SPIM Instructions

Instructions marked with a dagger (†) are pseudoinstructions.

Arithmetic Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., `add`) into the immediate form (e.g., `addi`) if the second argument is a constant.

Absolute Value

Put the absolute value of the integer from register `Rsrc` in register `Rdest`:

`abs Rdest, Rsrc` *Absolute Value* †

Add

Put the sum of the integers from registers `Rs` and `Rt` (or `Imm`) into register `Rd`:

`add Rd, Rs, Rt` *Addition (with overflow)*

0	Rs	Rt	Rd	0	0x20
6	5	5	5	5	6

`addu Rd, Rs, Rt` *Addition (without overflow)*

0	Rs	Rt	Rd	0	0x21
6	5	5	5	5	6

`addi Rt, Rs, Imm` *Addition Immediate (with overflow)*

8	Rs	Rt	Imm
6	5	5	16

`addiu Rt, Rs, Imm` *Addition Immediate (without overflow)*

9	Rs	Rt	Imm
6	5	5	16

Subtract

Put the difference of the integers from register `Rs` and `Rt` into register `Rd`:

`sub Rd, Rs, Rt` *Subtract (with overflow)*

0	Rs	Rt	Rd	0	0x22
6	5	5	5	5	6

`subu Rd, Rs, Rt` *Subtract (without overflow)*

0	Rs	Rt	Rd	0	0x23
6	5	5	5	5	6

Multiply

Put the product of registers `Rsrc1` and `Src2` into register `Rdest`:

`mul Rdest, Rsrc1, Src2` *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` *Multiply (with overflow)* †

`mulou Rdest, Rsrc1, Src2` *Unsigned Multiply (with overflow)* †

Multiply the contents of registers `Rs` and `Rt`. Leave the low-order word of the product in register `lo` and the high-word in register `hi`:

`mult Rs, Rt` *Multiply*

0	Rs	Rt	0	0x18
6	5	5	10	6

`multu Rs, Rt` *Unsigned Multiply*

0	Rs	Rt	0	0x19
6	5	5	10	6

Divide

Divide the integer in register `Rs` by the integer in register `Rt`. Leave the quotient in register `lo` and the remainder in register `hi`:

`div Rs, Rt` *Divide (with overflow)*

0	Rs	Rt	0	0x1a
6	5	5	10	6

`divu Rs, Rt` *Divide (without overflow)*

0	Rs	Rt	0	0x1b
6	5	5	10	6

Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

Put the quotient of the integers from register `Rsrc1` and `Src2` into register `Rdest`:

`div Rdest, Rsrc1, Src2` *Divide (with overflow)* †
`divu Rdest, Rsrc1, Src2` *Divide (without overflow)* †

Negative

Put the negative of the integer from register `Rsrc` into register `Rdest`:

`neg Rdest, Rsrc` *Negate Value (with overflow)* †
`negu Rdest, Rsrc` *Negate Value (without overflow)* †

Logical Operations

Put the logical AND of the integers from register `Rs` and register `Rt` (or the zero-extended immediate value `Imm`) into register `Rd`:

`and Rd, Rs, Rt` *AND*

0	Rs	Rt	Rd	0	0x24
6	5	5	5	5	6

`andi Rd, Rs, Imm` *AND Immediate*

0xc	Rs	Rd	Imm
6	5	5	16

Put the logical NOR of the integers from register `Rs` and `Rt` into register `Rd`:

`nor Rd, Rs, Rt` *NOR*

0	Rs	Rt	Rd	0	0x27
6	5	5	5	5	6

Put the bitwise logical negation of the integer from register `Rsrc` into register `Rdest`:

`not Rdest, Rsrc` *NOT* †

Put the logical OR of the integers from register `Rs` and `Rt` (or `Imm`) into register `Rd`:

`or Rd, Rs, Rt` *OR*

0	Rs	Rt	Rd	0	0x25
6	5	5	5	5	6

`ori Rt, Rs, Imm` *OR Immediate*

0xd	Rs	Rt	Imm
6	5	5	16

Put the logical XOR of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`:

`xor Rd, Rs, Rt` *XOR*

0	Rs	Rt	Rd	0	0x26
6	5	5	5	5	6

`xori Rt, Rs, Imm` *XOR Immediate*

0xe	Rs	Rt	Imm
6	5	5	16

Remainder

Put the remainder from dividing the integer in register `Rsrc1` by the integer in `Src2` into register `Rdest`:

`rem Rdest, Rsrc1, Src2` *Remainder* †
`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †

Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

Rotate and Shift Instructions

Rotate the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` and put the result in register `Rdest`:

`rol Rdest, Rsrc1, Src2` *Rotate Left* †

`ror Rdest, Rsrc1, Src2` *Rotate Right* †

Shift the contents of register `Rt` left (right) by the distance indicated by `Sa` (`Rs`) and put the result in register `Rd`:

`sll Rd, Rt, Sa` *Shift Left Logical*

0	Rs	Rt	Rd	Sa	0
6	5	5	5	5	6

`sllv Rd, Rt, Rs` *Shift Left Logical Variable*

0	Rs	Rt	Rd	0	4
6	5	5	5	5	6

`sra Rd, Rt, Sa` *Shift Right Arithmetic*

0	Rs	Rt	Rd	Sa	3
6	5	5	5	5	6

`srav Rd, Rt, Rs` *Shift Right Arithmetic Variable*

0	Rs	Rt	Rd	0	7
6	5	5	5	5	6

`srl Rd, Rt, Sa` *Shift Right Logical*

0	Rs	Rt	Rd	Sa	2
6	5	5	5	5	6

`srlv Rd, Rt, Rs` *Shift Right Logical Variable*

0	Rs	Rt	Rd	0	6
6	5	5	5	5	6

Constant-Manipulating Instructions

Move the immediate `imm` into register `Rdest`:

`li Rdest, imm`	*Load Immediate* †

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

`lui Rt, imm`	*Load Upper Immediate*

0xf	Rs	Rt	Imm
6	5	5	16

Comparison Instructions

In all instructions below, `Src2` can either be a register or an immediate value (a 16 bit integer).

Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to be 0 otherwise:

`seq Rdest, Rsrc1, Src2`	*Set Equal* †

Set register `Rdest` to 1 if register `Rsrc1` is greater than or equal to `Src2` and to 0 otherwise:

`sge Rdest, Rsrc1, Src2`	*Set Greater Than Equal* †
`sgeu Rdest, Rsrc1, Src2`	*Set Greater Than Equal Unsigned* †

Set register `Rdest` to 1 if register `Rsrc1` is greater than `Src2` and to 0 otherwise:

`sgt Rdest, Rsrc1, Src2`	*Set Greater Than* †
`sgtu Rdest, Rsrc1, Src2`	*Set Greater Than Unsigned* †

Set register `Rdest` to 1 if register `Rsrc1` is less than or equal to `Src2` and to 0 otherwise:

`sle Rdest, Rsrc1, Src2`	*Set Less Than Equal* †
`sleu Rdest, Rsrc1, Src2`	*Set Less Than Equal Unsigned* †

Set register `Rdest` to 1 if register `Rsrc1` is less than `Src2` (or `Imm`) and to 0 otherwise:

`slt Rd, Rs, Rt` *Set Less Than*

0	Rs	Rt	Rd	0	0x2a
6	5	5	5	5	6

`sltu Rd, Rs, Rt` *Set Less Than Unsigned*

0	Rs	Rt	Rd	0	0x2b
6	5	5	5	5	6

`slti Rd, Rs, Imm` *Set Less Than Immediate*

0xa	Rs	Rt	Imm
6	5	5	16

`sltiu Rd, Rs, Imm` *Set Less Than Unsigned Immediate*

0xb	Rs	Rt	Imm
6	5	5	16

Set register `Rdest` to 1 if register `Rsrc1` is not equal to `Src2` and to 0 otherwise:

`sne Rdest, Rsrc1, Src2` *Set Not Equal* †

Branch and Jump Instructions

In all instructions below, `Src2` can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit 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.

For branch instructions, the offset of the instruction at a label is computed by the assembler.

Unconditionally branch to the instruction at the label:

`b label` *Branch pseudoinstruction* †

Conditionally branch to the instruction at the label if coprocessor *z*'s condition flag is true (false):

`bc`*z*`t label` *Branch Coprocessor z True*

0x1*z*	8	1	Offset
6	5	5	16

`bc`*z*`f label` *Branch Coprocessor z False*

0x1*z*	8	0	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of register `Rs` equals the contents of register `Rt`:

`beq Rs, Rt, label` *Branch on Equal*

4	Rs	Rt	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of `Rsrc` equals 0:

`beqz Rsrc, label` *Branch on Equal Zero* †

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than or equal to `Src2`:

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †
`bgeu Rsrc1, Src2, label` *Branch on GTE Unsigned* †

Conditionally branch to the instruction at the label if the contents of `Rs` are greater than or equal to 0:

