This appendix provides a detailed description of the operation of each R4000 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

Figures at the end of this appendix list the bit encoding for the constant fields of each instruction, and the bit encoding for each individual instruction is included with that instruction.
Appendix A

A.1 Instruction Classes

CPU instructions are divided into the following classes:

- **Load** and **Store** instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is *base register + 16-bit immediate offset*.

- **Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.

- **Jump** and **Branch** instructions change the control flow of a program. Jumps are always made to absolute 26-bit word addresses (J-type format), or register addresses (R-type), for returns and dispatches. Branches have 16-bit offsets relative to the program counter (I-type). **Jump and Link** instructions save their return address in register 31.

- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPU instructions in Appendix B). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.

- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.
A.2 Instruction Formats

Every CPU instruction consists of a single word (32 bits) aligned on a word boundary and the major instruction formats are shown in Figure A-1.

**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>op</td>
<td>rs</td>
<td>rt</td>
<td>immediate</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>op</td>
<td>target</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>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **op**: 6-bit operation code
- **rs**: 5-bit source register specifier
- **rt**: 5-bit target (source/destination) or branch condition
- **immediate**: 16-bit immediate, branch displacement or address displacement
- **target**: 26-bit jump target address
- **rd**: 5-bit destination register specifier
- **shamt**: 5-bit shift amount
- **funct**: 6-bit function field

*Figure A-1 CPU Instruction Formats*
A.3 Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as rs, rt, immediate, etc.) are shown in lowercase names.

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.

Figures with the actual bit encoding for all the mnemonics are located at the end of this Appendix, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the Operation section describes the operation performed by each instruction using a high-level language notation. The R4000 can operate as either a 32- or 64-bit microprocessor and the operation for both modes is included with the instruction description.

Special symbols used in the notation are described in Table A-1.
CPU Instruction Set Details

### Table A-1  CPU Instruction Operation Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>Assignment.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x^</td>
<td>Replication of bit value x into a y-bit string. Note: x is always a single-bit value.</td>
</tr>
<tr>
<td>x_yz</td>
<td>Selection of bits y through z of bit string x. Little-endian bit notation is always 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 addition.</td>
</tr>
<tr>
<td>-</td>
<td>2’s complement or floating-point subtraction.</td>
</tr>
<tr>
<td>*</td>
<td>2’s complement or floating-point multiplication.</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>and</td>
<td>Bit-wise logical AND.</td>
</tr>
<tr>
<td>or</td>
<td>Bit-wise logical OR.</td>
</tr>
<tr>
<td>xor</td>
<td>Bit-wise logical XOR.</td>
</tr>
<tr>
<td>nor</td>
<td>Bit-wise logical NOR.</td>
</tr>
<tr>
<td>GPR[x]</td>
<td>General-Register x. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect.</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>Big-endian mode as configured at 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 (SR25 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 reversed by setting SR25. Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.</td>
</tr>
<tr>
<td>LLbit</td>
<td>Bit of state to specify synchronization instructions. Set by LL, cleared by ERET and Invalidate and read by SC.</td>
</tr>
<tr>
<td>T+i</td>
<td>Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked T+i: are executed at instruction cycle i relative to the start of execution of the instruction. Thus, an instruction which starts at time j executes operations marked T+i: at time i + j. The interpretation of the order of execution between two instructions or two operations which execute at the same time should be pessimistic; the order is not defined.</td>
</tr>
</tbody>
</table>
Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

\[
\text{GPR}[rt] \leftarrow \text{immediate} \ || \ 0^{16}
\]

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General-Purpose Register rt.

Example #2:

\[
(\text{immediate}_{15})^{16} \ || \ \text{immediate}_{15...0}
\]

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.
A.4 Load and Store Instructions

In the R4000 implementation, the instruction immediately following a load may use the loaded contents of the register. In such cases, the hardware interlocks, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code.

Two special instructions are provided in the R4000 implementation of the MIPS ISA, Load Linked and Store Conditional. 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.

In the load and store descriptions, the functions listed in Table A-2 are used to summarize the handling of virtual addresses and physical memory.

### Table A-2 Load and Store Common Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressTranslation</td>
<td>Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.</td>
</tr>
<tr>
<td>LoadMemory</td>
<td>Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the Access Type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.</td>
</tr>
<tr>
<td>StoreMemory</td>
<td>Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the Access Type field indicates which of each of the four bytes within the data word should be stored.</td>
</tr>
</tbody>
</table>
As shown in Table A-3, the Access Type field indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address in the addressed field. For a big-endian machine, this is the leftmost byte and contains the sign for a 2’s complement number; for a little-endian machine, this is the rightmost byte.

Table A-3  Access Type Specifications for Loads/Stores

<table>
<thead>
<tr>
<th>Access Type Mnemonic</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>

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low-order bits of the address.
A.5 Jump and Branch Instructions

All jump and branch instructions have an architectural delay of exactly one instruction. That is, the instruction immediately following a jump or branch (that is, occupying the delay slot) is always executed while the target instruction is being fetched from storage. A delay slot may not itself be occupied by a jump or branch instruction; however, this error is not detected and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the EPC register to point at the jump or branch instruction that precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.
A.6 Coprocessor Instructions

Coprocessors are alternate execution units, which have register files separate from the CPU. The MIPS architecture provides four coprocessor units, or classes, and these coprocessors have two register spaces, each space containing thirty-two 32-bit registers.

- The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor.
- The second space, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and the processor. Coprocessor instructions may alter registers in either space.

A.7 System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to/from coprocessors and to move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for writing to and reading from the CP0 registers.

Several CP0 instructions are defined to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.
ADD

Format:

ADD rd, rs, rt

Description:

The contents of general register rs and the contents of general register rt are added to form the result. The result is placed into general register rd. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

An overflow exception occurs if the carries out of bits 30 and 31 differ (2’s complement overflow). The destination register rd is not modified when an integer overflow exception occurs.

Operation:

<table>
<thead>
<tr>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPR[rd] ← (temp&lt;sub&gt;31&lt;/sub&gt;)&lt;sup&gt;32&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

Exceptions:

Integer overflow exception
ADDI  Add Immediate

Format:
ADDI  rt, rs, immediate

Description:
The 16-bit immediate is sign-extended and added to the contents of general register \( rs \) to form the result. The result is placed into general register \( rt \). In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

An overflow exception occurs if carries out of bits 30 and 31 differ (2’s complement overflow). The destination register \( rt \) is not modified when an integer overflow exception occurs.

Operation:

| 32 T: | GPR[rt] ← GPR[rs] + (immediate\(_{15}\))\(^{16} \) | immediate\(_{15...0}\) |
| 64 T: | temp ← GPR[rs] + (immediate\(_{15}\))\(^{48} \) | immediate\(_{15...0}\) |
|       | GPR[rt] ← (temp\(_{31}\))\(^{32} \) || temp\(_{31...0}\) |

Exceptions:
Integer overflow exception
ADDIU

Add Immediate Unsigned

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form the result. The result is placed into general register rt. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

\[
\begin{align*}
32 \quad T : & \quad \text{GPR} [rt] \leftarrow \text{GPR}[rs] + (\text{immediate}_{15})^{16} || \text{immediate}_{15...0} \\
64 \quad T : & \quad \text{temp} \leftarrow \text{GPR}[rs] + (\text{immediate}_{15})^{48} | | \text{immediate}_{15...0} \\
& \quad \text{GPR}[rt] \leftarrow (\text{temp}_{31})^{32} || \text{temp}_{31...0}
\end{align*}
\]

Exceptions:

None


**ADDU**  
Add Unsigned

**ADDU**

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

**Format:**

`ADDU rd, rs, rt`

**Description:**

The contents of general register `rs` and the contents of general register `rt` are added to form the result. The result is placed into general register `rd`. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the `ADD` instruction is that `ADDU` never causes an overflow exception.

**Operation:**

| 32 | T: | GPR[rd] ← GPR[rs] + GPR[rt] |
| 64 | T: | temp ← GPR[rs] + GPR[rt]  
GPR[rd] ← (temp₃₁)³² || temp₃₁...0 |

**Exceptions:**

None
AND

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>AND</td>
<td>100100</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:

AND rd, rs, rt

Description:

The contents of general register rs are combined with the contents of general register rt in a bit-wise logical AND operation. The result is placed into general register rd.

Operation:

32 T: GPR[rd] ← GPR[rs] and GPR[rt]

64 T: GPR[rd] ← GPR[rs] and GPR[rt]

Exceptions:

None
**ANDI**

**And 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>ANDI</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

ANDI rt, rs, immediate

**Description:**

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical AND operation. The result is placed into general register rt.

**Operation:**

| 32 | T: GPR[rt] ← 0^{16} || (immediate and GPR[rs]_{15...0}) |
| 64 | T: GPR[rt] ← 0^{48} || (immediate and GPR[rs]_{15...0}) |

**Exceptions:**

None
**BCzF**  
**Branch On Coprocessor z False**

<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>COPz</td>
<td>BC</td>
<td>BCF</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x*</td>
<td>0 1 0 0 0</td>
<td>0 0 0 0 0</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BCzF offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If coprocessor z’s condition signal (CpCond), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

**Operation:**

32  
T–1: condition ← not COC[z]  
T:  
target ← (offset_{15})^{14} || offset || 0^{2}  
T+1: if condition then  
PC ← PC + target  
endif

64  
T–1: condition ← not COC[z]  
T:  
target ← (offset_{15})^{46} || offset || 0^{2}  
T+1: if condition then  
PC ← PC + target  
endif

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.*
Exceptions:
- Coprocessor unusable exception

**Opcode Bit Encoding:**

<table>
<thead>
<tr>
<th>Bit #</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0F</td>
<td>0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>BC1F</td>
<td>0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>BC2F</td>
<td>0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Branch On Coprocessor z False (continued)
**BCzFL**  
**Branch On Coprocessor z**  
**False Likely**

<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>COPz</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>BCFL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BCzFL  offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor z’s condition line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.*
Appendix A

BCzFL  Branch On Coprocessor z
False Likely
(continued)

Operation:

<table>
<thead>
<tr>
<th>Bit #</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>T–1: condition ← not COC[z]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T: target ← (offset_{15})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T+1: if condition then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>T–1: condition ← not COC[z]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T: target ← (offset_{15})^46</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T+1: if condition then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions:
Coprocessor unusable exception

Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bit #</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCzFL</td>
<td>Bit #</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BC sub-opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Branch condition</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Coprocessor Unit Number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A-20  MIPS R4000 Microprocessor User’s Manual
**Format:**

BCzT  offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the coprocessor z’s condition signal (CpCond) is true, then the program branches to the target address, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

**Operation:**

<table>
<thead>
<tr>
<th>32</th>
<th>T–1: condition ← COC[z]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>target ← (offset15)_14</td>
</tr>
<tr>
<td>T+1</td>
<td>if condition then</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + target</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64</th>
<th>T–1: condition ← COC[z]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>target ← (offset15)_46</td>
</tr>
<tr>
<td>T+1</td>
<td>if condition then</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + target</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
</tbody>
</table>

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.*
## BCzT  Branch On Coprocessor z True

(continued)

### Exceptions:

Coprocessor unusable exception

### Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Bit # 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0T</td>
<td>0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>BC1T</td>
<td>0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>BC2T</td>
<td>0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>

- **BCzT**
- **Coprocessor Unit Number**
- **BC sub-opcode**
- **Branch condition**
Branch On Coprocessor z
True Likely

**Format:**

```
BCzTL  offset
```

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of coprocessor z’s condition line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

**Operation:**

```
32  T−1: condition ← COC[z]  
    T:  target ← (offset15)14 || offset || 02  
    T+1: if condition then  
        else  
            PC ← PC + target  
        endif  
        NullifyCurrentInstruction  
64  T−1: condition ← COC[z]  
    T:  target ← (offset16)46 || offset || 02  
    T+1: if condition then  
        else  
            PC ← PC + target  
        endif  
```

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.
## BCzTL

### Branch On Coprocessor z

**True Likely**

*(continued)*

#### Exceptions:

Coprocessor unusable exception

#### Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>BCzTL</th>
<th>Bit #</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC0TL</td>
<td>Bit #</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 0 0 1 0 0 0 0 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC1TL</td>
<td>Bit #</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC2TL</td>
<td>Bit #</td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Opcodes**: Coprocessor Unit Number, BC sub-opcode, Branch condition
BEQ Branch On 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>BEQ</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>

**Format:**

BEQ rs, rt, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

**Operation:**

32 T: \[\text{target} \leftarrow (\text{offset}_{15})^{14} || \text{offset} || 0^2 \]
\[\text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt])\]
T+1: if condition then
\[\text{PC} \leftarrow \text{PC} + \text{target}\]
endif

64 T: \[\text{target} \leftarrow (\text{offset}_{15})^{46} || \text{offset} || 0^2 \]
\[\text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt])\]
T+1: if condition then
\[\text{PC} \leftarrow \text{PC} + \text{target}\]
endif

**Exceptions:**

None
BEQL Branch On Equal Likely

Format:

BEQL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

\[
\begin{align*}
32 & \quad T: \quad \text{target} \leftarrow (\text{offset}_{16})^{14} || \text{offset} || 0^2 \\
& \quad \text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt]) \\
& \quad T+1: \quad \text{if condition then} \\
& \qquad \text{PC} \leftarrow \text{PC} + \text{target} \\
& \qquad \text{else} \\
& \qquad \quad \text{NullifyCurrentInstruction} \\
& \qquad \text{endif} \\
& \end{align*}
\]

\[
\begin{align*}
64 & \quad T: \quad \text{target} \leftarrow (\text{offset}_{16})^{46} || \text{offset} || 0^2 \\
& \quad \text{condition} \leftarrow (\text{GPR}[rs] = \text{GPR}[rt]) \\
& \quad T+1: \quad \text{if condition then} \\
& \qquad \text{PC} \leftarrow \text{PC} + \text{target} \\
& \qquad \text{else} \\
& \qquad \quad \text{NullifyCurrentInstruction} \\
& \qquad \text{endif} \\
& \end{align*}
\]

Exceptions:

None
BGEZ  Branch On Greater Than Or Equal To Zero

<table>
<thead>
<tr>
<th>REGIMM</th>
<th>rs</th>
<th>BGEZ</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1</td>
<td>0 0 0 0 1</td>
<td>0 0 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BGEZ rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

**Operation:**

32 T: target ← (offset15)14 || offset || 02 condition ← (GPR[rs]31 = 0) T+1: if condition then PC ← PC + target endif

64 T: target ← (offset15)46 || offset || 02 condition ← (GPR[rs]63 = 0) T+1: if condition then PC ← PC + target endif

**Exceptions:**

None
**BGEZAL** Branch On Greater Than Or Equal To Zero And Link

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

**Format:**

BGEZAL rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however.

**Operation:**

```
32 T: target ← (offset_{15})^{14} || offset || 0^2
     condition ← (GPR[rs]_{31} = 0)
     GPR[31] ← PC + 8
     T+1: if condition then
          PC ← PC + target
          endif

64 T: target ← (offset_{15})^{46} || offset || 0^2
     condition ← (GPR[rs]_{63} = 0)
     GPR[31] ← PC + 8
     T+1: if condition then
          PC ← PC + target
          endif
```

**Exceptions:**

None
BGEZALL  Branch On Greater Than Or Equal To Zero And Link Likely

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BGEZALL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td>1 0 0 1 1</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>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

BGEZALL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

32 T: target ← (offset15)14 || offset || 02
condition ← (GPR[rs]31 = 0)
GPR[31] ← PC + 8
T+1: if condition then
    PC ← PC + target
    else
    NullifyCurrentInstruction
    endif

64 T: target ← (offset15)46 || offset || 02
condition ← (GPR[rs]63 = 0)
GPR[31] ← PC + 8
T+1: if condition then
    PC ← PC + target
    else
    NullifyCurrentInstruction
    endif

Exceptions:

None
BGEZL

Branch On Greater Than Or Equal To Zero Likely

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

Format:

BGEZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

\[
\begin{align*}
32 &: \text{target} \leftarrow (\text{offset}_{15})^{14} || \text{offset} || 0^2 \\
&: \text{condition} \leftarrow (\text{GPR}[rs]_{31} = 0) \\
&: \text{T+1: if condition then} \\
&: \quad \text{PC} \leftarrow \text{PC} + \text{target} \\
&: \quad \text{else} \\
&: \quad \text{NullifyCurrentInstruction} \\
&: \quad \text{endif} \\
64 &: \text{target} \leftarrow (\text{offset}_{15})^{46} || \text{offset} || 0^2 \\
&: \text{condition} \leftarrow (\text{GPR}[rs]_{83} = 0) \\
&: \text{T+1: if condition then} \\
&: \quad \text{PC} \leftarrow \text{PC} + \text{target} \\
&: \quad \text{else} \\
&: \quad \text{NullifyCurrentInstruction} \\
&: \quad \text{endif}
\end{align*}
\]

Exceptions:

None
**BGTZ**  
**Branch On Greater Than Zero**

<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>BGTZ</td>
<td>0 0 0 1 1 1</td>
<td>rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>offset</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BGTZ rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs are compared to zero. If the contents of general register rs have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction.

**Operation:**

| 32 | target ← (offset15)14 || offset || 02  
condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 032)  
T+1: if condition then  
PC ← PC + target  
endif | 64 | target ← (offset15)46 || offset || 02  
condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 064)  
T+1: if condition then  
PC ← PC + target  
endif |

**Exceptions:**

None
### BGTZL

**Branch On Greater Than Zero Likely**

<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>00000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

BGTZL rs, offset

#### Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs are compared to zero. If the contents of general register rs have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

#### Operation:

32 T: target ← (offset_{15})^{14} || offset || 0^2  
   condition ← (GPR[rs]_{31} = 0) and (GPR[rs] \neq 0^{32})
   T+1: if condition then
     PC ← PC + target
     else
     NullifyCurrentInstruction
     endif

64 T: target ← (offset_{15})^{46} || offset || 0^2  
   condition ← (GPR[rs]_{63} = 0) and (GPR[rs] \neq 0^{64})
   T+1: if condition then
     PC ← PC + target
     else
     NullifyCurrentInstruction
     endif

#### Exceptions:

None
CPU Instruction Set Details

**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 0 0 0 1 1 0</td>
<td>rs</td>
<td>0</td>
<td>0 0 0 0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BLEZ rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs are compared to zero. If the contents of general register rs have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

**Operation:**

| 32 | T: target ← (offset_{15})^{14} || offset || 0^2 |
|----|-------------------------------------------------|
|    | condition ← (GPR[rs]_{31} = 1) or (GPR[rs] = 0^{32}) |
|    | T+1: if condition then |
|    | PC ← PC + target |
|    | endif |
| 64 | T: target ← (offset_{15})^{46} || offset || 0^2 |
|    | condition ← (GPR[rs]_{63} = 1) or (GPR[rs] = 0^{64}) |
|    | T+1: if condition then |
|    | PC ← PC + target |
|    | endif |

**Exceptions:**

None
## BLEZL

**Branch on Less Than Or Equal To Zero Likely**

<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>BLEZL</td>
<td>rs</td>
<td>0</td>
<td>000000</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BLEZL rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs is compared to zero. If the contents of general register rs have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

**Operation:**

32  T:  target ← (offset₁₅)₁₄ || offset || 0²
condition ← (GPR[rs]₃₁ = 1) or (GPR[rs] = 0³₂)
T+1:  if condition then
       PC ← PC + target
     else
       NullifyCurrentInstruction
     endif

64  T:  target ← (offset₁₅)₄₆ || offset || 0²
condition ← (GPR[rs]₆₃ = 1) or (GPR[rs] = 0₆₄)
T+1:  if condition then
       PC ← PC + target
     else
       NullifyCurrentInstruction
     endif

**Exceptions:**

None
**BLTZ**  
Branch On Less Than Zero

**Format:**

BLTZ rs, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction.

**Operation:**

| 32 | target ← (offset_{15})^{14} || offset || 0^2 |
|    | condition ← (GPR[rs]_{31} = 1) |
| T+1: if condition then |
|    | PC ← PC + target |
|    | endif |

| 64 | target ← (offset_{15})^{46} || offset || 0^2 |
|    | condition ← (GPR[rs]_{63} = 1) |
| T+1: if condition then |
|    | PC ← PC + target |
|    | endif |

**Exceptions:**

None
BLTZAL Branch On Less Than Zero And Link

<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>REGIMM</td>
<td>0 0 0 0 0 1</td>
<td>rs</td>
<td>BLTZAL</td>
<td>1 0 0 0 0</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
BLTZAL rs, offset

Description:
A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction with register 31 specified as rs is not trapped, however.

Operation:

32 T: target ← (offset15)14 || offset || 02
condition ← (GPR[rs]31 = 1)
GPR[31] ← PC + 8
T+1: if condition then
       PC ← PC + target
endif

64 T: target ← (offset15)46 || offset || 02
condition ← (GPR[rs]83 = 1)
GPR[31] ← PC + 8
T+1: if condition then
       PC ← PC + target
endif

Exceptions:
None
## BLTZALL  
**Branch On Less Than Zero And Link Likely**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM</td>
<td>rs</td>
<td>BLTZALL</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td></td>
<td>1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:

BLTZALL rs, offset

### Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, r31. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register rs may not be general register 31, because such an instruction is not restartable. An attempt to execute this instruction with register 31 specified as rs is not trapped, however. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

### Operation:

- **T**: target ← (offset15)14 || offset || 02  
  condition ← (GPR[rs]31 = 1)  
  GPR[31] ← PC + 8  
  T+1: if condition then  
  PC ← PC + target  
  else NullifyCurrentInstruction  
  endif

- **T**: target ← (offset15)46 || offset || 02  
  condition ← (GPR[rs]63 = 1)  
  GPR[31] ← PC + 8  
  T+1: if condition then  
  PC ← PC + target  
  else NullifyCurrentInstruction  
  endif

### Exceptions:

None
BLTZL  Branch On Less Than Zero Likely

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGIMM 000001</td>
<td>rs</td>
<td>BLTZL 00010</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the contents of general register rs have the sign bit set, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

\[
\begin{align*}
32 & & T: & \text{target} & \leftarrow (\text{offset}_{15})^{14} \mid \text{offset} \mid 0^2 \\
& & & \text{condition} & \leftarrow (\text{GPR}[rs]_{31} = 1) \\
& & T+1: & \text{if condition then} \\
& & & \text{PC} & \leftarrow \text{PC} + \text{target} \\
& & & \text{else} \\
& & & \text{NullifyCurrentInstruction} \\
& & & \text{endif} \\
64 & & T: & \text{target} & \leftarrow (\text{offset}_{15})^{46} \mid \text{offset} \mid 0^2 \\
& & & \text{condition} & \leftarrow (\text{GPR}[rs]_{63} = 1) \\
& & T+1: & \text{if condition then} \\
& & & \text{PC} & \leftarrow \text{PC} + \text{target} \\
& & & \text{else} \\
& & & \text{NullifyCurrentInstruction} \\
& & & \text{endif}
\end{align*}
\]

Exceptions:

None
### 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</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

BNE rs, rt, offset

**Description:**

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

**Operation:**

32

T: \( \text{target} \leftarrow (\text{offset}_{15})^{14} \| \text{offset} \| 0^2 \)  
condition \( \leftarrow (\text{GPR}[rs] \neq \text{GPR}[rt]) \)  
T+1: if condition then  
PC \( \leftarrow \) PC + target  
endif

64

T: \( \text{target} \leftarrow (\text{offset}_{15})^{46} \| \text{offset} \| 0^2 \)  
condition \( \leftarrow (\text{GPR}[rs] \neq \text{GPR}[rt]) \)  
T+1: if condition then  
PC \( \leftarrow \) PC + target  
endif

**Exceptions:**

None
BNEL Branch On Not Equal Likely

<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 0 1 0 1 0 1</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>

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

**Description:**
A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register rs and the contents of general register rt are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

**Operation:**

32

| T:  | target ← (offset_{15})^{14} || offset || 0^2 condition ← (GPR[rs] \neq GPR[rt]) |
|-----|---------------------------------|---------------------------------|
| T+1: | if condition then \ | else \ | endif \ |
| | PC ← PC + target \ | NullifyCurrentInstruction |

64

| T:  | target ← (offset_{15})^{46} || offset || 0^2 condition ← (GPR[rs] \neq GPR[rt]) |
|-----|---------------------------------|---------------------------------|
| T+1: | if condition then \ | else \ | endif \ |
| | PC ← PC + target \ | NullifyCurrentInstruction |

**Exceptions:**
None
### BREAK Breakpoint

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>code</td>
<td>BREAK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>0 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

```
BREAK
```

#### Description:

A breakpoint trap 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.

