

SPIM Quick Reference

This Web page was created by reformatting portions of the LaTeX source file of the public domain documentation distributed with the SPIM simulator.

Contents

- [MIPS Registers and Usage Convention](#)
- [Table of `sycalls`](#)
- [Assembler Directives](#)
- [SPIM Instruction Set](#)
 - [Arithmetic and Logical Instructions](#)
 - [Constant-Manipulating Instructions](#)
 - [Comparison Instructions](#)
 - [Branch and Jump Instructions](#)
 - [Load Instructions](#)
 - [Store Instructions](#)
 - [Data Movement Instructions](#)
 - [Floating Point Instructions](#)
 - [Exception and Trap Instructions](#)

MIPS Registers and Usage Convention

Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and results of a function
v1	3	Expression evaluation and results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)

s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

System Services

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		

Assembler Directives

`.align n`

Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.

`.ascii str`

Store the string in memory, but do not null-terminate it.

`.asciiz str`

Store the string in memory and null-terminate it.

`.byte b1, ..., bn`

Store the n values in successive bytes of memory.

`.data`

The following data items should be stored in the data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.double d1, ..., dn`

Store the n floating point double precision numbers in successive memory locations.

`.extern sym size`

Declare that the datum stored at *sym* is *size* bytes large and is a global symbol. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.

`.float f1, ..., fn`

Store the n floating point single precision numbers in successive memory locations.

`.globl sym`

Declare that symbol *sym* is global and can be referenced from other files.

`.half h1, ..., hn`

Store the n 16-bit quantities in successive memory halfwords.

`.kdata`

The following data items should be stored in the kernel data segment. If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.ktext`

The next items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.space n`

Allocate n bytes of space in the current segment (which must be the data segment in SPIM).

`.text`

The next items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, the items are stored beginning at address *addr*.

`.word w1, ..., wn`

Store the n 32-bit quantities in successive memory words.

SPIM Instruction Set

Arithmetic and Logical 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 constant.

`abs Rdest, Rsrc`

Absolute Value

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

`add Rdest, Rsrc1, Src2`

Addition (with overflow)

`addi Rdest, Rsrc1, Imm`

Addition Immediate (with overflow)

`addu Rdest, Rsrc1, Src2`

Addition (without overflow)

`addiu Rdest, Rsrc1, Imm`

Addition Immediate (without overflow)

Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

`and Rdest, Rsrc1, Src2`

AND

`andi Rdest, Rsrc1, Imm`

AND Immediate

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

`div Rsrc1, Rsrc2`

Divide (with overflow)

`divu Rsrc1, Rsrc2`

Divide (without overflow)

Divide the contents of the two registers. 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 conventions of the machine on which SPIM is run.

`div Rdest, Rsrc1, Src2`

Divide (with overflow)

`divu Rdest, Rsrc1, Src2`

Divide (without overflow)

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

<code>mul Rdest, Rsrc1, Src2</code>	Multiply (without overflow)
<code>mulo Rdest, Rsrc1, Src2</code>	Multiply (with overflow)
<code>mulou Rdest, Rsrc1, Src2</code>	Unsigned Multiply (with overflow)

Put the product of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

<code>mult Rsrc1, Rsrc2</code>	Multiply
<code>multu Rsrc1, Rsrc2</code>	Unsigned Multiply

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

<code>neg Rdest, Rsrc</code>	Negate Value (with overflow)
<code>negu Rdest, Rsrc</code>	Negate Value (without overflow)

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

<code>nor Rdest, Rsrc1, Src2</code>	NOR
-------------------------------------	-----

Put the logical NOR of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

<code>not Rdest, Rsrc</code>	NOT
------------------------------	-----

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

<code>or Rdest, Rsrc1, Src2</code>	OR
<code>ori Rdest, Rsrc1, Imm</code>	OR Immediate

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

<code>rem Rdest, Rsrc1, Src2</code>	Remainder
<code>remu Rdest, Rsrc1, Src2</code>	Unsigned Remainder

