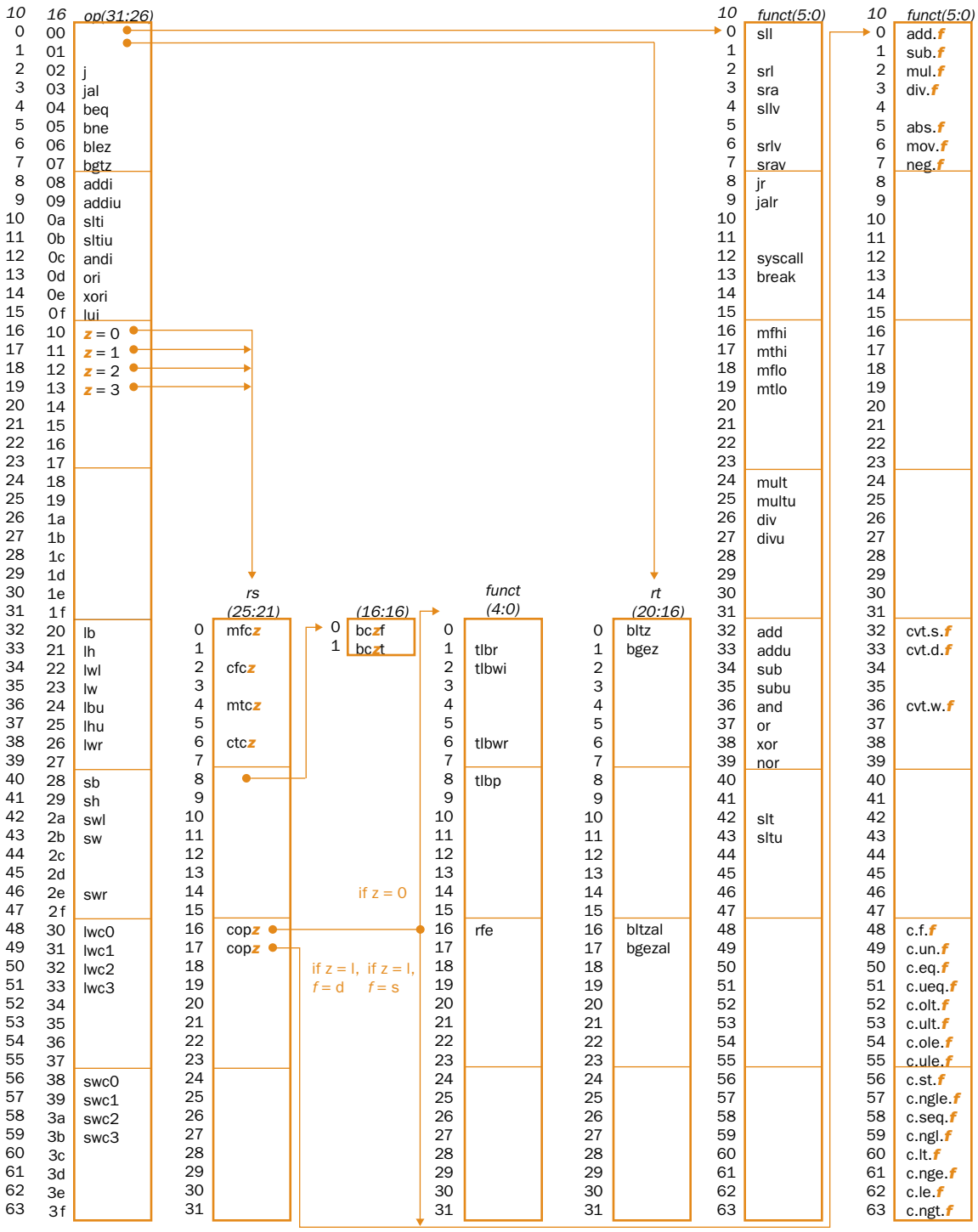


MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10 and the second shows base 16 for the op field (bits 31 to 26) in the third column. This op field completely specifies the MIPS operation except for 6 op values: 0, 1, 16, 17, 18, and 19. These operations are determined by other fields, identified by point-ers. The last field (funct) uses "*f*" to mean "s" if rs = 16 and op = 17 or "d" if rs = 17 and op = 17. The second field (rs) uses "*z*" to mean "0", "1", "2", or "3" if op = 16, 17, 18, or 19, respectively. If rs = 16, the operation is specified elsewhere: if *z* = 0, the operations are specified in the fourth field (bits 4 to 0); if *z* = 1, then the operations are in the last field with *f* = s. If rs = 17 and *z* = 1, then the operations are in the last field with *f* = d.