#### Operation:

```
32, 64 T: BreakpointException
```

#### Exceptions:

Breakpoint exception
Appendix A

CACHE op, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

If CP0 is not usable (User or Supervisor mode) the CP0 enable bit in the Status register is clear, and a coprocessor unusable exception is taken. The operation of this instruction on any operation/cache combination not listed below, or on a secondary cache when none is present, is undefined. The operation of this instruction on uncached addresses is also undefined.

The Index operation uses part of the virtual address to specify a cache block.

For a primary cache of $2^{\text{CACHEBITS}}$ bytes with $2^{\text{LINEBITS}}$ bytes per tag, $\text{vAddr}_{\text{CACHEBITS} \ldots \text{LINEBITS}}$ specifies the block.

For a secondary cache of $2^{\text{CACHEBITS}}$ bytes with $2^{\text{LINEBITS}}$ bytes per tag, $\text{pAddr}_{\text{CACHEBITS} \ldots \text{LINEBITS}}$ specifies the block.

Index Load Tag also uses $\text{vAddr}_{\text{LINEBITS} \ldots 3}$ to select the doubleword for reading ECC or parity. When the CE bit of the Status register is set, Hit WriteBack, Hit WriteBack Invalidate, Index WriteBack Invalidate, and Fill also use $\text{vAddr}_{\text{LINEBITS} \ldots 3}$ to select the doubleword that has its ECC or parity modified. This operation is performed unconditionally.

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.
Write back from a primary cache goes to the secondary cache (if there is one), otherwise to memory. Write back from a secondary cache always goes to memory. A secondary write back always writes the most recent data; the data comes from the primary data cache, if present, and modified (the $W$ bit is set). Otherwise the data comes from the specified secondary cache. The address to be written is specified by the cache tag and not the translated physical address.

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions.

Bits 17...16 of the instruction specify the cache as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I</td>
<td>primary instruction</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>primary data</td>
</tr>
<tr>
<td>2</td>
<td>SI</td>
<td>secondary instruction</td>
</tr>
<tr>
<td>3</td>
<td>SD</td>
<td>secondary data (or combined instruction/data)</td>
</tr>
</tbody>
</table>
Bits 20..18 (this value is listed under the **Code** column) of the instruction specify the operation as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I, SI</td>
<td>Index Invalidate</td>
<td>Set the cache state of the cache block to Invalid.</td>
</tr>
<tr>
<td>0</td>
<td>D</td>
<td>Index Writeback Invalidate</td>
<td>Examine the cache state and Writeback bit (W bit) of the primary data cache block at the index specified by the virtual address. If the state is not Invalid and the W bit is set, write the block back to the secondary cache (if present) or to memory (if no secondary cache). The address to write is taken from the primary cache tag. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR'd into the computed check bits during the write to the secondary cache for the addressed doubleword. Set the cache state of primary cache block to Invalid. The W bit is unchanged (and irrelevant because the state is Invalid).</td>
</tr>
<tr>
<td>0</td>
<td>SD</td>
<td>Index Writeback Invalidate</td>
<td>Examine the cache state of the secondary data cache block at the index specified by the physical address. If the block is dirty (the state is Dirty Exclusive or Dirty Shared), write the data back to memory. Like all secondary writebacks, the operation writes any modified data for the addresses from the primary data cache. The address to write is taken from the secondary cache tag. The PIdx field of the secondary tag is used to determine the locations in the primaries to check for matching primary blocks. In all cases, set the state of the secondary cache block and all matching primary subblocks to Invalid. No Invalidate is sent on the R4000's system interface.</td>
</tr>
<tr>
<td>1</td>
<td>All</td>
<td>Index Load Tag</td>
<td>Read the tag for the cache block at the specified index and place it into the TagLo and TagHi CP0 registers, ignoring any ECC or parity errors. Also load the data ECC or parity bits into the ECC register.</td>
</tr>
<tr>
<td>2</td>
<td>All</td>
<td>Index Store Tag</td>
<td>Write the tag for the cache block at the specified index from the TagLo and TagHi CP0 registers. The processor uses computed parity for the primary caches and the TagLo register in the case of the secondary cache.</td>
</tr>
<tr>
<td>Code</td>
<td>Caches</td>
<td>Name</td>
<td>Operation</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>--------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3</td>
<td>SD</td>
<td>Create Dirty</td>
<td>This operation is used to avoid loading data needlessly from memory when writing new contents into an entire cache block. If the cache block is valid but does not contain the specified address (a valid miss) the secondary block is vacated. The data is written back to memory if dirty and all matching blocks in both primary caches are invalidated. As usual during a secondary writeback, if the primary data cache contains modified data (matching blocks with W bit set) that modified data is written to memory. If the cache block is valid and contains the specified physical address (a hit), the operation cleans up the primary caches to avoid virtual aliases: all blocks in both primary caches that match the secondary line are invalidated without writeback. Note that the search for matching primary blocks uses the virtual index of the Pldx field of the secondary cache tag (the virtual index when the location was last used) and not the virtual index of the virtual address used in the operation (the virtual index where the location will now be used). If the secondary tag and address do not match (miss), or the tag and address do match (hit) and the block is in a shared state, an invalidate for the specified address is sent over the System interface. In all cases, the cache block tag must be set to the specified physical address, the cache state must be set to <strong>Dirty</strong> Exclusive, and the virtual index field set from the virtual address. The CH bit in the Status register is set or cleared to indicate a hit or miss.</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>Create Dirty</td>
<td>This operation is used to avoid loading data needlessly from secondary cache or memory when writing new contents into an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the secondary cache (if present) or otherwise to memory. In all cases, set the cache block tag to the specified physical address, set the cache state to <strong>Dirty</strong> Exclusive.</td>
</tr>
<tr>
<td>4</td>
<td>I,D</td>
<td>Hit Invalidate</td>
<td>If the cache block contains the specified address, mark the cache block invalid.</td>
</tr>
<tr>
<td>4</td>
<td>SI, SD</td>
<td>Hit Invalidate</td>
<td>If the cache block contains the specified address, mark the cache block invalid and also invalidate all matching blocks, if present, in the primary caches (the Pldx field of the secondary tag is used to determine the locations in the primaries to search). The CH bit in the Status register is set or cleared to indicate a hit or miss.</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>Hit Writeback</td>
<td>If the cache block contains the specified address, write the data back if it is dirty, and mark the cache block invalid. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the <strong>ECC</strong> register is XOR’d into the computed check bits during the write to the secondary cache for the addressed doubleword.</td>
</tr>
</tbody>
</table>
### Cache (continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Caches</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>SD</td>
<td>Hit Writeback Invalidate</td>
<td>If the cache block contains the specified address, write back the data (if dirty), and mark the secondary cache block and all matching blocks in both primary caches invalid. As usual with secondary writebacks, modified data in the primary data cache (matching block with the W bit set) is used during the writeback. The PIdx field of the secondary tag is used to determine the locations in the primaries to check for matching primary blocks. The CH bit in the Status register is set or cleared to indicate a hit or miss.</td>
</tr>
<tr>
<td>5</td>
<td>I</td>
<td>Fill</td>
<td>Fill the primary instruction cache block from secondary cache or memory. If the CE bit of the Status register is set, the content of the ECC register is used instead of the computed parity bits for addressed doubleword when written to the instruction cache. For the R4000PC, the cache is filled from memory. For the R4000SC and R4000MC, the cache is filled from the secondary cache whether or not the secondary cache block is valid or contains the specified address.</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>Hit Writeback</td>
<td>If the cache block contains the specified address, and the W bit is set, write back the data. The W bit is not cleared; a subsequent miss to the block will write it back again. This second writeback is redundant, but not incorrect. When a secondary cache is present, and the CE bit of the Status register is set, the content of the ECC register is XOR’d into the computed check bits during the write to the secondary cache for the addressed doubleword. Note: The W bit is not cleared during this operation due to an artifact of the implementation; the W bit is implemented as part of the data side of the cache array so that it can be written during a data write.</td>
</tr>
<tr>
<td>6</td>
<td>SD</td>
<td>Hit Writeback</td>
<td>If the cache block contains the specified address, and the cache state is Dirty Exclusive or Dirty Shared, data is written back to memory. The cache state is unchanged; a subsequent miss to the block causes it to be written back again. This second writeback is redundant, but not incorrect. The CH bit in the Status register is set or cleared to indicate a hit or miss. The writeback looks in the primary data cache for modified data, but does not invalidate or clear the Writeback bit in the primary data cache. Note: The state of the secondary block is not changed to clean during this operation because the W bit of matching sub-blocks cannot be cleared to put the primary block in a clean state.</td>
</tr>
<tr>
<td>6</td>
<td>I</td>
<td>Hit Writeback</td>
<td>If the cache block contains the specified address, data is written back unconditionally. When a secondary cache is present, and the CE bit of the Status register is set, the contents of the ECC register is XOR’d into the computed check bits during the write to the secondary cache for the addressed doubleword.</td>
</tr>
</tbody>
</table>
This operation is used to change the virtual index of secondary cache contents, avoiding unnecessary memory operations. If the cache block contains the specified address, invalidate matching blocks in the primary caches at the index formed by concatenating PIdx in the secondary cache tag (not the virtual address of the operation) and vAddr_{11...4}, and then set the virtual index field of the secondary cache tag from the specified virtual address. Modified data in the primary data cache is not preserved by the operation and should be explicitly written back before this operation. The CH bit in the Status register is set or cleared to indicate a hit or miss.

Operation:

32, 64 T: vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base]
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)
CacheOp (op, vAddr, pAddr)

Exceptions:
Coprocessor unusable exception
Appendix A

**Appendix A**

CFCz  Move Control From Coprocessor  CFCz

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPz</td>
<td>CF</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 x x*</td>
<td>0 0 0 1 0</td>
<td>0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

CFCz rt, rd

**Description:**

The contents of coprocessor control register rd of coprocessor unit z are loaded into general register rt.

This instruction is not valid for CP0.

**Operation:**

<table>
<thead>
<tr>
<th>Opcode Bit Encoding:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFCz Bit #31 30 29 28 27 26 25 24 23 22 21 0</td>
</tr>
<tr>
<td>CFC1 0 1 0 0 0 0 1 0 0 0 1 0</td>
</tr>
<tr>
<td>Bit #31 30 29 28 27 26 25 24 23 22 21 0</td>
</tr>
<tr>
<td>CFC2 0 1 0 0 1 0 0 0 0 1 0</td>
</tr>
</tbody>
</table>

**Exceptions:**

Coprocessor unusable exception

* Opcode Bit Encoding:
**COPz**

**Coprocessor Operation**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPz</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CO</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cofun</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description:**
A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache/memory system. Details of coprocessor operations are contained in Appendix B.

**Operation:**

32, 64 T: CoprocessorOperation (z, cofun)

**Exceptions:**
- Coprocessor unusable exception
- Coprocessor interrupt or Floating-Point Exception (R4000 CP1 only)

**Opcode Bit Encoding:**

<table>
<thead>
<tr>
<th>COPz Bit #</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Bit #</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>COP1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Bit #</td>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>COP2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- Opcode
- CO sub-opcode (see end of Appendix A)
- Coprocessor Unit Number
## CTCz

**Move Control to Coprocessor**

### Format:

CTCz rt, rd

### Description:

The contents of general register rt are loaded into control register rd of coprocessor unit z.

This instruction is not valid for CP0.

### Operation:

<table>
<thead>
<tr>
<th>32,64</th>
<th>T:</th>
<th>data ← GPR[rt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T + 1</td>
<td>CCR[z,rd] ← data</td>
<td></td>
</tr>
</tbody>
</table>

### Exceptions:

Coprocessor unusable

---

*See “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.*
**DADD** Doubleword Add

<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>DADD</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>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

DADD rd, rs, rt

**Description:**

The contents of general register rs and the contents of general register rt are added to form the result. The result is placed into general register rd.

An overflow exception occurs if the carries out of bits 62 and 63 differ (2’s complement overflow). The destination register rd is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

| 64 | T: | GPR[rd] ← GPR[rs] + GPR[rt] |

**Exceptions:**

- Integer overflow exception
- Reserved instruction exception (R4000 in 32-bit mode)
DADDI  Doubleword Add 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>DADDI 0 1 1 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

DADDI rt, rs, immediate

Description:

The 16-bit immediate is sign-extended and added to the contents of general register rs to form the result. The result is placed into general register rt.

An overflow exception occurs if carries out of bits 62 and 63 differ (2’s complement overflow). The destination register rt is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64
T: GPR[rt] ← GPR[rs] + (immediate_{15})^{48} + immediate_{15...0}
```

Exceptions:

- Integer overflow exception
- Reserved instruction exception (R4000 in 32-bit mode)
DADDIU  
Doubleword Add Immediate Unsigned  
DADDIU

31  26  25  21  20  16  15  0
<table>
<thead>
<tr>
<th>DADDIU</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>011001</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Format:
DADDIU rt, rs, immediate

Description:
The 16-bit immediate is sign-extended and added to the contents of general register rs to form the result. The result is placed into general register rt. No integer overflow exception occurs under any circumstances.
The only difference between this instruction and the DADDI instruction is that DADDIU never causes an overflow exception.
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

| 64 | T: | GPR [rt] ← GPR[rs] + (immediate_{15}^{48} || immediate_{15...0}) |

Exceptions:
Reserved instruction exception (R4000 in 32-bit mode)
DADDU Doubleword Add Unsigned

Format:
DADDU rd, rs, rt

Description:
The contents of general register rs and the contents of general register rt are added to form the result. The result is placed into general register rd.

No overflow exception occurs under any circumstances.

The only difference between this instruction and the DADD instruction is that DADDU never causes an overflow exception.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

| 64 | T: GPR[rd] ← GPR[rs] + GPR[rt] |

Exceptions:
Reserved instruction exception (R4000 in 32-bit mode)
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>0</td>
<td>00000000000000</td>
<td>DDIV</td>
<td>011110</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

DDIV rs, rt

Description:

The contents of general register rs are divided by the contents of general register rt, treating both operands as 2’s complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

| 64 | T–2: | LO | ← undefined |
|    |      | HI | ← undefined |
| T–1: | LO | ← undefined |
|      | HI | ← undefined |
| T:   | LO | ← GPR[rs] div GPR[rt] |
|      | HI | ← GPR[rs] mod GPR[rt] |

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)
**DDIVU Doubleword Divide Unsigned**

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDIVU rs, rt</td>
</tr>
</tbody>
</table>

**Description:**

The contents of general register rs are divided by the contents of general register rt, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

<table>
<thead>
<tr>
<th>64</th>
<th>T–2:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LO</td>
</tr>
<tr>
<td></td>
<td>HI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T–1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
DIV

**Format:**

DIV rs, rt

**Description:**

The contents of general register rs are divided by the contents of general register rt, treating both operands as 2’s complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more instructions.
DIV
Divide
(continued)

Operation:

| 32 | T–2: | LO  | ← undefined |
|    |      | HI  | ← undefined |
| T–1: | LO  | ← undefined |
|      | HI  | ← undefined |
| T:  | LO  | ← GPR[rs] div GPR[rt] |
|      | HI  | ← GPR[rs] mod GPR[rt] |
| 64 | T–2: | LO  | ← undefined |
|    |      | HI  | ← undefined |
| T–1: | LO  | ← undefined |
|      | HI  | ← undefined |
| T:  | q   | ← GPR[rs]_31...0 div GPR[rt]_31...0 |
|      | r   | ← GPR[rs]_31...0 mod GPR[rt]_31...0 |
|      | LO  | ← (q_{31})^{32} || q_{31...0} |
|      | HI  | ← (r_{31})^{32} || r_{31...0} |

Exceptions:

None
CPU Instruction Set Details

**DIVU**

**Divide Unsigned**

<table>
<thead>
<tr>
<th>Format:</th>
<th>DIVU rs, rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The contents of general register rs are divided by the contents of general register rt, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.</td>
</tr>
</tbody>
</table>

In 64-bit mode, the operands must be valid sign-extended, 32-bit values. This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register LO, and the remainder word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of HI or LO from writes by two or more instructions.
### DIVU

**Divide Unsigned**

**DIVU**

#### Operation:

<table>
<thead>
<tr>
<th>32</th>
<th>T–2:</th>
<th>LO</th>
<th>← undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td>T–1:</td>
<td>LO</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td>T:</td>
<td>LO</td>
<td>← (0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← (0</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>T–2:</td>
<td>LO</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td>T–1:</td>
<td>LO</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← undefined</td>
<td></td>
</tr>
<tr>
<td>T:</td>
<td>q</td>
<td>← (0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>← (0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LO</td>
<td>← (q_{31})^{32}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI</td>
<td>← (r_{31})^{32}</td>
<td></td>
</tr>
</tbody>
</table>

#### Exceptions:

None
DMFC0 Doubleword Move From System Control Coprocessor  

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>DMF</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6      5      5      5      11

Format:
DMFC0 rt, rd

Description:
The contents of coprocessor register rd of the CP0 are loaded into general register rt.

This operation is defined for the R4000 operating in 64-bit mode and in 32-bit kernel mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception. All 64-bits of the general register destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
64       T:    data ← CPR[0,rd]
65       T+1:  GPR[rt] ← data
```

Exceptions:
Coprocessor unusable exception
Reserved instruction exception  (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)
Appendix A

DMTC0  Doubleword Move To System Control Coprocessor  DMTC0

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>DMT</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1 0 0 0</td>
<td>0 0 1 0 1</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td>1 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

DMTC0 rt, rd

**Description:**

The contents of general register rt are loaded into coprocessor register rd of the CP0.

This operation is defined for the R4000 operating in 64-bit mode or in 32-bit kernel mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception.

All 64-bits of the coprocessor 0 register are written from the general register source. The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

**Operation:**

64  
T:  data ← GPR[rt]
T+1: CPR[0,rd] ← data

**Exceptions:**

Coprocessor unusable exception  (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)
Format:
DMULT rs, rt

Description:
The contents of general registers rs and rt are multiplied, treating both operands as 2's complement values. No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of HI or LO from writes by a minimum of two other instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

| 64 | T–2: | LO ← undefined |
|    | HI ← undefined |
| T–1: | LO ← undefined |
|    | HI ← undefined |
| T: | t ← GPR[rs] * GPR[rt] |
|    | LO ← t_{63...0} |
|    | HI ← t_{127...64} |

Exceptions:
Reserved instruction exception (R4000 in 32-bit mode)
Appendix A

**DMULTU**  Doubleword Multiply Unsigned  **DMULTU**

<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 000000</td>
<td>rs</td>
<td>rt</td>
<td>0000000000</td>
<td>DMULTU 011101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

DMULTU rs, rt

**Description:**

The contents of general register rs and the contents of general register rt are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of HI or LO from writes by a minimum of two instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

<table>
<thead>
<tr>
<th>64</th>
<th>T–2:</th>
<th>LO ← undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HI ← undefined</td>
</tr>
<tr>
<td>T–1:</td>
<td>LO ← undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI ← undefined</td>
<td></td>
</tr>
<tr>
<td>T:</td>
<td>t ← (0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LO ← t_{63...0}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HI ← t_{127...64}</td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
## DSLL: Doubleword Shift Left Logical

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>00000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSLL</td>
<td>111000</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:**

DSLL rd, rt, sa

**Description:**

The contents of general register rt are shifted left by sa bits, inserting zeros into the low-order bits. The result is placed in register rd.