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

<code>rol Rdest, Rsrc1, Src2</code>	Rotate Left
<code>ror Rdest, Rsrc1, Src2</code>	Rotate Right

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

<code>sll Rdest, Rsrc1, Src2</code>	Shift Left Logical
<code>sllv Rdest, Rsrc1, Rsrc2</code>	Shift Left Logical Variable
<code>sra Rdest, Rsrc1, Src2</code>	Shift Right Arithmetic
<code>srav Rdest, Rsrc1, Rsrc2</code>	Shift Right Arithmetic Variable
<code>srl Rdest, Rsrc1, Src2</code>	Shift Right Logical
<code>srlv Rdest, Rsrc1, Rsrc2</code>	Shift Right Logical Variable

Shift the contents of register `Rsrc1` left (right) by the distance indicated by `Src2` (`Rsrc2`) and put the result in register `Rdest`.

<code>sub Rdest, Rsrc1, Src2</code>	Subtract (with overflow)
<code>subu Rdest, Rsrc1, Src2</code>	Subtract (without overflow)

Put the difference of the integers from register `Rsrc1` and `Src2` into register `Rdest`.

<code>xor Rdest, Rsrc1, Src2</code>	XOR
<code>xori Rdest, Rsrc1, Imm</code>	XOR Immediate

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

Constant-Manipulating Instructions

<code>li Rdest, imm</code>	Load Immediate
----------------------------	----------------

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

<code>lui Rdest, imm</code>	Load Upper 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.

Comparison Instructions

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

<code>seq Rdest, Rsrc1, Src2</code>	Set Equal
-------------------------------------	-----------

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

<code>sge Rdest, Rsrc1, Src2</code>	Set Greater Than Equal
<code>sgeu Rdest, Rsrc1, Src2</code>	Set Greater Than Equal Unsigned

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

<code>sgt Rdest, Rsrc1, Src2</code>	Set Greater Than
<code>sgtu Rdest, Rsrc1, Src2</code>	Set Greater Than Unsigned

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

<code>sle Rdest, Rsrc1, Src2</code>	Set Less Than Equal
<code>sleu Rdest, Rsrc1, Src2</code>	Set Less Than Equal Unsigned

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

<code>slt Rdest, Rsrc1, Src2</code>	Set Less Than
<code>slti Rdest, Rsrc1, Imm</code>	Set Less Than Immediate
<code>sltu Rdest, Rsrc1, Src2</code>	Set Less Than Unsigned
<code>sltiu Rdest, Rsrc1, Imm</code>	Set Less Than Unsigned Immediate

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

<code>sne Rdest, Rsrc1, Src2</code>	Set Not Equal
-------------------------------------	---------------

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

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.

<code>b label</code>	Branch instruction
----------------------	--------------------

Unconditionally branch to the instruction at the label.

<code>bczt label</code>	Branch Coprocessor <i>z</i> True
<code>bczf label</code>	Branch Coprocessor <i>z</i> False

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

<code>beq Rsrc1, Src2, label</code>	Branch on Equal
-------------------------------------	-----------------

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

<code>beqz Rsrc, label</code>	Branch on Equal Zero
-------------------------------	----------------------

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

<code>bge Rsrc1, Src2, label</code>	Branch on Greater Than Equal
<code>bgeu Rsrc1, Src2, label</code>	Branch on GTE Unsigned

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

<code>bgez Rsrc, label</code>	Branch on Greater Than Equal Zero
-------------------------------	-----------------------------------

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

<code>bgezal Rsrc, label</code>	Branch on Greater Than Equal Zero And Link
---------------------------------	--

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

<code>bgt Rsrc1, Src2, label</code>	Branch on Greater Than
<code>bgtu Rsrc1, Src2, label</code>	Branch on Greater Than Unsigned

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

<code>bgtz Rsrc, label</code>	Branch on Greater Than Zero
-------------------------------	-----------------------------

Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater than 0.