Arithmetic and Logical Instructions

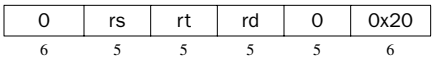
Absolute value

`abs rdest, rsrc` *pseudoinstruction*

Put the absolute value of register `rsrc` in register `rdest`.

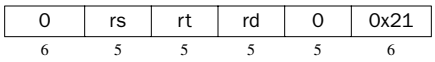
Addition (with overflow)

`add rd, rs, rt`



Addition (without overflow)

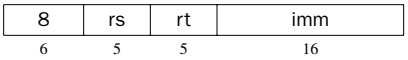
`addu rd, rs, rt`



Put the sum of registers `rs` and `rt` into register `rd`.

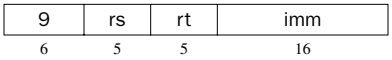
Addition immediate (with overflow)

`addi rt, rs, imm`



Addition immediate (without overflow)

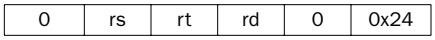
`addiu rt, rs, imm`



Put the sum of register `rs` and the sign-extended immediate into register `rt`.

AND

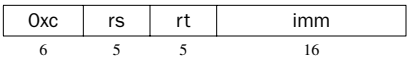
`and rd, rs, rt`



Put the logical AND of registers `rs` and `rt` into register `rd`.

AND immediate

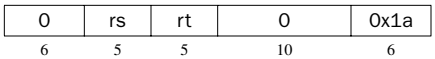
`andi rt, rs, imm`



Put the logical AND of register `rs` and the zero-extended immediate into register `rt`.

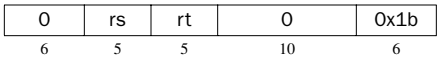
Divide (with overflow)

`div rs, rt`



Divide (without overflow)

`divu rs, rt`



Divide register `rs` by register `rt`. Leave the quotient in register `lo` and the remainder in register `hi`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

Divide (with overflow)

`div rdest, rsrc1, src2` *pseudoinstruction*

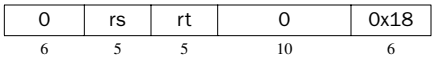
Divide (without overflow)

`divu rdest, rsrc1, src2` *pseudoinstruction*

Put the quotient of register `rsrc1` and `src2` into register `rdest`.

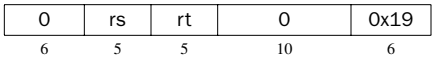
Multiply

`mult rs, rt`



Unsigned multiply

`multu rs, rt`



Multiply registers `rs` and `rt`. Leave the low-order word of the product in register `lo` and the high-order word in register `hi`.

Multiply (without overflow)

`mul rdest, rsrc1, src2` *pseudoinstruction*

Multiply (with overflow)

`mulo rdest, rsrc1, src2` *pseudoinstruction*

Unsigned multiply (with overflow)

`mulou rdest, rsrc1, src2` *pseudoinstruction*

Put the product of register `rsrc1` and `src2` into register `rdest`.

Negate value (with overflow)

`neg rdest, rsrc` *pseudoinstruction*

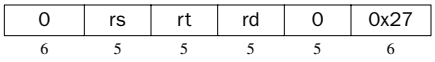
Negate value (without overflow)

`negu rdest, rsrc` *pseudoinstruction*

Put the negative of register `rsrc` into register `rdest`.

NOR

`nor rd, rs, rt`



Put the logical NOR of registers `rs` and `rt` into register `rd`.