**Operation:**

\[
64 \quad T: \quad s \leftarrow 0 \| sa \\
GPR[rd] \leftarrow GPR[rt]_{(63-s)} \| 0^s
\]

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
DSLLV

Doubleword Shift Left Logical Variable

\[
\begin{array}{cccccccc}
31 & 26 & 25 & 21 & 20 & 16 & 15 & 11 & 10 & 6 & 5 & 0 \\
\hline
\text{SPECIAL} & \text{rs} & \text{rt} & \text{rd} & 0 & 00000 & 0 & 010100 \\
0 & 5 & 5 & 5 & 5 & 6 \\
\end{array}
\]

**Format:**

DSLLV rd, rt, rs

**Description:**

The contents of general register rt are shifted left by the number of bits specified by the low-order six bits contained in general register rs, inserting zeros into the low-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

\[
\begin{align*}
64 & \quad T: \quad s \leftarrow \text{GPR}[rs]_{5...0} \\
& \quad \text{GPR}[rd] \leftarrow \text{GPR}[rt]_{(63-s)...0} \parallel 0^6
\end{align*}
\]

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
**Format:**

DSLL32 rd, rt, sa

**Description:**

The contents of general register rt are shifted left by 32+sa bits, inserting zeros into the low-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

\[
\begin{align*}
64 & \quad \text{T:} \quad s \leftarrow 1 \ || \ sa \\
& \quad \text{GPR[rd] } \leftarrow \text{GPR[rt]}_{(63-s)\ldots0} \ || \ 0^s
\end{align*}
\]

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
Format:
DSRA rd, rt, sa

Description:
The contents of general register rt are shifted right by sa bits, sign-extending the high-order bits. The result is placed in register rd.
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64 T: s ← 0 || sa
    GPR[rd] ← (GPR[rt]_{63})^s || GPR[rt]_{63..s}
```

Exceptions:
Reserved instruction exception (R4000 in 32-bit mode)
## DSRAV: Doubleword Shift Right Arithmetic Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0 0 0 0 0 0 0</td>
<td>DSRAV 0 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:

DSRAV rd, rt, rs

### Description:

The contents of general register `rt` are shifted right by the number of bits specified by the low-order six bits of general register `rs`, sign-extending the high-order bits. The result is placed in register `rd`.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

### Operation:

\[
\begin{align*}
64 & : \quad s \leftarrow \text{GPR}[rs]_{5...0} \\
& \quad \text{GPR}[rd] \leftarrow (\text{GPR}[rt]_{63}^6 \parallel \text{GPR}[rt]_{63...s})
\end{align*}
\]

### Exceptions:

- Reserved instruction exception (R4000 in 32-bit mode)
DSRA32 Doubleword Shift Right Arithmetic + 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>rd</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSRA32</td>
<td>1</td>
<td>1</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

DSRA32 rd, rt, sa

Description:

The contents of general register rt are shifted right by 32+sa bits, sign-extending the high-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

64 T: \( s \leftarrow 1 \ || \ sa \)

\[
GPR[rd] \leftarrow (GPR[rt]_{63})^s \ || \ GPR[rt]_{63...s}
\]

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)
## DSRL

### Doubleword Shift Right Logical

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>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>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:

DSRL rd, rt, sa

### Description:

The contents of general register rt are shifted right by sa bits, inserting zeros into the high-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

### Operation:

\[
\begin{align*}
64 & \quad T: \quad s & \leftarrow & 0 || sa \\
& & \text{GPR}[rd] & \leftarrow & 0^s || \text{GPR}[rt]_{63...s}
\end{align*}
\]

### Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)
Appendix A

DSRLV Doubleword Shift Right Logical Variable

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>DSRLV</td>
<td>010110</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:**

DSRLV rd, rt, rs

**Description:**

The contents of general register rt are shifted right by the number of bits specified by the low-order six bits of general register rs, inserting zeros into the high-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

| 64 T: | s ← GPR[rs]5...0 |
| GPR[rd] ← 0s || GPR[rt]63...s |

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
**DSRL32** Doubleword Shift Right Logical + 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>000000</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>DSRL32</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:**

DSRL32 rd, rt, sa

**Description:**

The contents of general register rt are shifted right by 32+sa bits, inserting zeros into the high-order bits. The result is placed in register rd.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

| 64 | T:  | s ← 1 || sa |
|----|-----|-------------|
|    | GPR[rd] ← 0^s || GPR[rt]_{63...s} |

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)
**DSUB**

**Doubleword Subtract**

<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>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>00000</td>
<td>DSUB</td>
<td>101110</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:**

DSUB rd, rs, rt

**Description:**

The contents of general register \( rt \) are subtracted from the contents of general register \( rs \) to form a result. The result is placed into general register \( rd \).

The only difference between this instruction and the DSUBU instruction is that DSUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (2’s complement overflow). The destination register \( rd \) is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

<table>
<thead>
<tr>
<th>64</th>
</tr>
</thead>
</table>

**Exceptions:**

- Integer overflow exception
- Reserved instruction exception (R4000 in 32-bit mode)
DSUBU  Doubleword Subtract 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>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>DSUBU</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

6 5 5 5 5 6

Format:

DSUBU rd, rs, rt

Description:

The contents of general register rt are subtracted from the contents of general register rs to form a result. The result is placed into general register rd.

The only difference between this instruction and the DSUB instruction is that DSUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

64  T:  GPR[rd] ← GPR[rs] – GPR[rt]

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)
ERET Exception Return

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CO</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></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:

ERET

Description:

ERET is the R4000 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing an error trap (SR2 = 1), then load the PC from the ErrorEPC and clear the ERL bit of the Status register (SR2). Otherwise (SR2 = 0), load the PC from the EPC, and clear the EXL bit of the Status register (SR1).

An ERET executed between a LL and SC also causes the SC to fail.

Operation:

32, 64

T: if SR2 = 1 then
   PC ← ErrorEPC
   SR ← SR31...3 || 0 || SR1...0
else
   PC ← EPC
   SR ← SR31...2 || 0 || SR0
endif
LLbit ← 0

Exceptions:

Coprocessor unusable exception
## J Jump

### Format:

J target

### Description:

The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

### Operation:

<table>
<thead>
<tr>
<th>32</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T:</td>
<td>temp ← target</td>
</tr>
<tr>
<td>T+1:</td>
<td>PC ← PC(_{31...28})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T:</td>
<td>temp ← target</td>
</tr>
<tr>
<td>T+1:</td>
<td>PC ← PC(_{63...28})</td>
</tr>
</tbody>
</table>

### Exceptions:

None
Appendix A

JAL  
Jump And Link

Format:
JAL target

Description:
The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, r31.

Operation:

| 32 | T: | temp ← target | GPR[31] ← PC + 8 | T+1: PC ← PC_{31..28} || temp || 0^2 |
|----|----|----------------|-----------------|----------------------------------|
| 64 | T: | temp ← target | GPR[31] ← PC + 8 | T+1: PC ← PC_{63..28} || temp || 0^2 |

Exceptions:
None
### JALR - Jump And Link Register

**Format:**
- JALR rs
- JALR rd, rs

**Description:**
The program unconditionally jumps to the address contained in general register rs, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register rd. The default value of rd, if omitted in the assembly language instruction, is 31.

Register specifiers rs and rd may not be equal, because such an instruction does not have the same effect when re-executed. However, an attempt to execute this instruction is not trapped, and the result of executing such an instruction is undefined.

Since instructions must be word-aligned, a **Jump and Link Register** instruction must specify a target register (rs) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

**Operation:**

<table>
<thead>
<tr>
<th>32, 64</th>
<th>T:</th>
<th>temp ← GPR [rs]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GPR[rd] ← PC + 8</td>
</tr>
</tbody>
</table>

**Exceptions:**
None
Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register rs, with a delay of one instruction.

Since instructions must be word-aligned, a Jump Register instruction must specify a target register (rs) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

<table>
<thead>
<tr>
<th>32, 64</th>
<th>T: temp ← GPR[rs]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T+1: PC ← temp</td>
</tr>
</tbody>
</table>

Exceptions:

None
**CPU Instruction Set Details**

**LB Load Byte**

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

**Format:**

`LB rt, offset(base)`

**Description:**

The 16-bit `offset` is sign-extended and added to the contents of general register `base` to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register `rt`.

**Operation:**

<table>
<thead>
<tr>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>T:  vAddr ← ((offset_{15})^{16}</td>
<td></td>
</tr>
<tr>
<td>(pAddr, uncached) ← AddressTranslation (vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>pAddr ← pAddr_{PSIZE - 1...3}</td>
<td></td>
</tr>
<tr>
<td>mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>byte ← vAddr_{2...0} xor BigEndianCPU^3</td>
<td></td>
</tr>
<tr>
<td>GPR[rt] ← (mem_{7+8*byte})^{24}</td>
<td></td>
</tr>
<tr>
<td>pAddr ← pAddr_{PSIZE - 1...3}</td>
<td></td>
</tr>
<tr>
<td>mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>byte ← vAddr_{2...0} xor BigEndianCPU^3</td>
<td></td>
</tr>
<tr>
<td>GPR[rt] ← (mem_{7+8*byte})^{56}</td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
## LBU

### Load Byte 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>LBU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 0</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

LBU rt, offset(base)

#### Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register rt.

#### Operation:

| 32 T: | vAddr ← (offset₁₅)₁₆ || offset₁₅...₀ + GPR[base] |
| --- | --- |
| vAddr ← AddressTranslation (vAddr, DATA) |
| pAddr ← pAddrPSIZE - 1...₃ || (pAddr₂...₀ xor ReverseEndian³) |
| mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA) |
| byte ← vAddr₂...₀ xor BigEndianCPU³ |
| GPR[rt] ← 0²⁴ || mem₇+₈* byte...₈* byte |

| 64 T: | vAddr ← (offset₁₅)₁₆ || offset₁₅...₀ + GPR[base] |
| --- | --- |
| vAddr ← AddressTranslation (vAddr, DATA) |
| pAddr ← pAddrPSIZE - 1...₃ || (pAddr₂...₀ xor ReverseEndian³) |
| mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA) |
| byte ← vAddr₂...₀ xor BigEndianCPU³ |
| GPR[rt] ← 0⁵₆ || mem₇+₈* byte...₈* byte |

#### Exceptions:

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
CPU Instruction Set Details

**LD Load Doubleword**

```plaintext
<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>LD</td>
<td>110111</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format:**

LD rt, offset(base)

**Description:**

The 16-bit `offset` is sign-extended and added to the contents of general register `base` to form a virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into general register `rt`.

If any of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

\[
\begin{align*}
\text{vAddr} & \leftarrow ((\text{offset}_{15})^{48} || \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
\text{mem} & \leftarrow \text{LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)} \\
\text{GPR}[\text{rt}] & \leftarrow \text{mem}
\end{align*}
\]

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit user mode
  R4000 in 32-bit supervisor mode)
LDCz Load Doubleword To Coprocessor

Format:
LDCz rt, offset(base)

Description:
The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The processor reads a doubleword from the addressed memory location and makes the data available to coprocessor unit z. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the rt field is non-zero.

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.
LDCz  Load Doubleword To Coprocessor

(continued)

**Operation:**

| 32  | T:  | vAddr ← (offset[15]^{16} || offset[15...0]) + GPR[base] |
|-----|-----|----------------------------------------------------------|
|     |     | (pAddr, uncached) ← AddressTranslation (vAddr, DATA)       |
|     |     | mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA) |
|     |     | COPzLD (rt, mem)                                          |

| 64  | T:  | vAddr ← ((offset[15]^{48} || offset[15...0]) + GPR[base] |
|-----|-----|----------------------------------------------------------|
|     |     | (pAddr, uncached) ← AddressTranslation (vAddr, DATA)       |
|     |     | mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA) |
|     |     | COPzLD (rt, mem)                                          |

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Coprocessor unusable exception

**Opcode Bit Encoding:**

<table>
<thead>
<tr>
<th>LDCz</th>
<th>Bit #31 30 29 28 27 26 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1</td>
<td>1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LDC2</th>
<th>Bit #31 30 29 28 27 26 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 1 0 1 1 0</td>
</tr>
</tbody>
</table>

- Opcode
- Coprocessor Unit Number
Appendix A

LDL Load Doubleword Left

<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>LDL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 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:**

LDL rt, offset(base)

**Description:**

This instruction can be used in combination with the LDR instruction to load a register with eight consecutive bytes from memory, when the bytes cross a doubleword boundary. LDL loads the left portion of the register with the appropriate part of the high-order doubleword; LDR loads the right portion of the register with the appropriate part of the low-order doubleword.

The LDL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte(s) of the register will not be changed.

![Memory and Register Diagram]

**Memory (big-endian)**

<table>
<thead>
<tr>
<th>address 8</th>
<th>address 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 10 11 12 13 14 15</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

**Register**

A B C D E F G H $24

**Before LDL** $24,3($0)

**After LDL** $24

A-86 MIPS R4000 Microprocessor User’s Manual
CPU Instruction Set Details

LDL

Load Doubleword Left
(continued)

The contents of general register \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LDL (or LDR) instruction which also specifies register \( rt \).

No address exceptions due to alignment are possible.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

\[
\begin{align*}
64 & \quad T: \quad v\text{Addr} \leftarrow ((\text{offset}_{15})^{48} || \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
& \quad \text{(pAddr, uncached)} \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
& \quad \text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1...3} || (\text{pAddr}_{2...0} \text{xor ReverseEndian}^3) \\
& \quad \text{if BigEndianMem} = 0 \text{ then} \\
& \quad \quad \text{pAddr} \leftarrow \text{pAddr}_{\text{PSIZE}-1...3} || 0^3 \\
& \quad \text{endif} \\
& \quad \text{byte} \leftarrow \text{vAddr}_{2...0} \text{xor BigEndianCPU}^3 \\
& \quad \text{mem} \leftarrow \text{LoadMemory (uncached, byte, pAddr, vAddr, DATA)} \\
& \quad \text{GPR}[rt] \leftarrow \text{mem}_{7+8\text{byte}...0} || \text{GPR}[rt]_{55...8\text{byte}...0}
\end{align*}
\]
**LDL**  
*Load Doubleword Left (continued)*

Given a doubleword in a register and a doubleword in memory, the operation of LDL is as follows:

<table>
<thead>
<tr>
<th>vAddr₂₀</th>
<th>destination</th>
<th>type</th>
<th>BigEndianCPU = 0</th>
<th></th>
<th>BigEndianCPU = 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LEM</td>
<td>BEM</td>
<td>LEM</td>
<td>BEM</td>
</tr>
<tr>
<td>0</td>
<td>P B C D E F G H</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>I J K L M N O P</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>O P C D E F G H</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>J K L M N O P H</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>N O P D E F G H</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>K L M N O P G H</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>M N O P E F G P</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>L M N O P F G H</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>L M N O P F G H</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>M N O P E F G H</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>K L M N O P G H</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>N O P D E F G H</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>J K L M N O P H</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>O P C D E F G H</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>I J K L M N O P</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>P B C D E F G H</td>
<td>0</td>
</tr>
</tbody>
</table>

*LEM*  
Little-endian memory (BigEndianMem = 0)

*BEM*  
BigEndianMem = 1

*Type*  
AccessType (see Table 2-1) sent to memory

*Offset*  
pAddr₂₀ sent to memory

**Exceptions:**

TLB refill exception  
TLB invalid exception  
Bus error exception  
Address error exception  
Reserved instruction exception (R4000 in 32-bit mode)
LDR
Load Doubleword Right

Format:

LDR rt, offset(base)

Description:

This instruction can be used in combination with the LDL instruction to load a register with eight consecutive bytes from memory, when the bytes cross a doubleword boundary. LDR loads the right portion of the register with the appropriate part of the low-order doubleword; LDL loads the left portion of the register with the appropriate part of the high-order doubleword.

The LDR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte(s) of the register will not be changed.

![Diagram of LDR instruction](image-url)
The contents of general register \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LDR (or LDL) instruction which also specifies register \( rt \).

No address exceptions due to alignment are possible.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

\[
\begin{align*}
64 & \quad T: \quad vAddr \leftarrow (\text{offset}_{15})^{48} || \text{offset}_{15...0} + \text{GPR}[\text{base}] \\
& \quad (\text{pAddr}, \text{uncached}) \leftarrow \text{AddressTranslation} (vAddr, \text{DATA}) \\
& \quad \text{pAddr} \leftarrow \text{pAddr}_{P\text{SIZE}-1...3} || (\text{pAddr}_{2...0} \text{ xor } \text{ReverseEndian}^3) \\
& \quad \text{if } \text{BigEndianMem} = 1 \text{ then} \\
& \quad \quad \text{pAddr} \leftarrow \text{pAddr}_{31...3} || 0^3 \\
& \quad \text{endif} \\
& \quad \text{byte} \leftarrow vAddr_{2...0} \text{ xor } \text{BigEndianCPU}^3 \\
& \quad \text{mem} \leftarrow \text{LoadMemory} (\text{uncached}, \text{byte}, \text{pAddr}, vAddr, \text{DATA}) \\
& \quad \text{GPR}[rt] \leftarrow \text{GPR}[rt]_{63...64-8*\text{byte}} || \text{mem}_{63...8*\text{byte}}
\end{align*}
\]
Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:

### LDR

**Register**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
</table>

**Memory**

<table>
<thead>
<tr>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>P</th>
</tr>
</thead>
</table>

### Table: LDR Operation

| vAddr<sub>2..0</sub> | destination | type | offset | BigEndianCPU = 0 | | BigEndianCPU = 1 | type | offset |
|-----------------|-------------|------|--------|------------------||------------------|------|--------|
| 0               | I J K L M N O | 7    | 0 0    | A B C D E F G I | 0 | 7 0    |
| 1               | A I J K L M O | 6    | 1 0    | A B C D E F I J | 1 | 6 0    |
| 2               | A B I J K L M | 5    | 2 0    | A B C D E I J K | 2 | 5 0    |
| 3               | A B C I J K L M | 4    | 3 0    | A B C D I J K L | 3 | 4 0    |
| 4               | A B C D I J K L | 3    | 4 0    | A B C I J K L M | 4 | 3 0    |
| 5               | A B C D E I J K | 2    | 5 0    | A B I J K L M N | 5 | 2 0    |
| 6               | A B C D E F I J | 1    | 6 0    | A I J K L M N O | 6 | 1 0    |
| 7               | A B C D E F G I | 0    | 7 0    | I J K L M N O P | 7 | 0 0    |

- **LEM**: Little-endian memory (BigEndianMem = 0)
- **BEM**: BigEndianMem = 1
- **Type**: AccessType (see Table 2-1) sent to memory
- **Offset**: pAddr<sub>2..0</sub> sent to memory

### Exceptions:

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)
### LH Load Halfword

<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>LH</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100001</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

LH rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register rt.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

**Operation:**

```plaintext
32 T:  
  vAddr ← ((offset_{15})^{16} || offset_{15...0}) + GPR[base]  
  (pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
  pAddr ← pAddr_{PSIZE - 1...3} || (pAddr_{2...0} xor (ReverseEndian || 0))  
  mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)  
  byte ← vAddr_{2...0} xor (BigEndianCPU^{2} || 0)  
  GPR[rt] ← (mem_{15+8*byte})^{16} || mem_{15+8*byte...8*byte}

64 T:  
  vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base]  
  (pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
  pAddr ← pAddr_{PSIZE - 1...3} || (pAddr_{2...0} xor (ReverseEndian || 0))  
  mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)  
  byte ← vAddr_{2...0} xor (BigEndianCPU^{2} || 0)  
  GPR[rt] ← (mem_{15+8*byte})^{48} || mem_{15+8*byte...8*byte}
```

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
**CPU Instruction Set Details**

**Format:**

LHU rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register rt.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

**Operation:**

| 32 | T: vAddr ← ((offset15)16 || offset15...0) + GPR[base] |
|----|-----------------------------------------------------|
|    | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|    | pAddr ← pAddrPSIZE - 1...3 || (pAddr2...0 xor (ReverseEndian || 0)) |
|    | mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA) |
|    | byte ← vAddr2...0 xor (BigEndianCPU2 || 0) |
|    | GPR[rt] ← 016 || mem15+8*byte...8*byte |
| 64 | T: vAddr ← ((offset15)48 || offset15...0) + GPR[base] |
|    | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|    | pAddr ← pAddrPSIZE - 1...3 || (pAddr2...0 xor (ReverseEndian || 0)) |
|    | mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA) |
|    | byte ← vAddr2...0 xor (BigEndianCPU2 || 0) |
|    | GPR[rt] ← 048 || mem15+8*byte...8*byte |

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus Error exception
- Address error exception
Format:

LL rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt. In 64-bit mode, the loaded word is sign-extended.

The processor begins checking the accessed word for modification by other processor and devices.

Load Linked and Store Conditional can be used to atomically update memory locations as shown:

```
L1:
  LL T1, (T0)
  ADD T2, T1, 1
  SC T2, (T0)
  BEQ T2, 0, L1
  NOP
```

This atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set. This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

The operation of LL is undefined if the addressed location is uncached and, for synchronization between multiple processors, the operation of LL is undefined if the addressed location is noncoherent. A cache miss that occurs between LL and SC may cause SC to fail, so no load or store operation should occur between LL and SC, otherwise the SC may never be successful. Exceptions also cause SC to fail, so persistent exceptions must be avoided. If either of the two least-significant bits of the effective address are non-zero, an address error exception takes place.
LL Load Linked (continued)

Operation:

32 T:  
vAddr ← ((offset_{15})^{16} || offset_{15...0}) + GPR[base]  
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
pAddr ← pAddr_{PSIZE-1...3} || (pAddr_{2...0} xor (ReverseEndian || 0^2))  
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)  
byte ← vAddr_{2...0} xor (BigEndianCPU || 0^2)  
GPR[rt] ← mem_{31+8*byte...8*byte}  
LLbit ← 1

64 T:  
vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base]  
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
pAddr ← pAddr_{PSIZE-1...3} || (pAddr_{2...0} xor (ReverseEndian || 0^2))  
mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)  
byte ← vAddr_{2...0} xor (BigEndianCPU || 0^2)  
GPR[rt] ← (mem_{31+8*byte})^{32} || mem_{31+8*byte...8*byte}  
LLbit ← 1

Exceptions:

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
Load Linked Doubleword (LLD)

**Format:**

LLD rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded into general register rt.

The processor begins checking the accessed word for modification by other processor and devices.

Load Linked Doubleword and Store Conditional Doubleword can be used to atomically update memory locations:

```
L1:
   LLD    T1, (T0)
   ADD    T2, T1, 1
   SCD    T2, (T0)
   BEQ    T2, 0, L1
   NOP
```

This atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set.
LLD Load Linked Doubleword  
(continued)

The operation of LLD is undefined if the addressed location is uncached and, for synchronization between multiple processors, the operation of LLD is undefined if the addressed location is noncoherent. A cache miss that occurs between LLD and SCD may cause SCD to fail, so no load or store operation should occur between LLD and SCD, otherwise the SCD may never be successful. Exceptions also cause SCD to fail, so persistent exceptions must be avoided.

