# *CPU Instruction Set Details*

# *A*

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.

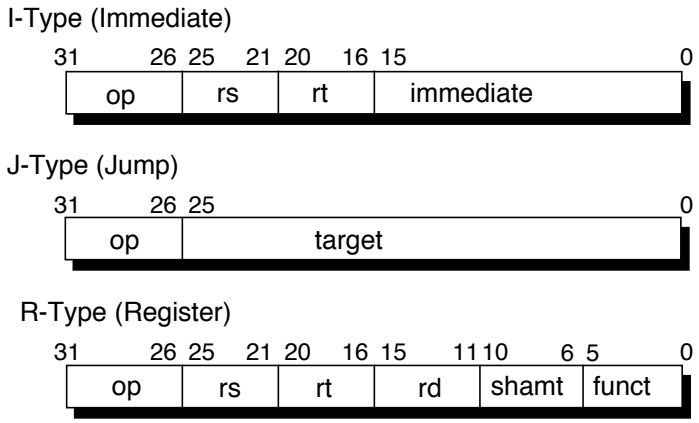
## A.1 Instruction Classes

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 offse*t.
- **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.




| | |
|---|---|
| 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.

*Table A-1 CPU Instruction Operation Notations*

| Symbol | Meaning |
|---|---|
| ← | Assignment. |
| \|\| | Bit string concatenation. |
| $x^y$ | Replication of bit value *x* into a *y*-bit string. Note: *x* is always a single-bit value. |
| $x_{y:z}$ | 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. |
| + | 2's complement or floating-point addition. |
| - | 2's complement or floating-point subtraction. |
| * | 2's complement or floating-point multiplication. |
| div | 2's complement integer division. |
| mod | 2's complement modulo. |
| / | Floating-point division. |
| < | 2's complement less than comparison. |
| and | Bit-wise logical AND. |
| or | Bit-wise logical OR. |
| xor | Bit-wise logical XOR. |
| nor | Bit-wise logical NOR. |
| GPR[*x*] | General-Register x. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect. |
| CPR[*z,x*] | Coprocessor unit *z*, general register *x*. |
| CCR[*z,x*] | Coprocessor unit *z*, control register *x*. |
| COC[*z*] | Coprocessor unit *z* condition signal. |
| BigEndianMem | 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. |
| ReverseEndian | 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 ($SR_{25}$ and User mode). |
| BigEndianCPU | The endianness for load and store instructions (0 → Little, 1 → Big). In User mode, this endianness may be reversed by setting $SR_{25}$. Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian. |
| LLbit | Bit of state to specify synchronization instructions. Set by *LL*, cleared by *ERET* and *Invalidate* and read by *SC*. |
| T+*i*: | 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. |

## Instruction Notation Examples

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

Example #1:

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

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:

$$(immediate_{15})^{16} \text{ || } immediate_{15...0}$$

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*

| Function | Meaning |
|---|---|
| AddressTranslation | 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. |
| LoadMemory | 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. |
| StoreMemory | 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. |

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*

| Access Type Mnemonic | Value | Meaning |
|---|---|---|
| DOUBLEWORD | 7 | 8 bytes (64 bits) |
| SEPTIBYTE | 6 | 7 bytes (56 bits) |
| SEXTIBYTE | 5 | 6 bytes (48 bits) |
| QUINTIBYTE | 4 | 5 bytes (40 bits) |
| WORD | 3 | 4 bytes (32 bits) |
| TRIPLEBYTE | 2 | 3 bytes (24 bits) |
| HALFWORD | 1 | 2 bytes (16 bits) |
| BYTE | 0 | 1 byte (8 bits) |

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.

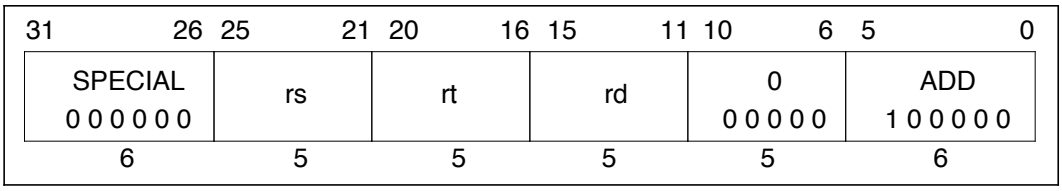
- The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor.
- The second space, *coprocessor control* registers, may only have their contents transferred directly between the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

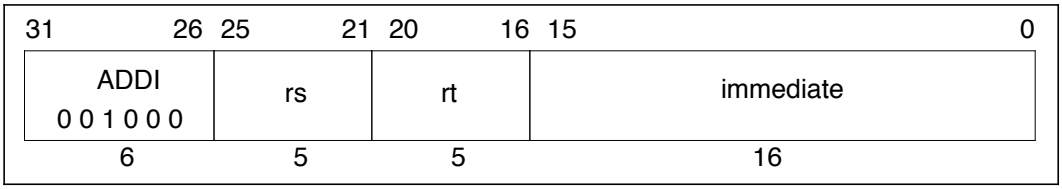
```
32    T:    GPR[rd] ←GPR[rs] + GPR[rt]

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

**Exceptions:**

Integer overflow exception

# ADDI Add Immediate ADDI




**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:**

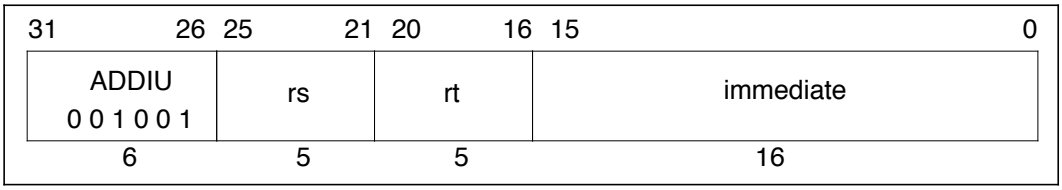
```
32    T:    GPR [rt] ← GPR[rs] +(immediate15)16 || immediate15...0

64    T:    temp ← GPR[rs] + (immediate15)48 || immediate15...0
            GPR[rt] ← (temp31)32 || temp31...0
```

**Exceptions:**

Integer overflow exception

# ADDIU Add Immediate Unsigned ADDIU




**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:**

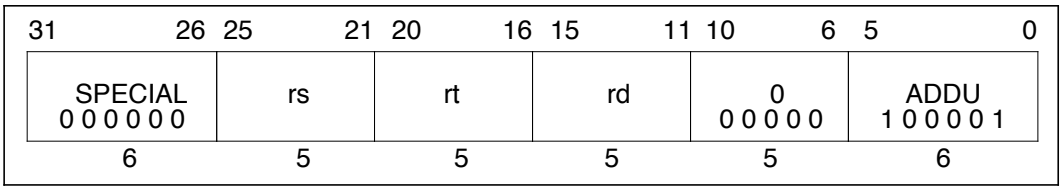
```
32    T:    GPR [rt] ← GPR[rs] + (immediate15)16 || immediate15...0

64    T:    temp ← GPR[rs] + (immediate15)48 || immediate15...0
            GPR[rt] ← (temp31)32 || temp31...0
```

**Exceptions:**

None

# ADDU Add Unsigned ADDU




**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:**

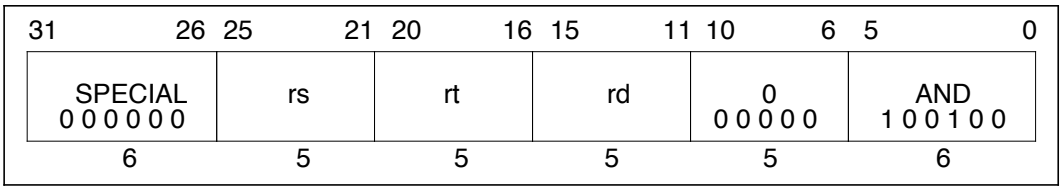
```
32    T:    GPR[rd] ←GPR[rs] + GPR[rt]

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

**Exceptions:**

None

# AND And AND




**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:**

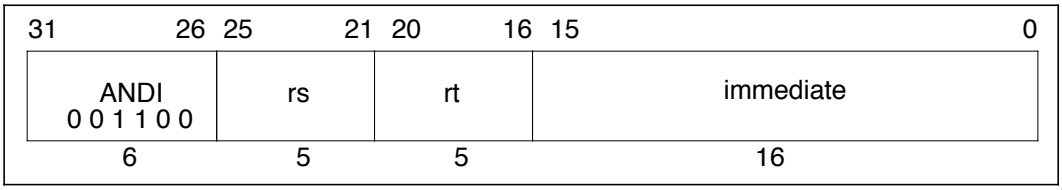
```
32    T:    GPR[rd] ← GPR[rs] and GPR[rt]

64    T:    GPR[rd] ← GPR[rs] and GPR[rt]
```

**Exceptions:**

None

# ANDI And Immediate ANDI




**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:**

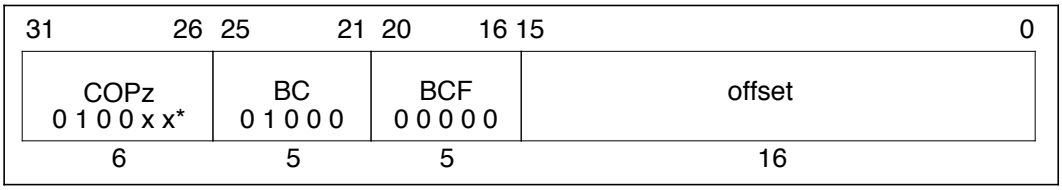
32 T: $GPR[rt] \leftarrow 0^{16} \,||\, (immediate \text{ and } GPR[rs]_{15...0})$

64 T: $GPR[rt] \leftarrow 0^{48} \,||\, (immediate \text{ and } GPR[rs]_{15...0})$

**Exceptions:**

None

# BCzF Branch On Coprocessor z False BCzF




**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 ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           endif

64    T–1: condition ← not COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           endif
```

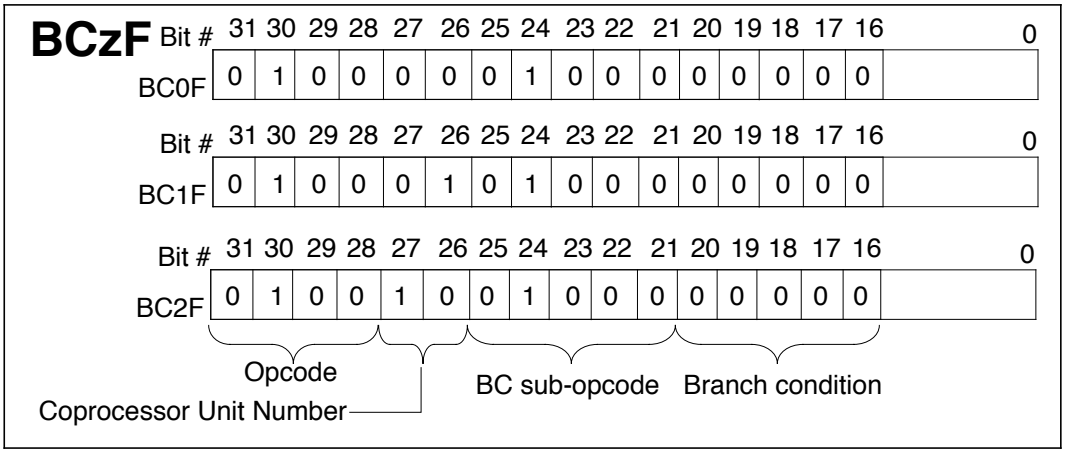
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# BCzF Branch On Coprocessor z False (continued) BCzF

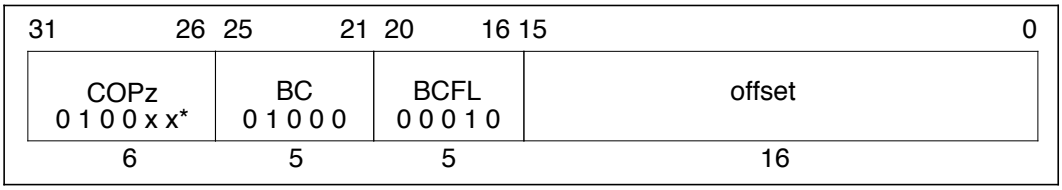
**Exceptions:**

Coprocessor unusable exception

**Opcode Bit Encoding:**



# BCzFL — Branch On Coprocessor z False Likely — BCzFL




**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.

# BCzFL — Branch On Coprocessor z False Likely (continued) — BCzFL

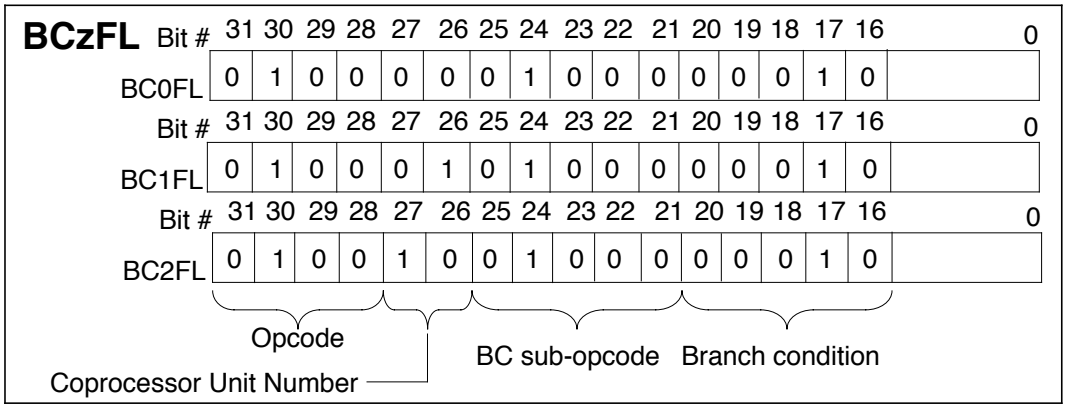
**Operation:**

```
32    T–1: condition ← not  COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           else
                        NullifyCurrentInstruction
           endif
64    T–1: condition ← not  COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           else
                        NullifyCurrentInstruction
           endif
```

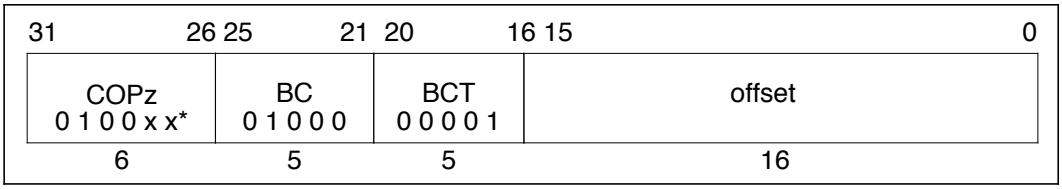
**Exceptions:**

Coprocessor unusable exception

**Opcode Bit Encoding:**



# BCzT Branch On Coprocessor z True BCzT




**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:**

```
32    T–1: condition ← COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           endif
64    T–1: condition ← COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           endif
```

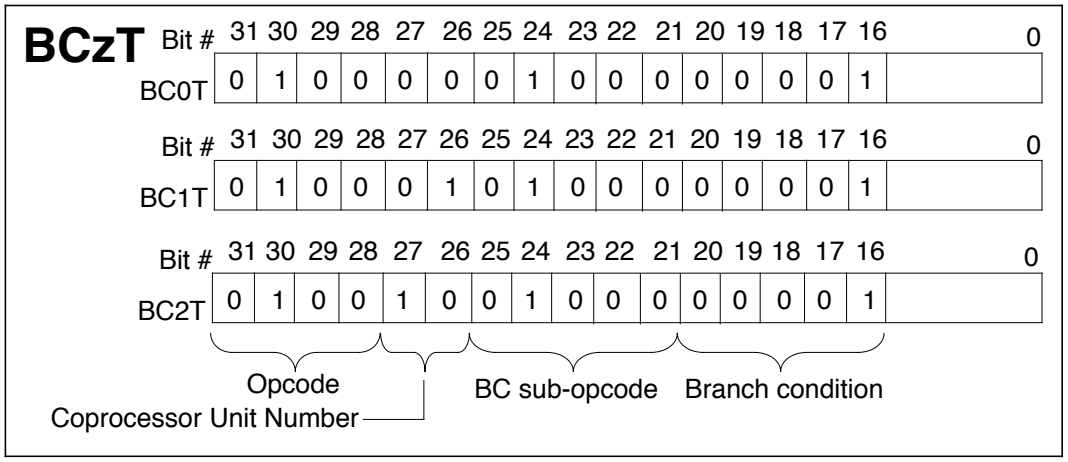
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# BCzT Branch On Coprocessor z True (continued) BCzT

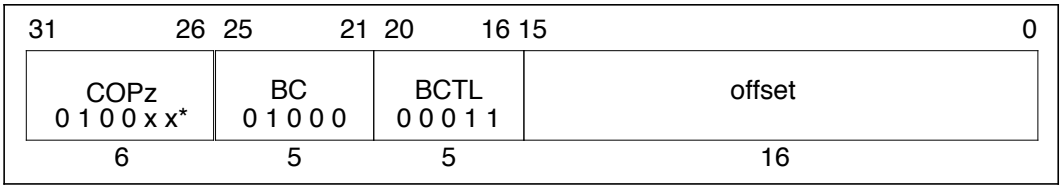
**Exceptions:**

Coprocessor unusable exception

**Opcode Bit Encoding:**



# BCzTL Branch On Coprocessor z True Likely BCzTL




**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 || 0^2
      T+1: if condition then
                 PC ← PC + target
           else
                 NullifyCurrentInstruction
           endif
64    T–1: condition ← COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                 PC ← PC + target
           else
                 NullifyCurrentInstruction
           endif
```

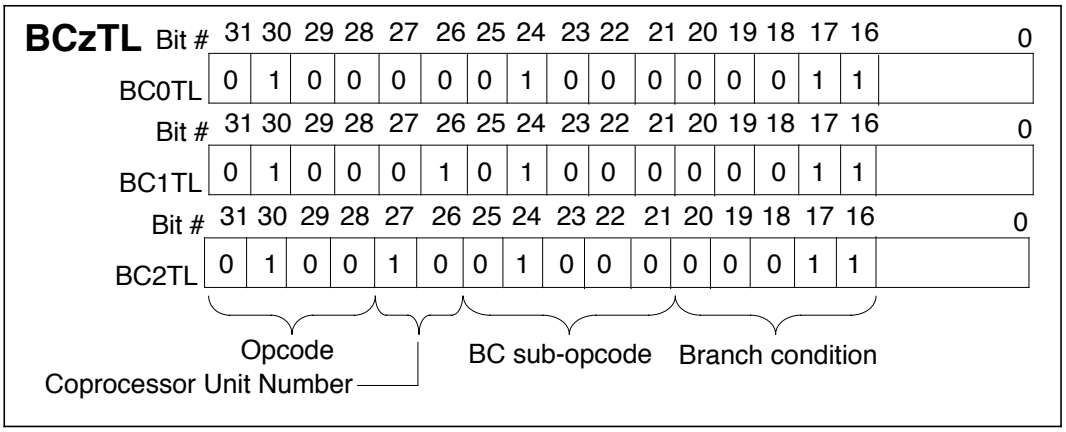
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# BCzTL — Branch On Coprocessor z True Likely (continued) — BCzTL

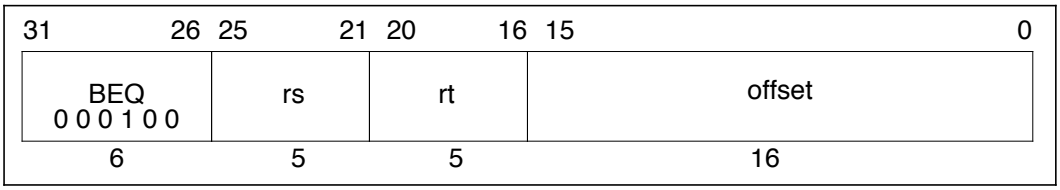
**Exceptions:**

Coprocessor unusable exception

**Opcode Bit Encoding:**



# BEQ Branch On Equal BEQ




**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.

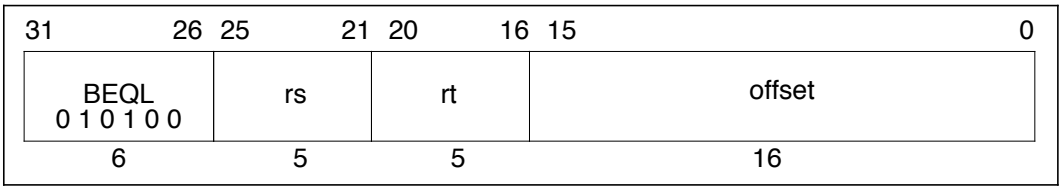
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            endif
```

**Exceptions:**

None

# BEQL Branch On Equal Likely BEQL




**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.

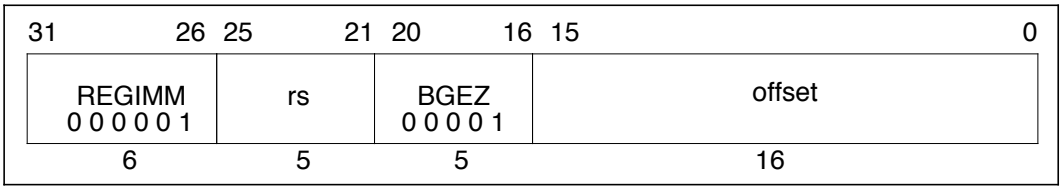
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                                  PC ← PC + target
            else
                              NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                              PC ← PC + target
            else
                              NullifyCurrentInstruction
            endif
```

**Exceptions:**

None

# BGEZ — Branch On Greater Than Or Equal To Zero — BGEZ




**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.

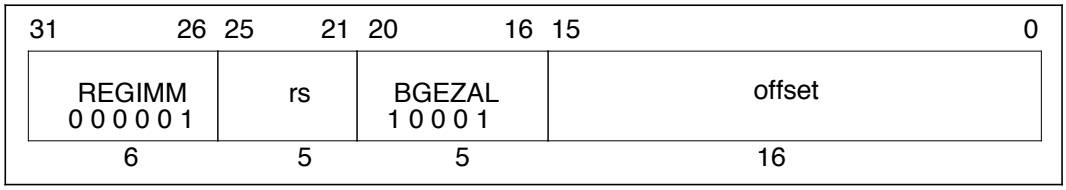
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0)
      T+1:  if condition then
                        PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            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 BGEZAL




**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.

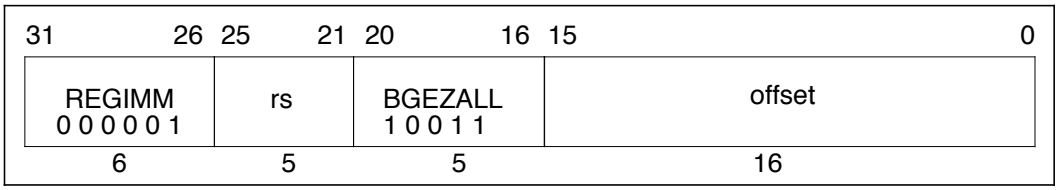
**Operation:**

```
32    T:    target ← (offset15)^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 ← (offset15)^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 BGEZALL




**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.

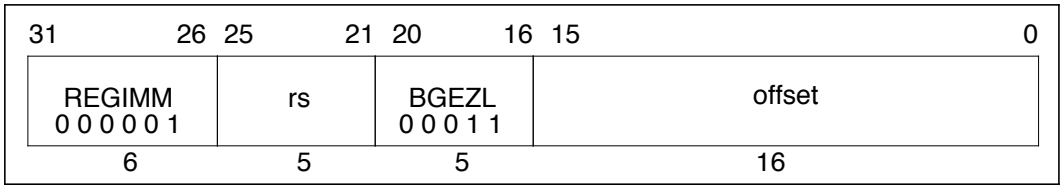
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            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 || 0^2
            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 — BGEZL




**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.

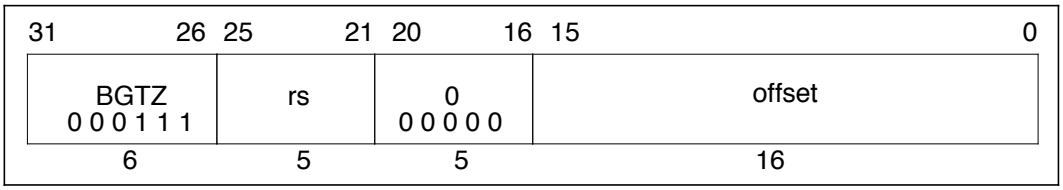
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

**Exceptions:**

None

# BGTZ Branch On Greater Than Zero BGTZ




**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.

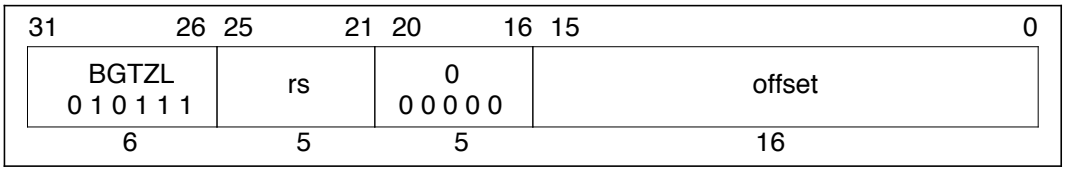
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 0^64)
      T+1:  if condition then
                  PC ← PC + target
            endif
```

**Exceptions:**

None

# BGTZL — Branch On Greater Than Zero Likely — BGTZL




**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.

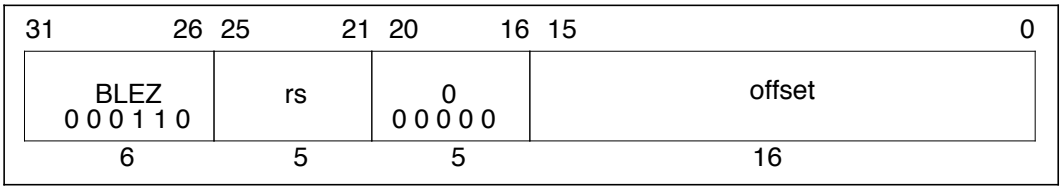
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 0^64)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