<code>ble Rsrc1, Src2, label</code>	Branch on Less Than Equal
<code>bleu Rsrc1, Src2, label</code>	Branch on LTE Unsigned

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

<code>blez Rsrc, label</code>	Branch on Less Than Equal Zero
-------------------------------	--------------------------------

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

<code>bgezal Rsrc, label</code>	Branch on Greater Than Equal Zero And Link
<code>bltzal Rsrc, label</code>	Branch on Less Than And Link

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

<code>blt Rsrc1, Src2, label</code>	Branch on Less Than
<code>bltu Rsrc1, Src2, label</code>	Branch on Less Than Unsigned

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

<code>bltz Rsrc, label</code>	Branch on Less Than Zero
-------------------------------	--------------------------

Conditionally branch to the instruction at the label if the contents of `Rsrc` are less than 0.

<code>bne Rsrc1, Src2, label</code>	Branch on Not Equal
-------------------------------------	---------------------

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

<code>bnez Rsrc, label</code>	Branch on Not Equal Zero
-------------------------------	--------------------------

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

<code>j label</code>	Jump
----------------------	------

Unconditionally jump to the instruction at the label.

jal label

Jump and Link

jalr Rsrc

Jump and Link Register

Unconditionally jump to the instruction at the label or whose address is in register *Rsrc*. Save the address of the next instruction in register 31.

jr Rsrc

Jump Register

Unconditionally jump to the instruction whose address is in register *Rsrc*.

Load Instructions

la Rdest, address

Load Address

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

lb Rdest, address

Load Byte

lbu Rdest, address

Load Unsigned Byte

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

ld Rdest, address

Load Double-Word

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

lh Rdest, address

Load Halfword

lhu Rdest, address

Load Unsigned Halfword

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

lw Rdest, address

Load Word

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

lwcz Rdest, address

Load Word Coprocessor *z*

Load the word at *address* into register *Rdest* of coprocessor *z* (0-3).

lwl Rdest, address

Load Word Left

lwr Rdest, address

Load Word Right

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

ulh Rdest, address

Unaligned Load Halfword

ulhu Rdest, address

Unaligned Load Halfword Unsigned

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

ulw Rdest, address

Unaligned Load Word

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

Store Instructions

sb Rsrc, address

Store Byte

Store the low byte from register *Rsrc* at *address*.

sd Rsrc, address

Store Double-Word

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

sh Rsrc, address

Store Halfword

Store the low halfword from register *Rsrc* at *address*.

sw Rsrc, address

Store Word

Store the word from register *Rsrc* at *address*.

swcz Rsrc, addressStore Word Coprocessor *z*

Store the word from register *Rsrc* of coprocessor *z* at *address*.

swl Rsrc, address

Store Word Left

swr Rsrc, address

Store Word Right

Store the left (right) bytes from register *Rsrc* at the possibly-unaligned *address*.

ush Rsrc, address

Unaligned Store Halfword

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

usw Rsrc, address

Unaligned Store Word

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

Data Movement Instructions

move Rdest, Rsrc

Move

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

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 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.

<code>mfhi Rdest</code>	Move From <code>hi</code>
<code>mflo Rdest</code>	Move From <code>lo</code>

Move the contents of the `hi` (`lo`) register to register `Rdest`.

<code>mfhi Rdest</code>	Move To <code>hi</code>
<code>mflo Rdest</code>	Move To <code>lo</code>

Move the contents register `Rdest` to the `hi` (`lo`) register.

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

<code>mfcz Rdest, CPsrc</code>	Move From Coprocessor <code>z</code>
--------------------------------	--------------------------------------

Move the contents of coprocessor `z`'s register `CPsrc` to CPU register `Rdest`.

<code>mfcl.d Rdest, FRsrc1</code>	Move Double From Coprocessor 1
-----------------------------------	--------------------------------

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

<code>mtcz Rsrc, CPdest</code>	Move To Coprocessor <code>z</code>
--------------------------------	------------------------------------

Move the contents of CPU register `Rsrc` to coprocessor `z`'s register `CPdest`.