This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

<table>
<thead>
<tr>
<th>64</th>
<th>T:</th>
</tr>
</thead>
<tbody>
<tr>
<td>vAddr ← (offset_{15})^{48}</td>
<td></td>
</tr>
<tr>
<td>(pAddr, uncached) ← AddressTranslation (vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)</td>
<td></td>
</tr>
<tr>
<td>GPR[rt] ← mem</td>
<td></td>
</tr>
<tr>
<td>LLbit ← 1</td>
<td></td>
</tr>
</tbody>
</table>

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)
Appendix A

**LUI**

**Load Upper Immediate**

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI rt, immediate</td>
</tr>
</tbody>
</table>

**Description:**
The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is placed into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

**Operation:**

| 32 T: | GPR[rt] ← immediate || 0^16 |
|-------|------------------------------|
| 64 T: | GPR[rt] ← (immediate_{15})^32 || immediate || 0^16 |

**Exceptions:**
None
CPU Instruction Set Details

**LW**

**Load Word**

**Format:**

LW rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt. In 64-bit mode, the loaded word is sign-extended. If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

**Operation:**

- **32 T:**
  
  \[
  \text{vAddr} \leftarrow ((\text{offset}_{15})^{16} || \text{offset}_{15...0}) + \text{GPR[base]}
  \]
  
  \[
  \text{pAddr} \leftarrow \text{AddressTranslation (vAddr, DATA)}
  \]
  
  \[
  \text{mem} \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)}
  \]
  
  \[
  \text{byte} \leftarrow \text{vAddr}_{2...0} \text{xor (BigEndianCPU || 0^2)}
  \]
  
  \[
  \text{GPR[rt]} \leftarrow \text{mem}_{31+8*\text{byte}...8*\text{byte}}
  \]

- **64 T:**
  
  \[
  \text{vAddr} \leftarrow ((\text{offset}_{15})^{48} || \text{offset}_{15...0}) + \text{GPR[base]}
  \]
  
  \[
  \text{pAddr} \leftarrow \text{AddressTranslation (vAddr, DATA)}
  \]
  
  \[
  \text{mem} \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)}
  \]
  
  \[
  \text{byte} \leftarrow \text{vAddr}_{2...0} \text{xor (BigEndianCPU || 0^2)}
  \]
  
  \[
  \text{GPR[rt]} \leftarrow (\text{mem}_{31+8*\text{byte}})^{32} || \text{mem}_{31+8*\text{byte}...8*\text{byte}}
  \]

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
Appendix A

LWCz  Load Word To Coprocessor  LWCz

<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>LWCz 1100xx*</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

LWCz rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The processor reads a word from the addressed memory location, and makes the data available to coprocessor unit z.

The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This instruction is not valid for use with CP0.

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.*
**LWCz**

**Load Word To Coprocessor**

(continued)

**LWCz**

**Operation:**

| 32 T: | vAddr ← ((offset_{15})^{16} || offset_{15...0}) + GPR[base] |
|-------|----------------------------------------------------------|
|       | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|       | pAddr ← pAddr_{PSIZE-1...3} || (pAddr_{2...0} xor (ReverseEndian || 0^2)) |
|       | mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA) |
|       | byte ← vAddr_{2...0} xor (BigEndianCPU || 0^2) |
|       | COPzLW (byte, rt, mem) |
| 64 T: | vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base] |
|       | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|       | pAddr ← pAddr_{PSIZE-1...3} || (pAddr_{2...0} xor (ReverseEndian || 0^2)) |
|       | mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA) |
|       | byte ← vAddr_{2...0} xor (BigEndianCPU || 0^2) |
|       | COPzLW (byte, rt, mem) |

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Coprocessor unusable exception

**Opcode Bit Encoding:**

<table>
<thead>
<tr>
<th>LWCz Bit #</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWC1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LWC2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- ** Opcode **
- ** Coprocessor Unit Number **
LWL Load Word Left

Format:

LWL rt, offset(base)

Description:

This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a word boundary. LWL loads the left portion of the register with the appropriate part of the high-order word; LWR loads the right portion of the register with the appropriate part of the low-order word.

The LWL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.

<table>
<thead>
<tr>
<th>address 4</th>
<th>memory (big-endian)</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>address 0</td>
<td>4 5 6 7</td>
<td>A B C D $24</td>
</tr>
</tbody>
</table>

LWL $24,1($0)

before

after

$24
The contents of general register \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LWL (or LWR) instruction which also specifies register \( rt \). No address exceptions due to alignment are possible.

**Operation:**

<table>
<thead>
<tr>
<th>Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 T</td>
<td>( \text{vAddr} \leftarrow (\text{offset}_{15})^{16}</td>
</tr>
<tr>
<td></td>
<td>(pAddr, uncached) \leftarrow \text{AddressTranslation (vAddr, DATA)}</td>
</tr>
<tr>
<td></td>
<td>pAddr \leftarrow pAddr_{\text{PSIZE}-1...3}</td>
</tr>
<tr>
<td></td>
<td>if BigEndianMem = 0 then</td>
</tr>
<tr>
<td></td>
<td>\hspace{1cm} pAddr \leftarrow pAddr_{\text{PSIZE}-1...2}</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
<tr>
<td></td>
<td>byte \leftarrow \text{vAddr}_{1...0} \text{ xor BigEndianCPU}^2</td>
</tr>
<tr>
<td></td>
<td>word \leftarrow \text{vAddr}_{2} \text{ xor BigEndianCPU}</td>
</tr>
<tr>
<td></td>
<td>mem \leftarrow \text{LoadMemory (uncached, 0</td>
</tr>
<tr>
<td></td>
<td>temp \leftarrow mem_{32*\text{word}+8*\text{byte}+7...32*\text{word}</td>
</tr>
<tr>
<td></td>
<td>GPR[rt] \leftarrow \text{temp}</td>
</tr>
<tr>
<td>64 T</td>
<td>( \text{vAddr} \leftarrow (\text{offset}_{15})^{48}</td>
</tr>
<tr>
<td></td>
<td>(pAddr, uncached) \leftarrow \text{AddressTranslation (vAddr, DATA)}</td>
</tr>
<tr>
<td></td>
<td>pAddr \leftarrow pAddr_{\text{PSIZE}-1...3}</td>
</tr>
<tr>
<td></td>
<td>if BigEndianMem = 0 then</td>
</tr>
<tr>
<td></td>
<td>\hspace{1cm} pAddr \leftarrow pAddr_{\text{PSIZE}-1...2}</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
<tr>
<td></td>
<td>byte \leftarrow \text{vAddr}_{1...0} \text{ xor BigEndianCPU}^2</td>
</tr>
<tr>
<td></td>
<td>word \leftarrow \text{vAddr}_{2} \text{ xor BigEndianCPU}</td>
</tr>
<tr>
<td></td>
<td>mem \leftarrow \text{LoadMemory (uncached, 0</td>
</tr>
<tr>
<td></td>
<td>temp \leftarrow mem_{32*\text{word}+8*\text{byte}+7...32*\text{word}</td>
</tr>
<tr>
<td></td>
<td>GPR[rt] \leftarrow (temp_{31})^{32}</td>
</tr>
</tbody>
</table>
Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:

### LWL

**Register**
- A
- B
- C
- D
- E
- F
- G
- H

**Memory**
- I
- J
- K
- L
- M
- N
- O
- P

<table>
<thead>
<tr>
<th>vAddr&lt;sub&gt;2,0&lt;/sub&gt;</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LEM</td>
<td>BEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S S S S P F G H</td>
<td>0</td>
<td>0 7</td>
<td>S S S I J K L</td>
<td>3</td>
<td>4 0</td>
</tr>
<tr>
<td>1</td>
<td>S S S S O P G H</td>
<td>1</td>
<td>0 6</td>
<td>S S S S J K L H</td>
<td>2</td>
<td>4 1</td>
</tr>
<tr>
<td>2</td>
<td>S S S S N O P H</td>
<td>2</td>
<td>0 5</td>
<td>S S S S K L G H</td>
<td>1</td>
<td>4 2</td>
</tr>
<tr>
<td>3</td>
<td>S S S S M N O P</td>
<td>3</td>
<td>0 4</td>
<td>S S S S L F G H</td>
<td>0</td>
<td>4 3</td>
</tr>
<tr>
<td>4</td>
<td>S S S S L F G H</td>
<td>0</td>
<td>4 3</td>
<td>S S S S M N O P</td>
<td>3</td>
<td>0 4</td>
</tr>
<tr>
<td>5</td>
<td>S S S S K L G H</td>
<td>1</td>
<td>4 2</td>
<td>S S S S N O P H</td>
<td>2</td>
<td>0 5</td>
</tr>
<tr>
<td>6</td>
<td>S S S S J K L H</td>
<td>2</td>
<td>4 1</td>
<td>S S S S O P G H</td>
<td>1</td>
<td>0 6</td>
</tr>
<tr>
<td>7</td>
<td>S S S S I J K L</td>
<td>3</td>
<td>4 0</td>
<td>S S S S P F G H</td>
<td>0</td>
<td>0 7</td>
</tr>
</tbody>
</table>

**Exceptions:**
- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
LWR

Load Word Right

Format:
LWR rt, offset(base)

Description:
This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a word boundary. LWR loads the right portion of the register with the appropriate part of the low-order word; LWL loads the left portion of the register with the appropriate part of the high-order word.

The LWR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, if bit 31 of the destination register is loaded, then the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.

LWR $24,4($0)

Before

<table>
<thead>
<tr>
<th>memory (big-endian)</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>A B C D</td>
</tr>
</tbody>
</table>

$24

After

| A B C 4 |
The contents of general register \( rt \) are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register \( rt \) and a following LWR (or LWL) instruction which also specifies register \( rt \). No address exceptions due to alignment are possible.

**Operation:**

\[
\begin{align*}
32 \quad T: & \quad vAddr \leftarrow ((\text{offset}_{15})^{16} \ || \ \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
& \quad (pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}) \\
& \quad pAddr \leftarrow pAddr_{\text{PSIZE}-1...3} \ || \ (pAddr_{2...0} \ xor \ ReverseEndian)^3 \\
& \quad \text{if BigEndianMem} = 1 \text{ then} \\
& \quad \quad pAddr \leftarrow pAddr_{\text{PSIZE}-31...3} \ || \ 0^3 \\
& \quad \quad \text{endif} \\
& \quad \text{byte} \leftarrow vAddr_{1...0} \ xor \ BigEndianCPU^2 \\
& \quad \text{word} \leftarrow vAddr_{2} \ xor \ BigEndianCPU \\
& \quad \text{mem} \leftarrow \text{LoadMemory}(\text{uncached}, 0 \ || \ \text{byte}, pAddr, vAddr, \text{DATA}) \\
& \quad \text{temp} \leftarrow \text{GPR}[rt]_{31...32-8*\text{byte}} \ || \ \text{mem}_{31+32*\text{word}...32*\text{word}+8*\text{byte}} \\
& \quad \text{GPR}[rt] \leftarrow \text{temp}
\end{align*}
\]

\[
\begin{align*}
64 \quad T: & \quad vAddr \leftarrow ((\text{offset}_{15})^{48} \ || \ \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
& \quad (pAddr, \text{uncached}) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}) \\
& \quad pAddr \leftarrow pAddr_{\text{PSIZE}-1...3} \ || \ (pAddr_{2...0} \ xor \ ReverseEndian)^3 \\
& \quad \text{if BigEndianMem} = 1 \text{ then} \\
& \quad \quad pAddr \leftarrow pAddr_{\text{PSIZE}-31...3} \ || \ 0^3 \\
& \quad \quad \text{endif} \\
& \quad \text{byte} \leftarrow vAddr_{1...0} \ xor \ BigEndianCPU^2 \\
& \quad \text{word} \leftarrow vAddr_{2} \ xor \ BigEndianCPU \\
& \quad \text{mem} \leftarrow \text{LoadMemory}(\text{uncached}, 0 \ || \ \text{byte}, pAddr, vAddr, \text{DATA}) \\
& \quad \text{temp} \leftarrow \text{GPR}[rt]_{31...32-8*\text{byte}} \ || \ \text{mem}_{31+32*\text{word}...32*\text{word}+8*\text{byte}} \\
& \quad \text{GPR}[rt] \leftarrow (\text{temp}_{31})^{32} \ || \ \text{temp}
\end{align*}
\]
LWR  Load Word Right
(continued)

Given a word in a register and a word in memory, the operation of LWR is as follows:

<table>
<thead>
<tr>
<th>vAddr_{2,0}</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
<th>BigEndianCPU = 0</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S S S S M N O P</td>
<td>0</td>
<td>0 4</td>
<td>X X X X E F G I</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X X X X E M N O</td>
<td>1</td>
<td>1 4</td>
<td>X X X X E F I J</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>X X X X E F M N</td>
<td>2</td>
<td>2 4</td>
<td>X X X X E I J K</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>X X X X E F G M</td>
<td>3</td>
<td>3 4</td>
<td>S S S S I J K L</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>S S S S I J K L</td>
<td>0</td>
<td>0 4</td>
<td>X X X X E F G M</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>X X X X E I J K</td>
<td>1</td>
<td>5 0</td>
<td>X X X X E F M N</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>X X X X E F I J</td>
<td>2</td>
<td>6 0</td>
<td>X X X X E M N O</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>X X X X E F G I</td>
<td>3</td>
<td>7 0</td>
<td>S S S S M N O P</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)
**BEM** BigEndianMem = 1
**Type**AccessType (see Table 2-1) sent to memory
**Offset**pAddr_{2..0} sent to memory
**S** sign-extend of destination_{31}
**X** either unchanged or sign-extend of destination_{31}

**Exceptions:**
- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
## LWU

**Load Word Unsigned**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<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>0</td>
</tr>
<tr>
<td>LWU</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 1 1</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Format:

LWU rt, offset(base)

### Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt. The loaded word is zero-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

### Operation:

64 T:  

\[
\begin{align*}
\text{vAddr} & \leftarrow ((\text{offset}_{15})^{48} \ || \ \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
(\text{pAddr, uncached}) & \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
\text{pAddr} & \leftarrow \text{pAddr}_{\text{PSIZE}-1...3} \ || \ (\text{pAddr}_{2...0} \ \text{xor (ReverseEndian || 0^2)}) \\
\text{mem} & \leftarrow \text{LoadMemory (uncached, WORD, pAddr, vAddr, DATA)} \\
\text{byte} & \leftarrow \text{vAddr}_{2...0} \ \text{xor (BigEndianCPU || 0^2)} \\
\text{GPR}[rt] & \leftarrow 0^{32} \ || \ \text{mem}_{31+8*\text{byte}...8*\text{byte}}
\end{align*}
\]

### Exceptions:

- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)
**MFC0**

**Move From System Control Coprocessor MFC0**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>MF</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>5</td>
<td>5</td>
<td>0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Format:**

MFC0 rt, rd

**Description:**

The contents of coprocessor register rd of the CP0 are loaded into general register rt.

**Operation:**

<table>
<thead>
<tr>
<th>32</th>
<th>T:</th>
<th>data ← CPR[0,rd]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T+1:</td>
<td>GPR[rt] ← data</td>
<td></td>
</tr>
</tbody>
</table>

| 64 | T: | data ← CPR[0,rd] |
|    | T+1: | GPR[rt] ← (data_{31})^{32} || data_{31}...0 |

**Exceptions:**

Coprocessor unusable exception
MFCz  Move From Coprocessor

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPz</td>
<td>MF</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

MFCz rt, rd

Description:

The contents of coprocessor register rd of coprocessor z are loaded into general register rt.

Operation:

32  T:  data ← CPR[z,rd]

64  T:  if rd0 = 0 then
       data ← CPR[z,rd4...1 || 0]31...0
       else
       data ← CPR[z,rd4...1 || 0]63...32
       endif

   T+1:  GPR[rt] ← (data31)32 || data

Exceptions:

Coprocessor unusable exception

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.
MFCz  Move From Coprocessor
(continued)  MFCz

Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>MFCz</th>
<th>Bit #31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFC0</td>
<td>0</td>
<td>1</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>MFC1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MFC2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<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>

- **Opcode**: 00 000 0 0 0 0 0 0 0 0 0 0 0 0
- **Coprocessor Suboperation**: 00 000 0 0 0 0 0 0 0 0 0 0 0 0
- **Coprocessor Unit Number**: 00 000 0 0 0 0 0 0 0 0 0 0 0 0
Appendix A

**MFHI**

**Move From HI**

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

**Format:**

MFHI rd

**Description:**

The contents of special register HI are loaded into general register rd.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the HI register: MULT, MULTU, DIV, DIVU, MTHI, DMULT, DMULTU, DDIV, DDIVU.

**Operation:**

| 32, 64 | T: | GPR[rd] ← HI |

**Exceptions:**

None
MFLO

Move From Lo

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

Format:

MFLO rd

Description:

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

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the LO register: MULT, MULTU, DIV, DIVU, MTLO, DMULT, DMULTU, DDIV, DDIVU.

Operation:

32, 64 T: GPR[rd] ← LO

Exceptions:

None
Format:

MTC0 rt, rd

Description:

The contents of general register rt are loaded into coprocessor register rd of CP0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

Operation:

<table>
<thead>
<tr>
<th>T</th>
<th>GPR[rt]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T+1</td>
<td>CPR[0,rd] ← data</td>
</tr>
</tbody>
</table>

Exceptions:

Coprocessor unusable exception
MTCz  
Move To Coprocessor  
MTCz

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTCz  rt, rd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The contents of general register rt are loaded into coprocessor register rd of coprocessor z.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
</tr>
</thead>
</table>
| 32  T: data ← GPR[rt]  
T+1: CPR[z, rd] ← data |
| 64  T: data ← GPR[rt]31...0  
T+1: if rd0 = 0  
CPR[z, rd4...1 || 0] ← CPR[z, rd4...1 || 0]63...32 || data  
else  
CPR[z, rd4...1 || 0] ← data || CPR[z, rd4...1 || 0]31...0  
endif |

<table>
<thead>
<tr>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coprocessor unusable exception</td>
</tr>
</tbody>
</table>

*Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>MTCz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit #31 30 29 28 27 26 25 24 23 22 21 0</td>
</tr>
<tr>
<td>COP0</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>COP1</td>
</tr>
<tr>
<td>0 1 0 0 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>COP2</td>
</tr>
<tr>
<td>0 1 0 0 1 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

 Opcode  Coprocessor Unit Number  Coprocessor Suboperation
**MTHI**  Move To HI  **MTHI**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</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>15</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MTHI  rs

**Description:**

The contents of general register rs are loaded into special register HI.

If a MTHI operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register LO are undefined.

**Operation:**

| 32,64 | T–2:  Hl ← undefined
|       | T–1:  HI ← undefined
|       | T:    HI ← GPR[rs]

**Exceptions:**

None
**MTLO**

**Move To LO**

<table>
<thead>
<tr>
<th>Format:</th>
<th>MTLO rs</th>
</tr>
</thead>
</table>

**Description:**

The contents of general register *rs* are loaded into special register *LO*.

If a MTLO operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *HI* are undefined.

**Operation:**

| 32,64 | T–2: LO ← undefined |
| T–1: LO ← undefined |
| T: LO ← GPR[rs] |

**Exceptions:**

None
**MULT** Multiply **MULT**

<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>MULT</td>
</tr>
</tbody>
</table>

**Format:**

MULT rs, rt

**Description:**

The contents of general registers $rs$ and $rt$ are multiplied, treating both operands as 32-bit 2’s complement values. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register $LO$, and the high-order word of the double result is loaded into special register $HI$.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of $HI$ or $LO$ from writes by a minimum of two other instructions.
MULT Multiply (continued) MULT

Operation:

| 32 | T–2: LO ← undefined  |
|    | HI ← undefined       |
|    | T–1: LO ← undefined  |
|    | HI ← undefined       |
|    | T: t ← GPR[rs] * GPR[rt] |
|    | LO ← t_{31...0}      |
|    | HI ← t_{63...32}     |
| 64 | T–2: LO ← undefined  |
|    | HI ← undefined       |
|    | T–1: LO ← undefined  |
|    | HI ← undefined       |
|    | T: t ← GPR[rs]_{31...0} * GPR[rt]_{31...0} |
|    | LO ← (t_{31})^{32} || t_{31...0} |
|    | HI ← (t_{63})^{32} || t_{63...32} |

Exceptions:
None
Appendix A

MULTU Multiply Unsigned MULTU

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

Format:

MULTU rs, rt

Description:

The contents of general register rs and the contents of general register rt are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word of the double result is loaded into special register HI.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of HI or LO from writes by a minimum of two instructions.
MULTU | Multiply Unsigned (continued) | MULTU

**Operation:**

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>T–2: LO</th>
<th>← undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T–1: LO</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T: t</td>
<td>← (0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LO</td>
<td>← t_{31...0}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← t_{63...32}</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>T–2: LO</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T–1: LO</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T: t</td>
<td>← (0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LO</td>
<td>← (t_{31})_{32}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HI</td>
<td>← (t_{63})_{32}</td>
</tr>
</tbody>
</table>

**Exceptions:**

None
**NOR**

<table>
<thead>
<tr>
<th>Format:</th>
<th>NOR rd, rs, rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The contents of general register $rs$ are combined with the contents of general register $rt$ in a bit-wise logical NOR operation. The result is placed into general register $rd$.</td>
</tr>
<tr>
<td>Exceptions:</td>
<td>None</td>
</tr>
</tbody>
</table>
**Format:**

OR rd, rs, rt

**Description:**

The contents of general register rs are combined with the contents of general register rt in a bit-wise logical OR operation. The result is placed into general register rd.

**Operation:**

| 32, 64 | T: GPR[rd] ← GPR[rs] or GPR[rt] |

**Exceptions:**

None
ORI  Or Immediate  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>ORI</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 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:

ORI rt, rs, immediate

Description:

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical OR operation. The result is placed into general register rt.

Operation:

32 T:  GPR[rt] ← GPR[rs]_{31...16} || (immediate or GPR[rs]_{15...0})
64 T:  GPR[rt] ← GPR[rs]_{63...16} || (immediate or GPR[rs]_{15...0})

Exceptions:

None
**SB Store Byte**

### Format:

SB rt, offset(base)

### Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The least-significant byte of register rt is stored at the effective address.

### Operation:

- **32 T:**
  \[
  vAddr \leftarrow ((\text{offset}_{15})^{16} \ || \ offset_{15...0}) + \text{GPR}[\text{base}]
  \]
  \[
  (pAddr, \text{uncached}) \leftarrow \text{AddressTranslation} \ (vAddr, \text{DATA})
  \]
  \[
  pAddr \leftarrow pAddr_{\text{PSIZE-1...3}} || (pAddr_{2...0} \ xor \ \text{ReverseEndian}^3)
  \]
  \[
  \text{byte} \leftarrow vAddr_{2...0} \ xor \ \text{BigEndianCPU}^3
  \]
  \[
  \text{data} \leftarrow \text{GPR}[rt]_{63-8*\text{byte}...0} || 0^{8*\text{byte}}
  \]
  \[
  \text{StoreMemory} \ (\text{uncached, BYTE, data, pAddr, vAddr, DATA})
  \]

- **64 T:**
  \[
  vAddr \leftarrow ((\text{offset}_{15})^{48} \ || \ offset_{15...0}) + \text{GPR}[\text{base}]
  \]
  \[
  (pAddr, \text{uncached}) \leftarrow \text{AddressTranslation} \ (vAddr, \text{DATA})
  \]
  \[
  pAddr \leftarrow pAddr_{\text{PSIZE-1...3}} || (pAddr_{2...0} \ xor \ \text{ReverseEndian}^3)
  \]
  \[
  \text{byte} \leftarrow vAddr_{2...0} \ xor \ \text{BigEndianCPU}^3
  \]
  \[
  \text{data} \leftarrow \text{GPR}[rt]_{63-8*\text{byte}...0} || 0^{8*\text{byte}}
  \]
  \[
  \text{StoreMemory} \ (\text{uncached, BYTE, data, pAddr, vAddr, DATA})
  \]

### Exceptions:

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
### Format:

\[
\text{SC \ rt, offset(base)}
\]

### Description:

The 16-bit offset is sign-extended and added to the contents of general register \( \text{base} \) to form a virtual address. The contents of general register \( \text{rt} \) are conditionally stored at the memory location specified by the effective address.