**Exceptions:**

None

# BLEZ — Branch on Less Than Or Equal To Zero — BLEZ




**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.

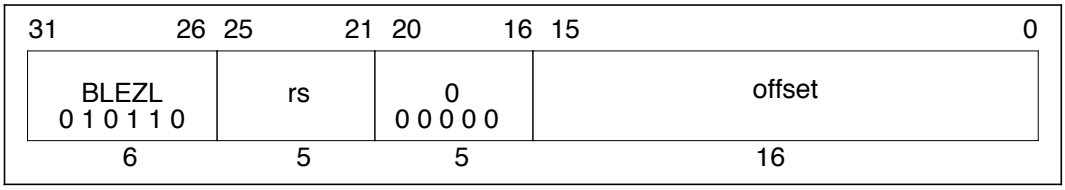
**Operation:**

```
32      T:    target ← (offset15)^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 ← (offset15)^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 BLEZL




**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.

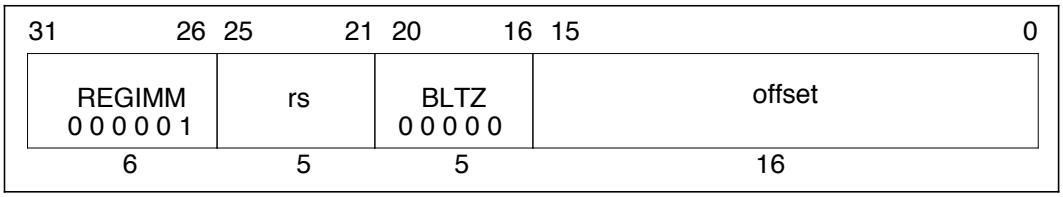
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1) or (GPR[rs] = 0^32)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1) or (GPR[rs] = 0^64)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

**Exceptions:**

None

# BLTZ Branch On Less Than Zero BLTZ




**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.

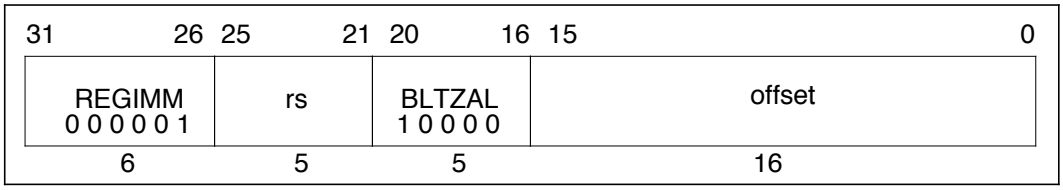
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^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 BLTZAL




**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.

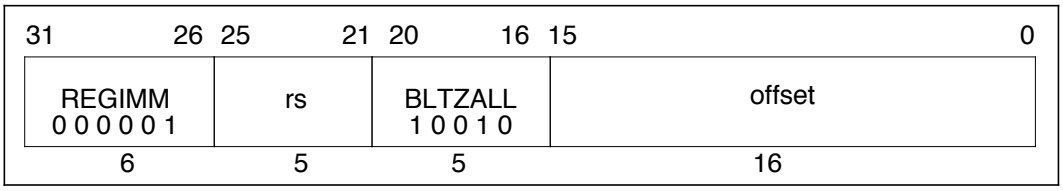
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 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 BLTZALL




**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.

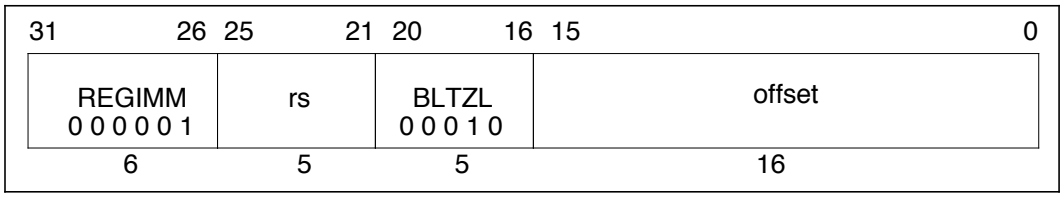
**Operation:**

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            else NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            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 BLTZL




**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.

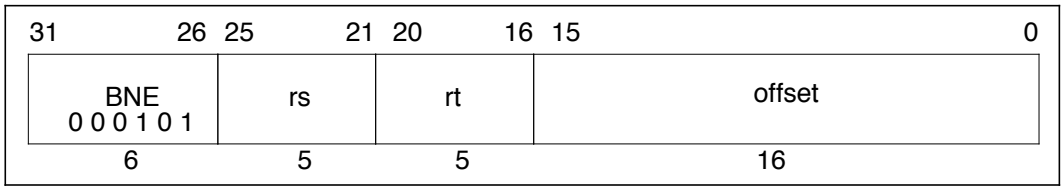
**Operation:**

```
32    T:    target ← (offset15)14 || offset || 02
            condition ← (GPR[rs]31 = 1)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)46 || offset || 02
            condition ← (GPR[rs]63 = 1)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

**Exceptions:**

None

# BNE Branch On Not Equal BNE




**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.

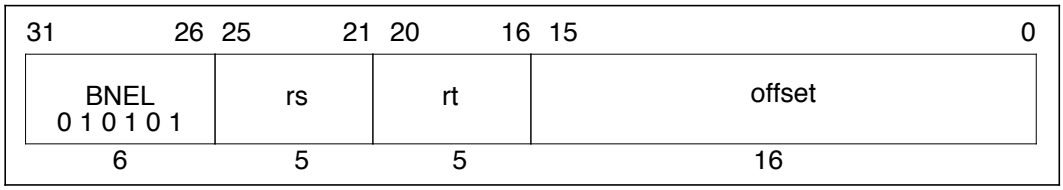
**Operation:**

```
32    T:     target ← (offset15)^14 || offset || 0^2
             condition ← (GPR[rs] ≠ GPR[rt])
      T+1:   if condition then
                    PC ← PC + target
             endif
64    T:     target ← (offset15)^46 || offset || 0^2
             condition ← (GPR[rs] ≠ GPR[rt])
      T+1:   if condition then
                    PC ← PC + target
             endif
```

**Exceptions:**

None

# BNEL Branch On Not Equal Likely BNEL




**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.

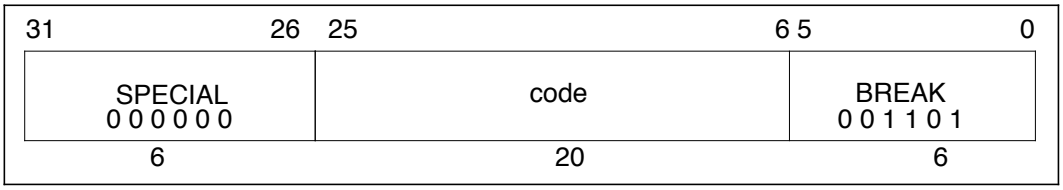
**Operation:**

```
32      T:    target ← (offset15)^14 || offset || 0^2
              condition ← (GPR[rs] ≠ GPR[rt])
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
64      T:    target ← (offset15)^46 || offset || 0^2
              condition ← (GPR[rs] ≠ GPR[rt])
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
```

**Exceptions:**

None

# BREAK Breakpoint BREAK




**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.

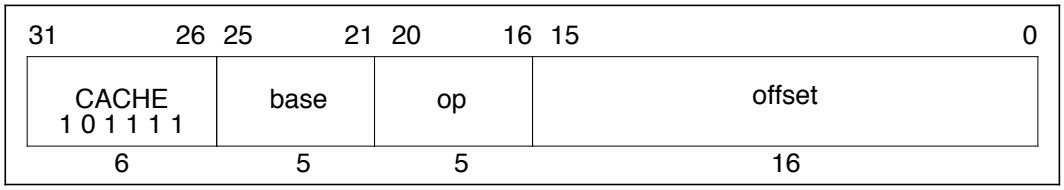
**Operation:**

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

**Exceptions:**

Breakpoint exception

# CACHE Cache CACHE




**Format:**

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 $2^{CACHEBITS}$ bytes with $2^{LINEBITS}$ bytes per tag, $vAddr_{CACHEBITS\ ...\ LINEBITS}$ specifies the block.

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

Index Load Tag also uses $vAddr_{LINEBITS...\ 3}$ 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 $vAddr_{LINEBITS\ ...\ 3}$ 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.

# CACHE Cache (continued) CACHE

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:

| Code | Name | Cache |
|---|---|---|
| 0 | I | primary instruction |
| 1 | D | primary data |
| 2 | SI | secondary instruction |
| 3 | SD | secondary data (or combined instruction/data) |

# CACHE Cache (continued) CACHE

Bits 20...18 (this value is listed under the **Code** column) of the instruction specify the operation as follows:

| Code | Caches | Name | Operation |
|---|---|---|---|
| 0 | I, SI | Index Invalidate | Set the cache state of the cache block to Invalid. |
| 0 | D | Index Writeback Invalidate | 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). |
| O | SD | Index Writeback Invalidate | 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. |
| 1 | All | Index Load Tag | Read the tag for the cache block at the specified index and place it iinto the *TagLo* and *TagHi* CP0 registers, ignoring any ECC or parity errors. Also load the data ECC or parity bits into the ECC register. |
| 2 | All | Index Store Tag | 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. |

# CACHE Cache (continued) CACHE

| Code | Caches | Name | Operation |
|---|---|---|---|
| 3 | SD | Create Dirty Exclusive | 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 *PIdx* 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 Dirty 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. |
| 3 | D | Create Dirty Exclusive | 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 Dirty Exclusive. |
| 4 | I,D | Hit Invalidate | If the cache block contains the specified address, mark the cache block invalid. |
| 4 | SI, SD | Hit Invalidate | 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 *PIdx* 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. |
| 5 | D | Hit Writeback Invalidate | 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 *ECC* register is XOR'd into the computed check bits during the write to the secondary cache for the addressed doubleword. |

# CACHE Cache (continued) CACHE

| Code | Caches | Name | Operation |
|---|---|---|---|
| 5 | SD | Hit Writeback Invalidate | 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. |
| 5 | I | Fill | 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. |
| 6 | D | Hit Writeback | 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. |
| 6 | SD | Hit Writeback | 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. |
| 6 | I | Hit Writeback | 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. |

# CACHE — Cache (continued) — CACHE

| Code | Caches | Name | Operation |
|---|---|---|---|
| 7 | SI,SD | Hit Set Virtual | 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. |

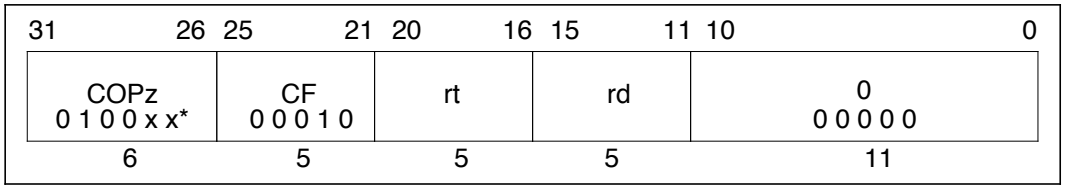
**Operation:**

```
32, 64    T:    vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                CacheOp (op, vAddr, pAddr)
```

**Exceptions:**

Coprocessor unusable exception

# CFCz — Move Control From Coprocessor — CFCz




**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.

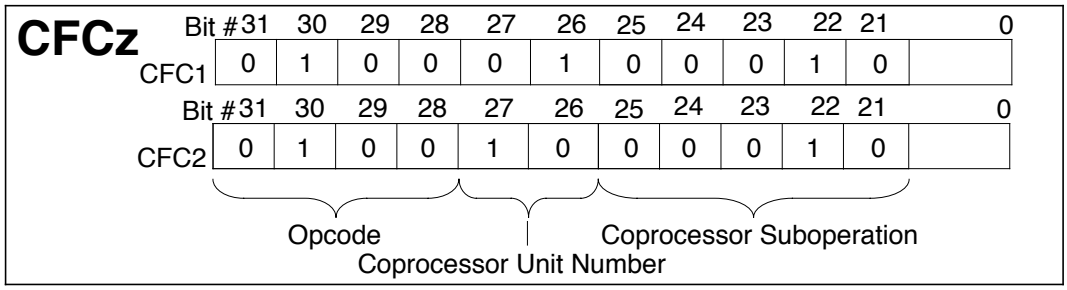
**Operation:**

```
32      T:    data ← CCR[z,rd]
        T+1:  GPR[rt] ← data
64      T:    data ← (CCR[z,rd]31)32 || CCR[z,rd]
        T+1:  GPR[rt] ← data
```

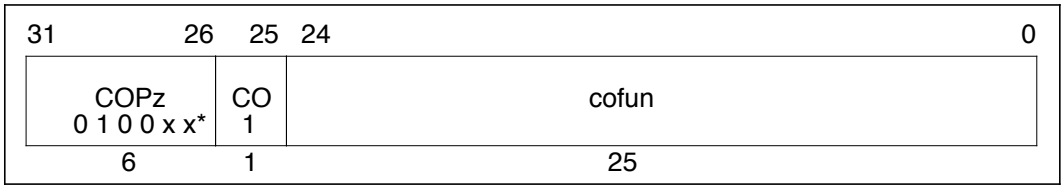
**Exceptions:**

Coprocessor unusable exception

***Opcode Bit Encoding:**



# COPz Coprocessor Operation COPz




**Format:**

COPz cofun

**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.

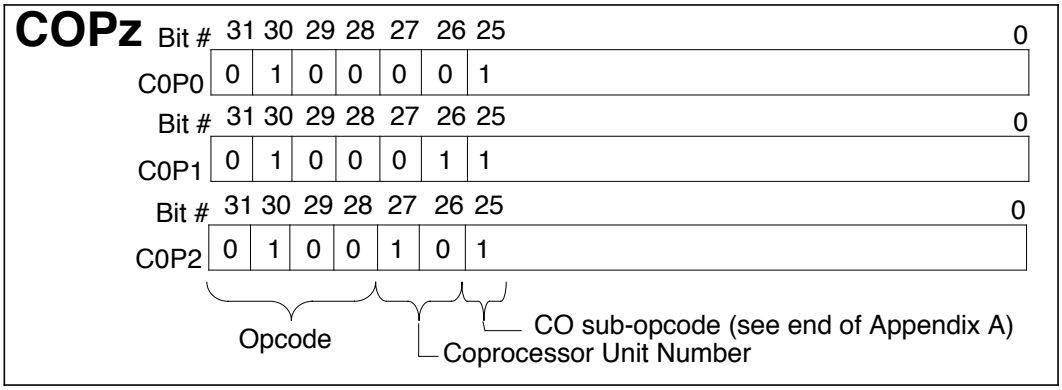
**Operation:**

```
32, 64    T:    CoprocessorOperation (z, cofun)
```

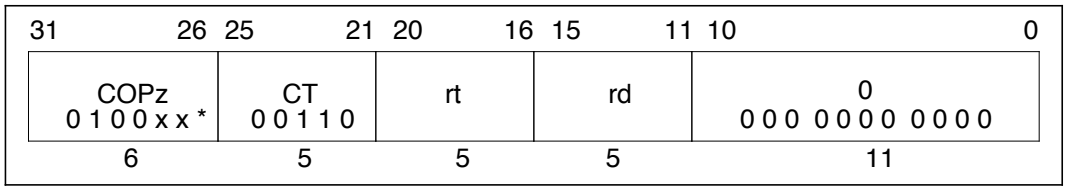
**Exceptions:**

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

***Opcode Bit Encoding:**



# CTCz Move Control to Coprocessor CTCz




**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:**

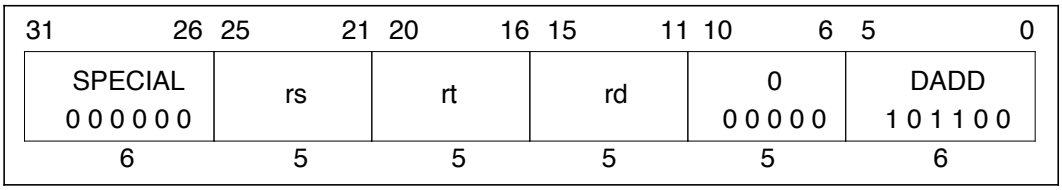
```
32,64    T:      data ← GPR[rt]
         T + 1:  CCR[z,rd] ← data
```

**Exceptions:**

Coprocessor unusable

*See "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# DADD Doubleword Add DADD




**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.

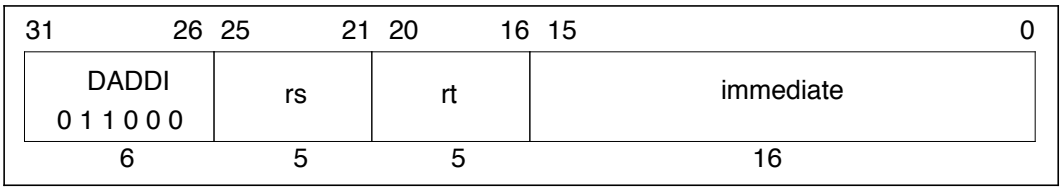
**Operation:**

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

**Exceptions:**

Integer overflow exception
Reserved instruction exception (R4000 in 32-bit mode)

# DADDI Doubleword Add Immediate DADDI




**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.

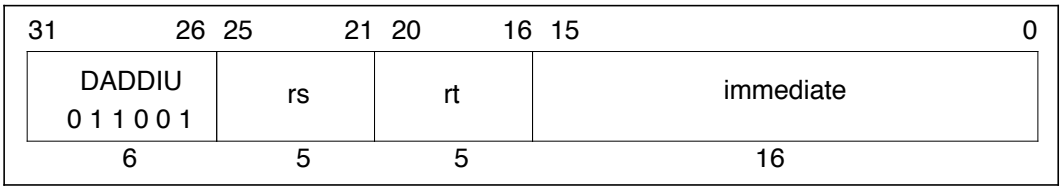
**Operation:**

```
64    T:    GPR [rt] ← GPR[rs] + (immediate15)48 || immediate15...0
```

**Exceptions:**

Integer overflow exception
Reserved instruction exception (R4000 in 32-bit mode)

# DADDIU Doubleword Add Immediate Unsigned DADDIU




**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.

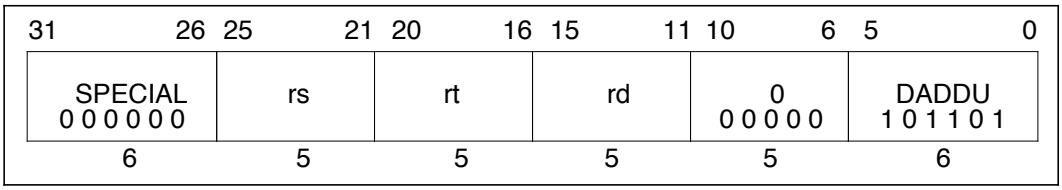
**Operation:**

```
64    T:    GPR [rt] ← GPR[rs] + (immediate15)48 || immediate15...0
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DADDU Doubleword Add Unsigned DADDU




**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.

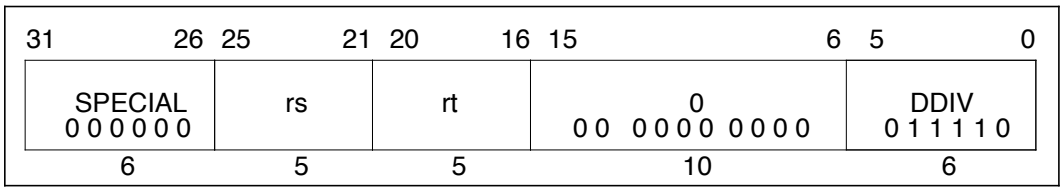
**Operation:**

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

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DDIV Doubleword Divide DDIV




**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.

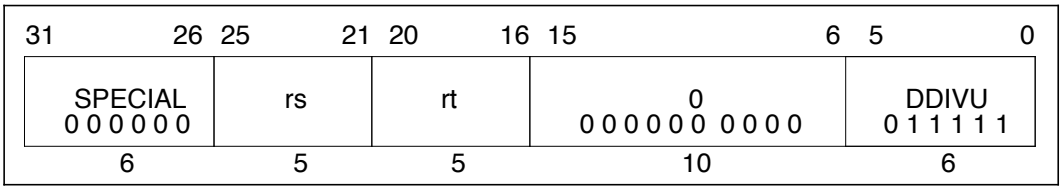
**Operation:**

```
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 DDIVU




**Format:**

DDIVU rs, rt

**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.

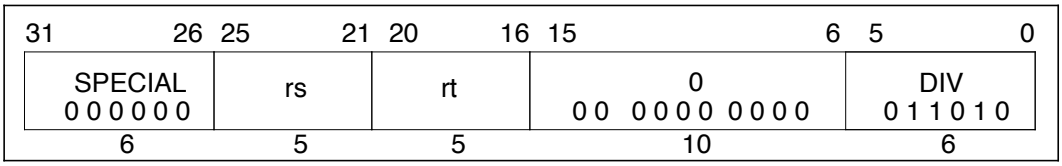
**Operation:**

```
64   T–2:   LO   ← undefined
            HI   ← undefined
       T–1:   LO   ← undefined
            HI   ← undefined
       T:     LO   ← (0 || GPR[rs]) div (0 || GPR[rt])
            HI   ← (0 || GPR[rs]) mod  (0 || GPR[rt])
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DIV — Divide — 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) — DIV

**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   ← (q31)32 || q31...0
            HI   ← (r31)32 || r31...0
```

**Exceptions:**

None

# DIVU Divide Unsigned DIVU


**Format:**

DIVU rs, rt

**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.

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 (continued) — DIVU

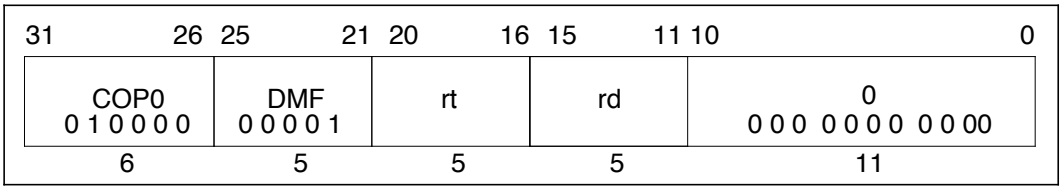
**Operation:**

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

**Exceptions:**

None

# DMFC0 Doubleword Move From System Control Coprocessor DMFC0




**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.

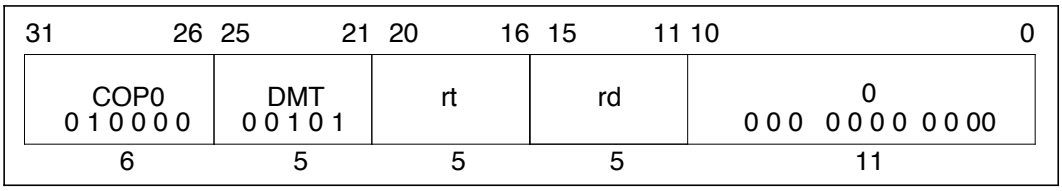
**Operation:**

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

**Exceptions:**

Coprocessor unusable exception
Reserved instruction exception (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)

# DMTC0 Doubleword Move To System Control Coprocessor DMTC0




**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.

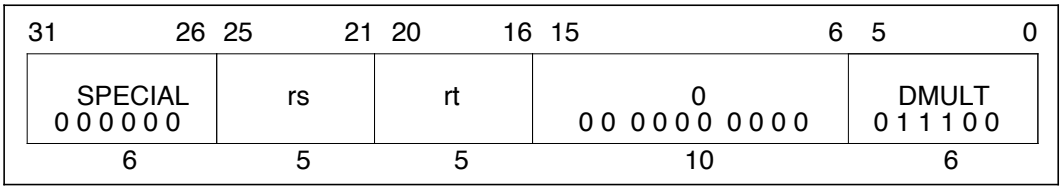
**Operation:**

```
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)

# DMULT Doubleword Multiply DMULT




**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.

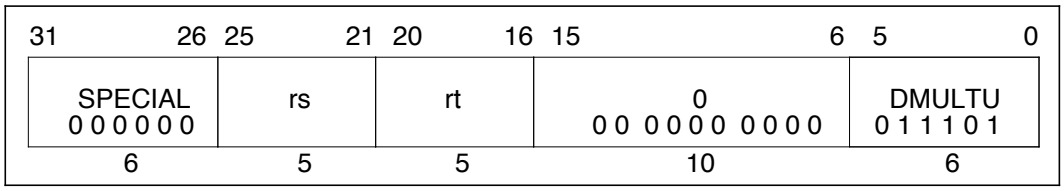
**Operation:**

```
64   T–2: LO   ← undefined
          HI   ← undefined
     T–1: LO   ← undefined
          HI   ← undefined
     T:   t    ← GPR[rs] * GPR[rt]
          LO   ← t63...0
          HI   ← t127...64
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DMULTU Doubleword Multiply Unsigned DMULTU




**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.

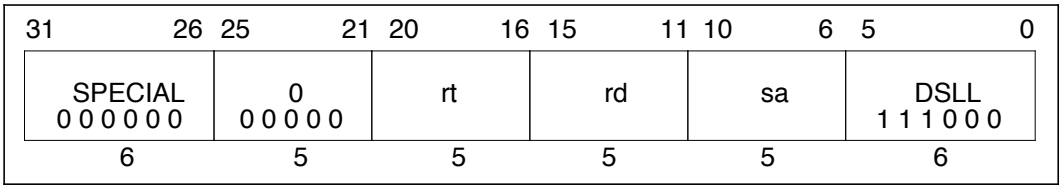
**Operation:**

```
64    T–2:  LO ← undefined
            HI ← undefined
      T–1:  LO ← undefined
            HI ← undefined
      T:    t ← (0 || GPR[rs]) * (0 || GPR[rt])
            LO ← t63...0
            HI ←t127...64
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSLL Doubleword Shift Left Logical DSLL




**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*.

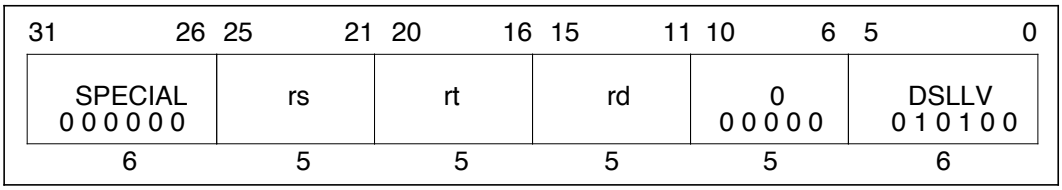
**Operation:**

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

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSLLV — Doubleword Shift Left Logical Variable — DSLLV




**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.

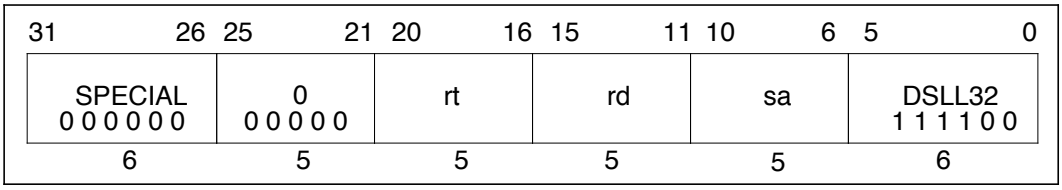
**Operation:**