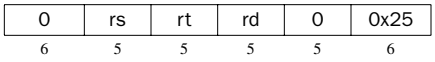
NOT

`not rdest, rsrc` *pseudoinstruction*

Put the bitwise logical negation of register `rsrc` into register `rdest`.

OR

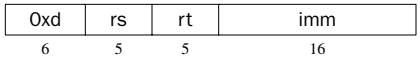
`or rd, rs, rt`



Put the logical OR of registers `rs` and `rt` into register `rd`.

OR immediate

`ori rt, rs, imm`



Put the logical OR of register `rs` and the zero-extended immediate into register `rt`.

Remainder

`rem rdest, rsrc1, rsrc2` *pseudoinstruction*

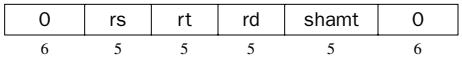
Unsigned remainder

`remu rdest, rsrc1, rsrc2` *pseudoinstruction*

Put the remainder of register `rsrc1` divided by register `rsrc2` into register `rdest`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

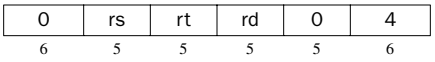
Shift left logical

`sll rd, rt, shamt`



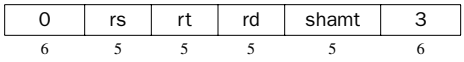
Shift left logical variable

`sllv rd, rt, rs`



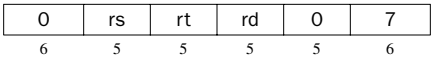
Shift right arithmetic

`sra rd, rt, shamt`



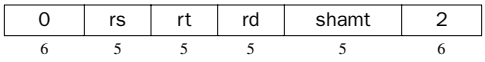
Shift right arithmetic variable

`srav rd, rt, rs`



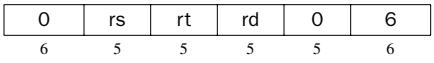
Shift right logical

`srl rd, rt, shamt`



Shift right logical variable

`srlv rd, rt, rs`



Shift register `rt` left (right) by the distance indicated by immediate `shamt` or the register `rs` and put the result in register `rd`. Note that argument `rs` is ignored for `sll`, `sra`, and `srl`.

Rotate left

`rol rdest, rsrc1, rsrc2` *pseudoinstruction*

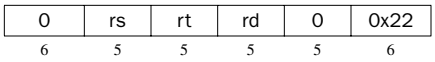
Rotate right

```
ror rdest, rsrc1, rsrc2          pseudoinstruction
```

Rotate register `rsrc1` left (right) by the distance indicated by `rsrc2` and put the result in register `rdest`.

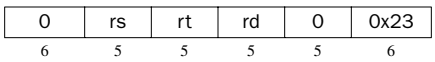
Subtract (with overflow)

`sub rd, rs, rt`



Subtract (without overflow)

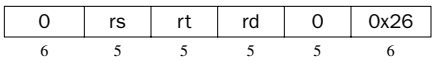
`subu rd, rs, rt`



Put the difference of registers `rs` and `rt` into register `rd`.

Exclusive OR

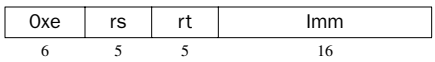
`xor rd, rs, rt`



Put the logical XOR of registers `rs` and `rt` into register `rd`.

XOR immediate

`xori rt, rs, imm`

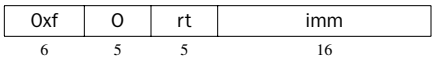


Put the logical XOR of register `rs` and the zero-extended immediate into register `rt`.

Constant-Manipulating Instructions

Load upper immediate

`lui rt, imm`



Load the lower halfword of the immediate `imm` into the upper halfword of register `rt`. The lower bits of the register are set to 0.

Load immediate

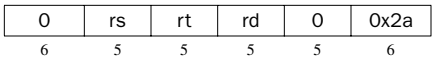
```
li rdest, imm                    pseudoinstruction
```

Move the immediate `imm` into register `rdest`.