`bgez Rs, label` *Branch on Greater Than Equal Zero*

1	Rs	1	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of `Rs` are greater than or equal to 0. Save the address of the next instruction in register 31:

`bgezal Rs, label` *Branch on Greater Than Equal Zero And Link*

1	Rs	0x11	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are greater than `Src2`:

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †
`bgtu Rsrc1, Src2, label` *Branch on Greater Than Unsigned* †

Conditionally branch to the instruction at the label if the contents of `Rs` are greater than 0:

`bgtz Rs, label` *Branch on Greater Than Zero*

7	Rs	0	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than or equal to `Src2`:

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †
`bleu Rsrc1, Src2, label` *Branch on LTE Unsigned* †

Conditionally branch to the instruction at the label if the contents of `Rs` are less than or equal to 0:

`blez Rs, label` *Branch on Less Than Equal Zero*

6	Rs	0	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of `Rs` are less than 0. Save the address of the next instruction in register 31:
`bltzal Rs, label` *Branch on Less Than And Link*

1	Rs	0x10	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are less than `Src2`:

`blt Rsrc1, Src2, label` *Branch on Less Than* †
`bltu Rsrc1, Src2, label` *Branch on Less Than Unsigned* †

Conditionally branch to the instruction at the label if the contents of `Rs` are less than 0:

`bltz Rs, label` *Branch on Less Than Zero*

1	Rs	0	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` are not equal to `Src2`:

`bne Rs, Rt, label` *Branch on Not Equal*

5	Rs	Rt	Offset
6	5	5	16

Conditionally branch to the instruction at the label if the contents of `Rsrc` are not equal to 0:

`bnez Rsrc, label` *Branch on Not Equal Zero* †

Unconditionally jump to the instruction at Target:

`j label` *Jump*

2	Target
6	26

Unconditionally jump to the instruction at Target. Save the address of the next instruction in register 31:

`jal label` *Jump and Link*

3	Target
6	26

Unconditionally jump to the instruction whose address is in register `Rs`. Save the address of the next instruction in register `Rd` (or in register 31, if `Rd` is omitted):

`jalr [Rd,] Rs` *Jump and Link Register*

0	Rs	0	Rd	0	9
6	5	5	5	5	6

Unconditionally jump to the instruction whose address is in register `Rs`:

`jr Rs` *Jump Register*

0	Rs	0	8
6	5	16	5

Load Instructions

Load computed *address*, not the contents of the location, into register `Rdest`:

`la Rdest, address` *Load Address* †

Load the byte at *address* (or at `Offset` + contents of register `Base`) into register `Rt`. The byte is sign-extended by the `lb`, but not the `lbu`, instruction:

`lb Rt, address|Offset(Base)` *Load Byte*

0x20	Base	Rt	Offset
6	5	5	16

`lbu Rt, address|Offset(Base)` *Load Unsigned Byte*

0x24	Base	Rt	Offset
6	5	5	16

Load the 64-bit quantity at *address* into registers `Rdest` and `Rdest + 1`:

`ld Rdest, address` *Load Double-Word* †

Load the 16-bit quantity (halfword) at *address* (or at `Offset` + contents of register `Base`) into register `Rt`. The halfword is sign-extended by the `lh`, but not the `lhu`, instruction:

`lh Rt, address|Offset(Base)` *Load Halfword*

0x21	Base	Rt	Offset
6	5	5	16

`lhu Rt, address|Offset(Base)` *Load Unsigned Halfword*

0x25	Base	Rt	Offset
6	5	5	16

Load the 16-bit *immediate* into the most significant 16 bits of register `Rt`:

`lui Rt, Imm` *Load Upper Immediate*

15	0	Rt	Imm
6	5	5	16

Load the 32-bit quantity (word) at *address* (or at `Offset` + contents of register `Base`) into register `Rt`:

`lw Rt, address|Offset(Base)` *Load Word*

0x23	Base	Rt	Offset
6	5	5	16

Load the word at *address* (or at `Offset` + contents of register `Base`) into register `Rt` of coprocessor z (0–3):

`lwc`z `Rt, address|Offset(Base)` *Load Word Coprocessor*

0x3z	Base	Rt	Offset
6	5	5	16

Load the left (right) bytes from the word at the possibly-unaligned *address* into register `Rdest`:

`lwl Rdest, address` *Load Word Left*

0x22	Rs	Rt	Offset
6	5	5	16

`lwr Rdest, address` *Load Word Right*

0x23	Rs	Rt	Offset
6	5	5	16

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

`ulh Rdest, address` *Unaligned Load Halfword* †
`ulhu Rdest, address` *Unaligned Load Halfword Unsigned* †