```
64          T:   s ← GPR[rs]5...0
                 GPR[rd]← GPR[rt](63–s)...0 || 0^s
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSLL32 Doubleword Shift Left Logical + 32 DSLL32




**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.

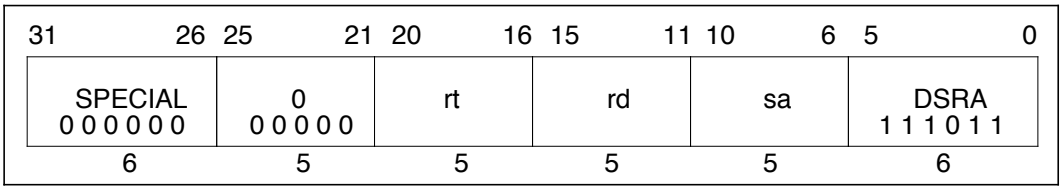
**Operation:**

```
64          T:   s ← 1 || sa
                 GPR[rd]← GPR[rt](63–s)...0 || 0^s
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSRA — Doubleword Shift Right Arithmetic — DSRA




**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.

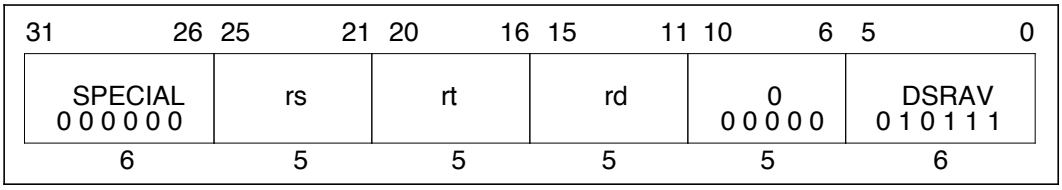
**Operation:**

```
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 — DSRAV




**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.

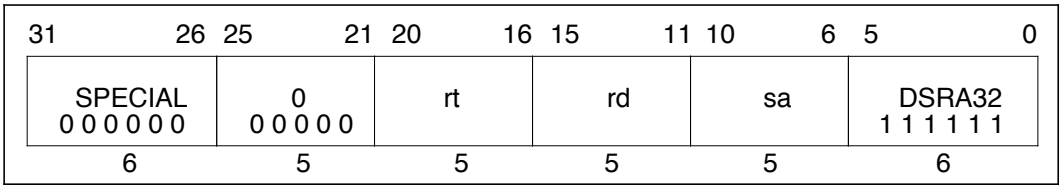
**Operation:**

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

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSRA32 Doubleword Shift Right Arithmetic + 32 DSRA32




**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.

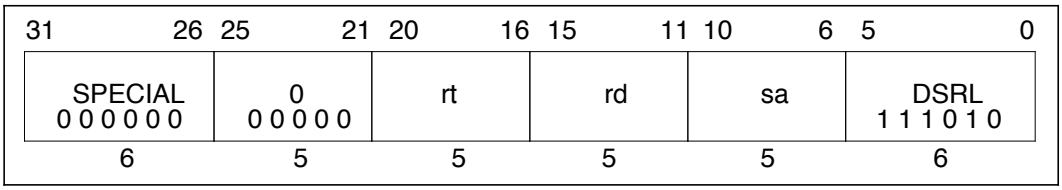
**Operation:**

```
64    T:    s ←1 || sa
            GPR[rd] ← (GPR[rt]63)^s || GPR[rt] 63...s
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSRL — Doubleword Shift Right Logical — DSRL




**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.

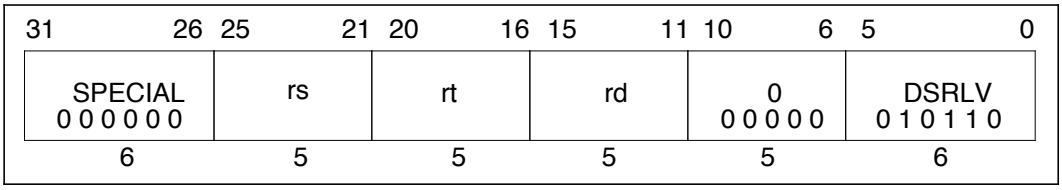
**Operation:**

```
64    T:    s ← 0 || sa
            GPR[rd] ← 0^s || GPR[rt]_63...s
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSRLV — Doubleword Shift Right Logical Variable — DSRLV




**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.

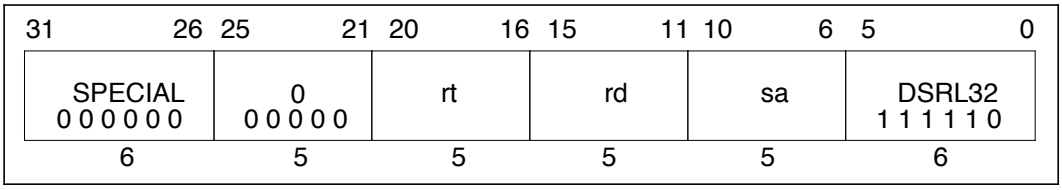
**Operation:**

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

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# DSRL32 Doubleword Shift Right Logical + 32 DSRL32




**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.

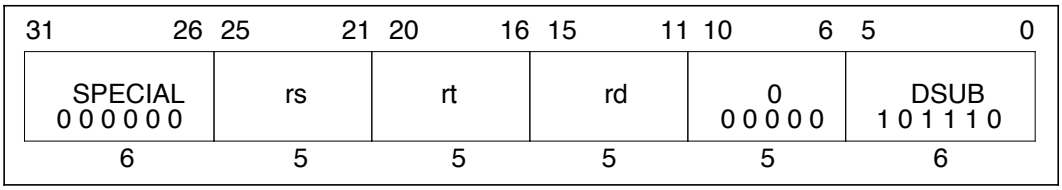
**Operation:**

```
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 DSUB




**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.

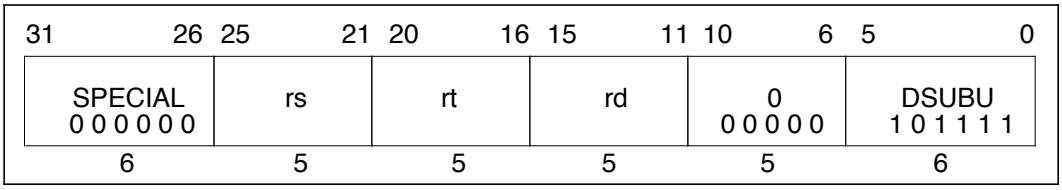
**Operation:**

```
64          T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

**Exceptions:**

Integer overflow exception
Reserved instruction exception (R4000 in 32-bit mode)

# DSUBU Doubleword Subtract Unsigned DSUBU




**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.

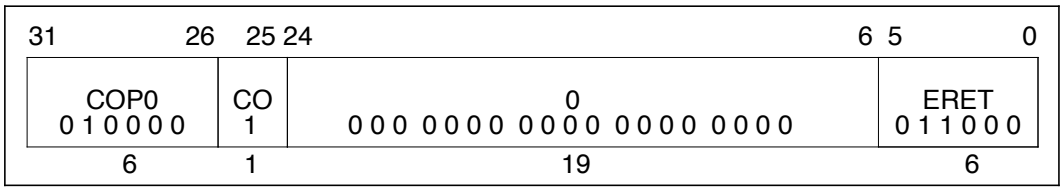
**Operation:**

```
64          T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

**Exceptions:**

Reserved instruction exception (R4000 in 32-bit mode)

# ERET Exception Return ERET




**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 ($SR_2 = 1$), then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register ($SR_2$). Otherwise ($SR_2 = 0$), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register ($SR_1$).

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

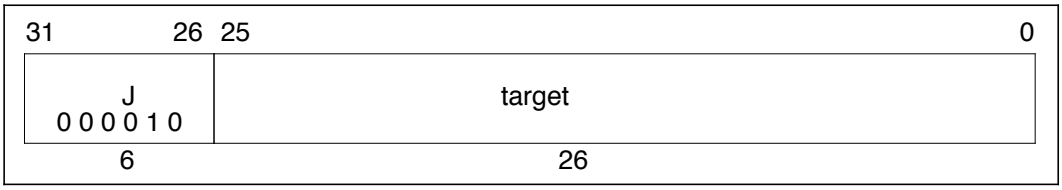
**Operation:**

```
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 — J




**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:**

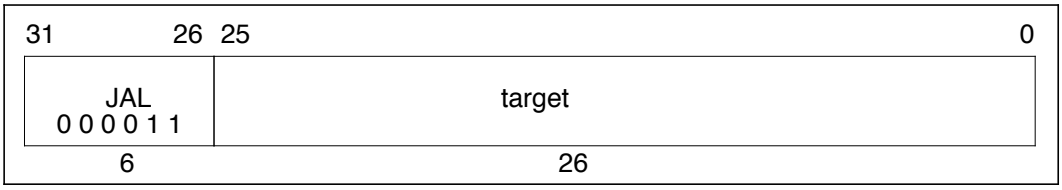
```
32      T:    temp ← target
        T+1:  PC ← PC31...28 || temp || 0^2

64      T:    temp ← target
        T+1:  PC ← PC63...28 || temp || 0^2
```

**Exceptions:**

None

# JAL — Jump And Link — JAL




**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*.

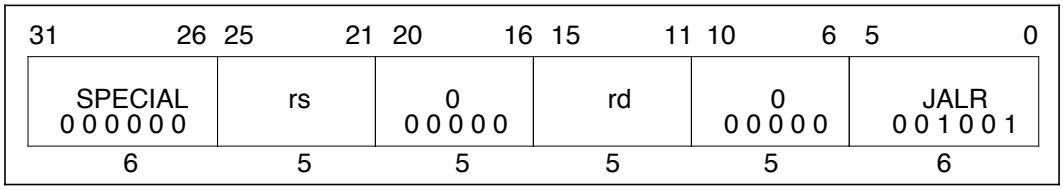
**Operation:**

```
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 JALR




**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.

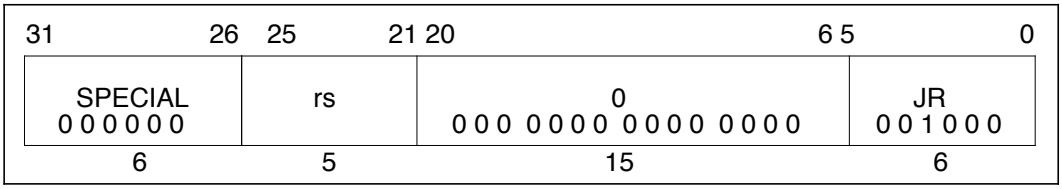
**Operation:**

```
32, 64      T:      temp ← GPR [rs]
                    GPR[rd] ← PC + 8
            T+1:    PC ← temp
```

**Exceptions:**

None

# JR — Jump Register — JR




**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.

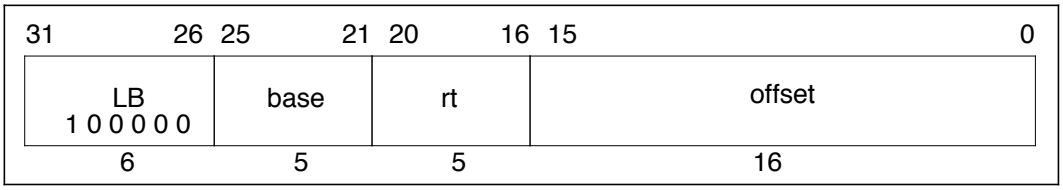
**Operation:**

```
32, 64      T:      temp ← GPR[rs]
            T+1:    PC ← temp
```

**Exceptions:**

None

# LB Load Byte LB




**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:**

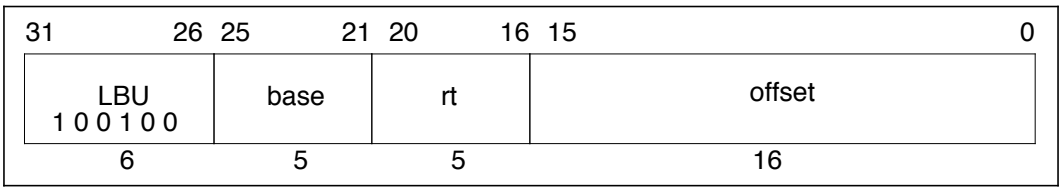
```
32    T:    vAddr ← ((offset15)16 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1 ... 3 || (pAddr2...0 xor ReverseEndian3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU3
            GPR[rt] ← (mem7+8*byte)24 || mem7+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 ReverseEndian3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU3
            GPR[rt] ← (mem7+8*byte)56 || mem7+8*byte...8*byte
```

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LBU Load Byte Unsigned LBU




**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:**

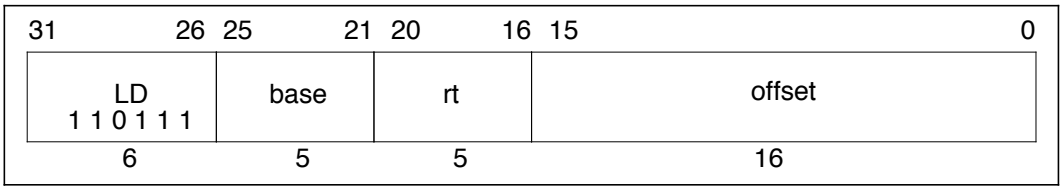
```
32    T:    vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1 ...3 || (pAddr2...0 xor ReverseEndian^3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU^3
            GPR[rt] ← 0^24 || mem7+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^3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU^3
            GPR[rt] ← 0^56 || mem7+8* byte...8* byte
```

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LD Load Doubleword LD




**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.

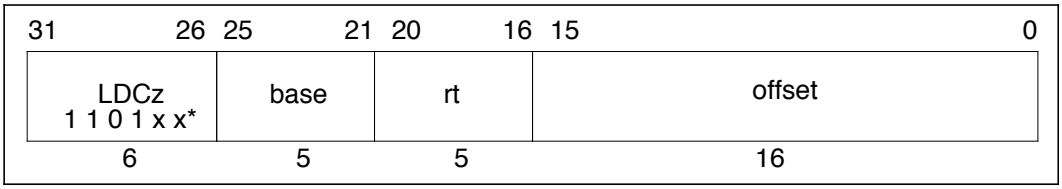
**Operation:**

```
64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          GPR[rt] ← mem
```

**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 LDCz




**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) LDCz

**Operation:**

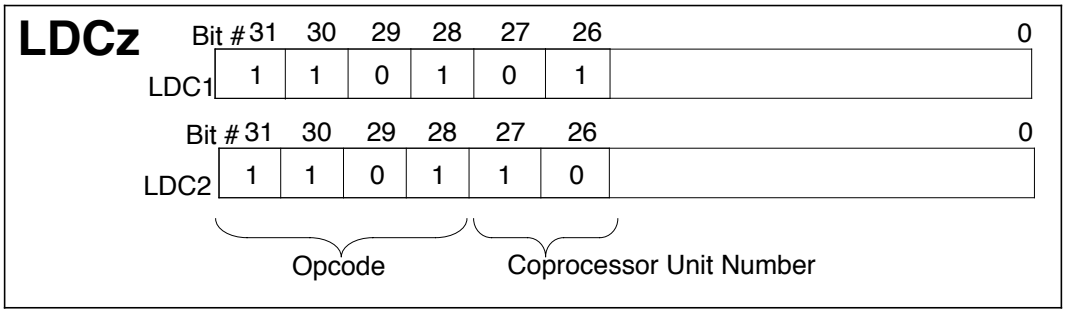
```
32    T:   vAddr ← ((offset15)16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)

64    T:   vAddr ← ((offset15)48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)
```

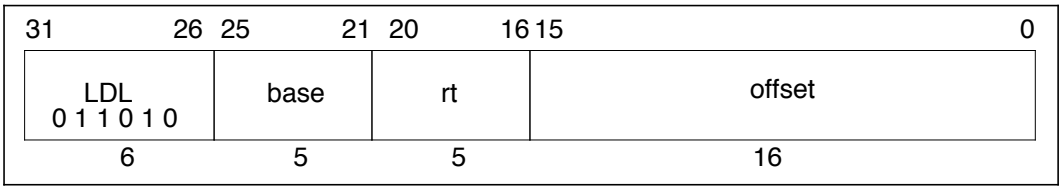
**Exceptions:**

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

**Opcode Bit Encoding:**



# LDL Load Doubleword Left LDL




**Format:**

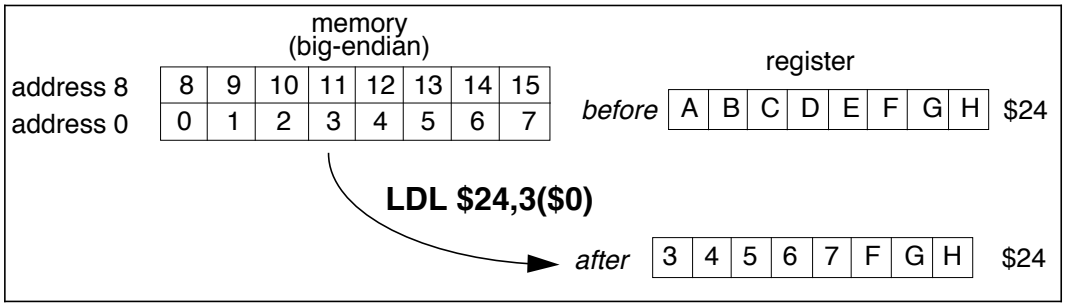
LDL rt, offset(base)

**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.



# LDL — Load Doubleword Left (continued) — LDL

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.

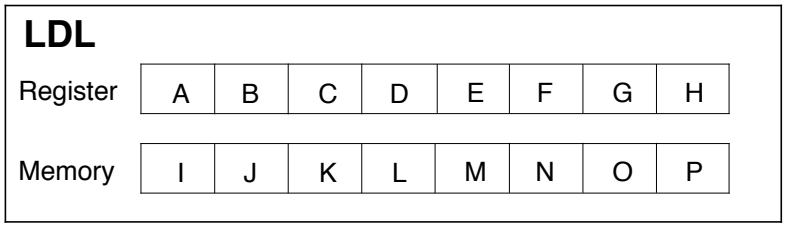
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

```
64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
          if BigEndianMem = 0 then
                    pAddr ← pAddrPSIZE–1...3 || 0^3
          endif
          byte ← vAddr2...0 xor BigEndianCPU^3
          mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
          GPR[rt] ← mem7+8*byte...0 || GPR[rt]55–8*byte...0
```

# LDL — Load Doubleword Left (continued) — LDL

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



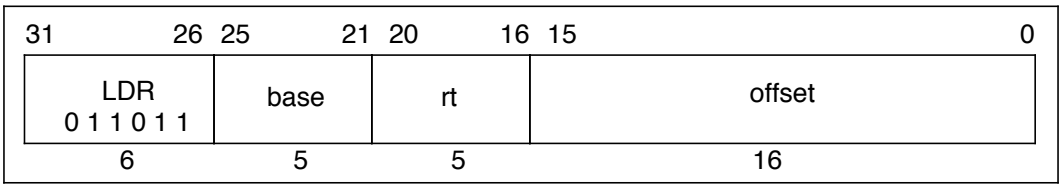

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | P B C D E F G H | 0 | 0 | 7 | I J K L M N O P | 7 | 0 | 0 |
| 1 | O P C D E F G H | 1 | 0 | 6 | J K L M N O P H | 6 | 0 | 1 |
| 2 | N O P D E F G H | 2 | 0 | 5 | K L M N O P G H | 5 | 0 | 2 |
| 3 | M N O P E F G P | 3 | 0 | 4 | L M N O P F G H | 4 | 0 | 3 |
| 4 | L M N O P F G H | 4 | 0 | 3 | M N O P E F G H | 3 | 0 | 4 |
| 5 | K L M N O P G H | 5 | 0 | 2 | N O P D E F G H | 2 | 0 | 5 |
| 6 | J K L M N O P H | 6 | 0 | 1 | O P C D E F G H | 1 | 0 | 6 |
| 7 | I J K L M N O P | 7 | 0 | 0 | P B C D E F G H | 0 | 0 | 7 |

*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
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

# LDR Load Doubleword Right LDR




**Format:**

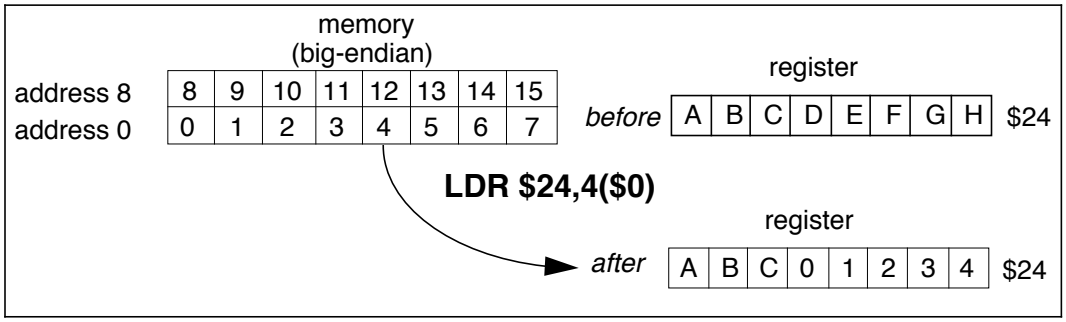
LDR rt, offset(base)

**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.



# LDR Load Doubleword Right (continued) LDR

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.

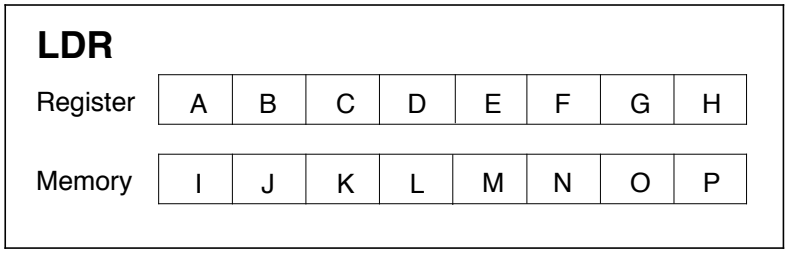
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

```
64    T:    vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE–1...3 || (pAddr2...0  xor  ReverseEndian^3)
            if BigEndianMem = 1 then
                  pAddr ←  pAddr31...3 || 0^3
            endif
            byte ← vAddr2...0 xor BigEndianCPU^3
            mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
            GPR[rt] ← GPR[rt]63...64-8*byte || mem63...8*byte
```

# LDR — Load Doubleword Right (continued) — LDR

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



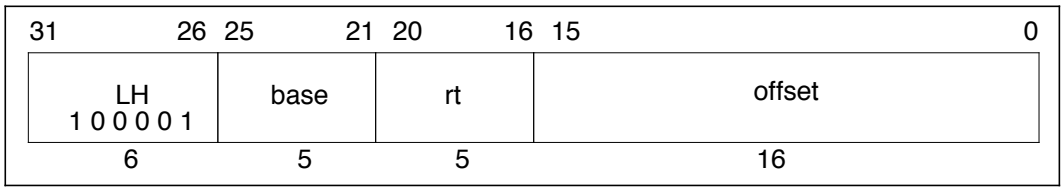

| vAddr$_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | I J K L M N O P | 7 | 0 | 0 | A B C D E F G I | 0 | 7 | 0 |
| 1 | A I J K L M N O | 6 | 1 | 0 | A B C D E F I J | 1 | 6 | 0 |
| 2 | A B I J K L M N | 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$_{2...0}$ 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 LH




**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.

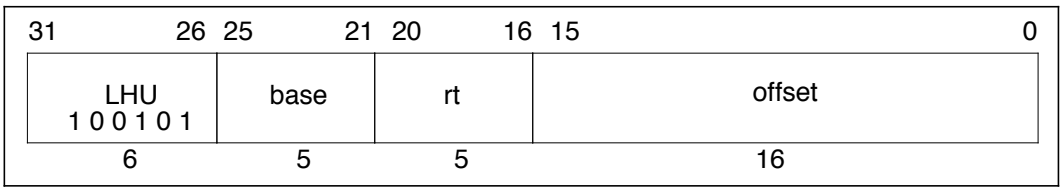
**Operation:**

```
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 (BigEndianCPU^2 || 0)
           GPR[rt] ← (mem15+8*byte)^16 || 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 (BigEndianCPU^2 || 0)
           GPR[rt] ← (mem15+8*byte)^48 || mem15+8*byte...8* byte
```

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LHU Load Halfword Unsigned LHU




**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:**

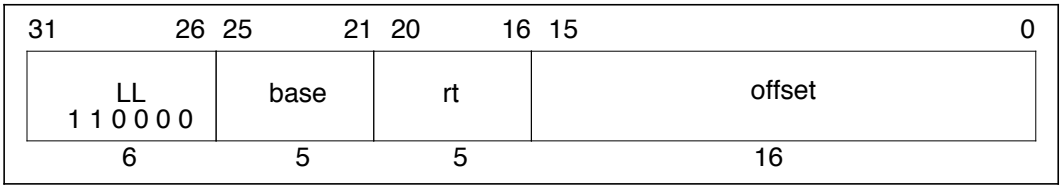
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           GPR[rt] ← 0^16 || 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^2 || 0))
           mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           GPR[rt] ← 0^48 || mem15+8*byte...8*byte
```