If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register \( \text{rt} \) after execution of the instruction. A successful store sets the contents of general register \( \text{rt} \) to 1; an unsuccessful store sets it to 0.

The operation of Store Conditional is undefined when the address is different from the address used in the last Load Linked.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the two least-significant bits of the effective address is non-zero, an address error exception takes place.

If this instruction should both fail and take an exception, the exception takes precedence.
SC Store Conditional (continued) SC

Operation:

32 T:  vAddr ← ((offset15)16 || offset15...0) + GPR[base]
        (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
        pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 02))
        data ← GPR[rt]63-8*byte...0 || 08*byte
        if LLbit then
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
        GPR[rt] ← 031 || LLbit

64 T:  vAddr ← ((offset15)48 || offset15...0) + GPR[base]
        (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
        pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 02))
        data ← GPR[rt]63-8*byte...0 || 08*byte
        if LLbit then
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
        GPR[rt] ← 063 || LLbit

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Format:

SCD rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are conditionally stored at the memory location specified by the effective address.

If any other processor or device has modified the physical address since the time of the previous Load Linked Doubleword instruction, or if an ERET instruction occurs between the Load Linked Doubleword instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register rt after execution of the instruction. A successful store sets the contents of general register rt to 1; an unsuccessful store sets it to 0.

The operation of Store Conditional Doubleword is undefined when the address is different from the address used in the last Load Linked Doubleword.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the three least-significant bits of the effective address is non-zero, an address error exception takes place.
If this instruction should both fail and take an exception, the exception takes precedence.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

| 64 | T: | vAddr ← \((\text{offset}_{15})^{\text{48}} || \text{offset}_{15...0}\) + GPR[base] |
|-----------------|-----------------|-----------------|
| (pAddr, uncached) ← AddressTranslation (vAddr, DATA) | data ← GPR[rt] | if LLbit then |
| StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA) | endif | GPR[rt] ← \(0^{63} || \text{LLbit}\) |

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)
Appendix A

SD Store Doubleword

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

Format:

SD rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are stored at the memory location specified by the effective address.

If either of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

\[
\begin{array}{c}
64 \quad T: \quad \text{vAddr} \leftarrow ((\text{offset}_{15})^{48} || \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
\quad (\text{pAddr, uncached}) \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
\quad \text{data} \leftarrow \text{GPR}[rt] \\
\quad \text{StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)}
\end{array}
\]

Exceptions:

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit user mode R4000 in 32-bit supervisor mode)
**Format:**

SDCz rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. Coprocessor unit z sources a doubleword, which the processor writes to the addressed memory location. The data to be stored is defined by individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the rt field is non-zero.

**Operation:**

| 32 | T: | vAddr ← ((offset15)^16 || offset15...0) + GPR[base] |
|----|----|--------------------------------------------------|
|    |    | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|    |    | data ← COPzSD(rt), |
|    |    | StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA) |

| 64 | T: | vAddr ← ((offset15)^48 || offset15...0) + GPR[base] |
|----|----|--------------------------------------------------|
|    |    | (pAddr, uncached) ← AddressTranslation (vAddr, DATA) |
|    |    | data ← COPzSD(rt), |
|    |    | StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA) |

*See the table, “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.
Appendix A

Exceptions:
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Coprocessor unusable exception

 Opcode Bit Encoding:

<table>
<thead>
<tr>
<th>SDCz</th>
<th>Bit #31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDC1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SDC2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

SDCz

Store Doubleword From Coprocessor (continued)

SDCz

SD opcode
Coprocessor Unit Number
Format:

SDL rt, offset(base)

Description:

This instruction can be used with the SDR instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a doubleword boundary. SDL stores the left portion of the register into the appropriate part of the high-order doubleword of memory; SDR stores the right portion of the register into the appropriate part of the low-order doubleword.

The SDL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

\[
\begin{align*}
64 & \quad T:\quad \text{vAddr} \leftarrow ((\text{offset}_{15})^{48} || \text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
\text{pAddr, uncached} & \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
pAddr & \leftarrow \text{pAddr}_{\text{PSIZE }-1...3} || (\text{pAddr}_{2...0} \text{ xor ReverseEndian}^3) \\
\text{If BigEndianMem } = 0 \text{ then} & \\
pAddr & \leftarrow \text{pAddr}_{31...3} || 0^3 \\
\text{endif} & \\
\text{byte} & \leftarrow \text{vAddr}_{2...0} \text{ xor BigEndianCPU}^3 \\
\text{data} & \leftarrow 0^{56-8*\text{byte}} || \text{GPR}[rt]_{63...56-8*\text{byte}} \\
\text{Storememory (uncached, byte, data, pAddr, vAddr, DATA)} &
\end{align*}
\]
Given a doubleword in a register and a doubleword in memory, the operation of SDL is as follows:

**Exceptions:**
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)

---

<table>
<thead>
<tr>
<th>vAddr_{2,0}</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I J K L M N O A</td>
<td>0</td>
<td>0</td>
<td>A B C D E F G H</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>I J K L M N A B</td>
<td>1</td>
<td>0</td>
<td>A B C D E F G</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>I J K L M A B C</td>
<td>2</td>
<td>0</td>
<td>I J A B C D E F</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>I J K L A B C D</td>
<td>3</td>
<td>0</td>
<td>I J K A B C D E</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>I J K A B C D E</td>
<td>4</td>
<td>0</td>
<td>I J K L A B C D</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>I J A B C D E F</td>
<td>5</td>
<td>0</td>
<td>I J K L M A B C</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>I A B C D E F G</td>
<td>6</td>
<td>0</td>
<td>I J K L M N A B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>A B C D E F G H</td>
<td>7</td>
<td>0</td>
<td>I J K L M N O A</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

*LEM*  
Little-endian memory (BigEndianMem = 0)

*BEM*  
BigEndianMem = 1

*Type*  
AccessType (see Table 2-1) sent to memory

*Offset*  
pAddr_{2,0} sent to memory
### SDR Store Doubleword Right

**Format:**

SDR rt, offset(base)

**Description:**

This instruction can be used with the SDL instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDR stores the right portion of the register into the appropriate part of the low-order doubleword; SDL stores the left portion of the register into the appropriate part of the low-order doubleword of memory.

The SDR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the high-order byte of the word in memory. No address exceptions due to alignment are possible.

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

**Example:**

Before SDR:

<table>
<thead>
<tr>
<th>memory (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E F G H $24</td>
</tr>
</tbody>
</table>

SDR $24,4($0):

<table>
<thead>
<tr>
<th>memory (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D E F G H $24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address 8</th>
<th>Address 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 9 10 11 12 13 14 15</td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
</tr>
</tbody>
</table>

**Register Before:** A B C D E F G H $24

**Register After:** A B C D E F G H $24

**Memory Before:** 8 9 10 11 12 13 14 15

**Memory After:** 8 9 10 11 12 13 14 15

**Address:** 8 0

**Address:** 8 0
## SDR

### Store Doubleword Right

(continued)

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

| 64 | T: vAddr ← ((offset[^15]^48 || offset[^15]...0) + GPR[base]) |
|----|---------------------------------------------------------------|
|    | (pAddr, uncached) ← AddressTranslation (vAddr, DATA)         |
|    | pAddr ← pAddr\_PSIZE – 1...3 || (pAddr[^2]...0 xor ReverseEndian[^3]) |
|    | If BigEndianMem = 0 then                                      |
|    | pAddr ← pAddr\_PSIZE – 31...3 || 0[^3]                       |
|    | endif                                                         |
|    | byte ← vAddr[^1]...0 xor BigEndianCPU[^3]                     |
|    | data ← GPR[rt][^63–8]*byte || 0[^8]*byte                    |
|    | StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr, DATA) |

---

MIPS R4000 Microprocessor User’s Manual

A-137
Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:

<table>
<thead>
<tr>
<th>vAddr_{2..0}</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
<th>destination</th>
<th>type</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A B C D E F G H</td>
<td>7</td>
<td>0 0</td>
<td>H J K L M N O P</td>
<td>0</td>
<td>7 0</td>
</tr>
<tr>
<td>1</td>
<td>B C D E F G H P</td>
<td>6</td>
<td>1 0</td>
<td>G H K L M N O P</td>
<td>1</td>
<td>6 0</td>
</tr>
<tr>
<td>2</td>
<td>C D E F G H O P</td>
<td>5</td>
<td>2 0</td>
<td>F G H L M N O P</td>
<td>2</td>
<td>5 0</td>
</tr>
<tr>
<td>3</td>
<td>D E F G H N O P</td>
<td>4</td>
<td>3 0</td>
<td>E F G H M N O P</td>
<td>3</td>
<td>4 0</td>
</tr>
<tr>
<td>4</td>
<td>E F G H M N O P</td>
<td>3</td>
<td>4 0</td>
<td>D E F G H N O P</td>
<td>4</td>
<td>3 0</td>
</tr>
<tr>
<td>5</td>
<td>F G H L M N O P</td>
<td>2</td>
<td>5 0</td>
<td>C D E F G H O P</td>
<td>5</td>
<td>2 0</td>
</tr>
<tr>
<td>6</td>
<td>G H K L M N O P</td>
<td>1</td>
<td>6 0</td>
<td>B C D E F G H P</td>
<td>6</td>
<td>1 0</td>
</tr>
<tr>
<td>7</td>
<td>H J K L M N O P</td>
<td>0</td>
<td>7 0</td>
<td>A B C D E F G H</td>
<td>7</td>
<td>0 0</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessType (see Table 2-1) sent to memory

**Offset** pAddr_{2..0} sent to memory

**Exceptions:**
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Reserved instruction exception (R4000 in 32-bit mode)
### SH Store Halfword

#### Format:

\[
\text{SH rt, offset(base)}
\]

#### Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form an unsigned effective address. The least-significant halfword of register rt is stored at the effective address. If the least-significant bit of the effective address is non-zero, an address error exception occurs.

#### Operation:

<table>
<thead>
<tr>
<th>32 T</th>
<th>( v\text{Addr} \leftarrow ((\text{offset}<em>{15})^{16} \parallel \text{offset}</em>{15...0}) + \text{GPR[base]} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( (p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}) )</td>
</tr>
<tr>
<td></td>
<td>( p\text{Addr} \leftarrow p\text{Addr}<em>{\text{PSIZE}-1...3} \parallel (p\text{Addr}</em>{2...0} \text{xor (ReverseEndian^2 \parallel 0)}) )</td>
</tr>
<tr>
<td></td>
<td>( \text{byte} \leftarrow v\text{Addr}_{2...0} \text{xor (BigEndianCPU^2 \parallel 0)} )</td>
</tr>
<tr>
<td></td>
<td>( \text{data} \leftarrow \text{GPR[rt]}_{63-8*\text{byte}...0} \parallel 0^{8*\text{byte}} )</td>
</tr>
<tr>
<td></td>
<td>( \text{StoreMemory(uncached, HALFWORD, data, pAddr, vAddr, DATA)} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64 T</th>
<th>( v\text{Addr} \leftarrow ((\text{offset}<em>{15})^{48} \parallel \text{offset}</em>{15...0}) + \text{GPR[base]} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( (p\text{Addr}, \text{uncached}) \leftarrow \text{AddressTranslation}(v\text{Addr}, \text{DATA}) )</td>
</tr>
<tr>
<td></td>
<td>( p\text{Addr} \leftarrow p\text{Addr}<em>{\text{PSIZE}-1...3} \parallel (p\text{Addr}</em>{2...0} \text{xor (ReverseEndian^2 \parallel 0)}) )</td>
</tr>
<tr>
<td></td>
<td>( \text{byte} \leftarrow v\text{Addr}_{2...0} \text{xor (BigEndianCPU^2 \parallel 0)} )</td>
</tr>
<tr>
<td></td>
<td>( \text{data} \leftarrow \text{GPR[rt]}_{63-8*\text{byte}...0} \parallel 0^{8*\text{byte}} )</td>
</tr>
<tr>
<td></td>
<td>( \text{StoreMemory(uncached, HALFWORD, data, pAddr, vAddr, DATA)} )</td>
</tr>
</tbody>
</table>

#### Exceptions:

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
SLL

**Shift Left Logical**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SLL</td>
<td>0</td>
<td>0</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:**

SLL rd, rt, sa

**Description:**

The contents of general register rt are shifted left by sa bits, inserting zeros into the low-order bits.

The result is placed in register rd.

In 64-bit mode, the 32-bit result is sign extended when placed in the destination register. It is sign extended for all shift amounts, including zero; SLL with a zero shift amount truncates a 64-bit value to 32 bits and then sign extends this 32-bit value. SLL, unlike nearly all other word operations, does not require an operand to be a properly sign-extended word value to produce a valid sign-extended word result.

**NOTE:** SLL with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLL with a zero shift to truncate 64-bit values, check the assembler you are using.

**Operation:**

| 32 T: | GPR[rd] ← GPR[rt]_{31– sa...0} || 0^{sa} |
| 64 T: | s ← 0 || sa |
| | temp ← GPR[rt]_{31– s...0} || 0^{s} |
| | GPR[rd] ← (temp_{31})^{32} || temp |

**Exceptions:**

None
Format:

SLLV rd, rt, rs

Description:

The contents of general register rt are shifted left the number of bits specified by the low-order five bits contained in general register rs, inserting zeros into the low-order bits.

The result is placed in register rd.

In 64-bit mode, the 32-bit result is sign extended when placed in the destination register. It is sign extended for all shift amounts, including zero; SLLV with a zero shift amount truncates a 64-bit value to 32 bits and then sign extends this 32-bit value. SLLV, unlike nearly all other word operations, does not require an operand to be a properly sign-extended word value to produce a valid sign-extended word result.

**NOTE:** SLLV with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLLV with a zero shift to truncate 64-bit values, check the assembler you are using.

Operation:

\[
\begin{align*}
\text{T:} & & s & \leftarrow \text{GP}[rs]_{4...0} \\
& & \text{GPR}[rd] & \leftarrow \text{GPR}[rt]_{(31-s)...0} \ || \ 0^s \\
\text{64} & & s & \leftarrow 0 \ || \ \text{GP}[rs]_{4...0} \\
& & \text{temp} & \leftarrow \text{GPR}[rt]_{(31-s)...0} \ || \ 0^s \\
& & \text{GPR}[rd] & \leftarrow (\text{temp}_{31})^{32} \ || \ \text{temp}
\end{align*}
\]

Exceptions:

None
### SLT

**Set On Less Than**

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Format:**

SLT rd, rs, rt

**Description:**

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one; otherwise the result is set to zero.

The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

**Operation:**

32 T: if GPR[rs] < GPR[rt] then
     GPR[rd] ← 0<sup>31</sup> || 1
else
     GPR[rd] ← 0<sup>32</sup>
endif

64 T: if GPR[rs] < GPR[rt] then
     GPR[rd] ← 0<sup>63</sup> || 1
else
     GPR[rd] ← 0<sup>64</sup>
endif

**Exceptions:**

None
**SLTI**  
Set On Less Than 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>SLTI</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 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:**

SLTI rt, rs, immediate

**Description:**

The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as signed integers, if rs is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero.

The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

**Operation:**

\[
32 \ T: \ \text{if } \text{GPR[rs]} < (\text{immediate}_{15})^{16} \ || \text{immediate}_{15...0} \ \text{then}
\]

\[
\text{GPR[rd]} \leftarrow 0^{31} || 1
\]

\[
\text{else}
\]

\[
\text{GPR[rd]} \leftarrow 0^{32}
\]

\[
\text{endif}
\]

\[
64 \ T: \ \text{if } \text{GPR[rs]} < (\text{immediate}_{15})^{48} \ || \text{immediate}_{15...0} \ \text{then}
\]

\[
\text{GPR[rd]} \leftarrow 0^{63} || 1
\]

\[
\text{else}
\]

\[
\text{GPR[rd]} \leftarrow 0^{64}
\]

\[
\text{endif}
\]

**Exceptions:**

None
Format:
SLTIU rt, rs, immediate

Description:
The 16-bit immediate is sign-extended and subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if rs is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero.

The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32 T: if (0 || GPR[rs]) < (immediate15)16 || immediate15...0 then
    GPR[rd] ← 031 || 1
else
    GPR[rd] ← 032
endif
64 T: if (0 || GPR[rs]) < (immediate15)48 || immediate15...0 then
    GPR[rd] ← 063 || 1
else
    GPR[rd] ← 064
endif
```

Exceptions:
None
### SLTU

#### Set On Less Than Unsigned

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>000000</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>00000</td>
<td>SLTU</td>
<td>101011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

SLTU rd, rs, rt

#### Description:

The contents of general register rt are subtracted from the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are less than the contents of general register rt, the result is set to one; otherwise the result is set to zero.

The result is placed into general register rd.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

#### Operation:

```plaintext
32 T: if (0 || GPR[rs]) < 0 || GPR[rt] then
    GPR[rd] ← 0^31 || 1
else
    GPR[rd] ← 0^32
endif

64 T: if (0 || GPR[rs]) < 0 || GPR[rt] then
    GPR[rd] ← 0^63 || 1
else
    GPR[rd] ← 0^64
endif
```

#### Exceptions:

None
Format:

SRA rd, rt, sa

Description:

The contents of general register rt are shifted right by sa bits, sign-extending the high-order bits.

The result is placed in register rd.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

32 T: \( \text{GPR}[rd] \leftarrow (\text{GPR}[rt]_{31})^{sa} \ || \ \text{GPR}[rt]_{31...sa} \)

64 T: \( s \leftarrow 0 \ || \ sa \)

\( \text{temp} \leftarrow (\text{GPR}[rt]_{31})^{s} \ || \ \text{GPR}[rt]_{31...s} \)

\( \text{GPR}[rd] \leftarrow (\text{temp}_{31})^{32} \ || \ \text{temp} \)

Exceptions:

None
SRAV  
Shift 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 0 0 0 0 0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SRAV 0 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
SRAV  rd, rt, rs

Description:
The contents of general register rt are shifted right by the number of bits specified by the low-order five bits of general register rs, sign-extending the high-order bits.
The result is placed in register rd.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

\[
\begin{align*}
32 & : s \leftarrow GPR[rs]_{4...0} \\
& \quad GPR[rd] \leftarrow (GPR[rt]_{31})^s \ || \ GPR[rt]_{31...s} \\
64 & : s \leftarrow GPR[rs]_{4...0} \\
& \quad temp \leftarrow (GPR[rt]_{31})^s \ || \ GPR[rt]_{31...s} \\
& \quad GPR[rd] \leftarrow (temp_{31})^{32} \ || \ temp
\end{align*}
\]

Exceptions:
None
SRL

Shift Right Logical

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL 0 0 0 0 0 0</td>
<td>rt</td>
<td>rd</td>
<td>sa</td>
<td>SRL 0 0 0 0 1 0</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>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SRL rd, rt, sa

**Description:**

The contents of general register rt are shifted right by sa bits, inserting zeros into the high-order bits.

The result is placed in register rd.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

**Operation:**

| 32 | T: | GPR[rd] ← 0^sa || GPR[rt]_{31...sa} |
|----|----|-------------------------------------|
| 64 | T: | s ← 0 || sa |
|    |    | temp ← 0^s || GPR[rt]_{31...s} |
|    |    | GPR[rd] ← (temp_{31})^{32} || temp |

**Exceptions:**

None
**SRLV**

**Shift Right Logical Variable**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPECIAL</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>SRLV</td>
<td></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:**

SRLV rd, rt, rs

**Description:**

The contents of general register rt are shifted right by the number of bits specified by the low-order five bits of general register rs, inserting zeros into the high-order bits.

The result is placed in register rd.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

**Operation:**

```
32 T:  s ← GPR[rs]_{4...0}
      GPR[rd] ← 0^s || GPR[rt]_{31...s}

64 T:  s ← GPR[rs]_{4...0}
      temp ← 0^s || GPR[rt]_{31...s}
      GPR[rd] ← (temp_{31})^{32} || temp
```

**Exceptions:**

None
SUB Subtract SUB

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| SPECIAL | 0 0 0 0 0 0 | rs | rt | rd | 0 | 0 0 0 0 0 | SUB | 1 0 0 0 1 0 |

6 | 5 | 5 | 5 | 5 | 5 | 6 |

Format:

SUB rd, rs, rt

Description:

The contents of general register rt are subtracted from the contents of general register rs to form a result. The result is placed into general register rd. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUBU instruction is that SUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ (2’s complement overflow). The destination register rd is not modified when an integer overflow exception occurs.

Operation:

| 64 | T: temp ← GPR[rs] - GPR[rt] |
|    | GPR[rd] ← (temp_{31})^{32} || temp_{31...0} |

Exceptions:

Integer overflow exception
SUBU Subtract 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>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>SUBU 1 0 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SUBU rd, rs, rt

**Description:**

The contents of general register rt are subtracted from the contents of general register rs to form a result.

The result is placed into general register rd.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUB instruction is that SUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

**Operation:**

| 64 T | temp ← GPR[rs] - GPR[rt] |
|      | GPR[rd] ← (temp_{31})^{32} || temp_{31}...0 |

**Exceptions:**

None
### 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></td>
<td>SW</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101011</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SW rt, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of general register rt are stored at the memory location specified by the effective address.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

**Operation:**

| 32 T: | vAddr ← ((offset<sub>15</sub>)<sup>16</sup> || offset<sub>15...0</sub>) + GPR[base] |
|-------|------------------------------------------------------------------|
|       | (pAddr, uncached) ← AddressTranslation (vAddr, DATA)             |
|       | pAddr ← pAddr<sub>PSIZE-1...3</sub> || (pAddr<sub>2...0</sub> xor (ReverseEndian || 0<sup>2</sup>) | |
|       | byte ← vAddr<sub>2...0</sub> xor (BigEndianCPU || 0<sup>2</sup>) |
|       | data ← GPR[rt]<sub>63-8*byte</sub> || 0<sup>8*byte</sup> |
|       | StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)           |

| 64 T: | vAddr ← ((offset<sub>15</sub>)<sup>48</sup> || offset<sub>15...0</sub>) + GPR[base] |
|-------|---------------------------------------------------------------------------------|
|       | (pAddr, uncached) ← AddressTranslation (vAddr, DATA)                             |
|       | pAddr ← pAddr<sub>PSIZE-1...3</sub> || (pAddr<sub>2...0</sub> xor (ReverseEndian || 0<sup>2</sup>) | |
|       | byte ← vAddr<sub>2...0</sub> xor (BigEndianCPU || 0<sup>2</sup>)                |
|       | data ← GPR[rt]<sub>63-8*byte</sub> || 0<sup>8*byte</sup>                  |
|       | StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)                           |

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
SWCz  Store Word From Coprocessor  SWCz

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

Format:

SWCz rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. Coprocessor unit z sources a word, which the processor writes to the addressed memory location.

The data to be stored is defined by individual coprocessor specifications.

This instruction is not valid for use with CP0.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

32 T:  vAddr ← ((offset15)16 || offset15...0) + GPR[base]
       (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
       pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 02))
       byte ← vAddr2...0 xor (BigEndianCPU || 02)
       data ← COPzSW (byte, rt)
       StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64 T:  vAddr ← ((offset15)48 || offset15...0) + GPR[base]
       (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
       pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 02))
       byte ← vAddr2...0 xor (BigEndianCPU || 02)
       data ← COPzSW (byte, rt)
       StoreMemory (uncached, WORD, data, pAddr, vAddr DATA)

*See the table “Opcode Bit Encoding” on next page, or “CPU Instruction Opcode Bit Encoding” at the end of Appendix A.
Exceptions:

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
- Coprocessor unusable exception

Opcodes Bit Encoding:

```
<table>
<thead>
<tr>
<th>SWCz</th>
<th>Bit #31 30 29 28 27 26 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWC1</td>
<td>1 1 1 0 0 1 1</td>
</tr>
<tr>
<td>SWC2</td>
<td>1 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>
```

- SW opcode
- Coprocessor Unit Number

---

**Format:**

SWL rt, offset(base)

**Description:**

This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a word boundary. SWL stores the left portion of the register into the appropriate part of the high-order word of memory; SWR stores the right portion of the register into the appropriate part of the low-order word.

The SWL instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.

---

### SWL Store Word Left

<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>SWL</td>
<td>base</td>
<td>rt</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101010</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**address 4**

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**address 0**

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

(before) $24$

after

SWL $24,1($0)
Operation:

32 T: \( \text{vAddr} \leftarrow ((\text{offset}_{15})^{16} \| \text{offset}_{15...0}) + \text{GPR}[/base]\)
(pAddr, uncached) \(\leftarrow\) AddressTranslation (vAddr, DATA)
\(\text{pAddr} \leftarrow \text{pAddr}_{\text{Psize}} - 1...3 \| (\text{pAddr}_{2...0} \text{ xor ReverseEndian})^3\)
If BigEndianMem = 0 then
\(\text{pAddr} \leftarrow \text{pAddr}_{31...2} \| 0^2\)
endif
byte \(\leftarrow \text{vAddr}_{1...0} \text{ xor BigEndianCPU}^2\)
if (vAddr\(_2\) xor BigEndianCPU) = 0 then
\(\text{data} \leftarrow 0^{32} \| 0^{24-8*\text{byte}} \| \text{GPR}[\text{rt}]_{31...24-8*\text{byte}}\)
else
\(\text{data} \leftarrow 0^{24-8*\text{byte}} \| \text{GPR}[\text{rt}]_{31...24-8*\text{byte}} \| 0^{32}\)
endif
StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)

64 T: \( \text{vAddr} \leftarrow ((\text{offset}_{15})^{48} \| \text{offset}_{15...0}) + \text{GPR}[/base]\)
(pAddr, uncached) \(\leftarrow\) AddressTranslation (vAddr, DATA)
\(\text{pAddr} \leftarrow \text{pAddr}_{\text{Psize}} - 1...3 \| (\text{pAddr}_{2...0} \text{ xor ReverseEndian})^3\)
If BigEndianMem = 0 then
\(\text{pAddr} \leftarrow \text{pAddr}_{31...2} \| 0^2\)
endif
byte \(\leftarrow \text{vAddr}_{1...0} \text{ xor BigEndianCPU}^2\)
if (vAddr\(_2\) xor BigEndianCPU) = 0 then
\(\text{data} \leftarrow 0^{32} \| 0^{24-8*\text{byte}} \| \text{GPR}[\text{rt}]_{31...24-8*\text{byte}}\)
else
\(\text{data} \leftarrow 0^{24-8*\text{byte}} \| \text{GPR}[\text{rt}]_{31...24-8*\text{byte}} \| 0^{32}\)
endif
StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

### SWL

<table>
<thead>
<tr>
<th>Register</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D</td>
<td>E F G H</td>
</tr>
<tr>
<td>I J K L</td>
<td>M N O P</td>
</tr>
</tbody>
</table>

#### SWL Store Word Left (Continued)

<table>
<thead>
<tr>
<th>vAddr_{2,0}</th>
<th>BigEndianCPU = 0</th>
<th></th>
<th>BigEndianCPU = 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>destination</td>
<td>type</td>
<td>offset</td>
<td>destination</td>
<td>type</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>0</td>
<td>I J K L M N O E</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>I J K L M N E F</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>I J K L M E F G</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>I J K L E F G H</td>
<td>3</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>I J K E M N O P</td>
<td>0</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>I J E F M N O P</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>I E F G M N O P</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>E F G H M N O P</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

- **LEM**: Little-endian memory (BigEndianMem = 0)
- **BEM**: BigEndianMem = 1
- **Type**: AccessType (see Table 2-1) sent to memory
- **Offset**: pAddr_{2,0} sent to memory

**Exceptions:**
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
Appendix A

**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>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>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

SWR rt, offset(base)

**Description:**

This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low-order word; SWL stores the left portion of the register into the appropriate part of the low-order word of memory.

The SWR instruction adds its sign-extended 16-bit offset to the contents of general register base to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then copies bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.

---

**Before:**

<table>
<thead>
<tr>
<th>address 4</th>
<th>address 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory (big-endian)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Register:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
</table>

$24$

**SWR $24,1(0)$**

---

**After:**

<table>
<thead>
<tr>
<th>address 4</th>
<th>address 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**SWR**  
**Store Word Right**  
(Continued)

**Operation:**

| 32 T: | vAddr ← ((offset_{15})^{16} || offset_{15...0}) + GPR[base]  
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
pAddr ← pAddr_{PSIZE - 1...3} || (pAddr_{2...0} xor ReverseEndian^3)  
If BigEndianMem = 0 then  
pAddr ← pAddr_{31...2} || 0²  
endif  
byte ← vAddr_{1...0} xor BigEndianCPU^2  
if (vAddr_{2} xor BigEndianCPU) = 0 then  
data ← 0³² || GPR[rt]_{31-8*byte...0} || 0³²  
else  
data ← GPR[rt]_{31-8*byte...0} || 0³²  
endif  
StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA) |
|---|---|
| 64 T: | vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base]  
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
pAddr ← pAddr_{PSIZE - 1...3} || (pAddr_{2...0} xor ReverseEndian^3)  
If BigEndianMem = 0 then  
pAddr ← pAddr_{31...2} || 0²  
endif  
byte ← vAddr_{1...0} xor BigEndianCPU^2  
if (vAddr_{2} xor BigEndianCPU) = 0 then  
data ← 0³² || GPR[rt]_{31-8*byte...0} || 0³²  
else  
data ← GPR[rt]_{31-8*byte...0} || 0³²  
endif  
StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA) |
Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:

**SWR**

**Store Word Right**

(Continued)

<table>
<thead>
<tr>
<th>vAddr_{2..0}</th>
<th>BigEndianCPU = 0</th>
<th>BigEndianCPU = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I J K L E F G H</td>
<td>0 4</td>
</tr>
<tr>
<td>1</td>
<td>I J K L F G H P</td>
<td>2 1 4</td>
</tr>
<tr>
<td>2</td>
<td>I J K L G H O P</td>
<td>1 2 4</td>
</tr>
<tr>
<td>3</td>
<td>I J K L H N O P</td>
<td>0 3 4</td>
</tr>
<tr>
<td>4</td>
<td>E F G H M N O P</td>
<td>3 4 0</td>
</tr>
<tr>
<td>5</td>
<td>F G H L M N O P</td>
<td>2 5 0</td>
</tr>
<tr>
<td>6</td>
<td>G H K L M N O P</td>
<td>1 6 0</td>
</tr>
<tr>
<td>7</td>
<td>H J K L M N O P</td>
<td>0 7 0</td>
</tr>
</tbody>
</table>

**LEM** Little-endian memory (BigEndianMem = 0)

**BEM** BigEndianMem = 1

**Type** AccessType (see Table 2-1) sent to memory

**Offset** pAddr_{2..0} sent to memory

**Exceptions:**

- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
SYNC Synchronize

### Format:

`SYNC`

### Description:

The SYNC instruction ensures that any loads and stores fetched *prior to* the present instruction are completed before any loads or stores *after* this instruction are allowed to start. Use of the SYNC instruction to serialize certain memory references may be required in a multiprocessor environment for proper synchronization. For example:

<table>
<thead>
<tr>
<th>Processor A</th>
<th>Processor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW R1, DATA</td>
<td>1: LW R2, FLAG</td>
</tr>
<tr>
<td>LI R2, 1</td>
<td>BEQ R2, R0, 1B</td>
</tr>
<tr>
<td>SYNC</td>
<td>NOP</td>
</tr>
<tr>
<td>SW R2, FLAG</td>
<td>SYNC</td>
</tr>
</tbody>
</table>

The SYNC in processor A prevents DATA being written after FLAG, which could cause processor B to read stale data. The SYNC in processor B prevents DATA from being read before FLAG, which could likewise result in reading stale data. For processors which only execute loads and stores in order, with respect to shared memory, this instruction is a NOP.

LL and SC instructions implicitly perform a SYNC.

This instruction is allowed in User mode.

### Operation:

32, 64 T: SyncOperation()
Appendix A

Format:

SYSCALL

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.

Operation:

| 32, 64 | T: SystemCallException |

Exceptions:

System Call exception
CPU Instruction Set Details

**TEQ**

**Trap If Equal**

<table>
<thead>
<tr>
<th>Format:</th>
<th>TEQ rs, rt</th>
</tr>
</thead>
</table>

**Description:**

The contents of general register *rt* are compared to general register *rs*. If the contents of general register *rs* are equal to the contents of general register *rt*, a trap exception occurs.

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.

**Operation:**

| 32, 64 | T: if GPR[rs] = GPR[rt] then TrapException endif |

**Exceptions:**

Trap exception
Format:
TEQI rs, immediate

Description:
The 16-bit immediate is sign-extended and compared to the contents of general register rs. If the contents of general register rs are equal to the sign-extended immediate, a trap exception occurs.

Operation:

32 T: if GPR[rs] = (immediate[15])^{16} || immediate_{15...0} then
    TrapException
endif

64 T: if GPR[rs] = (immediate[15])^{48} || immediate_{15...0} then
    TrapException
endif

Exceptions:
Trap exception
## TGE Trap If Greater Than Or Equal

### Format:

TGE rs, rt

### Description:

The contents of general register rt are compared to the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are greater than or equal to the contents of general register rt, a trap exception occurs.

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.

### Operation:

```plaintext
32, 64  T: if GPR[rs] ≥ GPR[rt] then
        TrapException
     endif
```

### Exceptions:

Trap exception
### TGEI Trap If Greater Than Or Equal Immediate

<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></td>
<td>REGIMM</td>
<td>rs</td>
<td>TGEI</td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 1</td>
<td>0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 6 | 5 | 5 | 16 |

#### Format:

TGEI rs, immediate

#### Description:

The 16-bit immediate is sign-extended and compared to the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are greater than or equal to the sign-extended immediate, a trap exception occurs.

#### Operation:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
<tr>
<td>T: if GPR[rs] ≥ (immediate\textsubscript{15})^{16} \parallel \text{immediate}_{15...0} then</td>
</tr>
<tr>
<td>TrapException</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>T: if GPR[rs] ≥ (immediate\textsubscript{15})^{48} \parallel \text{immediate}_{15...0} then</td>
</tr>
<tr>
<td>TrapException</td>
</tr>
<tr>
<td>endif</td>
</tr>
</tbody>
</table>

#### Exceptions:

Trap exception
TGEIU
Trap If Greater Than Or Equal Immediate Unsigned

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

Format:
TGEIU rs, immediate

Description:
The 16-bit immediate is sign-extended and compared to the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are greater than or equal to the sign-extended immediate, a trap exception occurs.

Operation:

```
32  T: if (0 || GPR[rs]) ≥ (0 || (immediate15)16 || immediate15...0) then
    TrapException
endif

64  T: if (0 || GPR[rs]) ≥ (0 || (immediate15)48 || immediate15...0) then
    TrapException
endif
```

Exceptions:
Trap exception
TGEU Trap If Greater Than Or Equal 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>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>TGEU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>10</td>
<td>1100001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

TGEU rs, rt

Description:

The contents of general register rt are compared to the contents of general register rs. Considering both quantities as unsigned integers, if the contents of general register rs are greater than or equal to the contents of general register rt, a trap exception occurs.

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.

Operation:

T: if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
   TrapException
endif

Exceptions:

Trap exception
TLBP  
Probe TLB For Matching Entry

<table>
<thead>
<tr>
<th>TLBP</th>
<th>Probe TLB For Matching Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>6 1 19 6</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

**TLBP**

**Description:**

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

**Operation:**

32 T: 

Index ← 1 || 0^25 || undefined^6

for i in 0...TLBEntries–1

if (TLB[i] 95...77 = EntryHi 31...12) and (TLB[i] 76 or (TLB[i] 71...64 = EntryHi 7...0)) then

Index ← 0^26 || i_5...0

endif

endfor

64 T: 

Index ← 1 || 0^25 || undefined^6

for i in 0...TLBEntries–1

if (TLB[i] 167...141 and not (0^15 || TLB[i] 216...205)) = EntryHi 39...13 and not (0^15 || TLB[i] 216...205)) and (TLB[i] 140 or (TLB[i] 135...128 = EntryHi 7...0)) then

Index ← 0^26 || i_5...0

endif

endfor

**Exceptions:**

Coprocessor unusable exception
Format:
TLBR

Description:
The \( G \) bit (which controls ASID matching) read from the TLB is written into both of the \( \text{EntryLo0} \) and \( \text{EntryLo1} \) registers.

The \( \text{EntryHi} \) and \( \text{EntryLo} \) registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB \( \text{Index} \) register. The operation is invalid (and the results are unspecified) if the contents of the TLB \( \text{Index} \) register are greater than the number of TLB entries in the processor.

Operation:

\[
\begin{array}{cccc}
32 & 6 & 5 & 0 \\
6 & 1 & 19 & 6 \\
\end{array}
\]

Exceptions:
Coprocessor unusable exception
Format:

TLBWI

Description:

The G bit of the TLB is written with the logical AND of the G bits in the EntryLo0 and EntryLo1 registers.

The TLB entry pointed at by the contents of the TLB Index register is loaded with the contents of the EntryHi and EntryLo registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB Index register are greater than the number of TLB entries in the processor.

Operation:

\[
\begin{align*}
32, 64 & : \quad \text{TLB[Index}_{5...0}] \leftarrow \\
& \quad \text{PageMask} \lor (\text{EntryHi and not PageMask}) \lor \text{EntryLo1} \lor \text{EntryLo0}
\end{align*}
\]

Exceptions:

Coprocessor unusable exception
## TLBWR

### Write Random TLB Entry

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP0</td>
<td>CO</td>
<td>00000000000000000000000000000000</td>
<td>TLBWR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

TLBWR

#### Description:

The G bit of the TLB is written with the logical AND of the G bits in the EntryLo0 and EntryLo1 registers.

The TLB entry pointed at by the contents of the TLB Random register is loaded with the contents of the EntryHi and EntryLo registers.

#### Operation:

\[
32, 64 T: \quad \text{TLB}[\text{Random}_{5...0}] \leftarrow \text{PageMask} \oplus (\text{EntryHi} \text{ and not PageMask}) \oplus \text{EntryLo1} \oplus \text{EntryLo0}
\]

#### Exceptions:

Coprocessor unusable exception
### TLT 

**Trap If 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>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPECIAL</strong></td>
<td>rs</td>
<td>rt</td>
<td>code</td>
<td><strong>TLT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Format:

TLT rs, rt

#### Description:

The contents of general register rt are compared to general register rs. Considering both quantities as signed integers, if the contents of general register rs are less than the contents of general register rt, a trap exception occurs.

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.

#### Operation:

```plaintext
32, 64  T:  if GPR[rs] < GPR[rt] then
    TrapException
  endif
```

#### Exceptions:

Trap exception
Appendix A

**TLTI**

**Trap If Less Than Immediate**

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLTI rs, immediate</td>
</tr>
</tbody>
</table>

**Description:**

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

**Operation:**

```
32 T: if GPR[rs] < (immediate_{15})^{16} || immediate_{15...0} then
   TrapException
   endif

64 T: if GPR[rs] < (immediate_{15})^{48} || immediate_{15...0} then
   TrapException
   endif
```

**Exceptions:**

Trap exception
**TLTIU** Trap If Less Than Immediate Unsigned  TLTIU

<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>REGIMM</td>
<td>000001</td>
<td>rs</td>
<td>TLTIU</td>
<td>01011</td>
<td>immediate</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>
<td></td>
</tr>
</tbody>
</table>

**Format:**

TLTIU rs, immediate

**Description:**

The 16-bit *immediate* is sign-extended and compared to the contents of general register rs. Considering both quantities as signed integers, if the contents of general register rs are less than the sign-extended *immediate*, a trap exception occurs.

**Operation:**

```plaintext
32 T: if (0 || GPR[rs]) < (0 || (immediate_{15})^{16} || immediate_{15...0}) then
    TrapException
endif

64 T: if (0 || GPR[rs]) < (0 || (immediate_{15})^{48} || immediate_{15...0}) then
    TrapException
endif
```

**Exceptions:**

Trap exception
**TLTU** Trap If Less Than Unsigned

<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</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLTU</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

TLTU rs, rt

**Description:**

The contents of general register $rt$ are compared to general register $rs$. Considering both quantities as unsigned integers, if the contents of general register $rs$ are less than the contents of general register $rt$, a trap exception occurs.

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.

**Operation:**

```
32, 64T:  if (0 || GPR[rs]) < (0 || GPR[rt]) then
          TrapException
          endif
```

**Exceptions:**

Trap exception
TNE Trap If Not Equal

Format:
TNE rs, rt

Description:
The contents of general register rt are compared to general register rs. If the contents of general register rs are not equal to the contents of general register rt, a trap exception occurs.

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.

Operation:

```
32, 64 T: if GPR[rs] ≠ GPR[rt] then
    TrapException
endif
```

Exceptions:
Trap exception
Format:
TNEI rs, immediate

Description:
The 16-bit immediate is sign-extended and compared to the contents of general register rs. If the contents of general register rs are not equal to the sign-extended immediate, a trap exception occurs.

Operation:

| 32  | T:  if GPR[rs] ≠ (immediate_{15})^{16} || immediate_{15}...0 then TrapException endif |
| 64  | T:  if GPR[rs] ≠ (immediate_{15})^{48} || immediate_{15}...0 then TrapException endif |

Exceptions:
Trap exception
### XOR: Exclusive Or

**Format:**

```
XOR rd, rs, rt
```

**Description:**

The contents of general register `rs` are combined with the contents of general register `rt` in a bit-wise logical exclusive OR operation.

The result is placed into general register `rd`.

**Operation:**

```
32, 64  T:  GPR[rd] ← GPR[rs] xor GPR[rt]
```

**Exceptions:**

None
**Xori**

**Exclusive OR 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>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

**Description:**

The 16-bit immediate is zero-extended and combined with the contents of general register rs in a bit-wise logical exclusive OR operation.

The result is placed into general register rt.

**Operation:**

| 32 | T: | GPR[rt] ← GPR[rs] xor (0^{16} || immediate) |
|----|----|---------------------------------------------|
| 64 | T: | GPR[rt] ← GPR[rs] xor (0^{48} || immediate) |

**Exceptions:**

None
CPU Instruction Opcode Bit Encoding

The remainder of this Appendix presents the opcode bit encoding for the CPU instruction set (ISA and extensions), as implemented by the R4000. Figure A-2 lists the R4000 Opcode Bit Encoding.

**Figure A-2   R4000 Opcode Bit Encoding**