Comparison Instructions

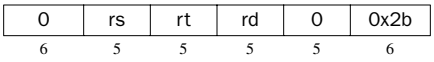
Set less than

`slt rd, rs, rt`



Set less than unsigned

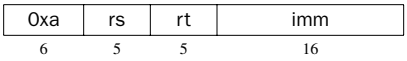
`sltu rd, rs, rt`



Set register `rd` to 1 if register `rs` is less than `rt`, and to 0 otherwise.

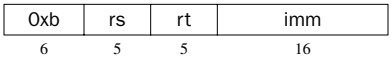
Set less than immediate

`slti rt, rs, imm`



Set less than unsigned immediate

`sltiu rt, rs, imm`



Set register `rt` to 1 if register `rs` is less than the sign-extended immediate, and to 0 otherwise.

Set equal

`seq rdest, rsrc1, rsrc2` *pseudoinstruction*

Set register `rdest` to 1 if register `rsrc1` equals `rsrc2`, and to 0 otherwise.

Set greater than equal

`sge rdest, rsrc1, rsrc2` *pseudoinstruction*

Set greater than equal unsigned

`sgeu rdest, rsrc1, rsrc2` *pseudoinstruction*

Set register `rdest` to 1 if register `rsrc1` is greater than or equal to `rsrc2`, and to 0 otherwise.

Set greater than

```
sgt rdest, rsrc1, rsrc2          pseudoinstruction
```

Set greater than unsigned

```
sgtu rdest, rsrc1, rsrc2         pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is greater than `rsrc2`, and to 0 otherwise.

Set less than equal

```
sle rdest, rsrc1, rsrc2          pseudoinstruction
```

Set less than equal unsigned

```
sleu rdest, rsrc1, rsrc2         pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is less than or equal to `rsrc2`, and to 0 otherwise.

Set not equal

```
sne rdest, rsrc1, rsrc2          pseudoinstruction
```

Set register `rdest` to 1 if register `rsrc1` is not equal to `rsrc2`, and to 0 otherwise.

Branch Instructions

Branch instructions use a signed 16-bit instruction *offset* field; hence they can jump $2^{15} - 1$ *instructions* (not bytes) forward or 2^{15} instructions backwards. The *jump* instruction contains a 26-bit address field.

In the descriptions below, the offsets are not specified. Instead, the instructions branch to a label. This is the form used in most assembly language programs because the distance between instructions is difficult to calculate when pseudoinstructions expand into several real instructions.

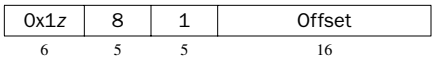
Branch instruction

```
b label                          pseudoinstruction
```

Unconditionally branch to the instruction at the label.

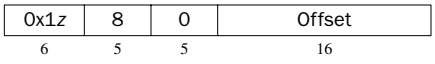
Branch coprocessor *z* true

`bczt label`



Branch coprocessor *z* false

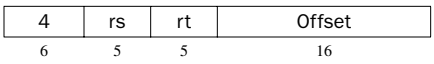
`bczf label`



Conditionally branch the number of instructions specified by the offset if *z*'s condition flag is true (false). *z* is 0, 1, 2, or 3. The floating-point unit is *z* = 1.

Branch on equal

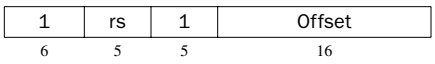
`beq rs, rt, label`



Conditionally branch the number of instructions specified by the offset if register `rs` equals `rt`.

Branch on greater than equal zero

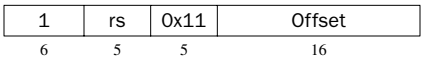
`bgez rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is greater than or equal to 0.

Branch on greater than equal zero and link

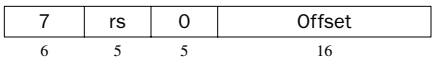
`bgezal rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is greater than or equal to 0. Save the address of the next instruction in register 31.

Branch on greater than zero

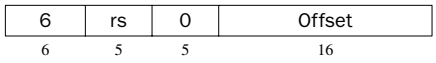
`bgtz rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is greater than 0.

Branch on less than equal zero

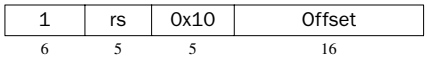
`blez rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is less than or equal to 0.