**Exceptions:**

| | |
|---|---|
| TLB refill exception | TLB invalid exception |
| Bus Error exception | Address error exception |

# LL Load Linked LL




**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) — LL

**Operation:**

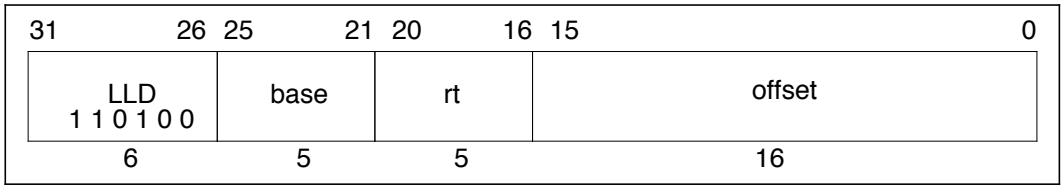
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← mem31+8*byte...8*byte
           LLbit ← 1

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← (mem31+8*byte)^32 || mem31+8*byte...8*byte
           LLbit ← 1
```

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LLD 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) LLD

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.

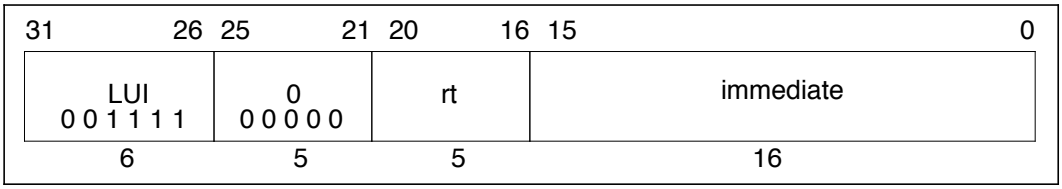
**Operation:**

```
64    T:    vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
```

**Exceptions:**

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

# LUI Load Upper Immediate LUI




**Format:**

LUI rt, immediate

**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:**

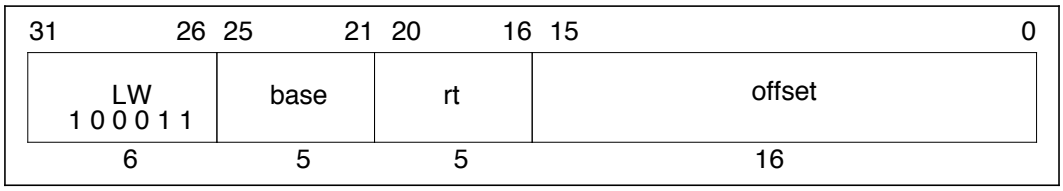
```
32    T:    GPR[rt] ← immediate || 0^16

64    T:    GPR[rt] ← (immediate_15)^32 || immediate || 0^16
```

**Exceptions:**

None

# LW Load Word LW




**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:**

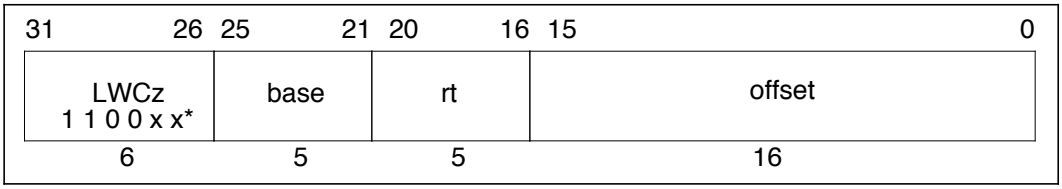
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← mem31+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 || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← (mem31+8*byte)^32 || mem31+8*byte...8*byte
```

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LWCz Load Word To Coprocessor LWCz




**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:**

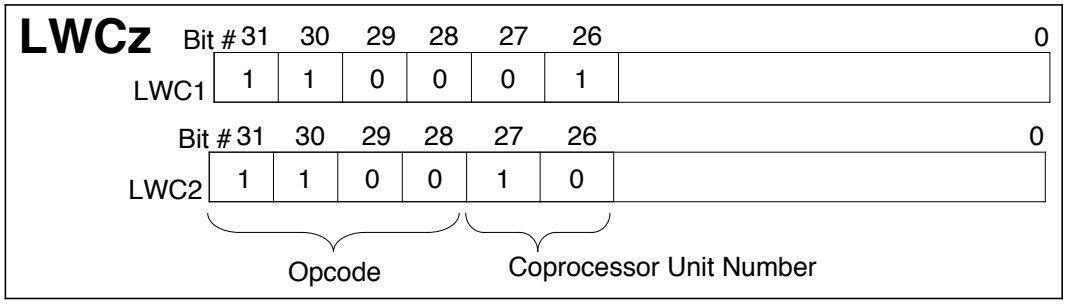
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           COPzLW (byte, rt, mem)

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base}
           (pAddr, uncached)← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           COPzLW (byte, rt, mem)
```

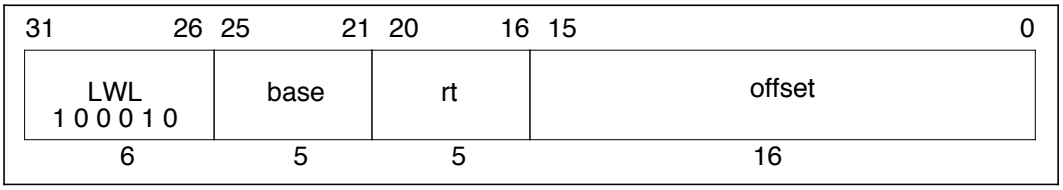
**Exceptions:**

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

**Opcode Bit Encoding:**



# LWL Load Word Left LWL




**Format:**

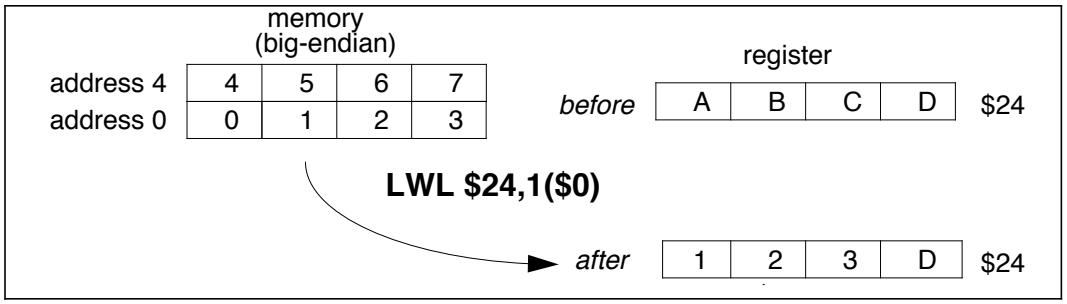
LWL rt, offset(base)

**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.



# LWL — Load Word Left (continued) — LWL

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.

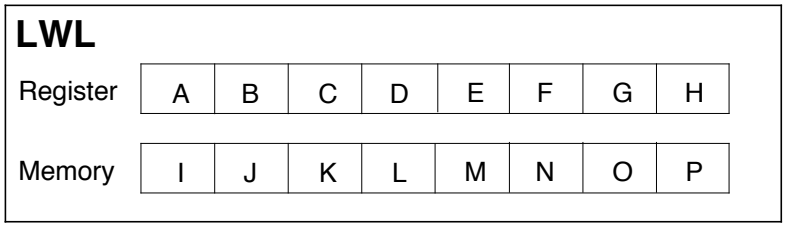
**Operation:**

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 0 then
                pAddr ← pAddrPSIZE–1...2 || 0^2
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← mem32*word+8*byte+7...32*word || GPR[rt]23-8*byte...0
           GPR[rt] ← temp

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 0 then
                pAddr ← pAddrPSIZE–1...2 || 0^2
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← mem32*word+8*byte+7...32*word || GPR[rt]23-8*byte...0
           GPR[rt] ← (temp31)^32 || temp
```

# LWL — Load Word Left (continued) — LWL

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



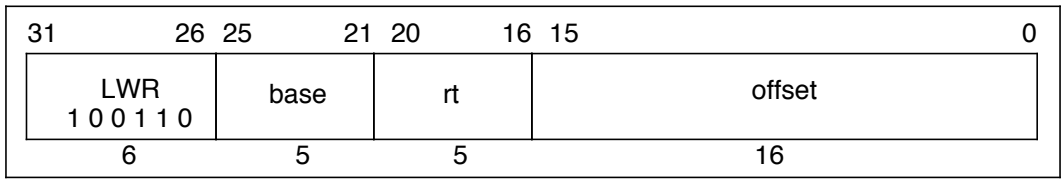

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | S S S S P F G H | 0 | 0 | 7 | S S S S I J K L | 3 | 4 | 0 |
| 1 | S S S S O P G H | 1 | 0 | 6 | S S S S J K L H | 2 | 4 | 1 |
| 2 | S S S S N O P H | 2 | 0 | 5 | S S S S K L G H | 1 | 4 | 2 |
| 3 | S S S S M N O P | 3 | 0 | 4 | S S S S L F G H | 0 | 4 | 3 |
| 4 | S S S S L F G H | 0 | 4 | 3 | S S S S M N O P | 3 | 0 | 4 |
| 5 | S S S S K L G H | 1 | 4 | 2 | S S S S N O P H | 2 | 0 | 5 |
| 6 | S S S S J K L H | 2 | 4 | 1 | S S S S O P G H | 1 | 0 | 6 |
| 7 | S S S S I J K L | 3 | 4 | 0 | S S S S P F G H | 0 | 0 | 7 |

| | |
|---|---|
| *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 $destination_{31}$ |

**Exceptions:**

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

# LWR Load Word Right LWR




**Format:**

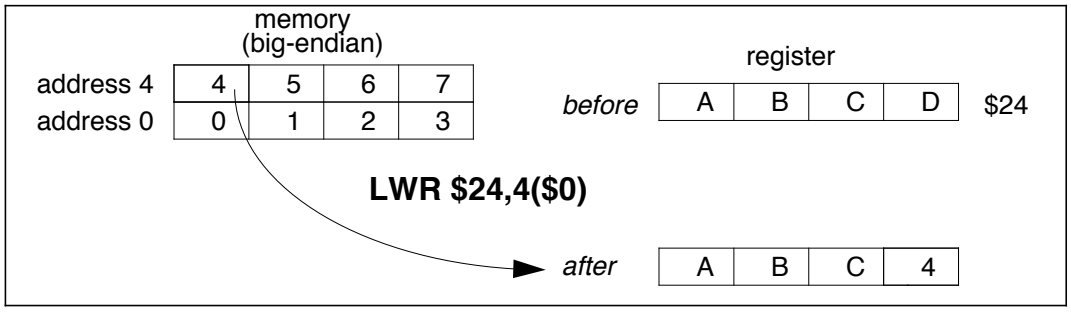
LWR rt, offset(base)

**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 Load Word Right (continued) LWR

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.

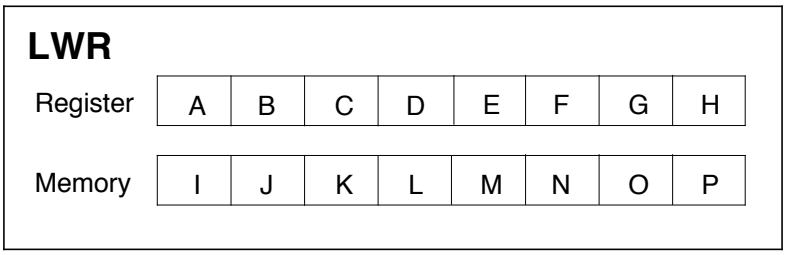
**Operation:**

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 1 then
                 pAddr ← pAddrPSIZE–31...3 || 0^3
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← GPR[rt]31...32-8*byte || mem31+32*word...32*word+8*byte
           GPR[rt] ← temp

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 1 then
                 pAddr ← pAddrPSIZE–31...3 || 0^3
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← GPR[rt]31...32-8*byte || mem31+32*word...32*word+8*byte
           GPR[rt] ← (temp31)^32 || temp
```

# LWR — Load Word Right (continued) — LWR

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



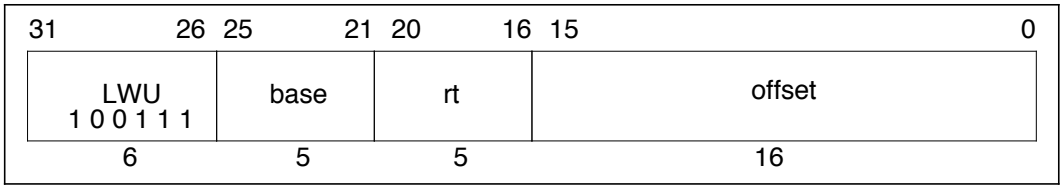

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | S S S S M N O P | 0 | 0 | 4 | X X X X E F G I | 0 | 7 | 0 |
| 1 | X X X X E M N O | 1 | 1 | 4 | X X X X E F I J | 1 | 6 | 0 |
| 2 | X X X X E F M N | 2 | 2 | 4 | X X X X E I J K | 2 | 5 | 0 |
| 3 | X X X X E F G M | 3 | 3 | 4 | S S S S I J K L | 3 | 4 | 0 |
| 4 | S S S S I J K L | 0 | 4 | 0 | X X X X E F G M | 0 | 3 | 4 |
| 5 | X X X X E I J K | 1 | 5 | 0 | X X X X E F M N | 1 | 2 | 4 |
| 6 | X X X X E F I J | 2 | 6 | 0 | X X X X E M N O | 2 | 1 | 4 |
| 7 | X X X X E F G I | 3 | 7 | 0 | S S S S M N O P | 3 | 0 | 4 |

| | |
|---|---|
| *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 LWU




**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.

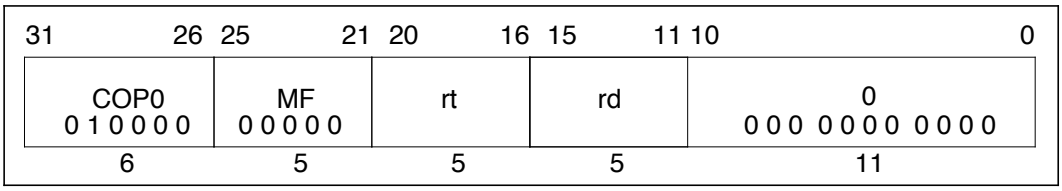
**Operation:**

```
64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← 0^32 || mem31+8*byte...8*byte
```

**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




**Format:**

MFC0 rt, rd

**Description:**

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

**Operation:**

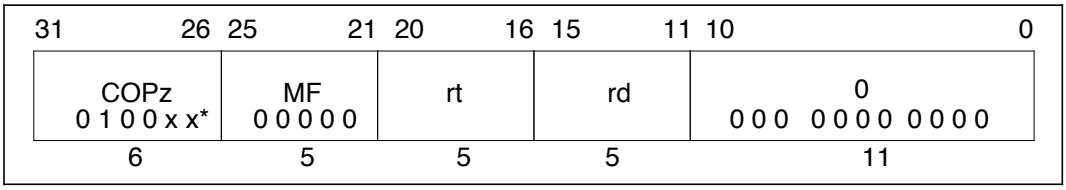
```
32    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← data

64    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← (data31)^32 || data31...0
```

**Exceptions:**

Coprocessor unusable exception

# MFCz Move From Coprocessor MFCz




**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]
        T+1:  GPR[rt] ← data
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
```

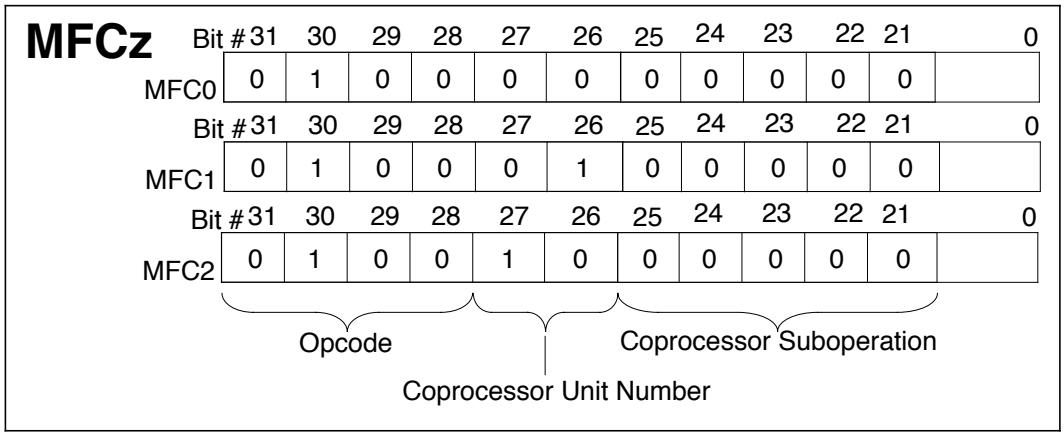
**Exceptions:**

Coprocessor unusable exception

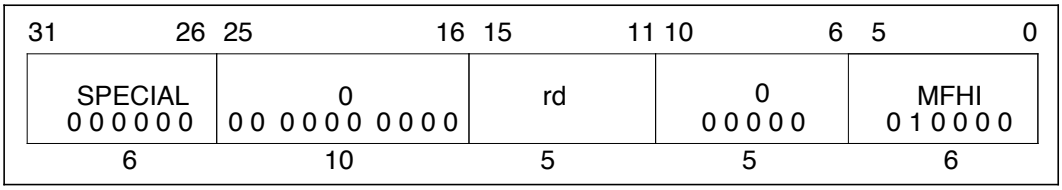
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# MFCz — Move From Coprocessor (continued) — MFCz

**Opcode Bit Encoding:**



# MFHI Move From HI MFHI




**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.

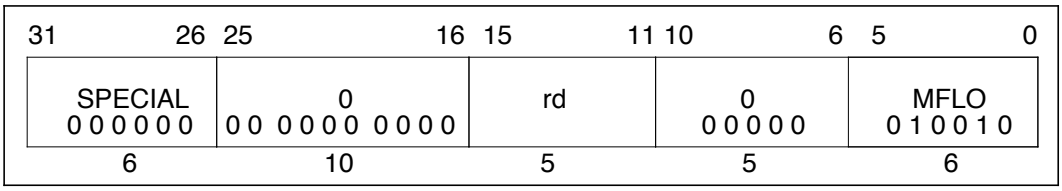
**Operation:**

```
32, 64      T:    GPR[rd] ← HI
```

**Exceptions:**

None

# MFLO Move From Lo MFLO




**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.

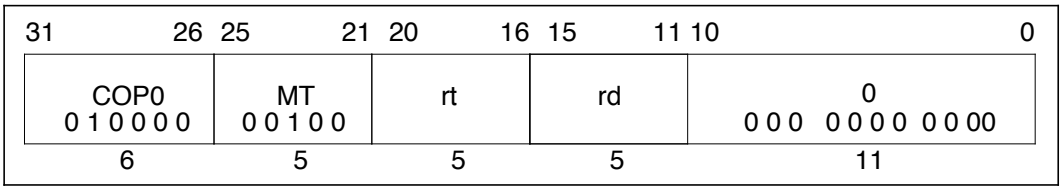
**Operation:**

```
32, 64      T:      GPR[rd] ← LO
```

**Exceptions:**

None

# MTC0 Move To System Control Coprocessor MTC0




**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.

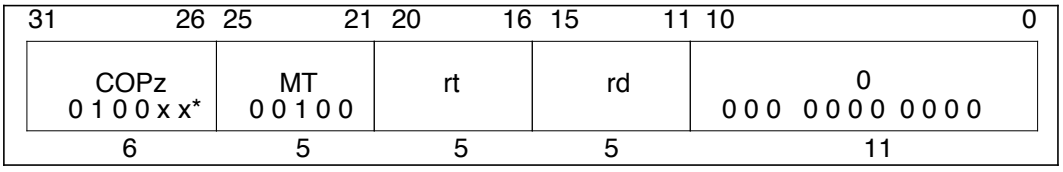
**Operation:**

```
32, 64    T:      data ← GPR[rt]
          T+1:    CPR[0,rd] ← data
```

**Exceptions:**

Coprocessor unusable exception

# MTCz — Move To Coprocessor — MTCz




**Format:**

MTCz rt, rd

**Description:**

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

**Operation:**

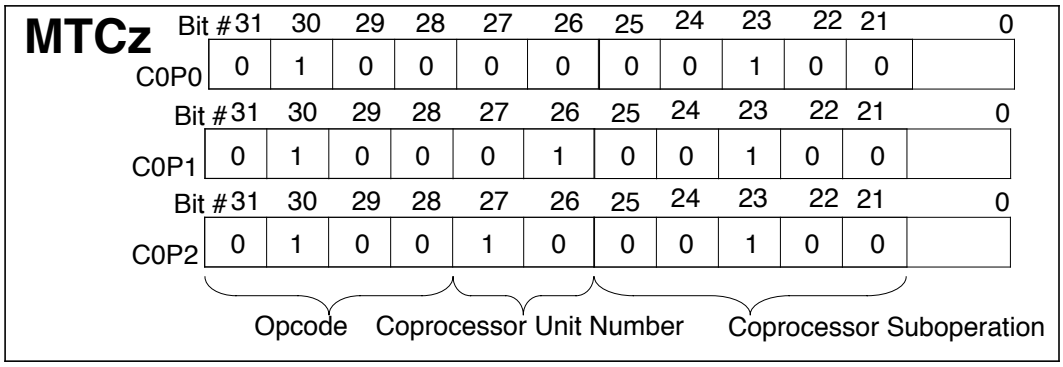
```
32    T:    data ← GPR[rt]
      T+1:  CPR[z,rd] ← data

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
```

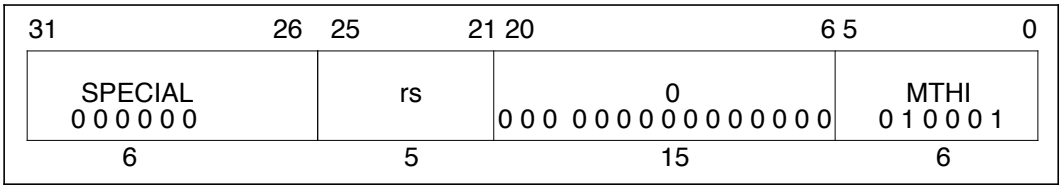
**Exceptions:**

Coprocessor unusable exception

***Opcode Bit Encoding:**



# MTHI Move To HI MTHI




**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.

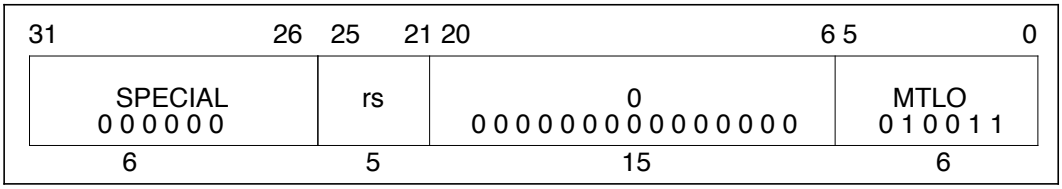
**Operation:**

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

**Exceptions:**

None

# MTLO Move To LO MTLO




**Format:**

MTLO rs

**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.

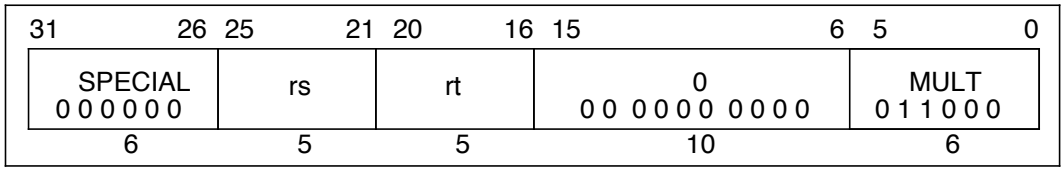
**Operation:**

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

**Exceptions:**

None

# MULT Multiply MULT




**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

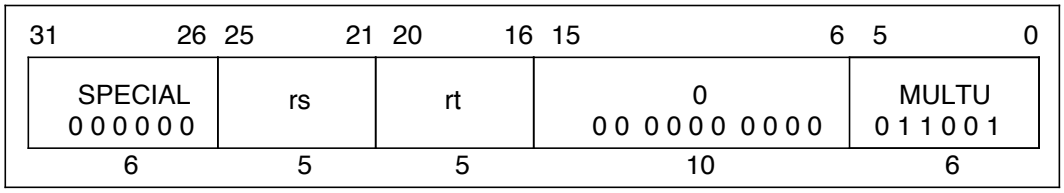
**Operation:**

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

**Exceptions:**

None

# MULTU Multiply Unsigned MULTU




**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

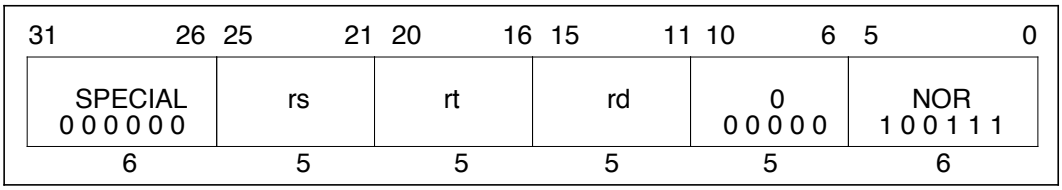
**Operation:**

```
32   T–2: LO   ← undefined
          HI   ← undefined
     T–1: LO   ← undefined
          HI   ← undefined
     T:   t    ← (0 || GPR[rs]) * (0 || GPR[rt])
          LO   ← t31...0
          HI   ← t63...32
64   T–2: LO   ← undefined
          HI   ← undefined
     T–1: LO   ← undefined
          HI   ← undefined
     T:   t    ← (0 || GPR[rs]31...0) * (0 || GPR[rt]31...0)
          LO   ← (t31)^32 || t31...0
          HI   ← (t63)^32 || t63...32
```

**Exceptions:**

None

# NOR Nor NOR




**Format:**

NOR rd, rs, rt

**Description:**

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*.

