COMP1521 17s2

SPIM Instruction Set

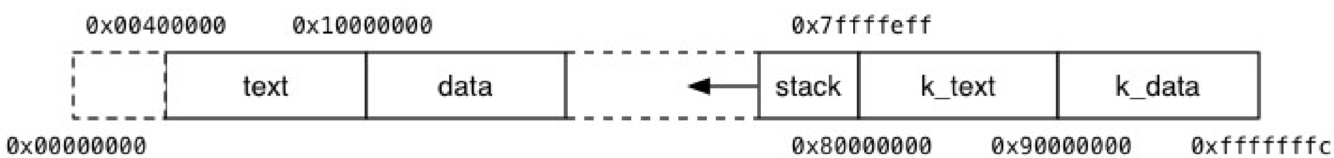
CNAME

An overview of the instruction set in the SPIM MIPS emulator. Based on a document from the University of Stuttgart.

The SPIM emulator implements instructions from the MIPS32 instruction set, as well as *pseudo-instructions* (which look like MIPS instrcutions, but are not actually provided on the MIPS32 hardware).

Architecture

MIPS has 32 × 32-bit general purpose registers and 16 × 64-bit floating point registers, as well a two special registers `Hi` and `Lo` for manipulating 64-bit integer quantities. In addition, it has a memory which is partitioned as follows:



Registers

The 32 general purpose registers can be referenced as `$0..$31`, or by symbolic names, and are used as follows:

Reg	Name	Description
`$0`	`zero`	the value 0, not changeable
`$1`	`$at`	**a**ssembler **t**emporary; reserved for assembler use
`$2,$3`	`$v0,$v1`	**v**alue from expression evaluation or function return
`$4..$7`	`$a0..$a3`	first four **a**rguments to a function/subroutine, if needed
`$8..$15`	`$t0..$t7`	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
`$16..$23`	`$s0..$s7`	**s**afe function variable; must not be overwritten by called subroutine
`$24..$25`	`$t8..$t9`	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
`$26..$27`	`$k0..$k1`	for **k**ernel use; may change unexpectedly
`$28`	`$gp`	**g**lobal **p**ointer (address of global area)
`$29`	`$sp`	**s**tack **p**ointer (top of stack)
`$30`	`$fp`	**f**rame **p**ointer (bottom of current stack frame)
`$30`	`$ra`	**r**eturn **a**ddress of most recent caller

The 16 floating point registers are referenced in pairs; each pair is 64-bits.

Reg	Description

Reg	Description
`$f0..$f2`	value floating-point expression evaluation or function return
`$f4..$f10`	temporary; must be saved by caller to subroutine; subroutine can overwrite
`$f12..$f14`	first two double-precision function arguments
`$f16..$f18`	temporary registers; used for expression evaluation
`$f20..$f30`	safe function variables; must be preserved across function calls

Instructions

Each instruction is written on a single line and has the general format

Label`:` *OpCode*`,` $Operand_1$`,` $Operand_2$`,` $Operand_3$

Some instructions have only one operand, others have two and many have three.

Operands

The following notation is used in describing operands in the description of instructions below.

Operand	Description
R_n	a register; R_s and R_t are sources, and R_d is a destination
Imm	a constant value; a literal constant in decimal or hexadecimal format
Label	a symbolic name which is associated with a memory address
Addr	a memory address, in one of the formats described below

Addressing Modes

Many instructions have one operand which is an address. Addresses can be written in a number of formats:

Format	Address
Label	the address associated with the label
(R_n)	the value stored in register R_n (indirect address)
Imm (R_n)	the sum of Imm and the value stored in register R_n
Label (R_n)	the sum of Label's address and the value stored in register R_n
Label ± Imm	the sum of Label's address and Imm
Label ± Imm (R_n)	the sum of Label's address, Imm and the value stored in register R_n

List of SPIM instructions

Real MIPS instructions are marked with a ✓. All other instructions are pseudoinstructions, special to the SPIM emulator. Operators in expressions have the same meaning as their C counterparts.