Branch on less than and link

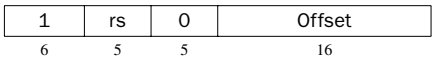
`bltzal rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is less than 0. Save the address of the next instruction in register 31.

Branch on less than zero

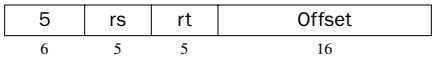
`bltz rs, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is less than 0.

Branch on not equal

`bne rs, rt, label`



Conditionally branch the number of instructions specified by the offset if register `rs` is not equal to `rt`.

Branch on equal zero

`beqz rsrc, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if `rsrc` equals 0.

Branch on greater than equal

`bge rsrc1, rsrc2, label` *pseudoinstruction*

Branch on greater than equal unsigned

`bgeu rsrc1, rsrc2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is greater than or equal to `rsrc2`.

Branch on greater than

`bgt rsrc1, src2, label` *pseudoinstruction*

Branch on greater than unsigned

`bgtu rsrc1, src2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is greater than `src2`.

Branch on less than equal

`ble rsrc1, src2, label` *pseudoinstruction*

Branch on less than equal unsigned

`bleu rsrc1, src2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is less than or equal to `src2`.

Branch on less than

`blt rsrc1, rsrc2, label` *pseudoinstruction*

Branch on less than unsigned

`bltu rsrc1, rsrc2, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc1` is less than `rsrc2`.

Branch on not equal zero

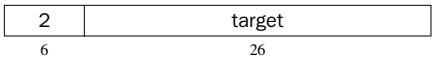
`bnez rsrc, label` *pseudoinstruction*

Conditionally branch to the instruction at the label if register `rsrc` is not equal to 0.

Jump Instructions

Jump

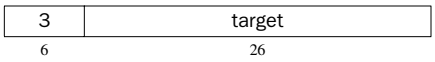
`j target`



Unconditionally jump to the instruction at target.

Jump and link

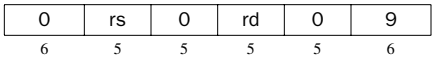
`jal target`



Unconditionally jump to the instruction at target. Save the address of the next instruction in register `$ra`.

Jump and link register

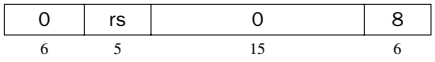
`jalr rs, rd`



Unconditionally jump to the instruction whose address is in register `rs`. Save the address of the next instruction in register `rd` (which defaults to 31).

Jump register

`jr rs`



Unconditionally jump to the instruction whose address is in register `rs`.

Load Instructions

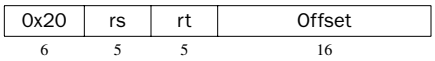
Load address

`la rdest, address` *pseudoinstruction*

Load computed *address*—not the contents of the location—into register `rdest`.

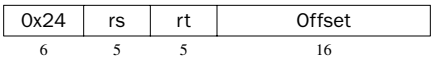
Load byte

`lb rt, address`



Load unsigned byte

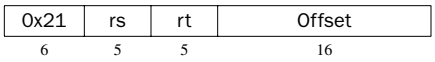
`lbu rt, address`



Load the byte at *address* into register `rt`. The byte is sign-extended by `lb`, but not by `lbu`.

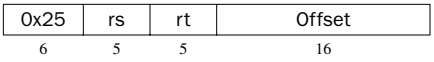
Load halfword

`lh rt, address`



Load unsigned halfword

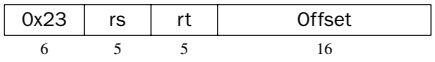
`lhu rt, address`



Load the 16-bit quantity (halfword) at *address* into register `rt`. The halfword is sign-extended by `lh`, but not by `lhu`.

Load word

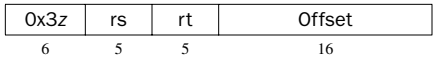
`lw rt, address`



Load the 32-bit quantity (word) at *address* into register `rt`.