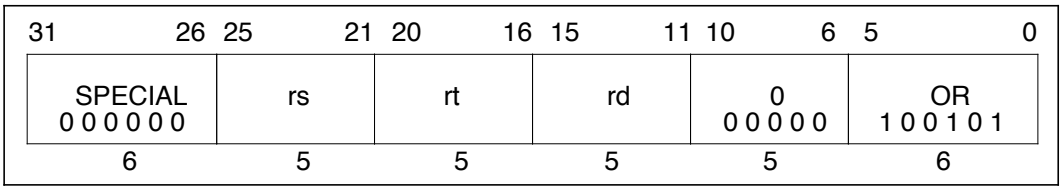
**Operation:**

```
32, 64      T:    GPR[rd] ← GPR[rs] nor GPR[rt]
```

**Exceptions:**

None

# OR Or OR




**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*.

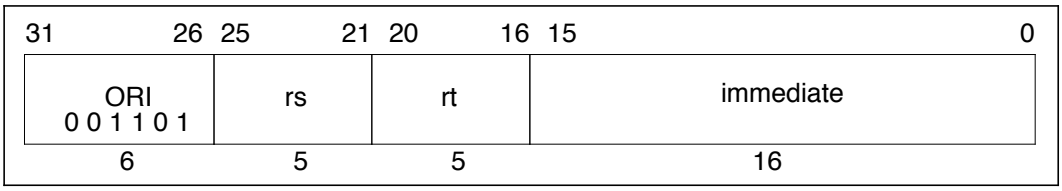
**Operation:**

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

**Exceptions:**

None

# ORI — Or Immediate — ORI




**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:**

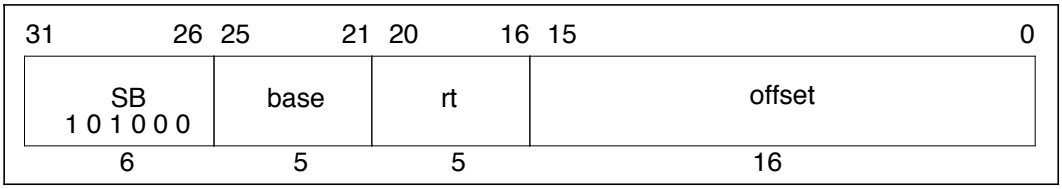
32 T: $GPR[rt] \leftarrow GPR[rs]_{31...16} \,||\, (immediate \text{ or } GPR[rs]_{15...0})$

64 T: $GPR[rt] \leftarrow GPR[rs]_{63...16} \,||\, (immediate \text{ or } GPR[rs]_{15...0})$

**Exceptions:**

None

# SB Store Byte SB




**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:**

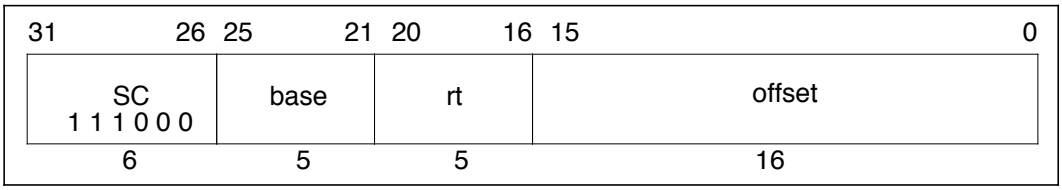
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor ReverseEndian^3)
           byte ← vAddr2...0 xor BigEndianCPU^3
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, BYTE, 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^3)
           byte ← vAddr2...0 xor BigEndianCPU^3
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

**Exceptions:**

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

# SC Store Conditional SC




**Format:**

SC 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 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 *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 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:**

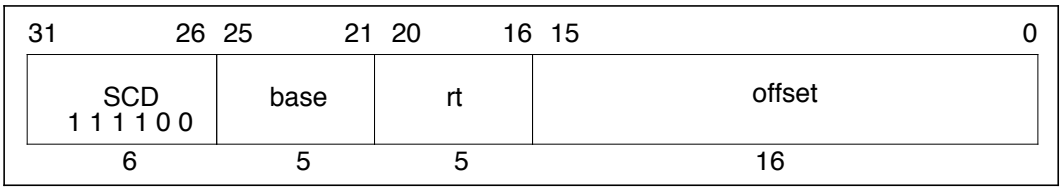
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           data  ← GPR[rt]63-8*byte...0 || 0^8*byte
           if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
           endif
           GPR[rt] ←  0^31 || LLbit

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           data  ← GPR[rt]63-8*byte...0 || 0^8*byte
           if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
           endif
           GPR[rt] ←  0^63 || LLbit
```

**Exceptions:**

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

# SCD Store Conditional Doubleword SCD




**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.

# SCD Store Conditional Doubleword (continued) SCD

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.

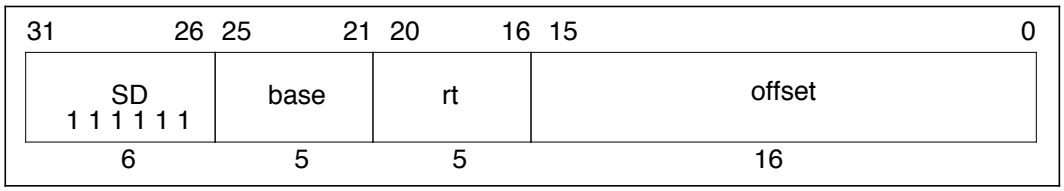
**Operation:**

```
64    T:    vAddr ← ((offset15)^48 || offset15..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 || 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)

# SD Store Doubleword SD




**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.

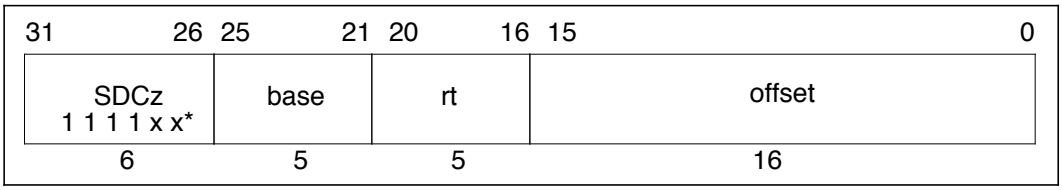
**Operation:**

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

**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)

# SDCz — Store Doubleword From Coprocessor — SDCz




**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)
```

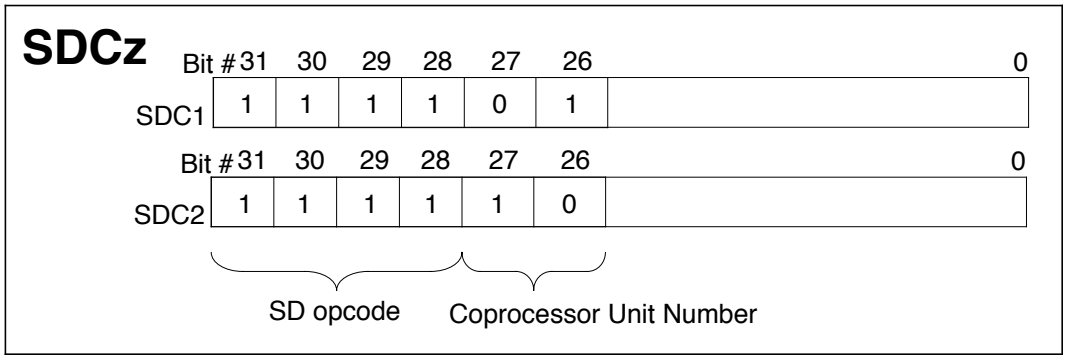
*See the table, "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# SDCz — Store Doubleword From Coprocessor (continued) — SDCz

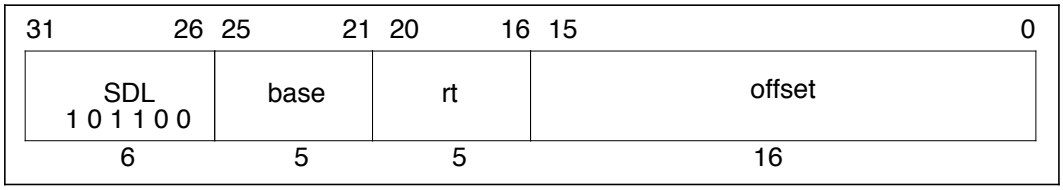
**Exceptions:**

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

**Opcode Bit Encoding:**



# SDL Store Doubleword Left SDL




**Format:**

SDL rt, offset(base)

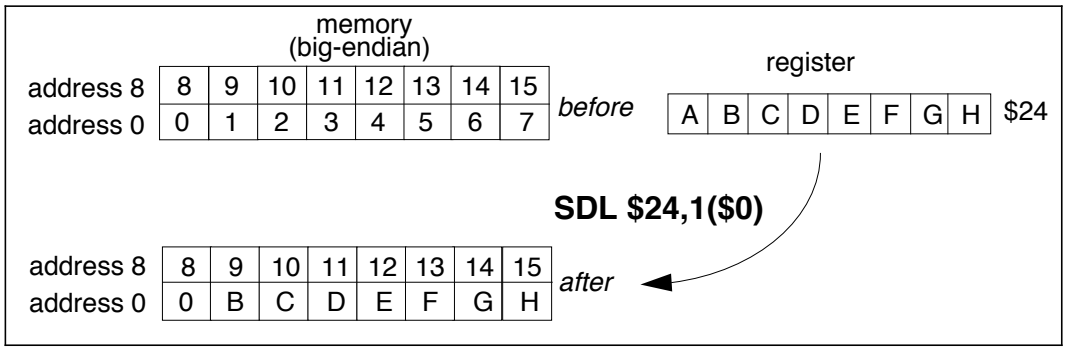
**Description:**

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.



# SDL Store Doubleword Left (continued) SDL

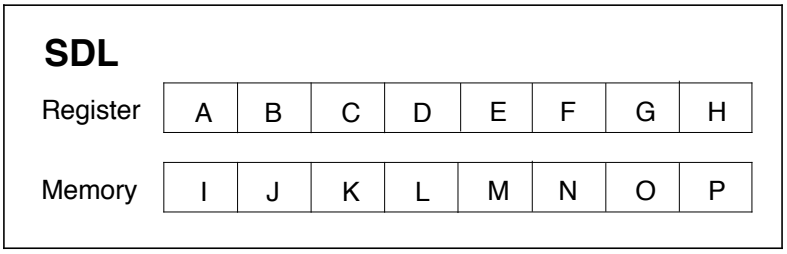
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

```
64   T:   vAddr ← ((offset15)^48 || offset 15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE –1...3 || (pAddr2...0 xor ReverseEndian^3)
          If BigEndianMem = 0 then
                pAddr ← pAddr31...3 || 0^3
          endif
          byte ← vAddr2...0 xor BigEndianCPU^3
          data ← 0^(56–8*byte) || GPR[rt]63...56–8*byte
          Storememory (uncached, byte, data, pAddr, vAddr, DATA)
```

# SDL — Store Doubleword Left (continued) — SDL

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



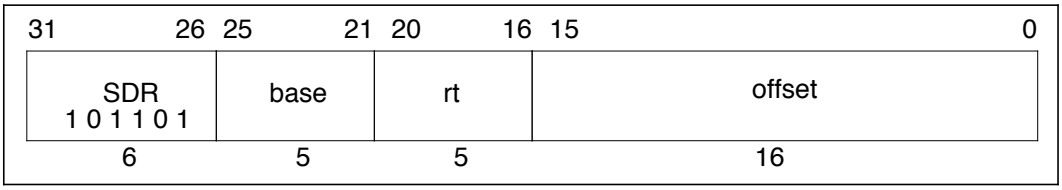

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset LEM | offset BEM | destination | type | offset LEM | offset BEM |
| 0 | I J K L M N O A | 0 | 0 | 7 | A B C D E F G H | 7 | 0 | 0 |
| 1 | I J K L M N A B | 1 | 0 | 6 | I A B C D E F G | 6 | 0 | 1 |
| 2 | I J K L M A B C | 2 | 0 | 5 | I J A B C D E F | 5 | 0 | 2 |
| 3 | I J K L A B C D | 3 | 0 | 4 | I J K A B C D E | 4 | 0 | 3 |
| 4 | I J K A B C D E | 4 | 0 | 3 | I J K L A B C D | 3 | 0 | 4 |
| 5 | I J A B C D E F | 5 | 0 | 2 | I J K L M A B C | 2 | 0 | 5 |
| 6 | I A B C D E F G | 6 | 0 | 1 | I J K L M N A B | 1 | 0 | 6 |
| 7 | A B C D E F G H | 7 | 0 | 0 | I J K L M N O A | 0 | 0 | 7 |

*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)

# SDR Store Doubleword Right SDR




**Format:**

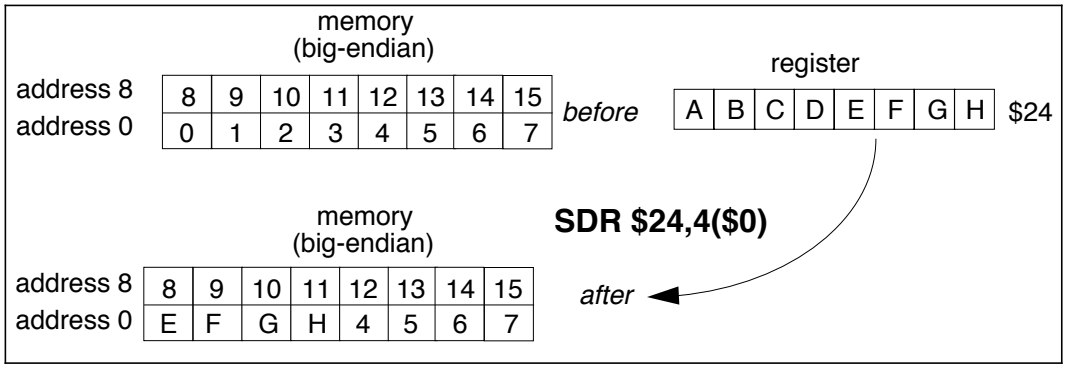
SDR rt, offset(base)

**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.



# SDR Store Doubleword Right (continued) SDR

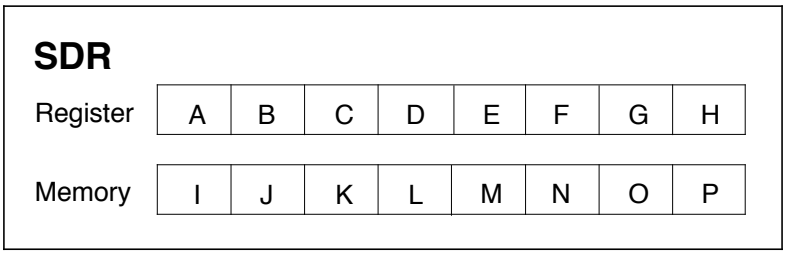
This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

**Operation:**

```
64    T:  vAddr ← ((offset15)48 || offset 15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian3)
          If BigEndianMem = 0 then
               pAddr ← pAddrPSIZE – 31...3 || 03
          endif
          byte ← vAddr1...0 xor BigEndianCPU3
          data ← GPR[rt]63–8*byte || 08*byte
          StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr, DATA)
```

# SDR Store Doubleword Right (continued) SDR

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



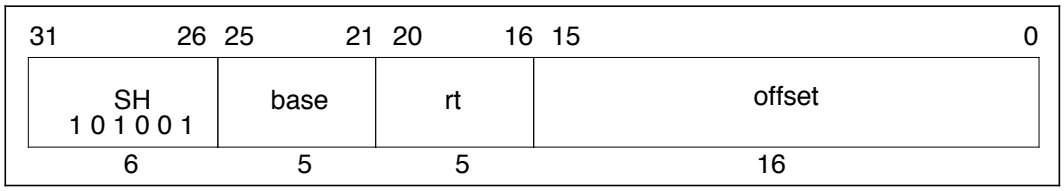

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | A B C D E F G H | 7 | 0 | 0 | H J K L M N O P | 0 | 7 | 0 |
| 1 | B C D E F G H P | 6 | 1 | 0 | G H K L M N O P | 1 | 6 | 0 |
| 2 | C D E F G H O P | 5 | 2 | 0 | F G H L M N O P | 2 | 5 | 0 |
| 3 | D E F G H N O P | 4 | 3 | 0 | E F G H M N O P | 3 | 4 | 0 |
| 4 | E F G H M N O P | 3 | 4 | 0 | D E F G H N O P | 4 | 3 | 0 |
| 5 | F G H L M N O P | 2 | 5 | 0 | C D E F G H O P | 5 | 2 | 0 |
| 6 | G H K L M N O P | 1 | 6 | 0 | B C D E F G H P | 6 | 1 | 0 |
| 7 | H J K L M N O P | 0 | 7 | 0 | A B C D E F G H | 7 | 0 | 0 |

*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 SH




**Format:**

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.

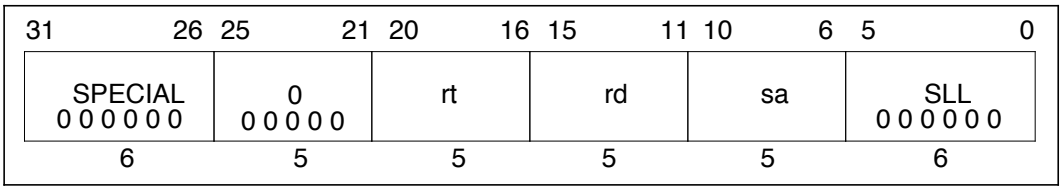
**Operation:**

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, HALFWORD, 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^2 || 0))
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

**Exceptions:**

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

# SLL — Shift Left Logical — SLL




**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:**

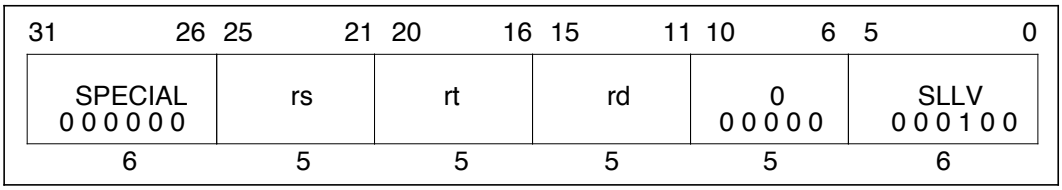
```
32    T:   GPR[rd] ← GPR[rt]31– sa...0 || 0sa

64    T:   s ← 0 || sa
           temp ← GPR[rt]31-s...0 || 0s
           GPR[rd] ← (temp31)32 || temp
```

**Exceptions:**

None

# SLLV Shift Left Logical Variable SLLV




**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:**

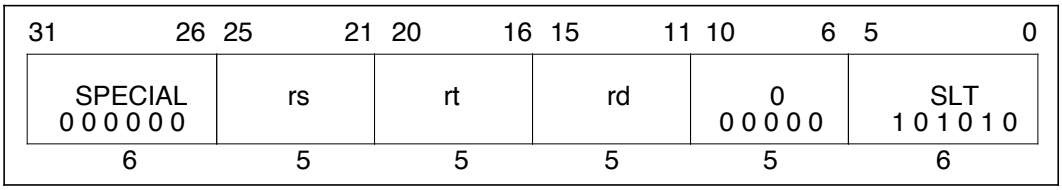
```
32    T:   s ← GP[rs]4...0
           GPR[rd]← GPR[rt](31–s)...0 || 0s

64    T:   s ← 0 || GP[rs]4...0
           temp ← GPR[rt](31-s)...0 || 0s
           GPR[rd] ← (temp31)32 || temp
```

**Exceptions:**

None

# SLT — Set On Less Than — SLT




**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.

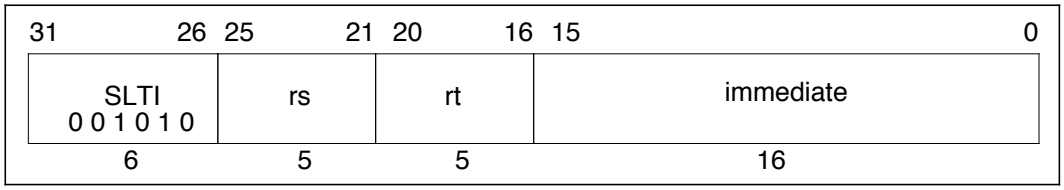
**Operation:**

```
32    T:    if GPR[rs] < GPR[rt] then
                  GPR[rd] ← 0^31 || 1
            else
                  GPR[rd] ← 0^32
            endif
64    T:    if GPR[rs] < GPR[rt] then
                  GPR[rd] ← 0^63 || 1
            else
                  GPR[rd] ← 0^64
            endif
```

**Exceptions:**

None

# SLTI Set On Less Than Immediate SLTI




**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.

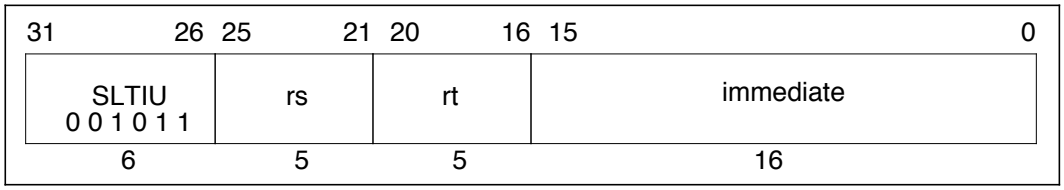
**Operation:**

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

**Exceptions:**

None

# SLTIU — Set On Less Than Immediate Unsigned — SLTIU




**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.

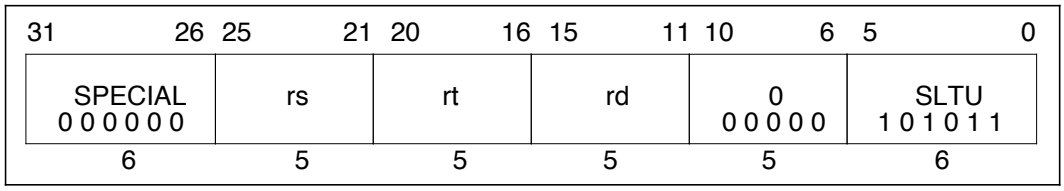
**Operation:**

```
32    T:   if (0 || GPR[rs]) < (immediate15)^16 || immediate15...0 then
                    GPR[rd] ← 0^31 || 1
           else
                    GPR[rd] ← 0^32
           endif
64    T:   if (0 || GPR[rs]) < (immediate15)^48 || immediate15...0 then
                    GPR[rd] ← 0^63 || 1
           else
                    GPR[rd] ← 0^64
           endif
```

**Exceptions:**

None

# SLTU Set On Less Than Unsigned SLTU




**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.

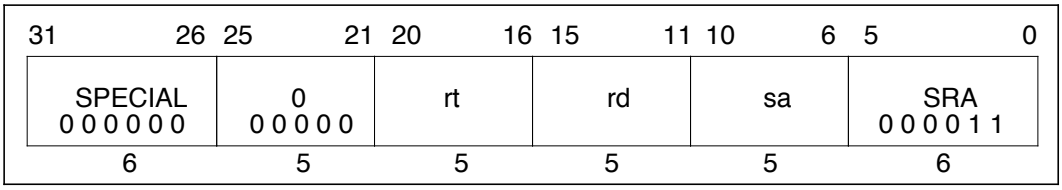
**Operation:**

```
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

# SRA Shift Right Arithmetic SRA




**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.

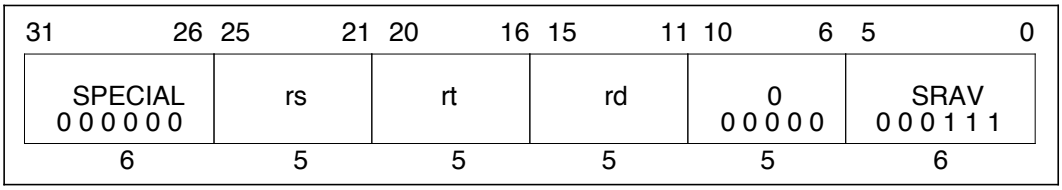
**Operation:**

```
32   T:   GPR[rd] ← (GPR[rt]31)sa || GPR[rt] 31...sa
64   T:   s ← 0 || sa
          temp ← (GPR[rt]31)s || GPR[rt] 31...s
          GPR[rd] ← (temp31)32 || temp
```

**Exceptions:**

None

# SRAV — Shift Right Arithmetic Variable — SRAV




**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:**

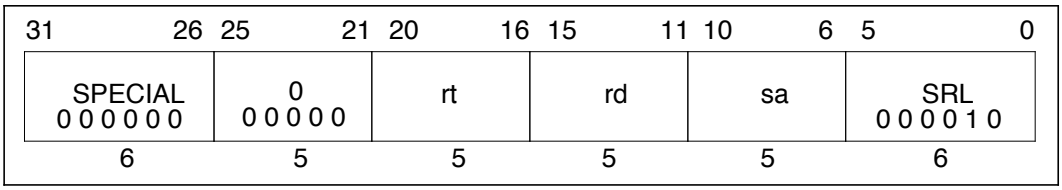
```
32    T:   s ← GPR[rs]4...0
           GPR[rd] ← (GPR[rt]31)^s || GPR[rt]31...s

64    T:   s ← GPR[rs]4...0
           temp ← (GPR[rt]31)^s || GPR[rt]31...s
           GPR[rd] ← (temp31)^32 || temp
```

**Exceptions:**

None

# SRL Shift Right Logical SRL




**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:**

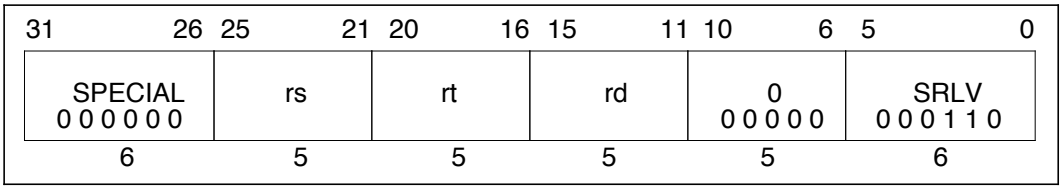
```
32   T:   GPR[rd] ← 0^sa || GPR[rt]31...sa

64   T:   s ← 0 || sa
          temp ← 0^s || GPR[rt]31...s
          GPR[rd] ← (temp31)^32 || temp
```