	Instruction		Description
✓	`add`	R_d, R_s, R_t	$R_d = R_s + R_t$ (signed)
✓	`addu`	R_d, R_s, R_t	$R_d = R_s + R_t$ (unsigned)
✓	`addi`	R_d, R_s, R_t	$R_d = R_s + R_t$ (unsigned)
✓	`sub`	R_d, R_s, R_t	$R_d = R_s - R_t$ (signed)
✓	`subu`	R_d, R_s, R_t	$R_d = R_s - R_t$ (unsigned)
✓	`div`	R_d, R_s	Lo = R_s / R_t, Hi = R_s % R_t (int division, signed)
✓	`divu`	R_d, R_s	Lo = R_s / R_t, Hi = R_s % R_t (int division, unsigned)
	`div`	R_d, R_s, R_t	$R_d = R_s / R_t$ (int division, signed)
	`divu`	R_d, R_s, R_t	$R_d = R_s / R_t$ (int division, unsigned)
	`rem`	R_d, R_s, R_t	$R_d = R_s / R_t$ (int division, signed)
	`remu`	R_d, R_s, R_t	$R_d = R_s / R_t$ (int division, unsigned)
	`mul`	R_d, R_s, R_t	$R_d = R_s * R_t$ (signed)
✓	`mult`	R_d, R_s	Lo = R_s * R_t (Lo = bits 0..31, Hi = bits 32..63, signed)
✓	`multu`	R_d, R_s	Lo = R_s * R_t (Lo = bits 0..31, Hi = bits 32..63, unsigned)
✓	`and`	R_d, R_s, R_t	$R_d = R_s \& R_t$
✓	`and`	R_d, R_s, Imm	$R_d = R_s$ & Imm
	`neg`	R_d, R_s	$R_d = \sim R_s$
✓	`nor`	R_d, R_s, R_t	$R_d = !(R_s \mid R_t)$
	`not`	R_d, R_s	$R_d = !R_s$
✓	`or`	R_d, R_s, R_t	$R_d = R_s \mid R_t$
✓	`ori`	R_d, R_s, Imm	$R_d = R_s$ | Imm
✓	`xor`	R_d, R_s, R_t	$R_d = R_s \wedge R_t$
✓	`xori`	R_d, R_s, Imm	$R_d = R_s$ ^ Imm
✓	`sll`	R_d, R_t, Imm	$R_d = R_t$ << Imm
✓	`sllv`	R_d, R_s, R_t	$R_d = R_t << R_s$
✓	`srl`	R_d, R_t, Imm	$R_d = R_t$ >> Imm
✓	`srlv`	R_d, R_s, R_t	$R_d = R_t >> R_s$
	`move`	R_d, R_s	$R_d = R_s$

	Instruction		Description
✓	`mfhi`	R_d	R_d = Hi
✓	`mflo`	R_d	R_d = Lo
	`la`	R_d, Addr	R_d = Addr
	`li`	R_d, Imm	R_d = Imm
✓	`lui`	R_d, Imm	R_d[0..15] = 0, R_d[16..31] = Imm
✓	`lb`	R_d, Addr	R_d = byte at Mem[Addr] (sign extended, Addr could be Label(R_t))
✓	`lw`	R_d, Addr	R_d = word at Mem[Addr] (Addr could be Label(R_t))
✓	`sb`	R_s, Addr	Mem[Addr] = R_s (sign extended, Addr could be Label(R_t))
✓	`sw`	R_s, Addr	Mem[Addr] = R_s (Addr could be Label(R_t))
✓	`slt`	R_d, R_s, R_t	R_d = 1 if R_s<R_t, R_d = 0 otherwise (signed)
✓	`slti`	R_d, R_s, Imm	R_d = 1 if R_s<Imm, R_d = 0 otherwise (signed)
✓	`sltu`	R_d, R_s, R_t	R_d = 1 if R_s<R_t, R_d = 0 otherwise (unsigned)
✓	`beq`	R_s, R_t, Label	branch to Label if R_s=R_t (signed)
	`beqz`	R_s, Label	branch to Label if R_s=0 (signed)
	`bge`	R_s, R_t, Label	branch to Label if R_s≥R_t (signed)
✓	`bgez`	R_s, Label	branch to Label if R_s≥0 (signed)
✓	`bgezal`	R_s, Label	branch to Label and and `$ra`=PC+8 if R_s≥0 (signed)
	`bgt`	R_s, R_t, Label	branch to Label if R_s>R_t (signed)
	`bgtu`	R_s, R_t, Label	branch to Label if R_s>R_t (unsigned)
✓	`bgtz`	R_s, Label	branch to Label if R_s>0 (signed)
	`blt`	R_s, R_t, Label	branch to Label if R_s<R_t (signed)
	`bltu`	R_s, R_t, Label	branch to Label if R_s<R_t (unsigned)
✓	`bltz`	R_s, Label	branch to Label if R_s<0 (signed)
✓	`bltzl`	R_s, Label	branch to Label and `$ra`=PC+8 if R_s<0 (signed)
✓	`bne`	R_s, R_t, Label	branch to Label if R_s&neq;R_t
✓	`bnez`	R_s, Label	branch to Label if R_s&neq;0
✓	`j`	Label	jump to Label (PC = Label)
✓	`jal`	Label	jump to Label and Link (`$ra` = PC+8; PC = Label)

	Instruction		Description
✓	`jr`	R_s	jump to location in R_s
✓	`jalr`	R_s	jump to location in R_s and Link (`$ra` = PC+8; PC = Label)
	`syscall`		invoke system service; service given in `$v0`

System Services

The SPIM emulator provides a number of mechanisms for interacting with the host system. These services are invoked via the `syscall` pseudo-instruction after storing the service code in the register `$v0`.

Service	Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = char *	
read_int	5		integer in `$v0`
read_float	6		float in `$v0`
read_double	7		double in `$v0`
read_string	8	`$a0` = buffer, `$a1` = length	string in buffer
sbrk	9	`$a0` = # bytes	extend data segment
exit	10		program exits

Directives

The SPIM assembler supports a number of directives, which allow things to be specified at assembly time.

Directive	Description
`.text`	the instructions following this directive are placed in the text segment of memory
`.data`	the data defined following this directive is placed in the data segment of memory
`.space` *n*	allocate *n* unitialised bytes of space in the data segment of memory
`.word` $val_1, val_2, \ldots$	store values in successive words in the data segment of memory
`.byte` $val_1, val_2, \ldots$	store values in succesive bytes in the data segment of memory
`.asciiz "`*string*`"`	store '\0'-terminated string in the data segment of memory