Load word coprocessor

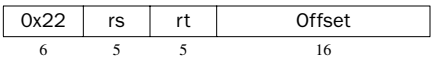
`lwcz rt, address`



Load the word at *address* into register `rt` of coprocessor `z` (0–3). The floating-point unit is *z* = 1.

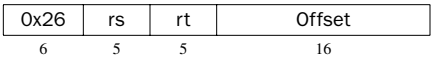
Load word left

`lwl rt, address`



Load word right

`lwr rt, address`



Load the left (right) bytes from the word at the possibly unaligned *address* into register `rt`.

Load doubleword

`ld rdest, address` *pseudoinstruction*

Load the 64-bit quantity at *address* into registers `rdest` and `rdest + 1`.

Unaligned load halfword

`ulh rdest, address` *pseudoinstruction*

Unaligned load halfword unsigned

`ulhu rdest, address` *pseudoinstruction*

Load the 16-bit quantity (halfword) at the possibly unaligned *address* into register `rdest`. The halfword is sign-extended by `ulh`, but not `ulhu`.

Unaligned load word

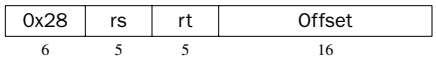
`ulw rdest, address` *pseudoinstruction*

Load the 32-bit quantity (word) at the possibly unaligned *address* into register `rdest`.

Store Instructions

Store byte

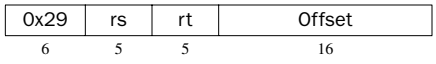
`sb rt, address`



Store the low byte from register `rt` at *address.*

Store halfword

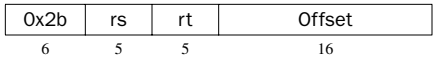
`sh rt, address`



Store the low halfword from register `rt` at *address*.

Store word

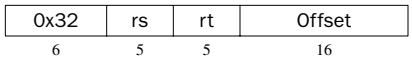
`sw rt, address`



Store the word from register `rt` at *address*.

Store word coprocessor

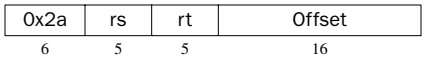
`swcz rt, address`



Store the word from register `rt` of coprocessor `z` at *address*. The floating-point unit is `z` = 1.

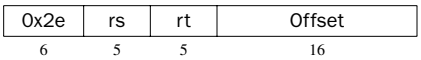
Store word left

`swl rt, address`



Store word right

`swr rt, address`



Store the left (right) bytes from register `rt` at the possibly unaligned *address*.

Store doubleword

`sd rsrc, address` *pseudoinstruction*

Store the 64-bit quantity in registers `rsrc` and `rsrc + 1` at *address*.

Unaligned store halfword

`ush rsrc, address` *pseudoinstruction*

Store the low halfword from register `rsrc` at the possibly unaligned *address*.

Unaligned store word

`usw rsrc, address` *pseudoinstruction*

Store the word from register `rsrc` at the possibly unaligned *address*.

Data Movement Instructions

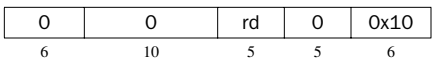
Move

`move rdest, rsrc` *pseudoinstruction*

Move register `rsrc` to `rdest`.

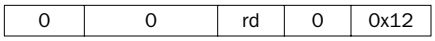
Move from hi

`mfhi rd`



Move from lo

`mflo rd`



The multiply and divide unit produces its result in two additional registers, `hi` and `lo`. These instructions move values to and from these registers. The multiply, divide, and remainder pseudoinstructions that make this unit appear to operate on the general registers move the result after the computation finishes.

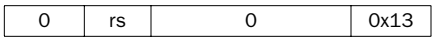
Move the `hi` (`lo`) register to register `rd`.

Move to hi

`mthi rs`

Move to lo

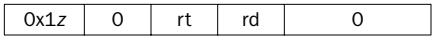
`mtlo rs`



Move register `rs` to the `hi` (`lo`) register.

Move from coprocessor *z*

`mfcz rt, rd`



Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

Move coprocessor *z*'s register `rd` to CPU register `rt`. The floating-point unit is coprocessor *z* = 1.