**Exceptions:**

None

# SRLV Shift Right Logical Variable SRLV




**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:**

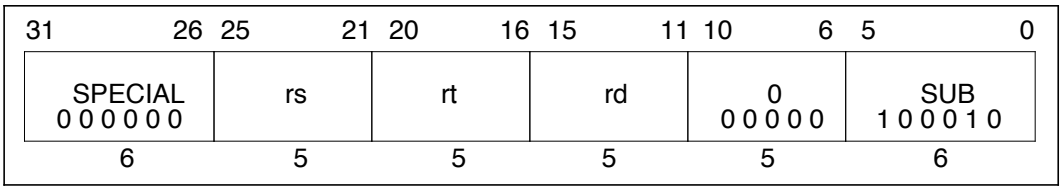
```
32   T:   s ← GPR[rs]4...0
          GPR[rd] ← 0^s || GPR[rt]31...s

64   T:   s ← GPR[rs]4...0
          temp ← 0^s || GPR[rt]31...s
          GPR[rd] ← (temp31)^32 || temp
```

**Exceptions:**

None

# SUB — Subtract — SUB




**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:**

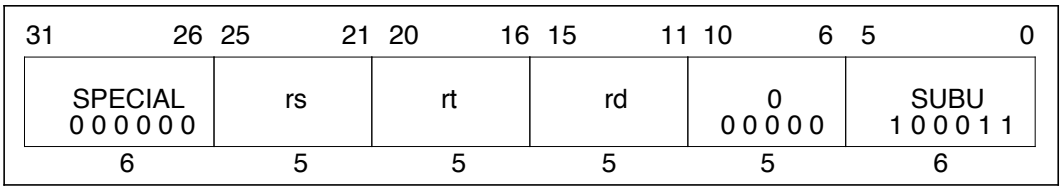
```
32    T:    GPR[rd] ← GPR[rs] – GPR[rt]

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

**Exceptions:**

Integer overflow exception

# SUBU Subtract Unsigned SUBU




**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:**

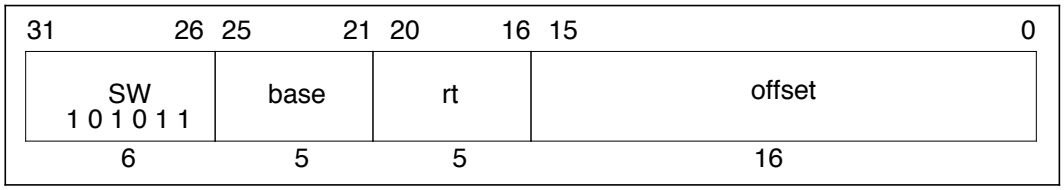
```
32    T:    GPR[rd] ← GPR[rs] – GPR[rt]

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

**Exceptions:**

None

# SW Store Word SW




**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.

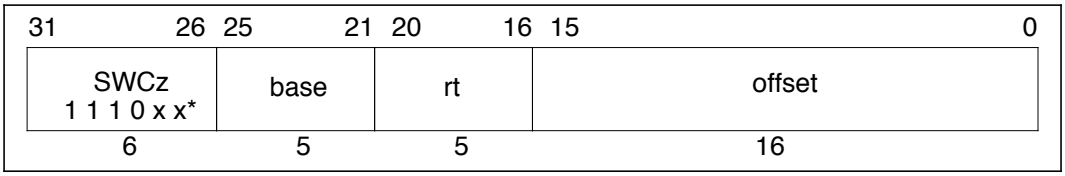
**Operation:**

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           data ← GPR[rt]63-8*byte || 0^8*byte
           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 || 0^2)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           data ← GPR[rt]63-8*byte || 0^8*byte
           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




**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)
```

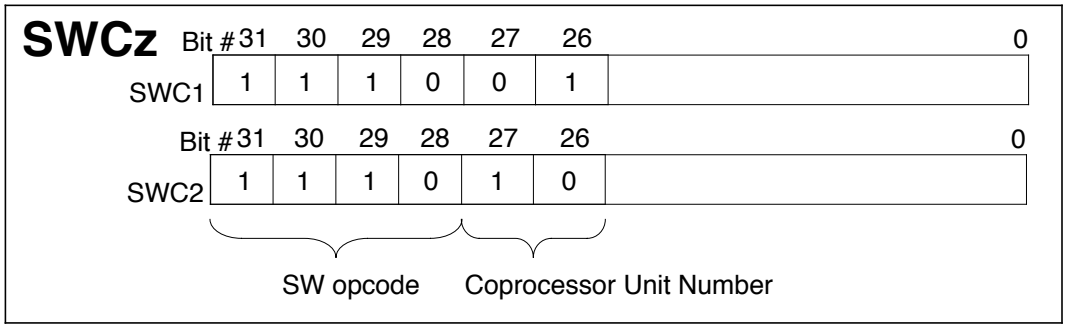
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

# SWCz Store Word From Coprocessor (Continued) SWCz

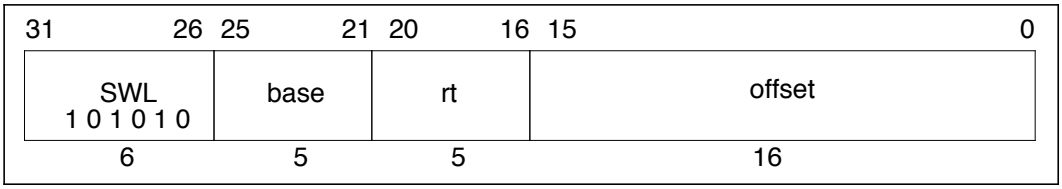
**Exceptions:**

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

**Opcode Bit Encoding:**



# SWL Store Word Left SWL




**Format:**

SWL rt, offset(base)

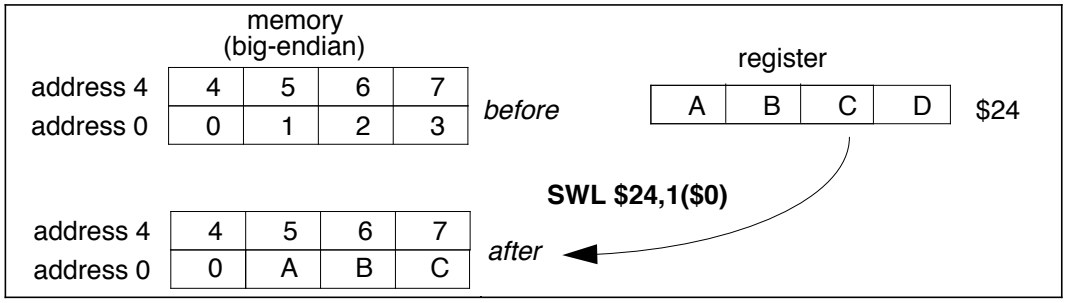
**Description:**

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.



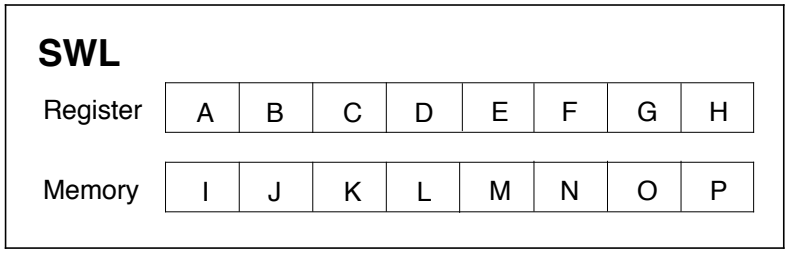
# SWL — Store Word Left (Continued) — SWL

**Operation:**

```
32      T:  vAddr ← ((offset15)^16 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || 0^24-8*byte || GPR[rt]31...24-8*byte
            else
                data ← 0^24-8*byte || GPR[rt]31...24-8*byte || 0^32
            endif
            Storememory (uncached, byte, data, pAddr, vAddr, DATA)
64      T:  vAddr ← ((offset15)^48 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || 0^24-8*byte || GPR[rt]31...24-8*byte
            else
                data ← 0^24-8*byte || GPR[rt]31...24-8*byte || 0^32
            endif
            StoreMemory(uncached, byte, data, pAddr, vAddr, DATA)
```

# SWL — Store Word Left (Continued) — SWL

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



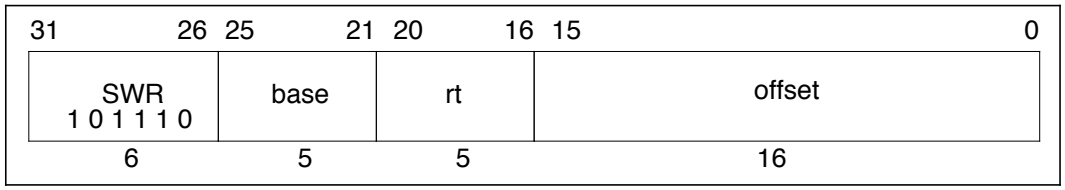

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset LEM | offset BEM | destination | type | offset LEM | offset BEM |
| 0 | I J K L M N O E | 0 | 0 | 7 | E F G H M N O P | 3 | 4 | 0 |
| 1 | I J K L M N E F | 1 | 0 | 6 | I E F G M N O P | 2 | 4 | 1 |
| 2 | I J K L M E F G | 2 | 0 | 5 | I J E F M N O P | 1 | 4 | 2 |
| 3 | I J K L E F G H | 3 | 0 | 4 | I J K E M N O P | 0 | 4 | 3 |
| 4 | I J K E M N O P | 0 | 4 | 3 | I J K L E F G H | 3 | 0 | 4 |
| 5 | I J E F M N O P | 1 | 4 | 2 | I J K L M E F G | 2 | 0 | 5 |
| 6 | I E F G M N O P | 2 | 4 | 1 | I J K L M N E F | 1 | 0 | 6 |
| 7 | E F G H M N O P | 3 | 4 | 0 | I J K L M N O E | 0 | 0 | 7 |

*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

# SWR Store Word Right SWR




**Format:**

SWR rt, offset(base)

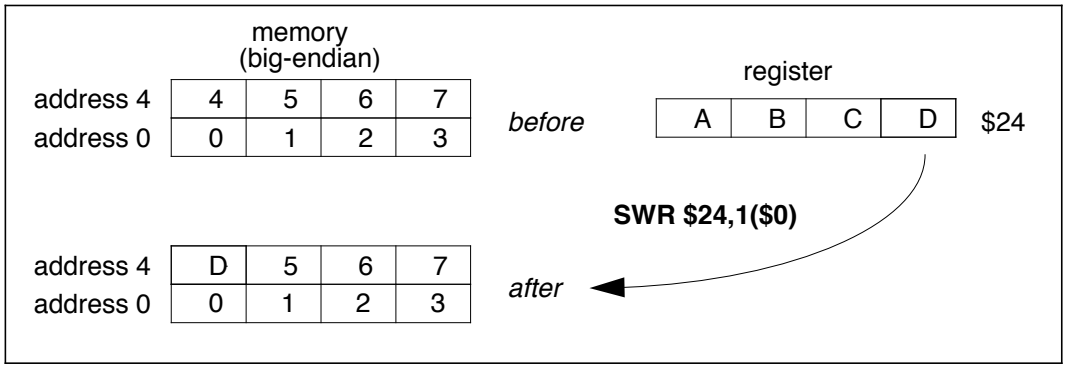
**Description:**

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.



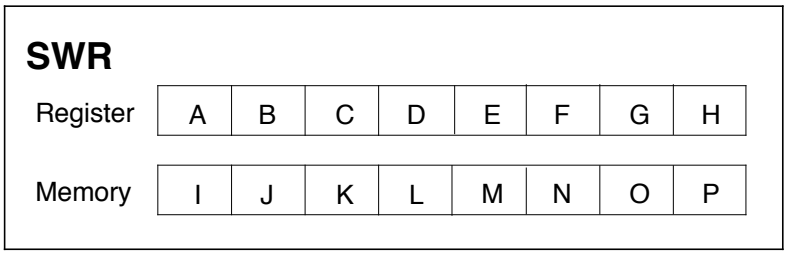
# SWR — Store Word Right (Continued) — SWR

**Operation:**

```
32      T:  vAddr ← ((offset15)^16 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || GPR[rt]31-8*byte...0 || 0^8*byte
            else
                data ← GPR[rt]31-8*byte...0 || 0^8*byte || 0^32
            endif
            Storememory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
64      T:  vAddr ← ((offset15)^48 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || GPR[rt]31-8*byte...0 || 0^8*byte
            else
                data ← GPR[rt]31-8*byte...0 || 0^8*byte || 0^32
            endif
            StoreMemory(uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

# SWR — Store Word Right (Continued) — SWR

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



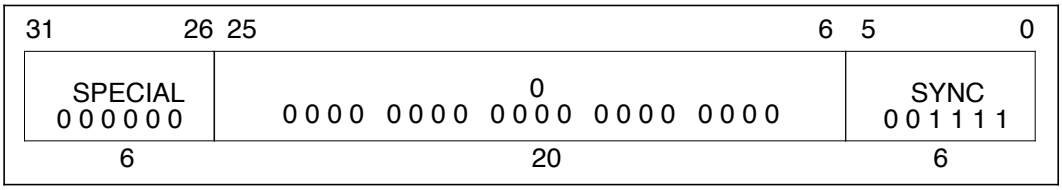

| $vAddr_{2..0}$ | BigEndianCPU = 0 | | | | BigEndianCPU = 1 | | | |
|---|---|---|---|---|---|---|---|---|
| | destination | type | offset | | destination | type | offset | |
| | | | LEM | BEM | | | LEM | BEM |
| 0 | I J K L E F G H | 3 | 0 | 4 | H J K L M N O P | 0 | 7 | 0 |
| 1 | I J K L F G H P | 2 | 1 | 4 | G H K L M N O P | 1 | 6 | 0 |
| 2 | I J K L G H O P | 1 | 2 | 4 | F G H L M N O P | 2 | 5 | 0 |
| 3 | I J K L H N O P | 0 | 3 | 4 | E F G H M N O P | 3 | 4 | 0 |
| 4 | E F G H M N O P | 3 | 4 | 0 | I J K L H N O P | 0 | 3 | 4 |
| 5 | F G H L M N O P | 2 | 5 | 0 | I J K L G H O P | 1 | 2 | 4 |
| 6 | G H K L M N O P | 1 | 6 | 0 | I J K L F G H P | 2 | 1 | 4 |
| 7 | H J K L M N O P | 0 | 7 | 0 | I J K L E F G H | 3 | 0 | 4 |

*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 SYNC




**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:

| Processor A | | Processor B | | |
|---|---|---|---|---|
| SW | R1, DATA | 1: | LW | R2, FLAG |
| LI | R2, 1 | | BEQ | R2, R0, 1B |
| SYNC | | | NOP | |
| SW | R2, FLAG | | SYNC | |
| | | | LW | R1, DATA |

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.

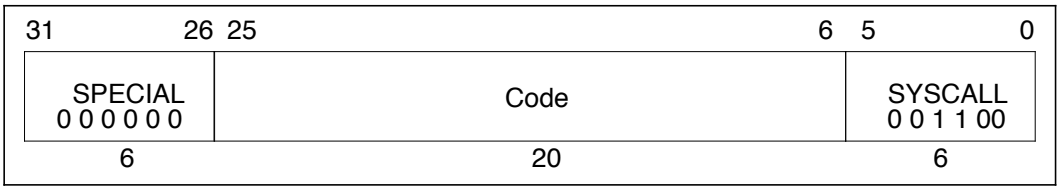
**Operation:**

```
32, 64      T:    SyncOperation()
```

**Exceptions:**

None

# SYSCALL System Call SYSCALL




**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.

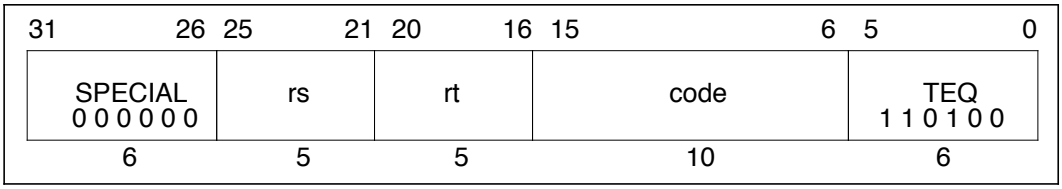
**Operation:**

```
32, 64   T:     SystemCallException
```

**Exceptions:**

System Call exception

# TEQ Trap If Equal TEQ




**Format:**

TEQ rs, rt

**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.

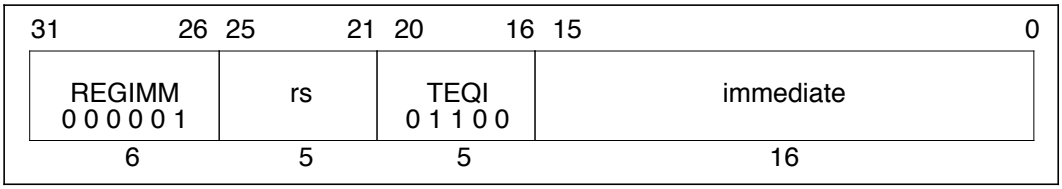
**Operation:**

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

**Exceptions:**

Trap exception

# TEQI Trap If Equal Immediate TEQI




**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:**

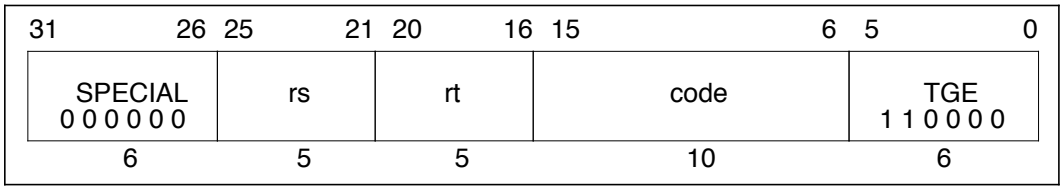
```
32    T:    if GPR[rs] = (immediate15)16 || immediate15...0 then
              TrapException
            endif

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

**Exceptions:**

Trap exception

# TGE Trap If Greater Than Or Equal TGE




**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.

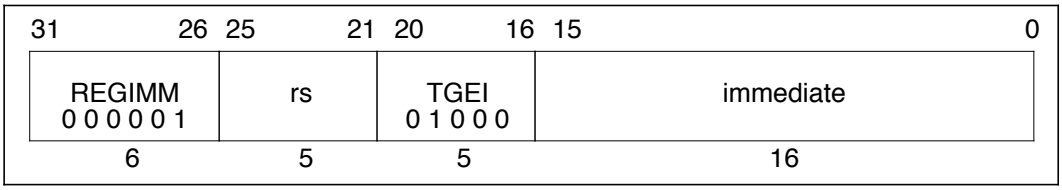
**Operation:**

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

**Exceptions:**

Trap exception

# TGEI Trap If Greater Than Or Equal Immediate TGEI




**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.

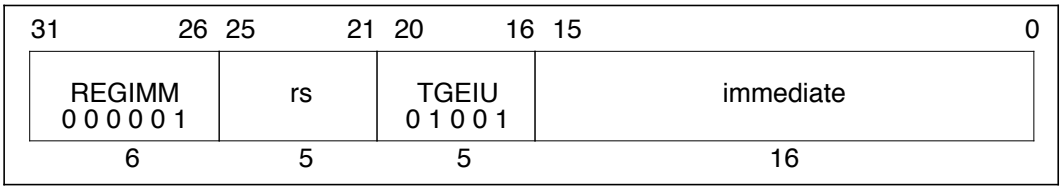
**Operation:**

```
32    T:  if GPR[rs] ≥ (immediate15)16 || immediate15...0 then
              TrapException
          endif
64    T:  if GPR[rs] ≥ (immediate15)48 || immediate15...0 then
              TrapException
          endif
```

**Exceptions:**

Trap exception

# TGEIU — Trap If Greater Than Or Equal Immediate Unsigned — TGEIU




**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.

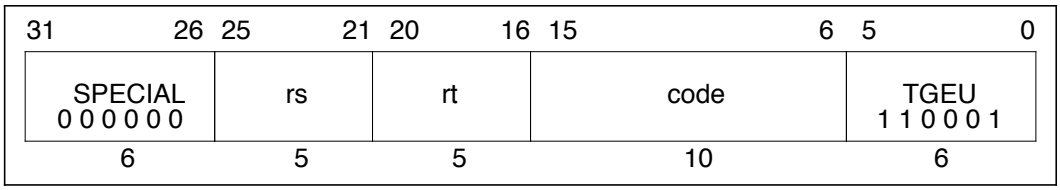
**Operation:**

```
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 TGEU




**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.

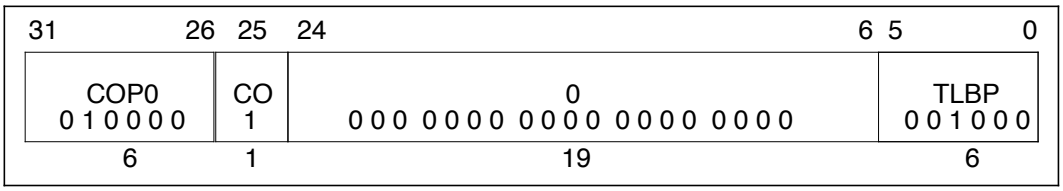
**Operation:**

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

**Exceptions:**

Trap exception

# TLBP Probe TLB For Matching Entry TLBP




**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:**

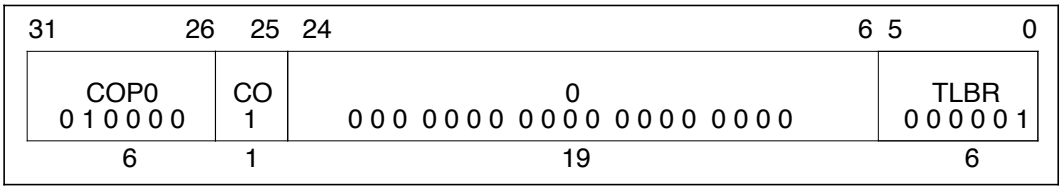
```
32    T:   Index← 1 || 0^25 || undefined^6
           for i in 0...TLBEntries–1
                if (TLB[i]95...77 = EntryHi31...12) and  (TLB[i]76 or
                (TLB[i]71...64 = EntryHi7...0)) then
                     Index ← 0^26 || i 5...0
                endif
           endfor

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))
                 = EntryHi39...13) and not (0^15 || TLB[i]216...205)) and
                (TLB[i]140 or (TLB[i]135...128 = EntryHi7...0)) then
                     Index ← 0^26 || i 5...0
                endif
           endfor
```

**Exceptions:**

Coprocessor unusable exception

# TLBR Read Indexed TLB Entry TLBR




**Format:**

TLBR

**Description:**

The *G* bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers.

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

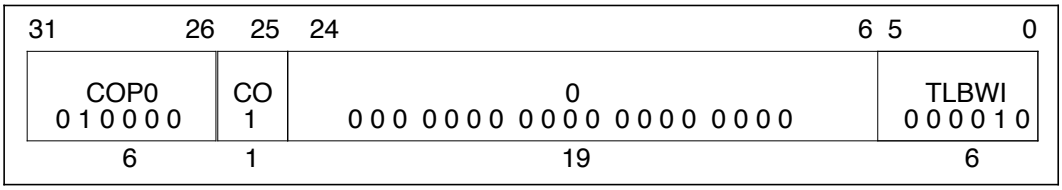
```
32    T:  PageMask ← TLB[Index5...0]127...96
          EntryHi ← TLB[Index5...0]95...64 and not TLB[Index5...0]127...96
          EntryLo1 ←TLB[Index5...0]63...32
          EntryLo0 ← TLB[Index5...0]31...0

64    T:  PageMask ← TLB[Index5...0]255...192
          EntryHi ← TLB[Index5...0]191...128 and not TLB[Index5...0]255...192
          EntryLo1 ←TLB[Index5...0]127...65 || TLB[Index5...0]140
          EntryLo0 ← TLB[Index5...0]63...1 || TLB[Index5...0]140
```

**Exceptions:**

Coprocessor unusable exception

# TLBWI Write Indexed TLB Entry TLBWI




**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.

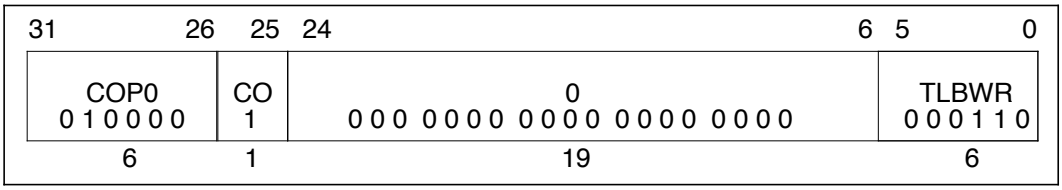
**Operation:**

```
32, 64T:    TLB[Index5...0] ←
                PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

**Exceptions:**

Coprocessor unusable exception

# TLBWR Write Random TLB Entry TLBWR




**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.

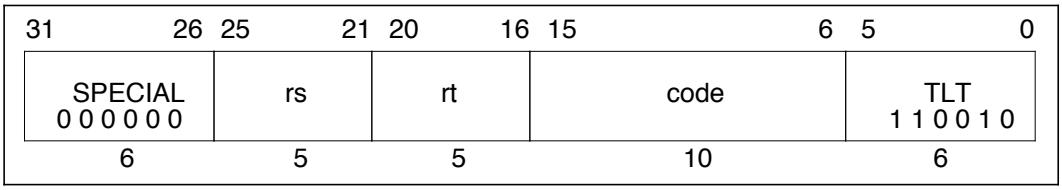
**Operation:**

```
32, 64T:    TLB[Random5...0] ←
                PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

**Exceptions:**

Coprocessor unusable exception

# TLT Trap If Less Than TLT




**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.

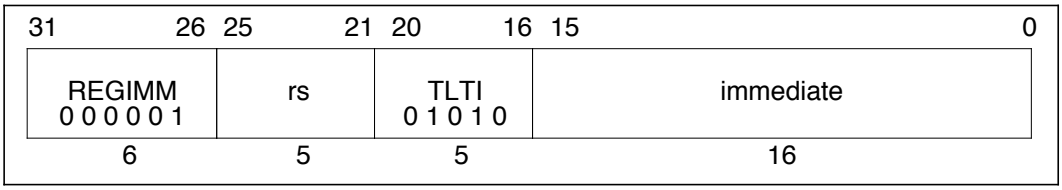
**Operation:**

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

**Exceptions:**

Trap exception

# TLTI Trap If Less Than Immediate TLTI




**Format:**

TLTI 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.