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

`ulw Rdest, address` *Unaligned Load Word* †

Store Instructions

Store the low byte from register `Rt` at *address*:

`sb Rt, address` *Store Byte*

0x28	Rs	Rt	Offset
6	5	5	16

Store the 64-bit quantity in registers `Rsrc` and `Rsrc + 1` at *address*:

`sd Rsrc, address` *Store Double-Word* †

Store the low halfword from register `Rt` at *address*:

`sh Rt, address` *Store Halfword*

0x29	Rs	Rt	Offset
6	5	5	16

Store the word from register `Rt` at *address*:

`sw Rt, address` *Store Word*

0x2b	Rs	Rt	Offset
6	5	5	16

Store the word from register `Rt` of coprocessor z at *address*:

`swc`z `Rt, address` *Store Word Coprocessor*

0x3(1−z)	Rs	Rt	Offset
6	5	5	16

Store the left (right) bytes from register `Rt` at the possibly-unaligned *address*:

`swl Rt, address` *Store Word Left*

0x2a	Rs	Rt	Offset
6	5	5	16

`swr Rt, address` *Store Word Right*

0x2e	Rs	Rt	Offset
6	5	5	16

Store the low halfword from register `Rsrc` at the possibly-unaligned *address*:

`ush Rsrc, address` *Unaligned Store Halfword* †

Store the word from register `Rsrc` at the possibly-unaligned *address*:

`usw Rsrc, address` *Unaligned Store Word* †

Data Movement Instructions

Move the contents of `Rsrc` to `Rdest`:

`move Rdest, Rsrc` *Move* †

The multiply and divide unit produces its result in two additional registers, `hi` and `lo`. The following instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

Move the contents of the hi (lo) register to register `Rd`:

`mfhi Rd` *Move From hi*

0	0	Rd	0	0x10
6	10	5	5	6

`mflo Rd` *Move From lo*

0	0	Rd	0	0x12
6	10	5	5	6

Move the contents of register `Rs` to the hi (lo) register:

`mthi Rs` *Move To hi*

0	Rs	0	0x11
6	5	15	6

`mtlo Rs` *Move To lo*

0	Rs	0	0x13
6	5	15	6

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

Move the contents of coprocessor *z*'s register `Rd` to CPU register `Rt`:

`mfc`*z* `Rt, Rd` *Move From Coprocessor z*

0x1*z*	0	Rt	Rd	0
6	5	5	5	11

Move the contents of floating point registers `FRsrc1` and `FRsrc1 + 1` to CPU registers `Rdest` and `Rdest + 1`:

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* [†]

Move the contents of CPU register `Rt` to coprocessor z's register `Rd`:

`mtc`z `Rt, Rd` *Move To Coprocessor z*

0x1z	4	Rt	Rd	0
6	5	5	5	11

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. To simplify matters, floating point operations only use even-numbered registers—including instructions that operate on single floats.

Values are moved in or out of these registers one word (32-bits) at a time by the `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 real instructions below, `Fs` and `Fd` are floating-point registers. In the pseudoinstructions, `FRdest`, `FRsrc1`, `FRsrc2`, and `FRsrc` are floating point registers (e.g., `$f2`).

Compute the absolute value of the floating float double (single) in register `Fs` and put it in register `Fd`:

`abs.d Fd, Fs` *Floating Point Absolute Value Double*

0x11	1	0	Fs	Fd	5
6	5	5	5	5	6

`abs.s Fd, Fs` *Floating Point Absolute Value Single*

0x11	0	0	Fs	Fd	5
6	5	5	5	5	6

Compute the sum of the floating float doubles (singles) in registers `Fs` and `Ft` and put it in register `Fd`:

`add.d Fd, Fs, Ft` *Floating Point Addition Double*

0x11	1	Ft	Fs	Fd	0
6	5	5	5	5	6

`add.s Fd, Fs, Ft` *Floating Point Addition Single*

0x11	0	Ft	Fs	Fd	0
6	5	5	5	5	6

Compare the floating point double in register `Fs` against the one in `Ft` and set the floating point condition flag `FC` true if they are equal:

`c.eq.d Fs, Ft` *Compare Equal Double*

0x11	1	Ft	Fs	Fd	FC	2
6	5	5	5	5	2	4

`c.eq.s Fs, Ft` *Compare Equal Single*

0x11	0	Ft	Fs	Fd	FC	2
6	5	5	5	5	2	4

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:

`c.le.d Fs, Ft` *Compare Less Than Equal Double*

0x11	1	Ft	Fs	0	FC	2
6	5	5	5	5	2	4

`c.le.s Fs, Ft` *Compare Less Than Equal Single*

0x11	0	Ft	Fs	0	FC	2
6	5	5	5	5	2	4

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:

`c.lt.d Fs, Ft` *Compare Less Than Double*

0x11	1	Ft	Fs	0	FC	0xc
6	5	5	5	5	2	4

`c.lt.s Fs, Ft` *Compare Less Than Single*

0x11	0	Ft	Fs	0	FC	0xc
6	5	5	5	5	2	4

Convert the single precision floating point number or integer in register `Fs` to a double precision number and put it in register `Fd`:

`cvt.d.s Fd, Fs` *Convert Single to Double*

0x11	1	0	Fs	Fd	0x21
6	5	5	5	5	6

`cvt.d.w Fd, Fs` *Convert Integer to Double*

0x11	0	0	Fs	Fd	0x21
6	5	5	5	5	6

Convert the double precision floating point number or integer in register `Fs` to a single precision number and put it in register `Fd`:

`cvt.s.d Fd, Fs` *Convert Double to Single*

0x11	1	0	Fs	Fd	0x20
6	5	5	5	5	6

`cvt.s.w Fd, Fs` *Convert Integer to Single*

0x11	0	0	Fs	Fd	0x20
6	5	5	5	5	6

Convert the double or single precision floating point number in register `Fs` to an integer and put it in register `Fd`:

`cvt.w.d Fd, Fs` *Convert Double to Integer*

0x11	1	0	Fs	Fd	0x24
6	5	5	5	5	6

`cvt.w.s Fd, Fs` *Convert Single to Integer*

0x11	0	0	Fs	Fd	0x24
6	5	5	5	5	6

Compute the quotient of the floating float doubles (singles) in registers `Fs` and `Ft` and put it in register `Fd`:

`div.d Fd, Fs, Ft` *Floating Point Divide Double*

0x11	1	Ft	Fs	Fd	3
6	5	5	5	5	6

`div.s Fd, Fs, Ft` *Floating Point Divide Single*

0x11	0	Ft	Fs	Fd	3
6	5	5	5	5	6

Load the floating float double (single) at `address` into register `FRdest`:

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †

Move the floating float double (single) from register `Fs` to register `Fd`:

`mov.d Fd, Fs` *Move Floating Point Double*

0x11	1	0	Fs	Fd	6
6	5	5	5	5	6

`mov.s Fd, Fs` *Move Floating Point Single*

0x11	0	0	Fs	Fd	6
6	5	5	5	5	6

Compute the product of the floating float doubles (singles) in registers `Fs` and `Ft` and put it in register `Fd`:

`mul.d Fd, Fs, Ft` *Floating Point Multiply Double*

0x11	1	Ft	Fs	Fd	2
6	5	5	5	5	6

`mul.s Fd, Fs, Ft` *Floating Point Multiply Single*

0x11	0	Ft	Fs	Fd	2
6	5	5	5	5	6

Negate the floating point double (single) in register `Fs` and put it in register `Fd`:

`neg.d Fd, Fs` *Negate Double*

0x11	1	0	Fs	Fd	7
6	5	5	5	5	6

`neg.s Fd, Fs` *Negate Single*

0x11	0	0	Fs	Fd	7
6	5	5	5	5	6

Store the floating float double (single) in register `FRdest` at `address`: Store the floating float double (single) in register `FRdest` at `address`:

`s.d FRdest, address` *Store Floating Point Double* †
`s.s FRdest, address` *Store Floating Point Single* †

Compute the difference of the floating float doubles (singles) in registers `Fs` and `Ft` and put it in register `Fd`:

`sub.d Fd, Fs, Ft` *Floating Point Subtract Double*

0x11	1	Ft	Fs	Fd	1
6	5	5	5	5	6

`sub.s Fd, Fs, Ft` *Floating Point Subtract Single*

0x11	0	Ft	Fs	Fd	1
6	5	5	5	5	6

Exception and Trap Instructions

Restore the Status register:

`rfe` *Return From Exception*

0x11	1	0	0x20
6	1	19	6

Register `$v0` contains the number of the system call (see Table **??**) provided by SPIM:

`syscall` *System Call*

0x11	0	0xc
6	20	6

Cause exception *n*. Exception 1 is reserved for the debugger:

`break n` *Break*

0x11	code	0xd
6	20	6

Do nothing:

`nop` *No operation*

0	0	0	0	0	0
6	5	5	5	5	6