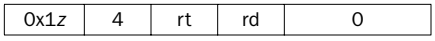
Move double from coprocessor 1

`mfc1.d rdest, frsrc1` *pseudoinstruction*

Move floating-point registers `frsrc1` and `frsrc1 + 1` to CPU registers `rdest` and `rdest + 1`.

Move to coprocessor *z*

`mtcz rd, rt`



Move CPU register `rt` to coprocessor *z*'s register `rd`.

Floating-Point Instructions

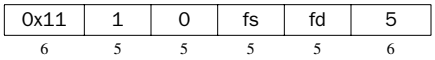
The MIPS has a floating-point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating-point numbers. This coprocessor has its own registers, which are numbered `$f0`–`$f31`. Because these registers are only 32 bits wide, two of them are required to hold doubles, so only floating-point registers with even numbers can hold double precision values.

Values are moved in or out of these registers one word (32 bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` instructions described above or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudoinstructions described below. The flag set by floating-point comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions.

In the actual instructions below, bits 21–26 are 0 for single precision and 1 for double precision. In the pseudoinstructions below, `fdest` is a floating-point register (e.g., `$f2`).

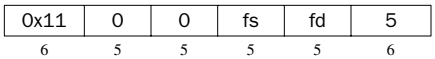
Floating-point absolute value double

`abs.d fd, fs`



Floating-point absolute value single

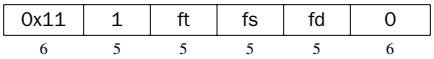
`abs.s fd, fs`



Compute the absolute value of the floating-point double (single) in register `fs` and put it in register `fd`.

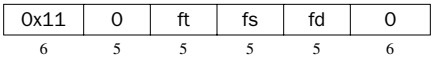
Floating-point addition double

`add.d fd, fs, ft`



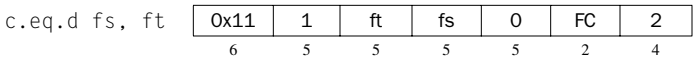
Floating-point addition single

`add.s fd, fs, ft`

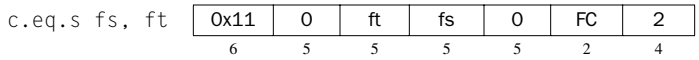


Compute the sum of the floating-point doubles (singles) in registers `fs` and `ft` and put it in register `fd`.

Compare equal double

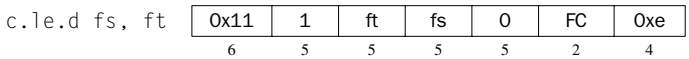


Compare equal single

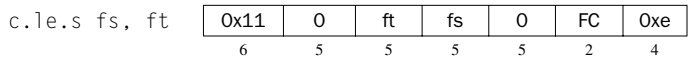


Compare the floating-point double in register `fs` against the one in `ft` and set the floating-point condition flag true if they are equal. Use the `bc1t` or `bc1f` instructions to test the value of this flag.

Compare less than equal double

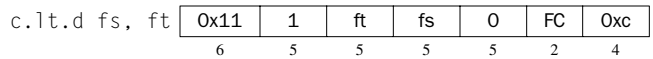


Compare less than equal single

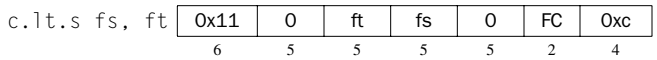


Compare the floating-point double in register `fs` against the one in `ft` and set the floating-point condition flag true if the first is less than or equal to the second. Use the `bc1t` or `bc1f` instructions to test the value of this flag.

Compare less than double

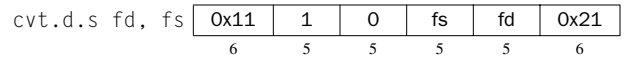


Compare less than single

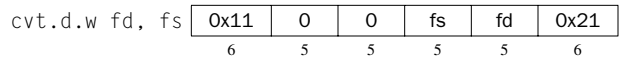


Compare the floating-point double in register `fs` against the one in `ft` and set the condition flag true if the first is less than the second. Use the `bc1t` or `bc1f` instructions to test the value of this flag.