```plaintext

<table>
<thead>
<tr>
<th>31..29</th>
<th>28..26</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SPECIAL REGIMM J</td>
</tr>
<tr>
<td>1</td>
<td>ADDI    ADDIU SLTI SLTIU</td>
</tr>
<tr>
<td>2</td>
<td>COP0    COP1 COP2</td>
</tr>
<tr>
<td>3</td>
<td>DADDle  DADDIUE LDLle LDRle</td>
</tr>
<tr>
<td>4</td>
<td>LB      LH LWL LW  LBU</td>
</tr>
<tr>
<td>5</td>
<td>SB      SH SWL SW  SDLc</td>
</tr>
<tr>
<td>6</td>
<td>LL      LWCl LWc2</td>
</tr>
<tr>
<td>7</td>
<td>SC      SWCl SWc2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5..3</th>
<th>2..0</th>
</tr>
</thead>
<tbody>
<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>ADDD</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>TGE</td>
</tr>
<tr>
<td>7</td>
<td>DSSLle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18..16</th>
<th>16..14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BLTZ</td>
</tr>
<tr>
<td>1</td>
<td>TGE1</td>
</tr>
<tr>
<td>2</td>
<td>BLTZAL</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>23..21</th>
<th>21..19</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MF</td>
</tr>
<tr>
<td>1</td>
<td>BC</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20..19</th>
<th>19..17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CF</td>
</tr>
<tr>
<td>1</td>
<td>MT</td>
</tr>
<tr>
<td>2</td>
<td>DMTe</td>
</tr>
<tr>
<td>3</td>
<td>CT</td>
</tr>
</tbody>
</table>
```

MIPS R4000 Microprocessor User's Manual

A-181
Key:

* Operation codes marked with an asterisk cause reserved instruction exceptions in all current implementations and are reserved for future versions of the architecture.

γ Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.

δ Operation codes marked with a delta are valid only for R4000 processors with CP0 enabled, and cause a reserved instruction exception on other processors.

φ Operation codes marked with a phi are invalid but do not cause reserved instruction exceptions in R4000 implementations.

ξ Operation codes marked with a xi cause a reserved instruction exception on R4000 processors.

χ Operation codes marked with a chi are valid only on R4000.

ε Operation codes marked with epsilon are valid when the processor is operating either in the Kernel mode or in the 64-bit non-Kernel (User or Supervisor) mode. These instructions cause a reserved instruction exception if 64-bit operation is not enabled in User or Supervisor mode.
This appendix provides a detailed description of each floating-point unit (FPU) instruction (refer to Appendix A for a detailed description of the CPU instructions). The instructions are listed alphabetically, and any exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate causes and the manner of handling exceptions are omitted from the instruction descriptions in this appendix (refer to Chapter 7 for detailed descriptions of floating-point exceptions and handling).

Figure B-3 at the end of this appendix lists the entire bit encoding for the constant fields of the floating-point instruction set; the bit encoding for each instruction is included with that individual instruction.
B.1 Instruction Formats

There are three basic instruction format types:

• I-Type, or Immediate instructions, which include load and store operations
• M-Type, or Move instructions
• R-Type, or Register instructions, which include the two- and three-register floating-point operations.

The instruction description subsections that follow show how these three basic instruction formats are used by:

• Load and store instructions
• Move instructions
• Floating-Point computational instructions
• Floating-Point branch instructions

Floating-point instructions are mapped onto the MIPS coprocessor instructions, defining coprocessor unit number one (CP1) as the floating-point unit.

Each operation is valid only for certain formats. Implementations may support some of these formats and operations through emulation, but they only need to support combinations that are valid (marked V in Table B-1). Combinations marked R in Table B-1 are not currently specified by this architecture, and cause an unimplemented operation trap. They will be available for future extensions to the architecture.
## Table B-1  Valid FPU Instruction Formats

<table>
<thead>
<tr>
<th>Operation</th>
<th>Single</th>
<th>Double</th>
<th>Word</th>
<th>Longword</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>SUB</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>MUL</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>DIV</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>SQRT</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>ABS</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>MOV</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>TRUNC.L</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUND.L</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEIL.L</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOOR.L</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRUNC.W</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROUND.W</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEIL.W</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOOR.W</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT.S</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>CVT.D</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT.W</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVT.L</td>
<td>V</td>
<td>V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>V</td>
<td>V</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
The coprocessor branch on condition true/false instructions can be used to logically negate any predicate. Thus, the 32 possible conditions require only 16 distinct comparisons, as shown in Table B-2 below.

**Table B-2  Logical Negation of Predicates by Condition True/False**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
<th>Greater Than</th>
<th>Less Than</th>
<th>Equal</th>
<th>Unordered</th>
<th>Invalid Operation Exception If Unordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>0</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>UN</td>
<td>OR</td>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>EQ</td>
<td>NEQ</td>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>UEQ</td>
<td>OGL</td>
<td>3</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>OLT</td>
<td>UGE</td>
<td>4</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>ULT</td>
<td>OGE</td>
<td>5</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>OLE</td>
<td>UGT</td>
<td>6</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>ULE</td>
<td>OGT</td>
<td>7</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>SF</td>
<td>ST</td>
<td>8</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>NGLE</td>
<td>GLE</td>
<td>9</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>SEQ</td>
<td>SNE</td>
<td>10</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NGL</td>
<td>GL</td>
<td>11</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>LT</td>
<td>NLT</td>
<td>12</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>NGE</td>
<td>GE</td>
<td>13</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>LE</td>
<td>NLE</td>
<td>14</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>NGT</td>
<td>GT</td>
<td>15</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Floating-Point Loads, Stores, and Moves

All movement of data between the floating-point coprocessor and memory is accomplished by coprocessor load and store operations, which reference the floating-point coprocessor General Purpose registers. These operations are unformatted; no format conversions are performed and, therefore, no floating-point exceptions can occur due to these operations.

Data may also be directly moved between the floating-point coprocessor and the processor by move to coprocessor and move from coprocessor instructions. Like the floating-point load and store operations, move to/from operations perform no format conversions and never cause floating-point exceptions.

An additional pair of coprocessor registers are available, called Floating-Point Control registers for which the only data movement operations supported are moves to and from processor General Purpose registers.

Floating-Point Operations

The floating-point unit operation set includes:

- floating-point add
- floating-point subtract
- floating-point multiply
- floating-point divide
- floating-point square root
- convert between fixed-point and floating-point formats
- convert between floating-point formats
- floating-point compare

These operations satisfy the requirements of IEEE Standard 754 requirements for accuracy. Specifically, these operations obtain a result which is identical to an infinite-precision result rounded to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for conversion functions, mixed-format operations are not provided.
B.2 Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as \( f_s \), \( ft \), \( immediate \), 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 in Figure B-3 at the end of this appendix, and are also included with each individual instruction.

In the instruction description examples that follow, the Operation section describes the operation performed by each instruction using a high-level language notation.

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

<table>
<thead>
<tr>
<th>Example #1:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{GPR}[rt] \leftarrow \text{immediate} \</td>
<td></td>
</tr>
<tr>
<td>Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General Purpose Register ( rt ).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example #2:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{immediate}_{15})^{16} \</td>
<td></td>
</tr>
<tr>
<td>Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.</td>
<td></td>
</tr>
</tbody>
</table>
B.3 Load and Store Instructions

In the R4000 implementation, the instruction immediately following a load may use the contents of the register being loaded. In such cases, the hardware interlocks, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code.

The behavior of the load store instructions is dependent on the width of the FGRs.

- When the FR bit in the Status register equals zero, the Floating-Point General registers (FGRs) are 32-bits wide.
- When the FR bit in the Status register equals one, the Floating-Point General registers (FGRs) are 64-bits wide.

In the load and store operation descriptions, the functions listed in Table B-3 are used to summarize the handling of virtual addresses and physical memory.

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressTranslation</td>
<td>Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.</td>
</tr>
<tr>
<td>LoadMemory</td>
<td>Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the Access Type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.</td>
</tr>
<tr>
<td>StoreMemory</td>
<td>Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the Access Type field indicates which of each of the four bytes within the data word should be stored.</td>
</tr>
</tbody>
</table>
Figure B-1 shows the I-Type instruction format used by load and store operations.

I-Type (Immediate)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op</td>
<td>base</td>
<td>ft</td>
<td></td>
<td></td>
<td>offset</td>
</tr>
</tbody>
</table>

- op is a 6-bit operation code
- base is the 5-bit base register specifier
- ft is a 5-bit source (for stores) or destination (for loads) FPA register specifier
- offset is the 16-bit signed immediate offset

All coprocessor loads and stores reference aligned data items. Thus, for word loads and stores, the access type field is always WORD, and the low-order two bits of the address must always be zero.

For doubleword loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero.

Regardless of byte-numbering order (endianness), the address specifies that byte which has the smallest byte-address in the addressed field. For a big-endian machine, this is the leftmost byte; for a little-endian machine, this is the rightmost byte.
B.4 Computational Instructions

Computational instructions include all of the arithmetic floating-point operations performed by the FPU.

Figure B-2 shows the R-Type instruction format used for computational operations.

R-Type (Register)

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<td>6</td>
</tr>
<tr>
<td></td>
<td>COP1</td>
<td>fmt</td>
<td>ft</td>
<td>fs</td>
<td>fd</td>
<td>function</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- COP1 is a 6-bit operation code
- fmt is a 5-bit format specifier
- fs is a 5-bit source1 register
- ft is a 5-bit source2 register
- fd is a 5-bit destination register
- function is a 6-bit function field

Figure B-2  Computational Instruction Format

The function field indicates the floating-point operation to be performed.

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by the 5-bit format field; decoding for this field is shown in Table B-4.

<table>
<thead>
<tr>
<th>Code</th>
<th>Mnemonic</th>
<th>Size</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>S</td>
<td>single</td>
<td>Binary floating-point</td>
</tr>
<tr>
<td>17</td>
<td>D</td>
<td>double</td>
<td>Binary floating-point</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>W</td>
<td>single</td>
<td>32-bit binary fixed-point</td>
</tr>
<tr>
<td>21</td>
<td>L</td>
<td>longword</td>
<td>64-bit binary fixed-point</td>
</tr>
<tr>
<td>22–31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Table B-5 lists all floating-point instructions.
Table B-5  Floating-Point Instructions and Operations

<table>
<thead>
<tr>
<th>Code (5:0)</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>1</td>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>2</td>
<td>MUL</td>
<td>Multiply</td>
</tr>
<tr>
<td>3</td>
<td>DIV</td>
<td>Divide</td>
</tr>
<tr>
<td>4</td>
<td>SQRT</td>
<td>Square root</td>
</tr>
<tr>
<td>5</td>
<td>ABS</td>
<td>Absolute value</td>
</tr>
<tr>
<td>6</td>
<td>MOV</td>
<td>Move</td>
</tr>
<tr>
<td>7</td>
<td>NEG</td>
<td>Negate</td>
</tr>
<tr>
<td>8</td>
<td>ROUND.L</td>
<td>Convert to 64-bit (long) fixed-point, rounded to nearest/even</td>
</tr>
<tr>
<td>9</td>
<td>TRUNC.L</td>
<td>Convert to 64-bit (long) fixed-point, rounded toward zero</td>
</tr>
<tr>
<td>10</td>
<td>CEIL.L</td>
<td>Convert to 64-bit (long) fixed-point, rounded to +∞</td>
</tr>
<tr>
<td>11</td>
<td>FLOOR.L</td>
<td>Convert to 64-bit (long) fixed-point, rounded to −∞</td>
</tr>
<tr>
<td>12</td>
<td>ROUND.W</td>
<td>Convert to single fixed-point, rounded to nearest/even</td>
</tr>
<tr>
<td>13</td>
<td>TRUNC.W</td>
<td>Convert to single fixed-point, rounded toward zero</td>
</tr>
<tr>
<td>14</td>
<td>CEIL.W</td>
<td>Convert to single fixed-point, rounded to +∞</td>
</tr>
<tr>
<td>15</td>
<td>FLOOR.W</td>
<td>Convert to single fixed-point, rounded to −∞</td>
</tr>
<tr>
<td>16–31</td>
<td>–</td>
<td>Reserved</td>
</tr>
<tr>
<td>32</td>
<td>CVT.S</td>
<td>Convert to single floating-point</td>
</tr>
<tr>
<td>33</td>
<td>CVT.D</td>
<td>Convert to double floating-point</td>
</tr>
<tr>
<td>34</td>
<td>–</td>
<td>Reserved</td>
</tr>
<tr>
<td>35</td>
<td>–</td>
<td>Reserved</td>
</tr>
<tr>
<td>36</td>
<td>CVT.W</td>
<td>Convert to 32-bit binary fixed-point</td>
</tr>
<tr>
<td>37</td>
<td>CVT.L</td>
<td>Convert to 64-bit (long) binary fixed-point</td>
</tr>
<tr>
<td>38–47</td>
<td>–</td>
<td>Reserved</td>
</tr>
<tr>
<td>48–63</td>
<td>C</td>
<td>Floating-point compare</td>
</tr>
</tbody>
</table>
In the following pages, the notation FGR refers to the 32 General Purpose registers FGR0 through FGR31 of the FPU, and FPR refers to the floating-point registers of the FPU.

- When the FR bit in the Status register (SR(26)) equals zero, only the even floating-point registers are valid and the 32 General Purpose registers of the FPU are 32-bits wide.
- When the FR bit in the Status register (SR(26)) equals one, both odd and even floating-point registers may be used and the 32 General Purpose registers of the FPU are 64-bits wide.

The following routines are used in the description of the floating-point operations to retrieve the value of an FPR or to change the value of an FGR:

```plaintext
value ← ValueFPR(fpr,fmt)

if SR26 = 1 then /* 64-bit wide FGRs */
  case fmt of
    S, W:
      value ← FGR[fpr]31...0
      return
    D, L:
      value ← FGR[fpr]
      return
  endcase
elseif fpr0 = 0 then /* valid specifier, 32-bit wide FGRs */
  case fmt of
    S, W:
      value ← FGR[fpr]
      return
    D, L:
      value ← FGR[fpr+1] || FGR[fpr]
      return
  endcase
else /* undefined result for odd 32-bit reg #s */
  value ← undefined
endif
```
StoreFPR(fpr, fmt, value)

if SR_{26} = 1 then /* 64-bit wide FGRs */
   case fmt of
      S, W:
         FGR[fpr] ← undefined^{32} || value
         return
      D, L:
         FGR[fpr] ← value
         return
   endcase
elseif fpr_0 = 0 then /* valid specifier, 32-bit wide FGRs */
   case fmt of
      S, W:
         FGR[fpr+1] ← undefined
         FGR[fpr] ← value
         return
      D, L:
         FGR[fpr+1] ← value_{63...32}
         FGR[fpr] ← value_{31...0}
         return
   endcase
else /* undefined result for odd 32-bit reg #s */
   undefined_result
endif
ABS.fmt fd, fs

Description:
The contents of the FPU register specified by $fs$ are interpreted in the specified format and the arithmetic absolute value is taken. The result is placed in the floating-point register specified by $fd$.

The absolute value operation is arithmetic; a NaN operand signals invalid operation.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

Operation:

$$T: \text{StoreFPR}(fd, fmt, \text{AbsoluteValue(ValueFPR}(fs, fmt)))$$

Exceptions:

- Coprocessor unusable exception
- Coprocessor exception trap

Coprocessor Exceptions:

- Unimplemented operation exception
- Invalid operation exception
ADD.fmt Floating-Point Add ADD.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>ft</td>
<td>fs</td>
<td>fd</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

ADD.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by fs and ft are interpreted in the specified format and arithmetically added. The result is rounded as if calculated to infinite precision and then rounded to the specified format (fmt), according to the current rounding mode. The result is placed in the floating-point register (FPR) specified by fd.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

Operation:

T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) + ValueFPR(ft, fmt))

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception
FPU Instruction Set Details

BC1F Branch On FPA False (Coprocessor 1)

Format:
BC1F offset

Description:
A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the result of the last floating-point compare is false (zero), the program branches to the target address, with a delay of one instruction.

There must be at least one instruction between C.cond.fmt and BC1F.

Operation:

<table>
<thead>
<tr>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>T–1:</td>
<td>condition ← not COC[1]</td>
</tr>
<tr>
<td>T:</td>
<td>target ← (offset\textsubscript{15})^{14}</td>
</tr>
<tr>
<td>T+1:</td>
<td>if condition then</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + target</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
</tbody>
</table>

Exceptions:
Coprocessor unusable exception
Appendix B

BC1FL  Branch On FPU False Likely  (Coprocessor 1)  BC1FL

<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>COP1</td>
<td>BC</td>
<td>BCFL</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 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

BC1FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the result of the last floating-point compare is false (zero), the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

There must be at least one instruction between C.cond.fmt and BC1FL.

Operation:

<table>
<thead>
<tr>
<th>32</th>
<th>T–1: condition ← not COC[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T:</td>
<td>target ← (offset16)14</td>
</tr>
<tr>
<td>T+1:</td>
<td>if condition then</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + target</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>NullifyCurrentInstruction</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64</th>
<th>T–1: condition ← not COC[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>T:</td>
<td>target ← (offset16)46</td>
</tr>
<tr>
<td>T+1:</td>
<td>if condition then</td>
</tr>
<tr>
<td></td>
<td>PC ← PC + target</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>NullifyCurrentInstruction</td>
</tr>
<tr>
<td></td>
<td>endif</td>
</tr>
</tbody>
</table>

Exceptions:

Coprocessor unusable exception
Format:

BC1T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the result of the last floating-point compare is true (one), the program branches to the target address, with a delay of one instruction.

There must be at least one instruction between C.cond.fmt and BC1T.

Operation:

| 32   | T–1:  | condition ← COP1[1] |
|      | T:    | target ← (offset15)[14] || offset || 0^2 |
|      | T+1:  | if condition then    |
|      |      | PC ← PC + target     |
|      |      | endif                |

| 64   | T–1:  | condition ← COP1[1] |
|      | T:    | target ← (offset15)[46] || offset || 0^2 |
|      | T+1:  | if condition then    |
|      |      | PC ← PC + target     |
|      |      | endif                |

Exceptions:

Coprocessor unusable exception
Appendix B

BC1TL  Branch On FPU True Likely  (Coprocessor 1)  BC1TL

<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>COP1</td>
<td>BC</td>
<td>BCTL</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 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

BC1TL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. If the result of the last floating-point compare is true (one), the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

There must be at least one instruction between C.cond.fmt and BC1TL.

Operation:

\[
\begin{align*}
32 & \quad \text{T–1: } \text{condition } \leftarrow \text{COC}[1] \\
T & \quad \text{target } \leftarrow \text{(offset}_{15}^{14} \ || \text{ offset } || \ 0^2 \\
T+1 & \quad \text{if condition then} \\
& \quad \text{PC } \leftarrow \text{PC } + \text{target} \\
& \quad \text{else} \\
& \quad \text{NullifyCurrentInstruction} \\
& \quad \text{endif} \\
64 & \quad \text{T–1: } \text{condition } \leftarrow \text{COC}[1] \\
T & \quad \text{target } \leftarrow \text{(offset}_{15}^{46} \ || \text{ offset } || \ 0^2 \\
T+1 & \quad \text{if condition then} \\
& \quad \text{PC } \leftarrow \text{PC } + \text{target} \\
& \quad \text{else} \\
& \quad \text{NullifyCurrentInstruction} \\
& \quad \text{endif}
\end{align*}
\]

Exceptions:

Coprocessor unusable exception
FPU Instruction Set Details

**Format:**

\[ \text{C.cond.fmt } \text{fs, ft} \]

**Description:**

The contents of the floating-point registers specified by \( \text{fs} \) and \( \text{ft} \) are interpreted in the specified format, \( \text{fmt} \), and arithmetically compared.

A result is determined based on the comparison and the conditions specified in the \( \text{cond} \) field. If one of the values is a Not a Number (NaN), and the high-order bit of the \( \text{cond} \) field is set, an invalid operation exception is taken. After a one-instruction delay, the condition is available for testing with branch on floating-point coprocessor condition instructions. There must be at least one instruction between the compare and the branch.

Comparisons are exact and can neither overflow nor underflow. Four mutually-exclusive relations are possible results: less than, equal, greater than, and unordered. The last case arises when one or both of the operands are NaN; every NaN compares unordered with everything, including itself.

Comparisons ignore the sign of zero, so +0 = –0.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the \( \text{FR} \) bit in the \( \text{Status} \) register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the \( \text{FR} \) bit in the \( \text{Status} \) register equals one, both even and odd register numbers are valid.

\[ \begin{array}{ccccccccc}
31 & 26 & 25 & 21 & 20 & 16 & 15 & 11 & 10 & 6 & 5 & 4 & 3 & 0 \\
\hline
\text{COP1} & \text{fmt} & \text{ft} & \text{fs} & 0 & \text{FC*} & \text{cond*} \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

*See “FPU Instruction Opcode Bit Encoding” at the end of Appendix B.
C.cond.fmt  Floating-Point Compare (continued)  C.cond.fmt

Operation:

\[
T: \begin{align*}
&\text{if NaN(ValueFPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then} \\
&\quad \text{less } \leftarrow \text{false} \\
&\quad \text{equal } \leftarrow \text{false} \\
&\quad \text{unordered } \leftarrow \text{true} \\
&\quad \text{if cond}_3 \text{ then} \\
&\quad \quad \text{signal InvalidOperationException} \\
&\quad \text{endif} \\
&\quad \text{else} \\
&\quad \quad \text{less } \leftarrow \text{ValueFPR}(fs, fmt) < \text{ValueFPR}(ft, fmt) \\
&\quad \quad \text{equal } \leftarrow \text{ValueFPR}(fs, fmt) = \text{ValueFPR}(ft, fmt) \\
&\quad \quad \text{unordered } \leftarrow \text{false} \\
&\quad \text{endif} \\
&\text{condition } \leftarrow (\text{cond}_2 \text{ and less}) \text{ or } (\text{cond}_1 \text{ and equal}) \text{ or} \\
&\quad (\text{cond}_0 \text{ and unordered}) \\
&\text{FCR}_{31}^{23} \leftarrow \text{condition} \\
&\text{COC}_1 \leftarrow \text{condition}
\end{align*}
\]

Exceptions:

- Coprocessor unusable
- Floating-Point exception

Coprocessor Exceptions:

- Unimplemented operation exception
- Invalid operation exception
**CEIL.L.fmt**

**Floating-Point Ceiling to Long Fixed-Point Format**

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

**Format:**

CEIL.L.fmt fd, fs

**Description:**

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $+\infty$ (2).

This instruction is valid only for conversion from single- or double-precision floating-point formats. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of $-2^{63}$ to $2^{63} - 1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63} - 1$ is returned.
CEIL.L.fmt Floating-Point Ceiling to Long Fixed-Point Format (continued)

Operation:

\[
T: \text{StoreFPR}(fd, L, \text{ConvertFmt}(%\text{ValueFPR}(fs, fmt), fmt, L))
\]

Exceptions:
- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:
- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
CEIL.W.fmt

Floating-Point Ceiling to Single Fixed-Point Format

Format:

CEIL.W.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $+\infty$ (2).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of $-2^{31}$ to $2^{31} - 1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{31} - 1$ is returned.
CEIL.W.fmt

Floating-Point
Ceiling to Single
Fixed-Point Format
(continued)

Operation:

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

Exceptions:

- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
### CFC1: Move Control Word From FPU (Coprocessor 1)

**Format:**

\[
\text{CFC1 rt, fs}
\]

**Description:**

The contents of the FPU control register \(fs\) are loaded into general register \(rt\).

This operation is only defined when \(fs\) equals 0 or 31.

The contents of general register \(rt\) are undefined for the instruction immediately following CFC1.

**Operation:**

\[
\begin{align*}
32 & \quad \text{T: } \quad \text{temp } & \leftarrow & \text{FCR}[fs] \\
& \quad \text{T+1: } \quad \text{GPR}[rt] & \leftarrow & \text{temp}
\end{align*}
\]

\[
\begin{align*}
64 & \quad \text{T: } \quad \text{temp } & \leftarrow & \text{FCR}[fs] \\
& \quad \text{T+1: } \quad \text{GPR}[rt] & \leftarrow & (\text{temp}_{31})^{32} \parallel \text{temp}
\end{align*}
\]

**Exceptions:**

Coprocessor unusable exception
CTC1 Move Control Word To FPU (Coprocessor 1)

Format:

CTC1 rt, fs

Description:

The contents of general register rt are loaded into FPU control register fs. This operation is only defined when fs equals 0 or 31.

Writing to Control Register 31, the floating-point Control/Status register, causes an interrupt or exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs. The contents of floating-point control register fs are undefined for the instruction immediately following CTC1.

Operation:

| 32 | T: temp ← GPR[rt] |
| 64 | T+1: FCR[fs] ← temp |
|    | COC[1] ← FCR[31]_{23} |

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Division by zero exception
Inexact exception
Overflow exception
Underflow exception
Format:

CVT.D.fmt fd, fs

Description:

The contents of the floating-point register specified by fs is interpreted in the specified source format, fmt, and arithmetically converted to the double binary floating-point format. The result is placed in the floating-point register specified by fd.

This instruction is valid only for conversions from single floating-point format, 32-bit or 64-bit fixed-point format.

If the single floating-point or single fixed-point format is specified, the operation is exact. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

Operation:

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

T: StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception
CVT.L.fmt  Convert to Long Fixed-Point Format

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

Format:

CVT.L.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd. This instruction is valid only for conversions from single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of $-2^{63}$ to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

Operation:

T:  StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
### CVT.S.fmt

**Floating-Point Convert to Single Floating-Point Format**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>fmt</td>
<td>0</td>
<td>fs</td>
<td>fd</td>
<td>CVT.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>00000</td>
<td>00000</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

\[
\text{CVT.S.fmt fd, fs}
\]

**Description:**

The contents of the floating-point register specified by \( fs \) are interpreted in the specified source format, \( fmt \), and arithmetically converted to the single binary floating-point format. The result is placed in the floating-point register specified by \( fd \). Rounding occurs according to the currently specified rounding mode.

This instruction is valid only for conversions from double floating-point format, or from 32-bit or 64-bit fixed-point format. The operation is not defined if bit 0 of any register specification is set and the \( FR \) bit in the \( Status \) register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the \( FR \) bit in the \( Status \) register equals one, both even and odd register numbers are valid.

**Operation:**

\[
T: \text{StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))}
\]

**Exceptions:**

- Coprocessor unusable exception
- Floating-Point exception

**Coprocessor Exceptions:**

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
- Underflow exception
**CVT.W.fmt** *Convert to Fixed-Point Format*

<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>fs</td>
<td>fd</td>
<td>CVT.W</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>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

`CVT.W.fmt fd, fs`

**Description:**

The contents of the floating-point register specified by `fs` are interpreted in the specified source format, `fmt`, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by `fd`. This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the `FR` bit in the `Status` register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the `FR` bit in the `Status` register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of $-2^{31}$ to $2^{31}-1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31}-1$ is returned.

**Operation:**

```
T: StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

**Exceptions:**

- Coprocessor unusable exception
- Floating-Point exception

**Coprocessor Exceptions:**

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
**FPU Instruction Set Details**

**DIV.fmt**  
**Floating-Point Divide**  
**DIV.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>ft</td>
<td>fs</td>
<td>fd</td>
<td>DIV</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:**

DIV.fmt fd, fs, ft

**Description:**

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and the value in the fs field is divided by the value in the ft field. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid for only single or double precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

**Operation:**

\[ T: \text{StoreFPR} (fd, fmt, \text{ValueFPR}(fs, fmt)/\text{ValueFPR}(ft, fmt)) \]

**Exceptions:**

- Coprocessor unusable exception
- Floating-Point exception

**Coprocessor Exceptions:**

- Unimplemented operation exception
- Invalid operation exception
- Division-by-zero exception
- Inexact exception
- Overflow exception
- Underflow exception
Appendix B

**Format:**

DMFC1 rt, fs

**Description:**

The contents of register \( fs \) from the floating-point coprocessor is stored into processor register \( rt \).