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 a word (32-bits) at a time by `lwcl`, `swcl`, `mtcl`, and `mfcl` 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 `bclt` and `bclf` instructions. In all instructions below, `FRdest`, `FRsrc1`, `FRsrc2`, and `FRsrc` are floating point registers (e.g., `f2`).

<code>abs.d FRdest, FRsrc</code>	Floating Point Absolute Value Double
<code>abs.s FRdest, FRsrc</code>	Floating Point Absolute Value Single

Compute the absolute value of the floating float double (single) in register `FRsrc` and put it in register `FRdest`.

<code>add.d FRdest, FRsrc1, FRsrc2</code>	Floating Point Addition Double
---	--------------------------------

add.s FRdest, FRsrc1, FRsrc2

Floating Point Addition Single

Compute the sum of the floating float doubles (singles) in registers **FRsrc1** and **FRsrc2** and put it in register **FRdest**.

c.eq.d FRsrc1, FRsrc2

Compare Equal Double

c.eq.s FRsrc1, FRsrc2

Compare Equal Single

Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the floating point condition flag true if they are equal.

c.le.d FRsrc1, FRsrc2

Compare Less Than Equal Double

c.le.s FRsrc1, FRsrc2

Compare Less Than Equal Single

Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the floating point condition flag true if the first is less than or equal to the second.

c.lt.d FRsrc1, FRsrc2

Compare Less Than Double

c.lt.s FRsrc1, FRsrc2

Compare Less Than Single

Compare the floating point double in register **FRsrc1** against the one in **FRsrc2** and set the condition flag true if the first is less than the second.

cvt.d.s FRdest, FRsrc

Convert Single to Double

cvt.d.w FRdest, FRsrc

Convert Integer to Double

Convert the single precision floating point number or integer in register **FRsrc** to a double precision number and put it in register **FRdest**.

cvt.s.d FRdest, FRsrc

Convert Double to Single

cvt.s.w FRdest, FRsrc

Convert Integer to Single

Convert the double precision floating point number or integer in register **FRsrc** to a single precision number and put it in register **FRdest**.

cvt.w.d FRdest, FRsrc

Convert Double to Integer

cvt.w.s FRdest, FRsrc

Convert Single to Integer

Convert the double or single precision floating point number in register **FRsrc** to an integer and put it in register **FRdest**.

div.d FRdest, FRsrc1, FRsrc2

Floating Point Divide Double

div.s FRdest, FRsrc1, FRsrc2

Floating Point Divide Single

Compute the quotient of the floating float doubles (singles) in registers **FRsrc1** and **FRsrc2** and put it in register **FRdest**.

l.d FRdest, address

Load Floating Point Double

l.s FRdest, address

Load Floating Point Single

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

`mov.d FRdest, FRsrc`

Move Floating Point Double

`mov.s FRdest, FRsrc`

Move Floating Point Single

Move the floating float double (single) from register `FRsrc` to register `FRdest`.

`mul.d FRdest, FRsrc1, FRsrc2`

Floating Point Multiply Double

`mul.s FRdest, FRsrc1, FRsrc2`

Floating Point Multiply Single

Compute the product of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

`neg.d FRdest, FRsrc`

Negate Double

`neg.s FRdest, FRsrc`

Negate Single

Negate the floating point double (single) in register `FRsrc` and put it in register `FRdest`.

`s.d FRdest, address`

Store Floating Point Double

`s.s FRdest, address`

Store Floating Point Single

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

`sub.d FRdest, FRsrc1, FRsrc2`

Floating Point Subtract Double

`sub.s FRdest, FRsrc1, FRsrc2`

Floating Point Subtract Single

Compute the difference of the floating float doubles (singles) in registers `FRsrc1` and `FRsrc2` and put it in register `FRdest`.

Exception and Trap Instructions

`rfe`

Return From Exception

Restore the Status register.

`syscall`

System Call

Register `v0` contains the number of the system call (see [System Services](#)) provided by SPIM.

`break n`

Break

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

`nop`

No operation

Do nothing.