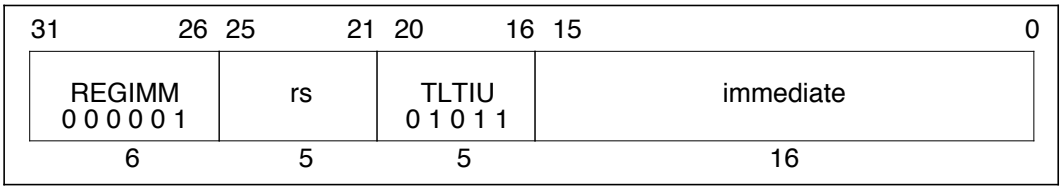
**Operation:**

```
32    T:  if GPR[rs] < (immediate15)16 || immediate15...0 then
              TrapException
          endif
64    T:  if GPR[rs] < (immediate15)48 || immediate15...0 then
              TrapException
          endif
```

**Exceptions:**

Trap exception

# TLTIU Trap If Less Than Immediate Unsigned TLTIU




**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:**

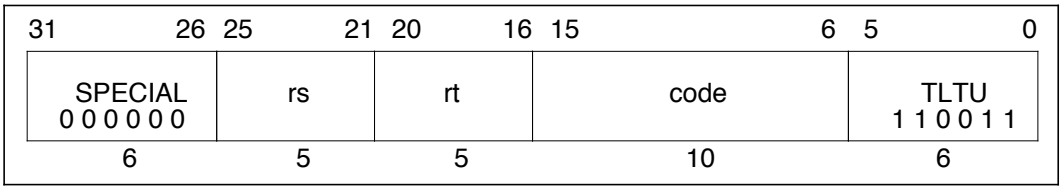
```
32    T:   if (0 || GPR[rs]) < (0 || (immediate15)16 || immediate15...0) then
               TrapException
           endif

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

**Exceptions:**

Trap exception

# TLTU Trap If Less Than Unsigned TLTU




**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.

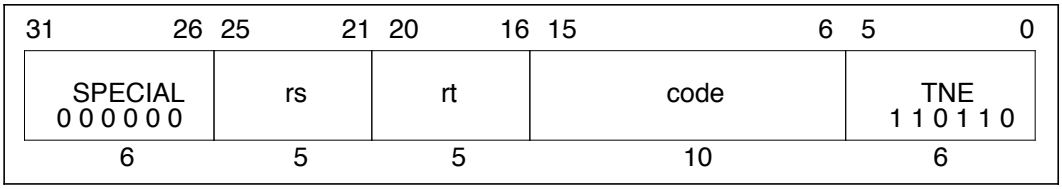
**Operation:**

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

**Exceptions:**

Trap exception

# TNE Trap If Not Equal TNE




**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.

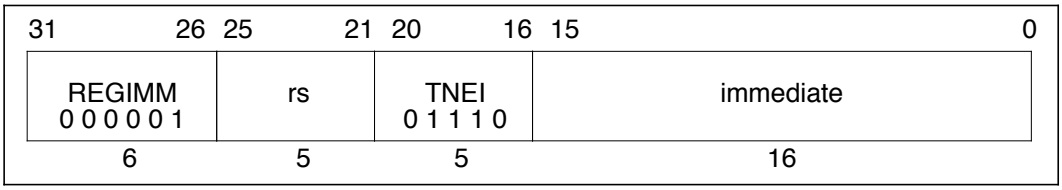
**Operation:**

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

**Exceptions:**

Trap exception

# TNEI Trap If Not Equal Immediate TNEI




**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.

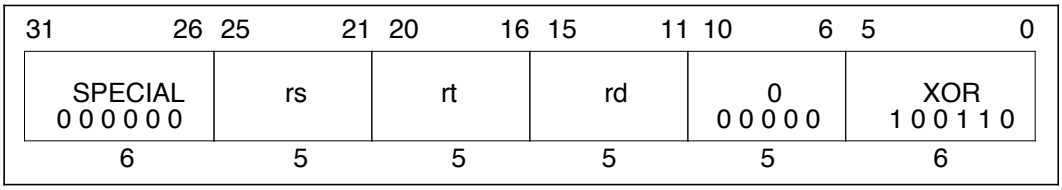
**Operation:**

```
32    T:   if GPR[rs] ≠ (immediate15)16 || immediate15...0 then
             TrapException
           endif
64    T:   if GPR[rs] ≠ (immediate15)48 || immediate15...0 then
             TrapException
           endif
```

**Exceptions:**

Trap exception

# XOR Exclusive Or XOR




**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*.

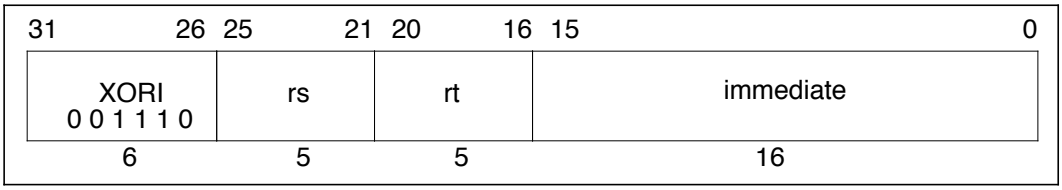
**Operation:**

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

**Exceptions:**

None

# XORI Exclusive OR Immediate XORI




**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.

**Opcode** (28...26 / 31...29)

| 31...29 \ 28...26 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SPECIAL | REGIMM | J | JAL | BEQ | BNE | BLEZ | BGTZ |
| 1 | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI |
| 2 | COP0 | COP1 | COP2 | * | BEQL | BNEL | BLEZL | BGTZL |
| 3 | DADDIε | DADDIUε | LDLε | LDRε | * | * | * | * |
| 4 | LB | LH | LWL | LW | LBU | LHU | LWR | LWUε |
| 5 | SB | SH | SWL | SW | SDLε | SDRε | SWR | CACHE δ |
| 6 | LL | LWC1 | LWC2 | * | LLDε | LDC1 | LDC2 | LDε |
| 7 | SC | SWC1 | SWC2 | * | SCDε | SDC1 | SDC2 | SDε |

**SPECIAL function** (2...0 / 5...3)

| 5...3 \ 2...0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SLL | * | SRL | SRA | SLLV | * | SRLV | SRAV |
| 1 | JR | JALR | * | * | SYSCALL | BREAK | * | SYNC |
| 2 | MFHI | MTHI | MFLO | MTLO | DSLLVε | * | DSRLVε | DSRAVε |
| 3 | MULT | MULTU | DIV | DIVU | DMULTε | DMULTUε | DDIVε | DDIVUε |
| 4 | ADD | ADDU | SUB | SUBU | AND | OR | XOR | NOR |
| 5 | * | * | SLT | SLTU | DADDε | DADDUε | DSUBε | DSUBUε |
| 6 | TGE | TGEU | TLT | TLTU | TEQ | * | TNE | * |
| 7 | DSLLε | * | DSRLε | DSRAε | DSLL32ε | * | DSRL32ε | DSRA32ε |

**REGIMM rt** (18...16 / 20...19)

| 20...19 \ 18...16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BLTZ | BGEZ | BLTZL | BGEZL | * | * | * | * |
| 1 | TGEI | TGEIU | TLTI | TLTIU | TEQI | * | TNEI | * |
| 2 | BLTZAL | BGEZAL | BLTZALL | BGEZALL | * | * | * | * |
| 3 | * | * | * | * | * | * | * | * |

**COPz rs** (23...21 / 25, 24)

| 25, 24 \ 23...21 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | MF | DMFε | CF | γ | MT | DMTε | CT | γ |
| 1 | BC | γ | γ | γ | γ | γ | γ | γ |
| 2 | CO | | | | | | | |
| 3 | | | | | | | | |

*Figure A-2 R4000 Opcode Bit Encoding*

**COPz rt**

| 20...19 \ 18...16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BCF | BCT | BCFL | BCTL | γ | γ | γ | γ |
| 1 | γ | γ | γ | γ | γ | γ | γ | γ |
| 2 | γ | γ | γ | γ | γ | γ | γ | γ |
| 3 | γ | γ | γ | γ | γ | γ | γ | γ |

**CP0 Function**

| 5 ... 3 \ 2 ... 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | φ | TLBR | TLBWI | φ | φ | φ | TLBWR | φ |
| 1 | TLBP | φ | φ | φ | φ | φ | φ | φ |
| 2 | ξ | φ | φ | φ | φ | φ | φ | φ |
| 3 | ERET χ | φ | φ | φ | φ | φ | φ | φ |
| 0 | φ | φ | φ | φ | φ | φ | φ | φ |
| 1 | φ | φ | φ | φ | φ | φ | φ | φ |
| 2 | φ | φ | φ | φ | φ | φ | φ | φ |
| 3 | φ | φ | φ | φ | φ | φ | φ | φ |

*Figure A-2 (cont.) R4000 Opcode Bit Encoding*

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.

# *FPU Instruction Set Details*

# *B*

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*

| Operation | Source Format | | | |
|---|---|---|---|---|
| | **Single** | **Double** | **Word** | **Longword** |
| ADD | V | V | R | R |
| SUB | V | V | R | R |
| MUL | V | V | R | R |
| DIV | V | V | R | R |
| SQRT | V | V | R | R |
| ABS | V | V | R | R |
| MOV | V | V | | |
| NEG | V | V | R | R |
| TRUNC.L | V | V | | |
| ROUND.L | V | V | | |
| CEIL.L | V | V | | |
| FLOOR.L | V | V | | |
| TRUNC.W | V | V | | |
| ROUND.W | V | V | | |
| CEIL.W | V | V | | |
| FLOOR.W | V | V | | |
| CVT.S | | V | V | V |
| CVT.D | V | | V | V |
| CVT.W | V | V | | |
| CVT.L | V | V | | |
| C | V | V | R | R |

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*

| Condition | | | Relations | | | | Invalid Operation Exception If Unordered |
|---|---|---|---|---|---|---|---|
| Mnemonic | | Code | Greater Than | Less Than | Equal | Unordered | |
| True | False | | | | | | |
| F | T | 0 | F | F | F | F | No |
| UN | OR | 1 | F | F | F | T | No |
| EQ | NEQ | 2 | F | F | T | F | No |
| UEQ | OGL | 3 | F | F | T | T | No |
| OLT | UGE | 4 | F | T | F | F | No |
| ULT | OGE | 5 | F | T | F | T | No |
| OLE | UGT | 6 | F | T | T | F | No |
| ULE | OGT | 7 | F | T | T | T | No |
| SF | ST | 8 | F | F | F | F | Yes |
| NGLE | GLE | 9 | F | F | F | T | Yes |
| SEQ | SNE | 10 | F | F | T | F | Yes |
| NGL | GL | 11 | F | F | T | T | Yes |
| LT | NLT | 12 | F | T | F | F | Yes |
| NGE | GE | 13 | F | T | F | T | Yes |
| LE | NLE | 14 | F | T | T | F | Yes |
| NGT | GT | 15 | F | T | T | T | Yes |

## 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 *fs, ft, immediate*, and so on) are shown in lower-case. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs = base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

In some instructions, the instruction subfields *op* and *function* can have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. For instance, in the floating-point ADD instruction we use *op* = COP1 and *function* = ADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Bit encodings for mnemonics are shown 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:

Example #1:

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

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:

$$(immediate_{15})^{16} \,||\, immediate_{15...0}$$

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.

## 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 *FGR*s.

- When the *FR* bit in the *Status* register equals zero, the *Floating-Point General* registers (*FGR*s) are 32-bits wide.
- When the *FR* bit in the *Status* register equals one, the *Floating-Point General* registers (*FGR*s) 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 B-3 Load and Store Common Functions*

| Function | Meaning |
|---|---|
| AddressTranslation | 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. |
| LoadMemory | 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. |
| StoreMemory | 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. |

Figure B-1 shows the I-Type instruction format used by load and store operations.

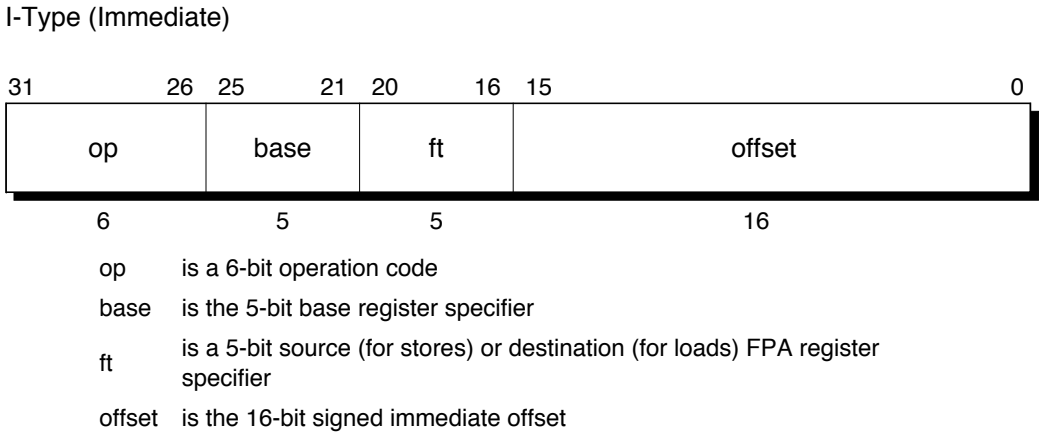



*Figure B-1 Load and Store Instruction Format*

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.

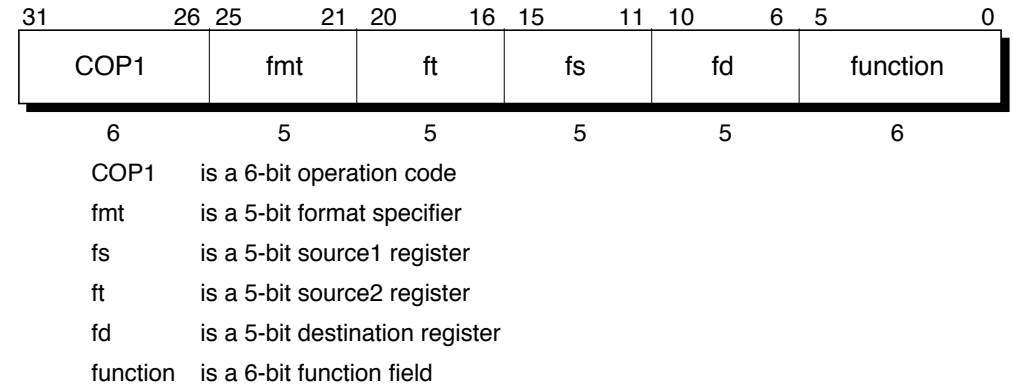



*Figure B-2 Computational Instruction Format*

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 B-4 Format Field Decoding*

| Code | Mnemonic | Size | Format |
|---|---|---|---|
| 16 | S | single | Binary floating-point |
| 17 | D | double | Binary floating-point |
| 18 | Reserved | | |
| 19 | Reserved | | |
| 20 | W | single | 32-bit binary fixed-point |
| 21 | L | longword | 64-bit binary fixed-point |
| 22–31 | Reserved | | |

Table B-5 lists all floating-point instructions.

*Table B-5 Floating-Point Instructions and Operations*

| Code (5: 0) | Mnemonic | Operation |
|---|---|---|
| 0 | ADD | Add |
| 1 | SUB | Subtract |
| 2 | MUL | Multiply |
| 3 | DIV | Divide |
| 4 | SQRT | Square root |
| 5 | ABS | Absolute value |
| 6 | MOV | Move |
| 7 | NEG | Negate |
| 8 | ROUND.L | Convert to 64-bit (long) fixed-point, rounded to nearest/even |
| 9 | TRUNC.L | Convert to 64-bit (long) fixed-point, rounded toward zero |
| 10 | CEIL.L | Convert to 64-bit (long) fixed-point, rounded to +∞ |
| 11 | FLOOR.L | Convert to 64-bit (long) fixed-point, rounded to -∞ |
| 12 | ROUND.W | Convert to single fixed-point, rounded to nearest/even |
| 13 | TRUNC.W | Convert to single fixed-point, rounded toward zero |
| 14 | CEIL.W | Convert to single fixed-point, rounded to + ∞ |
| 15 | FLOOR.W | Convert to single fixed-point, rounded to – ∞ |
| 16–31 | – | Reserved |
| 32 | CVT.S | Convert to single floating-point |
| 33 | CVT.D | Convert to double floating-point |
| 34 | – | Reserved |
| 35 | – | Reserved |
| 36 | CVT.W | Convert to 32-bit binary fixed-point |
| 37 | CVT.L | Convert to 64-bit (long) binary fixed-point |
| 38–47 | – | Reserved |
| 48–63 | C | Floating-point compare |

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:

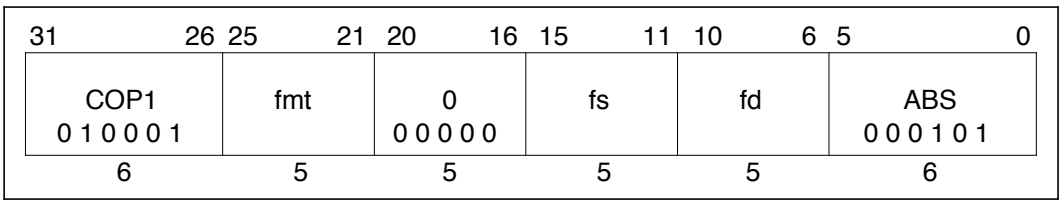
```
value ← ValueFPR(fpr,fmt)

 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 SR26 = 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 fpr0 = 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] ← value63...32
            FGR[fpr] ← value31...0
            return
    endcase
else   /* undefined result for odd 32-bit reg #s */
    undefined_result
endif
```

# ABS.fmt Floating-Point Absolute Value ABS.fmt




**Format:**

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:    StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

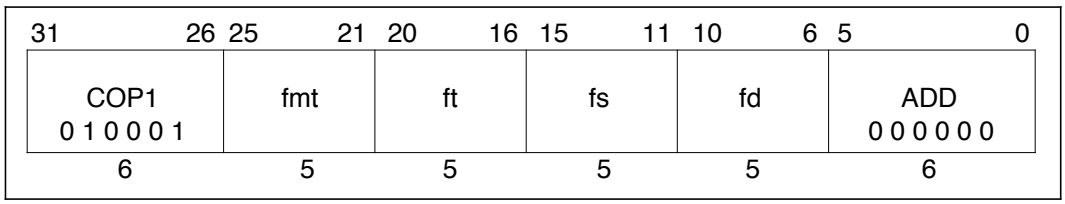
**Exceptions:**

Coprocessor unusable exception
Coprocessor exception trap

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception

# ADD.fmt Floating-Point Add ADD.fmt




**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))
```

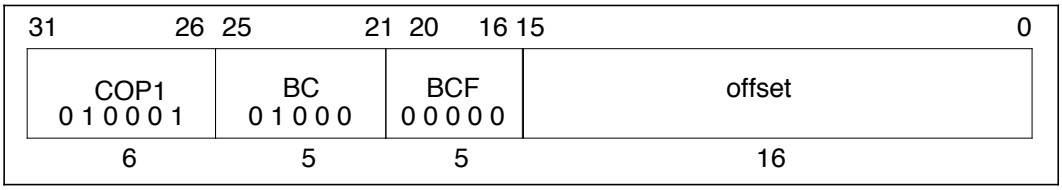
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

# BC1F Branch On FPA False (Coprocessor 1) BC1F




**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:**

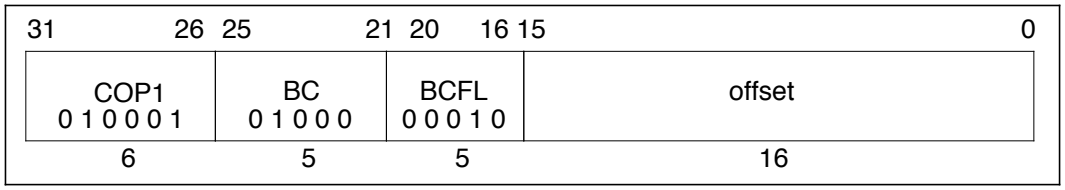
```
32      T–1:    condition ← not COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif

64      T–1:    condition ← not COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif
```

**Exceptions:**

Coprocessor unusable exception

# BC1FL Branch On FPU False Likely (Coprocessor 1) BC1FL




**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.

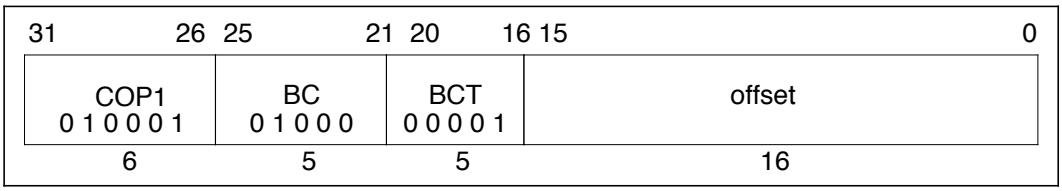
**Operation:**

```
32      T–1:    condition ← not COC[1]
        T:      target ← (offset15)14 || offset || 02
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
64      T–1:    condition ← not COC[1]
        T:      target ← (offset15)46 || offset || 02
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
```

**Exceptions:**

Coprocessor unusable exception

# BC1T — Branch On FPU True (Coprocessor 1) — BC1T




**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:**

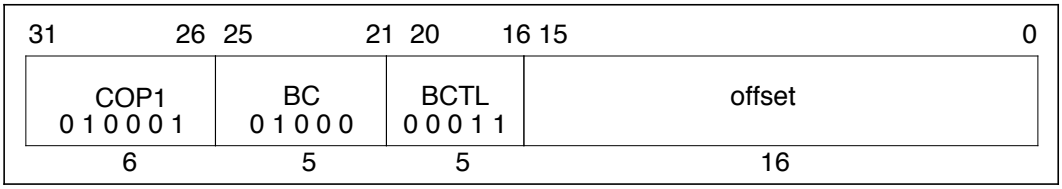
```
32      T–1:    condition ← COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif

64      T–1:    condition ← COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif
```

**Exceptions:**

Coprocessor unusable exception

# BC1TL Branch On FPU True Likely (Coprocessor 1) BC1TL




**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.

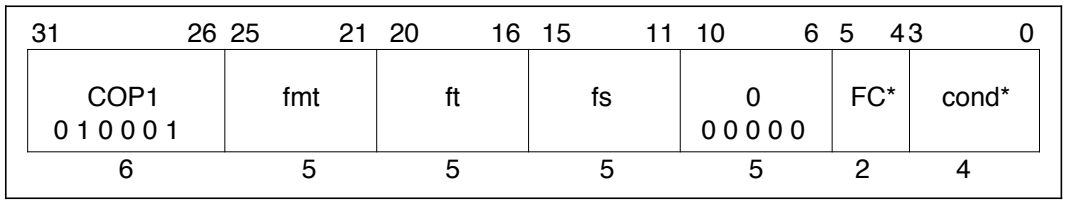
**Operation:**

```
32      T–1:    condition ← COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
64      T–1:    condition ← COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
```

**Exceptions:**

Coprocessor unusable exception

# C.cond.fmt Floating-Point Compare C.cond.fmt




**Format:**

C.cond.fmt fs, ft

**Description:**

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified format, *fmt*, and arithmetically compared.

A result is determined based on the comparison and the conditions specified in the *cond* field. If one of the values is a Not a Number (NaN), and the high-order bit of the *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 *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.

*See “FPU Instruction Opcode Bit Encoding” at the end of Appendix B.

