Simple SPIM Quick Reference

MIPS Registers and Usage Convention

Register Name	Number	Usage	Register Name	Number	Usage
zero	0	Constant 0	s0	16	Saved temporary (preserved across call)
at	1	Reserved for assembler	s1	17	Saved temporary (preserved across call)
v0	2	Evaluation and results of a function	s2	18	Saved temporary (preserved across call)
v1	3	Evaluation and results of a function	s3	19	Saved temporary (preserved across call)
a0	4	Argument 1	s4	20	Saved temporary (preserved across call)
a1	5	Argument 2	s5	21	Saved temporary (preserved across call)
a2	6	Argument 3	s6	22	Saved temporary (preserved across call)
a3	7	Argument 4	s7	23	Saved temporary (preserved across call)
t0	8	Temporary (not preserved across call)	t8	24	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)	t9	25	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)	k0	26	Reserved for OS kernel
t3	11	Temporary (not preserved across call)	k1	27	Reserved for OS kernel
t4	12	Temporary (not preserved across call)	gp	28	Pointer to global area
t5	13	Temporary (not preserved across call)	sp	29	Stack pointer
t6	14	Temporary (not preserved across call)	fp	30	Frame pointer
t7	15	Temporary (not preserved across call)	ra	31	Return address (used by function call)

System Services

Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_string	8	`$a0` = buffer, `$a1` = length	
exit	10		

Assembler Directives

`.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*.

`.globl sym`
: Declare that symbol `sym` is global and can be referenced from other files.

`.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.

Adressing modes

Format	Address Computation
(register)	contents of register
imm	immediate
symbol	address of symbol

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 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)
`addu Rdest, Rsrc1, Src2` Addition (without overflow)

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

`and Rdest, Rsrc1, Src2` AND

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`.

`mul Rdest, Rsrc1, Src2` Multiply (without overflow)
`mulo Rdest, Rsrc1, Src2` Multiply (with overflow)
`mulou Rdest, Rsrc1, Src2` Unsigned Multiply (with overflow)

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

`mult Rsrc1, Rsrc2` Multiply
`multu Rsrc1, Rsrc2` 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`.

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

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

`nor Rdest, Rsrc1, Src2` NOR

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

`not Rdest, Rsrc` NOT

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

`or Rdest, Rsrc1, Src2` OR

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

`rem Rdest, Rsrc1, Src2` Remainder
`remu Rdest, Rsrc1, Src2` 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.

`rol Rdest, Rsrc1, Src2` Rotate Left
`ror Rdest, Rsrc1, Src2` Rotate Right

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

`sll Rdest, Rsrc1, Src2` Shift Left Logical
`sllv Rdest, Rsrc1, Rsrc2` Shift Left Logical Variable
`sra Rdest, Rsrc1, Src2` Shift Right Arithmetic
`srav Rdest, Rsrc1, Rsrc2` Shift Right Arithmetic Variable
`srl Rdest, Rsrc1, Src2` Shift Right Logical
`srlv Rdest, Rsrc1, Rsrc2` 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`.

`sub Rdest, Rsrc1, Src2` Subtract (with overflow)
`subu Rdest, Rsrc1, Src2` Subtract (without overflow)

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

`xor Rdest, Rsrc1, Src2` XOR

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

Constant-Manipulating Instructions

`li Rdest, imm` Load Immediate

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

`lui Rdest, imm` 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).

`seq Rdest, Rsrc1, Src2` Set Equal

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

`slt Rdest, Rsrc1, Src2` Set Less Than
`sltu Rdest, Rsrc1, Src2` Set Less Than Unsigned

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

`sne Rdest, Rsrc1, Src2` 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.

`b label` Branch instruction

Unconditionally branch to the instruction at the label.

`bczt label` Branch Coprocessor *z* True
`bczf label` Branch Coprocessor *z* False

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

`beq Rsrc1, Src2, label` Branch on Equal

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

`beqz Rsrc, label` Branch on Equal Zero

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

`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 register `Rsrc1` are greater than or equal to `Src2`.

`bgez Rsrc, label` 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.

`bgezal Rsrc, label` 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.

`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 register `Rsrc1` are greater than `Src2`.

`bgtz Rsrc, label` Branch on Greater Than Zero

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

`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 register `Rsrc1` are less than or equal to `Src2`.

`blez Rsrc, label` 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.

`bgezal Rsrc, label` Branch on Greater Than Equal Zero And Link
`bltzal Rsrc, label` 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.

`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 register `Rsrc1` are less than `Src2`.

`bltz Rsrc, label` Branch on Less Than Zero

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

`bne Rsrc1, Src2, label` Branch on Not Equal

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

`bnez Rsrc, label` Branch on Not Equal Zero

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

`j label` 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

S tore the word from register `Rsrc` at *address*.

`swcz Rsrc, address` Store 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.

`mfhi Rdest` Move From hi
`mflo Rdest` Move From lo

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

`mthi Rdest` Move To hi
`mtlo Rdest` Move To lo

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.

`mfcz Rdest, CPsrc` Move From Coprocessor *z*

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

`mfc1.d Rdest, FRsrc1` Move Double From Coprocessor 1

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

`mtcz Rsrc, CPdest` Move To Coprocessor *z*

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