The contents of general register \( rt \) are undefined for the instruction immediately following DMFC1.

The \( FR \) bit in the Status register specifies whether all 32 registers of the R4000 are addressable. When \( FR \) equals zero, this instruction is not defined when the least significant bit of \( fs \) is non-zero. When \( FR \) is set, \( fs \) may specify either odd or even registers.

**Operation:**

| 64 | T: | if SR26 = 1 then /* 64-bit wide FGRs */ data ← FGR[fs]  
|    |    | elsif fs_0 = 0 then /* valid specifier, 32-bit wide FGRs */ data ← FGR[fs+1] || FGR[fs]  
|    |    | else /* undefined for odd 32-bit reg #s */ data ← undefined64  
|    | endif  
|    | T+1: GPR[rt] ← data |

**Exceptions:**

Coprocessor unusable exception

**Coprocessor Exceptions:**

Unimplemented operation exception
DMTC1

**Doubleword Move To Floating-Point Coprocessor**

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>DMT</td>
<td>rt</td>
<td>fs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

DMTC1 rt, fs

**Description:**

The contents of general register rt are loaded into coprocessor register fs of the CP1.

The contents of floating-point register fs are undefined for the instruction immediately following DMTC1.

The FR bit in the Status register specifies whether all 32 registers of the R4000 are addressable. When FR equals zero, this instruction is not defined when the least significant bit of fs is non-zero. When FR equals one, fs may specify either odd or even registers.

**Operation:**

```plaintext
64 T:  data ← GPR[rt]
T+1:  if SR26 = 1 then  /* 64-bit wide FGRs */  
      FGR[fs] ← data
       else  fs0 = 0 then  /*valid specifier, 32-bit wide valid FGRs */  
       FGR[fs+1] ← data63...32
       FGR[fs] ← data31...0
       else  /* undefined result for odd 32-bit reg #s */  
       undefined_result
       endif
```

**Exceptions:**

Coprocessor unusable exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Format:

FLOOR.L.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to \(-\infty\) (3).

This instruction is valid only for conversion from single- or double-precision floating-point formats.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of \(-2^{63}\) to \(2^{63} – 1\), the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and \(2^{63} – 1\) is returned.
FLOOR.L.fmt  Floating-Point Floor to Long Fixed-Point Format (continued)

Operation:

\[
T: \text{StoreFPR}(fd, L, \text{ConvertFmt}(\text{ValueFPR}(fs, fmt), fmt, L))
\]

Exceptions:
- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:
- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
FLOOR.W.fmt Floor to Single Fixed-Point Format

<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>00000</td>
<td>fs</td>
<td>fd</td>
<td>FLOOR.W</td>
<td>001111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

FLOOR.W.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $-\infty$ (RM = 3).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of $-2^{31}$ to $2^{31}-1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31}-1$ is returned.
**FLOOR.W.fmt**

*Floating-Point Floor to Single Fixed-Point Format (continued)*

**Operation:**

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

**Exceptions:**

- Coprocessor unusable exception
- Floating-Point exception

**Coprocessor Exceptions:**

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
LDC1  Load Doubleword to FPU  
(Coprocessor 1)

<table>
<thead>
<tr>
<th>Format:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDC1 ft, offset(base)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The 16-bit offset is sign-extended and added to the contents of general register base to form an unsigned effective address.</td>
</tr>
<tr>
<td>In 32-bit mode, the contents of the doubleword at the memory location specified by the effective address is loaded into registers ft and ft+1 of the floating-point coprocessor. This instruction is not valid, and is undefined, when the least significant bit of ft is non-zero.</td>
</tr>
<tr>
<td>In 64-bit mode, the contents of the doubleword at the memory location specified by the effective address are loaded into the 64-bit register ft of the floating point coprocessor.</td>
</tr>
<tr>
<td>The FR bit of the Status register (SR26) specifies whether all 32 registers of the R4000 are addressable. If FR equals zero, this instruction is not defined when the least significant bit of ft is non-zero. If FR equals one, ft may specify either odd or even registers.</td>
</tr>
<tr>
<td>If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.</td>
</tr>
</tbody>
</table>
## LDC1

**Load Doubleword to FPU**  
(Coprocessor 1)  
(continued)

### Operation:

| 32  | T: | vAddr ← ((offset_{15})^{16} || offset_{15...0}) + GPR[base] |
|-----|----|------------------------------------------------------------------|
| 64  | T: | vAddr ← ((offset_{15})^{48} || offset_{15...0}) + GPR[base]   |

32, 64  
(pAddr, uncached) ← AddressTranslation (vAddr, DATA)  
data ← LoadMemory(uncached, DOUBLEWORD, pAddr, vAddr, DATA)  
if SR_{26} = 1 then  /* 64-bit wide FGRs */  
FGR[ft] ← data  
elseif ft_{0} = 0 then  /* valid specifier, 32-bit wide FGRs */  
FGR[ft+1] ← data_{63...32}  
FGR[ft] ← data_{31...0}  
else  /* undefined result if odd */  
undefined_result  
endif

### Exceptions:

- Coprocessor unusable
- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
**LWC1**

**Load Word to FPU**

(Coprocessor 1)

<table>
<thead>
<tr>
<th>Bit Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>LWC1</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

**Format:**

LWC1 ft, offset(base)

**Description:**

The 16-bit offset is sign-extended and added to the contents of general register base to form an unsigned effective address. The contents of the word at the memory location specified by the effective address is loaded into register ft of the floating-point coprocessor.

The FR bit of the Status register specifies whether all 64-bit Floating-Point registers are addressable. If FR equals zero, LWC1 loads either the high or low half of the 16 even Floating-Point registers. If FR equals one, LWC1 loads the low 32-bits of both even and odd Floating-Point registers.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.
LWC1  Load Word to FPU  
(Coprocessor 1)  
(continued)

Operation:

| 32 | T: \( \text{vAddr} \leftarrow ((\text{offset}_{15} \ll 16) || \text{offset}_{15...0}) + \text{GPR[base]} \) |
|----|---|
| 64 | T: \( \text{vAddr} \leftarrow ((\text{offset}_{15} \ll 48) || \text{offset}_{15...0}) + \text{GPR[base]} \) |

32, 64  
\( (\text{pAddr, uncached}) \leftarrow \text{AddressTranslation (vAddr, DATA)} \)  
\( \text{pAddr} \leftarrow \text{pAddr}_{PSIZE-1...3} || (\text{pAddr}_{2...0} \text{xor (ReverseEndian || 0^2)}) \)  
\( \text{mem} \leftarrow \text{LoadMemory(uncached, WORD, pAddr, vAddr, DATA)} \)  
\( \text{byte} \leftarrow \text{vAddr}_{2...0} \text{xor (BigEndianCPU || 0^2)} \)  
/* “mem” is aligned 64-bits from memory. Pick out correct bytes. */  
if \( \text{SR}_{26} = 1 \) then /* 64-bit wide FGRs */  
\( \text{FGR[ft]} \leftarrow \text{undefined}_{32} \ || \ \text{mem}_{31+8*\text{byte}...8*\text{byte}} \)  
ext else /* 32-bit wide FGRs */  
\( \text{FGR[ft]} \leftarrow \text{mem}_{31+8*\text{byte}...8*\text{byte}} \)  
endif

Exceptions:

- Coprocessor unusable
- TLB refill exception
- TLB invalid exception
- Bus error exception
- Address error exception
Appendix B

MFC1  Move From FPU (Coprocessor 1)  MFC1

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>MF</td>
<td>rt</td>
<td>fs</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

MFC1 rt, fs

Description:
The contents of register fs from the floating-point coprocessor are stored into processor register rt.
The contents of register rt are undefined for the instruction immediately following MFC1.
The FR bit of the Status register specifies whether all 32 registers of the R4000 are addressable. If FR equals zero, MFC1 stores either the high or low half of the 16 even Floating-Point registers. If FR equals one, MFC1 stores the low 32-bits of both even and odd Floating-Point registers.

Operation:

32 T: data ← FGR[fs]31...0  
T+1: GPR[rt] ← data  

64 T: data ← FGR[fs]31...0  
T+1: GPR[rt] ← (data31)32 || data

Exceptions:
Coprocessor unusable exception
**MOV.fmt**

**Floating-Point Move**

**MOV.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>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>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MOV.fmt fd, fs

**Description:**

The contents of the FPU register specified by fs are interpreted in the specified format and are copied into the FPU register specified by fd.

The move operation is non-arithmetic; no IEEE 754 exceptions occur as a result of the instruction.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

**Operation:**

T: StoreFPR(fd, fmt, ValueFPR(fs, fmt))

**Exceptions:**

Coprocessor unusable exception

Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
MTC1
Move To FPU
(Coprocessor 1)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>COP1</td>
<td>MT</td>
<td>rt</td>
<td>fs</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 0 1</td>
<td>0 0 1 0 0</td>
<td>5</td>
<td>5</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:
MTC1 rt, fs

Description:
The contents of register rt are loaded into the FPU general register at location fs.
The contents of floating-point register fs is undefined for the instruction immediately following MTC1.
The FR bit of the Status register specifies whether all 32 registers of the R4000 are addressable. If FR equals zero, MTC1 loads either the high or low half of the 16 even Floating-Point registers. If FR equals one, MTC1 loads the low 32-bits of both even and odd Floating-Point registers.

Operation:

32,64 T: data ← GPR[rt]31...0
T+1: if SR26 = 1 then /* 64-bit wide FGRs */
     FGR[fs] ← undefined32 || data
else /* 32-bit wide FGRs */
     FGR[fs] ← data
endif

Exceptions:
Coprocessor unusable exception
FPU Instruction Set Details

MUL.fmt Floating-Point Multiply MUL.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>ft</td>
<td>fs</td>
<td>fd</td>
<td>MUL</td>
<td>0 0 0 0 1 0</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

MUL.fmt fd, fs, ft

**Description:**

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and arithmetically multiplied. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

**Operation:**

\[ T: \text{StoreFPR} (fd, \text{fmt}, \text{ValueFPR}(fs, \text{fmt}) \times \text{ValueFPR}(ft, \text{fmt})) \]

**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception
### NEG.fmt: Floating-Point Negate

**Format:**

\[
\text{NEG.fmt \( fd, fs \)}
\]

**Description:**

The contents of the FPU register specified by \( fs \) are interpreted in the specified format and the arithmetic negation is taken (polarity of the sign-bit is changed). The result is placed in the FPU register specified by \( fd \).

The negate operation is arithmetic; an NaN operand signals invalid operation.

This instruction is valid only for single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the \( FR \) bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the \( FR \) bit in the Status register equals one, both even and odd register numbers are valid.

**Operation:**

\[
T: \text{StoreFPR}(fd, fmt, \text{Negate(ValueFPR(fs, fmt)))}
\]

**Exceptions:**

- Coprocessor unusable exception
- Floating-Point exception

**Coprocessor Exceptions:**

- Unimplemented operation exception
- Invalid operation exception
The contents of the floating-point register specified by \( fs \) are interpreted in the specified source format, \( fmt \), and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by \( fd \).

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (0).

This instruction is valid only for conversion from single- or double-precision floating-point formats.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of \(-2^{63}\) to \(2^{63}-1\), the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and \(2^{63}-1\) is returned.
ROUND.L.fmt  Floating-Point
Round to Long
Fixed-Point Format
(continued)

Operation:

| T: | StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L)) |

Exceptions:

- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
FPU Instruction Set Details

ROUND.W.fmt Floating-Point Round to Single Fixed-Point Format

<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>00000</td>
<td>fs</td>
<td>fd</td>
<td>ROUND.W</td>
<td>001100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

ROUND.W.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to the nearest/even (RM = 0).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of $-2^{31}$ to $2^{31} - 1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31} - 1$ is returned.
ROUND.W.fmt  Floating-Point  ROUND.W.fmt
Round to Single  ROUND.W.fmt
Fixed-Point Format  (continued)

Operation:

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

Exceptions:

- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:

- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
**Format:**

SDC1 ft, offset(base)

**Description:**

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address.

In 32-bit mode, the contents of registers *ft* and *ft+1* from the floating-point coprocessor are stored at the memory location specified by the effective address. This instruction is not valid, and is undefined, when the least significant bit of *ft* is non-zero.

In 64-bit mode, the 64-bit register *ft* is stored to the contents of the doubleword at the memory location specified by the effective address. The *FR* bit of the *Status* register (SR26) specifies whether all 32 registers of the R4000 are addressable. When FR equals zero, this instruction is not defined if the least significant bit of *ft* is non-zero. If FR equals one, *ft* may specify either odd or even registers.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.
SDC1 Store Doubleword from FPU
(Coprocessor 1)
(continued)

Operation:

\[
\begin{array}{ll}
32 & T: \text{vAddr} \leftarrow (\text{offset}_{15})_{16} \ || \ (\text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
64 & T: \text{vAddr} \leftarrow (\text{offset}_{15})_{48} \ || \ (\text{offset}_{15...0}) + \text{GPR}[\text{base}] \\
\end{array}
\]

\[
\begin{array}{l}
32,64 \text{ (pAddr, uncached)} \leftarrow \text{AddressTranslation (vAddr, DATA)} \\
\quad \text{if SR}_{26} = 1 \quad /* 64-bit wide FGRs */ \\
\quad \quad \text{data} \leftarrow \text{FGR}[ft] \\
\quad \text{elseif } ft_{0} = 0 \text{ then } /* valid specifier, 32-bit wide FGRs */ \\
\quad \quad \text{data} \leftarrow \text{FGR}[ft+1] \ || \ \text{FGR}[ft] \\
\quad \text{else } /* \text{undefined for odd 32-bit reg #s } */ \\
\quad \quad \text{data} \leftarrow \text{undefined}_{64} \\
\quad \text{endif} \\
\quad \text{StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)}
\end{array}
\]

Exceptions:

- Coprocessor unusable
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
FPU Instruction Set Details

SQRT.fmt Floating-Point Square Root SQRT.fmt

<table>
<thead>
<tr>
<th></th>
<th>COP1</th>
<th>fmt</th>
<th>0</th>
<th>fs</th>
<th>fd</th>
<th>SQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>010001</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>000100</td>
</tr>
</tbody>
</table>

Format:

SQRT.fmt fd, fs

Description:

The contents of the floating-point register specified by fs are interpreted in the specified format and the positive arithmetic square root is taken. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. If the value of fs corresponds to –0, the result will be –0. The result is placed in the floating-point register specified by fd.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

Operation:

T: StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))

Exceptions:

Coprocessor usable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Appendix B

SUB.fmt Floating-Point Subtract SUB.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>ft</td>
<td>fs</td>
<td>fd</td>
<td>SUB</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>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Format:

SUB.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by fs and ft are interpreted in the specified format and the value in the ft field is subtracted from the value in the fs field. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by fd. This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

Operation:

T: StoreFPR (fd, fmt, ValueFPR(fs, fmt) – ValueFPR(ft, fmt))

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception
SWC1 Store Word from FPU (Coprocessor 1)

Format:
SWC1 ft, offset(base)

Description:
The 16-bit offset is sign-extended and added to the contents of general register base to form an unsigned effective address. The contents of register ft from the floating-point coprocessor are stored at the memory location specified by the effective address.

The FR bit of the Status register specifies whether all 64-bit floating-point registers are addressable.

If FR equals zero, SWC1 stores either the high or low half of the 16 even floating-point registers.

If FR equals one, SWC1 stores the low 32-bits of both even and odd floating-point registers.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.
SWC1 Store Word from FPU (Coprocessor 1) (continued)

Operation:

<table>
<thead>
<tr>
<th>Type</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 T</td>
<td>vAddr ← ((offset_15^16</td>
</tr>
<tr>
<td>64 T</td>
<td>vAddr ← ((offset_15^48</td>
</tr>
</tbody>
</table>

32, 64 (pAddr, uncached) ← AddressTranslation (vAddr, DATA)

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

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

/* the bytes of the word are put in the correct byte lanes in
  * "data" for a 64-bit path to memory */

if SR26 = 1 then /* 64-bit wide FGRs */
  data ← FGR[ft]^63-8*byte...0 || 0^8*byte
else /* 32-bit wide FGRs */
  data ← 0^32-8*byte || FGR[ft] || 0^8*byte
endif

StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exceptions:

- Coprocessor unusable
- TLB refill exception
- TLB invalid exception
- TLB modification exception
- Bus error exception
- Address error exception
TRUNC.L.fmt  Floating-Point Truncate to Long Fixed-Point Format

<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>TRUNC.L</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 0 0 0 0</td>
<td>0 0 0 0 0</td>
<td>0 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Format:**

TRUNC.L.fmt fd, fs

**Description:**

The contents of the floating-point register specified by fs are interpreted in the specified source format, fmt, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (1).

This instruction is valid only for conversion from single- or double-precision floating-point formats.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of $-2^{63}$ to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.
TRUNC.L.fmt  Floating-Point Truncate to Long Fixed-Point Format (continued)

Operation:

T: StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))

Exceptions:
- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:
- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
**TRUNC.W.fmt**  
Floating-Point Truncate to Single Fixed-Point Format

<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>TRUNC.W</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Format:**

TRUNC.W.fmt fd, fs

**Description:**

The contents of the FPU register specified by fs are interpreted in the specified source format fmt and arithmetically converted to the single fixed-point format. The result is placed in the FPU register specified by fd.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (RM = 1).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the FR bit in the Status register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the FR bit in the Status register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of \(-2^{31}\) to \(2^{31}-1\), an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and \(-2^{31}\) is returned.
TRUNC.W.fmt  Floating-Point Format
Truncate to Single
Fixed-Point Format
(continued)

Operation:

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

Exceptions:
- Coprocessor unusable exception
- Floating-Point exception

Coprocessor Exceptions:
- Invalid operation exception
- Unimplemented operation exception
- Inexact exception
- Overflow exception
### FPU Instruction Opcode Bit Encoding

<table>
<thead>
<tr>
<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>28...26</td>
<td>Opcode</td>
<td>23...21</td>
<td>18...16</td>
<td>20...19</td>
<td>25...24</td>
<td>23...21</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>2</td>
<td>COP1</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>
</tr>
<tr>
<td>4</td>
<td></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>
</tr>
<tr>
<td>6</td>
<td>LWC1</td>
<td>LDC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SWC1</td>
<td>SDC1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure B-3  Bit Encoding for FPU Instructions**
Key:

γ Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.

δ Operation codes marked with a delta cause unimplemented operation exceptions in all current implementations and are reserved for future versions of the architecture.

η Operation codes marked with an eta are valid only when MIPS III instructions are enabled. Any attempt to execute these without MIPS III instructions enabled causes an unimplemented operation exception.

### Figure B-3 (cont.) Bit Encoding for FPU Instructions

<table>
<thead>
<tr>
<th>5...3</th>
<th>2...0 function</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>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>
</tr>
<tr>
<td>1</td>
<td>ROUND.Lη</td>
<td>TRUNC.Lη</td>
<td>CEIL.Lη</td>
<td>FLOOR.Lη</td>
<td>ROUND.W</td>
<td>TRUNC.W</td>
<td>CEIL.W</td>
<td>FLOOR.W</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>δ δ δ δ δ δ δ δ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>δ δ δ δ δ δ δ δ</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>CVT.D</td>
<td>δ δ</td>
<td>CVT.W</td>
<td>CVT.Lη</td>
<td>δ δ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>δ δ δ δ δ δ δ δ</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</td>
<td>C.UN</td>
<td>C.EQ</td>
<td>C.UEQ</td>
<td>C.OLT</td>
<td>C.ULT</td>
<td>C.OLE</td>
<td>C.ULE</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C.SF</td>
<td>C.NGLE</td>
<td>C.SEQ</td>
<td>C.NGL</td>
<td>C.LT</td>
<td>C.NGE</td>
<td>C.LE</td>
<td>C.NGT</td>
<td></td>
</tr>
</tbody>
</table>