# C.cond.fmt — Floating-Point Compare (continued) — C.cond.fmt

**Operation:**

```
T:      if NaN(ValueFPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
                less ← false
                equal ← false
                unordered ← true
                if cond3 then
                    signal InvalidOperationException
                endif
        else
                less ← ValueFPR(fs, fmt) < ValueFPR(ft, fmt)
                equal ← ValueFPR(fs, fmt) = ValueFPR(ft, fmt)
                unordered ← false
        endif
        condition ← (cond2 and less) or (cond1 and equal) or
                        (cond0 and unordered)
        FCR[31]23 ← condition
        COC[1] ← condition
```

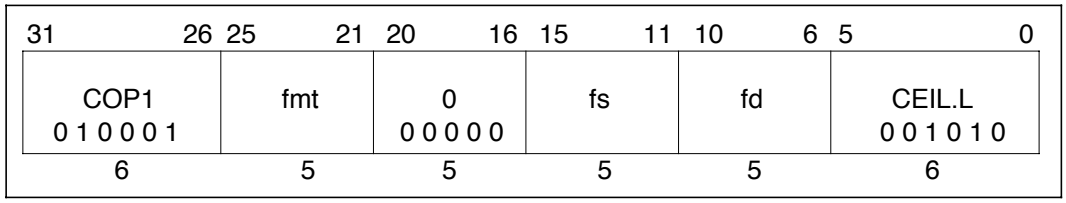
**Exceptions:**

Coprocessor unusable
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception

# CEIL.L.fmt Floating-Point Ceiling to Long Fixed-Point Format CEIL.L.fmt




**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 +∞ (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) CEIL.L.fmt

**Operation:**

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

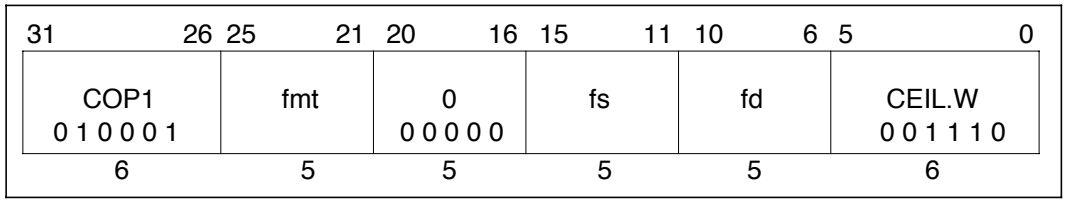
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# CEIL.W.fmt Floating-Point Ceiling to Single Fixed-Point Format CEIL.W.fmt




**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) CEIL.W.fmt

**Operation:**

```
T:      StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

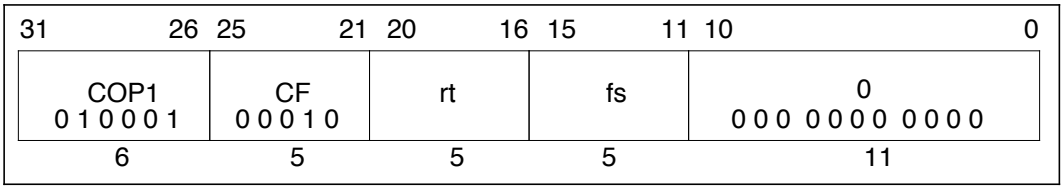
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# CFC1 Move Control Word From FPU (Coprocessor 1) CFC1




**Format:**

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:**

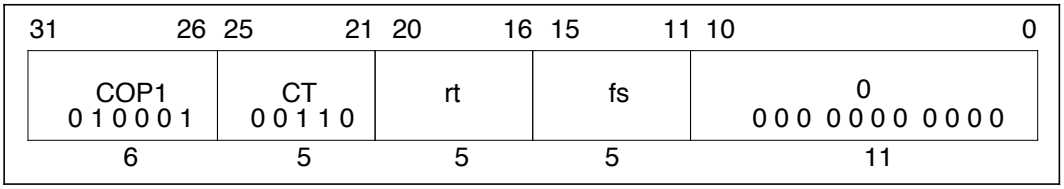
```
32      T:    temp ← FCR[fs]
        T+1:  GPR[rt] ← temp

64      T:    temp ← FCR[fs]
        T+1:  GPR[rt] ← (temp31)32 || temp
```

**Exceptions:**

Coprocessor unusable exception

# CTC1 Move Control Word To FPU (Coprocessor 1) CTC1




**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]
        T+1:    FCR[fs] ← temp
                COC[1] ← FCR[31]23

64      T:      temp ← GPR[rt]31...0
        T+1:    FCR[fs] ← temp
                COC[1] ← FCR[31]23
```

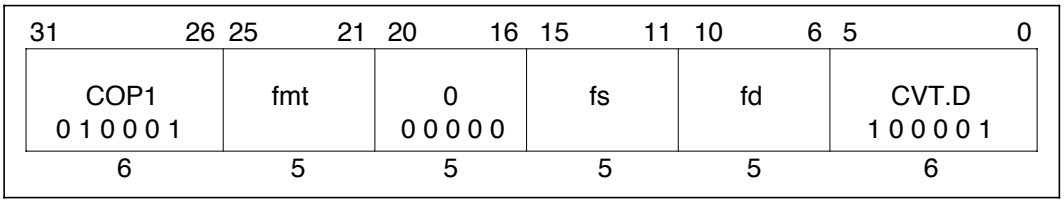
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Division by zero exception
Inexact exception
Overflow exception
Underflow exception

# CVT.D.fmt Floating-Point Convert to Double Floating-Point Format CVT.D.fmt




**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:**

```
T:    StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
```

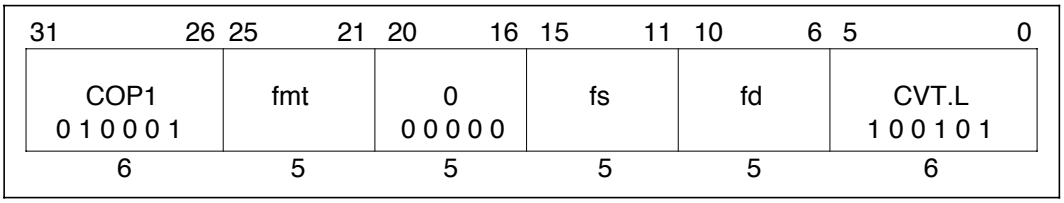
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

# CVT.L.fmt Floating-Point Convert to Long Fixed-Point Format CVT.L.fmt




**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))
```

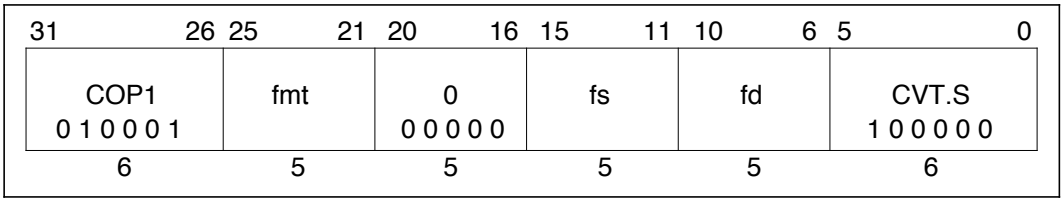
**Exceptions:**

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 CVT.S.fmt




**Format:**

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:    StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))
```

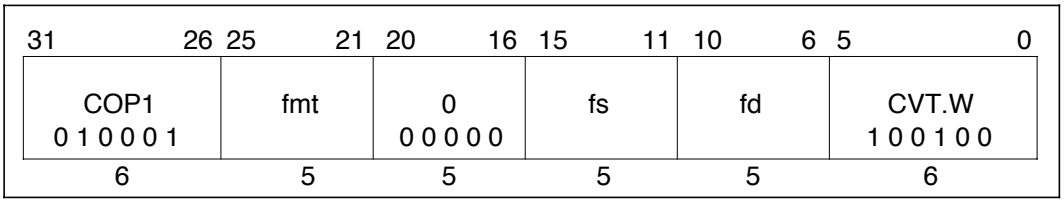
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

# CVT.W.fmt Floating-Point Convert to Fixed-Point Format CVT.W.fmt




**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))
```

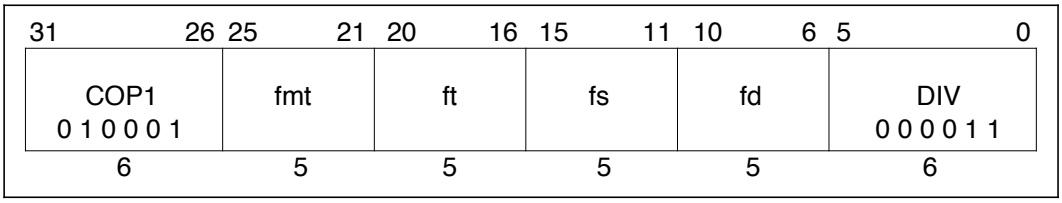
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# DIV.fmt Floating-Point Divide DIV.fmt




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

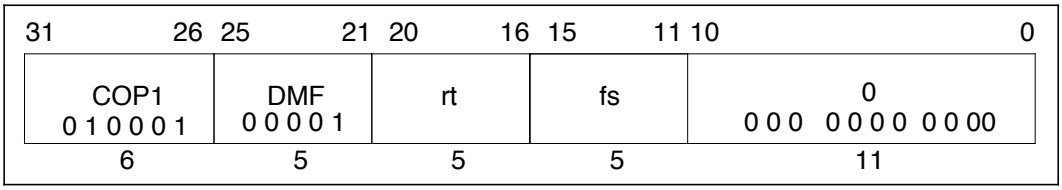
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Division-by-zero exception
Inexact exception
Overflow exception
Underflow exception

# DMFC1 Doubleword Move From Floating-Point Coprocessor DMFC1




**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]
                elseif  fs0 = 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
```

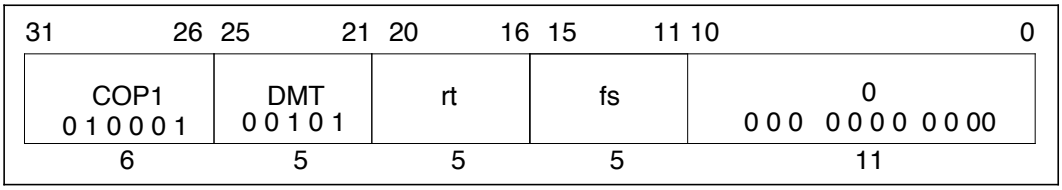
**Exceptions:**

Coprocessor unusable exception

**Coprocessor Exceptions:**

Unimplemented operation exception

# DMTC1 Doubleword Move To Floating-Point Coprocessor DMTC1




**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:**

```
64    T:    data ← GPR[rt]

      T+1:  if SR26 = 1 then   /* 64-bit wide FGRs */
                FGR[fs] ← data
            elseif 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
```

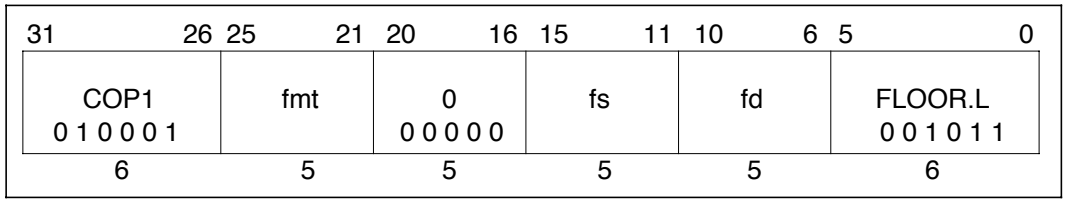
**Exceptions:**

Coprocessor unusable exception

**Coprocessor Exceptions:**

Unimplemented operation exception

# FLOOR.L.fmt Floating-Point Floor to Long Fixed-Point Format FLOOR.L.fmt




**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 -∞ (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) FLOOR.L.fmt

**Operation:**

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

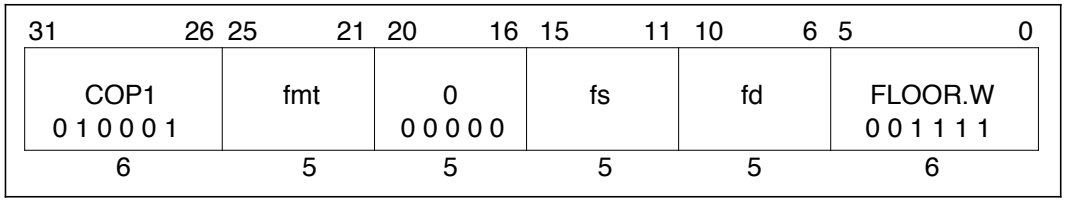
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# FLOOR.W.fmt Floating-Point Floor to Single Fixed-Point Format FLOOR.W.fmt




**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) FLOOR.W.fmt

**Operation:**

```
T:      StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

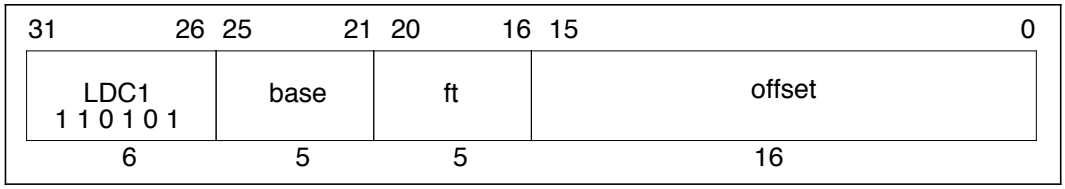
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# LDC1 Load Doubleword to FPU (Coprocessor 1) LDC1




**Format:**

LDC1 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 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.

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.

The *FR* bit of the *Status* register ($SR_{26}$) 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.

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

# LDC1 Load Doubleword to FPU (Coprocessor 1) (continued) LDC1

**Operation:**

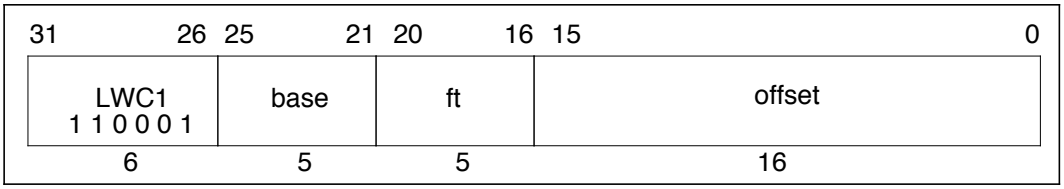
```
32      T:      vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
64      T:      vAddr ← ((offset15)^48 || offset15...0) + GPR[base]

32, 64          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                data ← LoadMemory(uncached, DOUBLEWORD, pAddr, vAddr, DATA)
                if SR26 = 1 then   /* 64-bit wide FGRs */
                    FGR[ft] ← data
                elseif ft0 = 0 then   /* valid specifier, 32-bit wide FGRs */
                    FGR[ft+1] ← data63...32
                    FGR[ft] ← data31...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) LWC1




**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) LWC1

**Operation:**

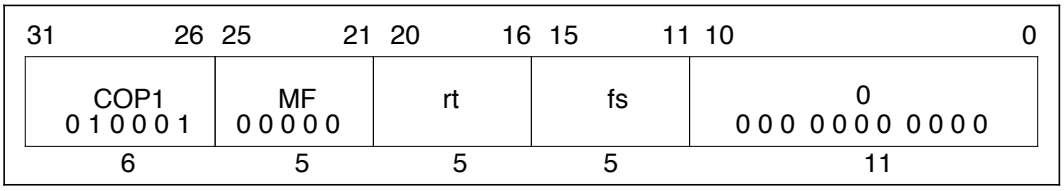
```
32      T:      vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
64      T:      vAddr ← ((offset15)^48 || offset15...0) + GPR[base]

32, 64          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
                mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
                byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
                /* "mem" is aligned 64-bits from memory.  Pick out correct bytes. */
                if SR26 = 1 then   /* 64-bit wide FGRs */
                    FGR[ft] ← undefined^32 || mem31+8*byte...8*byte
                else   /* 32-bit wide FGRs */
                    FGR[ft] ← mem31+8*byte...8*byte
                endif
```

**Exceptions:**

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

# MFC1 Move From FPU (Coprocessor 1) MFC1




**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:**

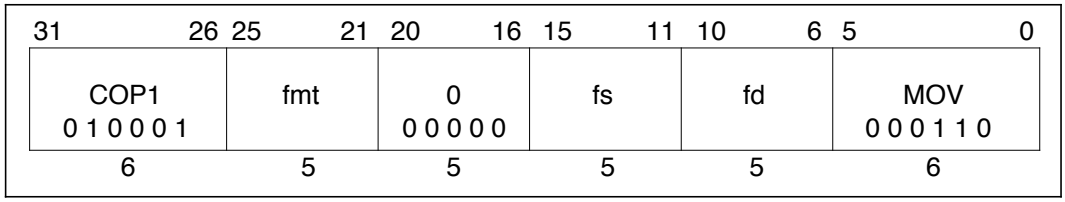
```
32    T:    data ← FGR[fs]31...0
      T+1:  GPR[rt] ← data

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




**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))
```

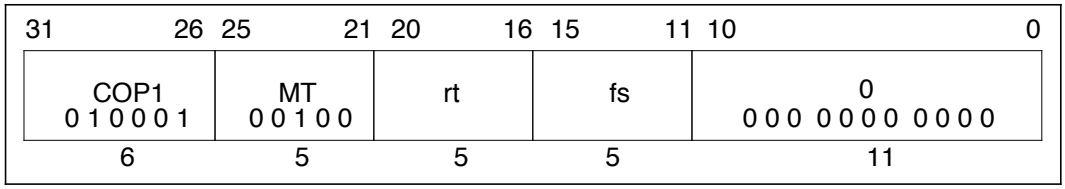
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception

# MTC1 Move To FPU (Coprocessor 1) MTC1




**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.

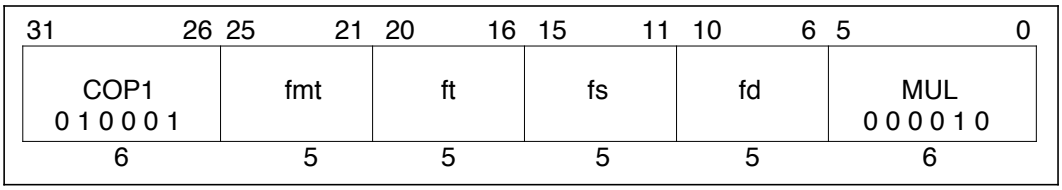
**Operation:**

```
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

# MUL.fmt Floating-Point Multiply MUL.fmt




**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:    StoreFPR (fd, fmt, ValueFPR(fs, fmt) * ValueFPR(ft, fmt))
```

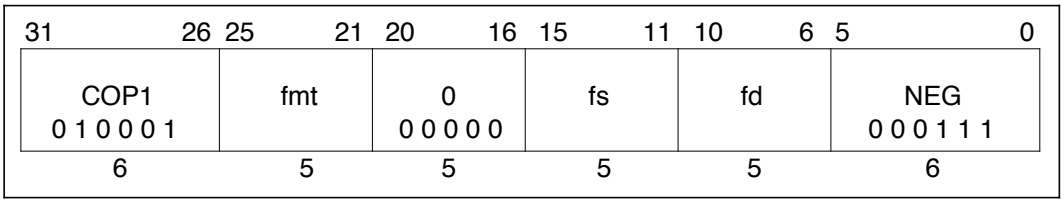
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

# NEG.fmt Floating-Point Negate NEG.fmt




**Format:**

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:    StoreFPR(fd, fmt, Negate(ValueFPR(fs, fmt)))
```

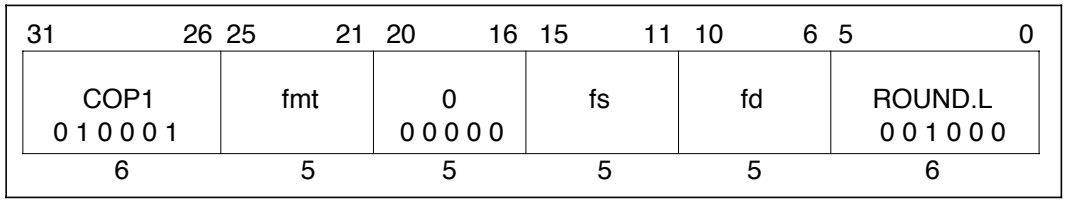
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception

# ROUND.L.fmt Floating-Point Round to Long Fixed-Point Format ROUND.L.fmt




**Format:**

ROUND.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 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) ROUND.L.fmt

**Operation:**

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

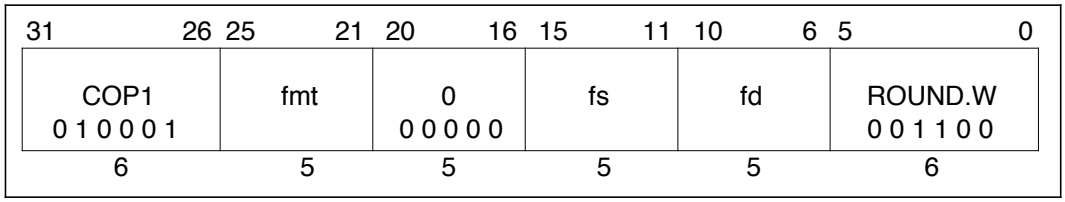
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# ROUND.W.fmt Floating-Point Round to Single Fixed-Point Format ROUND.W.fmt




**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 to Single Fixed-Point Format (continued) ROUND.W.fmt

**Operation:**

```
T:      StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

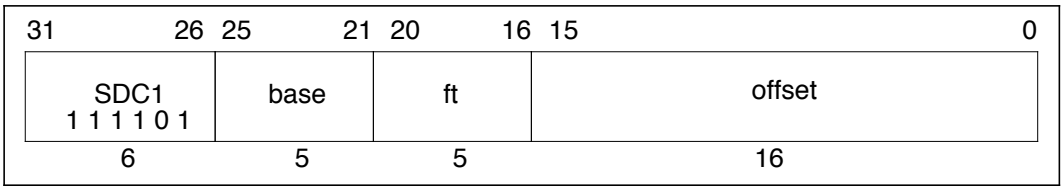
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

# SDC1 Store Doubleword from FPU (Coprocessor 1) SDC1




**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 ($SR_{26}$) 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) SDC1

**Operation:**

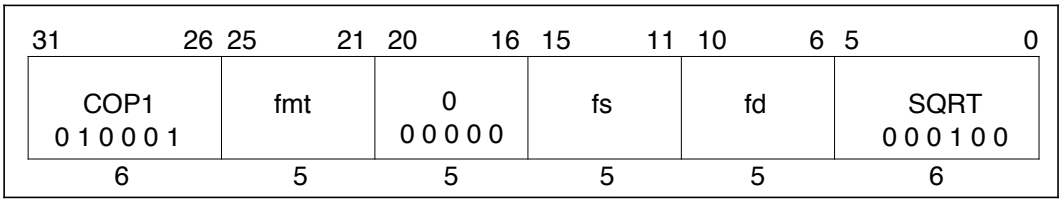
```
32      T:      vAddr ← (offset15)16 || offset15...0) + GPR[base]
64      T:      vAddr ← (offset15)48 || offset15...0) + GPR[base]

32,64           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                if SR26 = 1   /* 64-bit wide FGRs */
                    data ← FGR[ft]
                elseif  ft0 = 0 then   /* valid specifier, 32-bit wide FGRs */
                    data ← FGR[ft+1] || FGR[ft]
                else   /* undefined for odd 32-bit reg #s */
                    data ⇐ undefined64
                endif
                StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

**Exceptions:**

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

# SQRT.fmt Floating-Point Square Root SQRT.fmt




**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)))
```

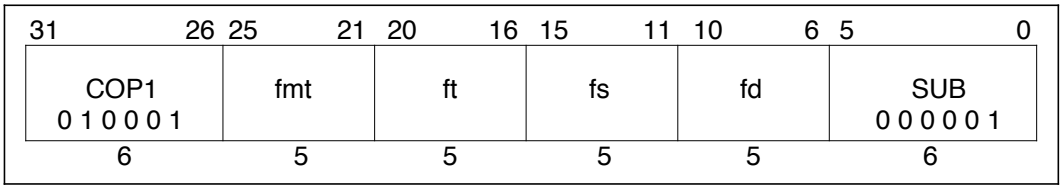
**Exceptions:**

Coprocessor unusable exception
Floating-Point exception

**Coprocessor Exceptions:**

Unimplemented operation exception
Invalid operation exception
Inexact exception

# SUB.fmt Floating-Point Subtract SUB.fmt




**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))
```

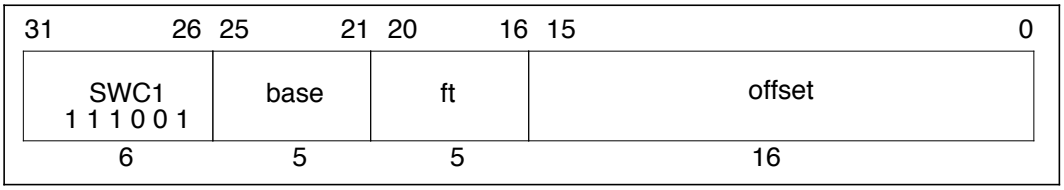
**Exceptions:**

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) SWC1




**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) SWC1

**Operation:**

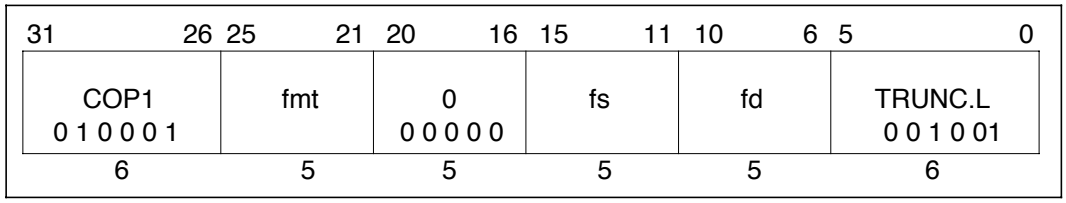
```
32      T:      vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
64      T:      vAddr ← ((offset15)^48 || offset15...0) + GPR[base]

32, 64          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
                byte ← vAddr2...0 xor (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 TRUNC.L.fmt




**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) TRUNC.L.fmt

**Operation:**

```
T:      StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

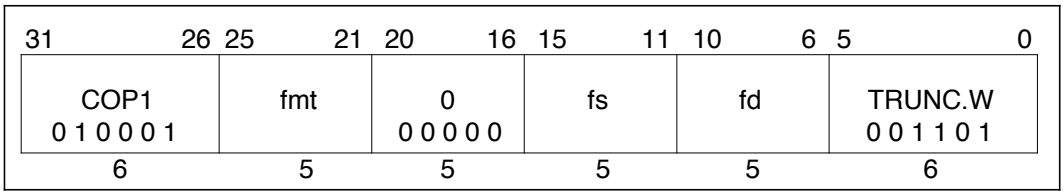
**Exceptions:**

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 TRUNC.W.fmt




**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 Truncate to Single Fixed-Point Format (continued) TRUNC.W.fmt

**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 Opcode Bit Encoding

**Opcode**

| 31...29 \ 28...26 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | COP1 | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | LWC1 | | | | LDC1 | | |
| 7 | | SWC1 | | | | SDC1 | | |

**sub**

| 25...24 \ 23...21 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | MF | DMFη | CF | δ | MT | DMTη | CT | δ |
| 1 | BC | δ | δ | δ | δ | δ | δ | δ |
| 2 | S | D | δ | δ | W | Lη | δ | δ |
| 3 | δ | δ | δ | δ | δ | δ | δ | δ |

**br**

| 20...19 \ 18...16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BCF | BCT | BCFL | BCTL | γ | γ | γ | γ |
| 1 | γ | γ | γ | γ | γ | γ | γ | γ |
| 2 | γ | γ | γ | γ | γ | γ | γ | γ |
| 3 | γ | γ | γ | γ | γ | γ | γ | γ |

*Figure B-3 Bit Encoding for FPU Instructions*

| 5...3 \ 2...0 | function 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | ADD | SUB | MUL | DIV | SQRT | ABS | MOV | NEG |
| 1 | ROUND.Lη | TRUNC.Lη | CEIL.Lη | FLOOR.Lη | ROUND.W | TRUNC.W | CEIL.W | FLOOR.W |
| 2 | δ | δ | δ | δ | δ | δ | δ | δ |
| 3 | δ | δ | δ | δ | δ | δ | δ | δ |
| 4 | CVT.S | CVT.D | δ | δ | CVT.W | CVT.Lη | δ | δ |
| 5 | δ | δ | δ | δ | δ | δ | δ | δ |
| 6 | C.F | C.UN | C.EQ | C.UEQ | C.OLT | C.ULT | C.OLE | C.ULE |
| 7 | C.SF | C.NGLE | C.SEQ | C.NGL | C.LT | C.NGE | C.LE | C.NGT |

*Figure B-3 (cont.) 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.