Convert single to double

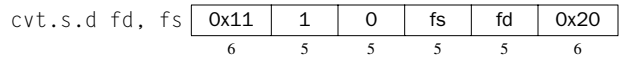


Convert integer to double

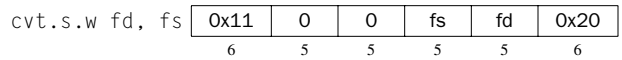


Convert the single precision floating-point number or integer in register `fs` to a double precision number and put it in register `fd`.

Convert double to single

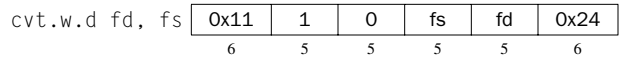


Convert integer to single

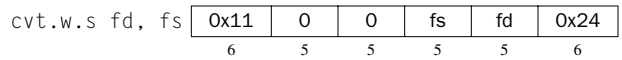


Convert the double precision floating-point number or integer in register `fs` to a single precision number and put it in register `fd`.

Convert double to integer

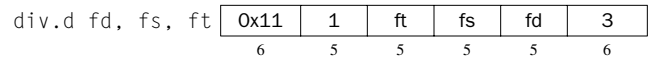


Convert single to integer

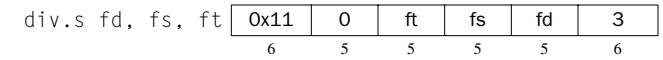


Convert the double or single precision floating-point number in register `fs` to an integer and put it in register `fd`.

Floating-point divide double



Floating-point divide single



Compute the quotient of the floating-point doubles (singles) in registers `fs` and `ft` and put it in register `fd`.

Load floating-point double

`l.d fdest, address` *pseudoinstruction*

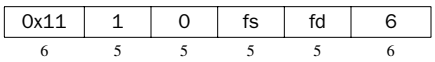
Load floating-point single

`l.s fdest, address` *pseudoinstruction*

Load the floating-point double (single) at `address` into register `fdest`.

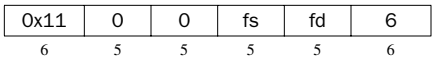
Move floating-point double

`mov.d fd, fs`



Move floating-point single

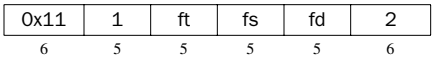
`mov.s fd, fs`



Move the floating-point double (single) from register `fs` to register `fd`.

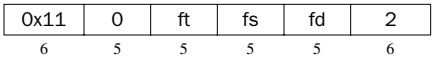
Floating-point multiply double

`mul.d fd, fs, ft`



Floating-point multiply single

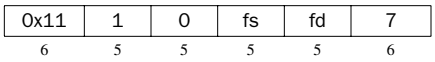
`mul.s fd, fs, ft`



Compute the product of the floating-point doubles (singles) in registers `fs` and `ft` and put it in register `fd`.

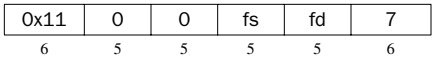
Negate double

`neg.d fd, fs`



Negate single

`neg.s fd, fs`



Negate the floating-point double (single) in register `fs` and put it in register `fd`.

Store floating-point double

`s.d fdest, address` *pseudoinstruction*

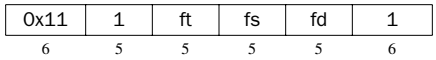
Store floating-point single

`s.s fdest, address` *pseudoinstruction*

Store the floating-point double (single) in register `fdest` at `address`.

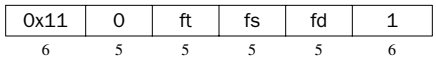
Floating-point subtract double

`sub.d fd, fs, ft`



Floating-point subtract single

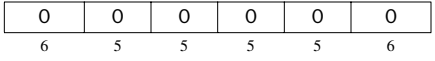
`sub.s fd, fs, ft`



Compute the difference of the floating-point doubles (singles) in registers `fs` and `ft` and put it in register `fd`.

No operation

nop



Do nothing.