



MIPS reference sheet

SPIM system calls			
Call code in \$v0	Service	Arguments	Returns Notes on call
1	Print integer	\$a0 = value to print	- value is signed
4	Print string	\$a0 = address of string to print	- string must be terminated with '\0'
5	Input integer	-	\$v0 = entered integer value is signed
8	Input string	\$a0 = address to store string at. \$a1 = maximum number of chars.	- returns if (\$a1 - 1) characters or Enter typed; the string is terminated with '\0'
10	Exit	-	- ends simulation
11	Print char	\$a0 = character to print	- upper 24 bits of \$a0 are ignored
12	Input char	-	\$v0 = sign or zero extended depending on the character underlying system

General-purpose registers

Number	Name	Purpose
R00	\$zero	provides the constant zero
R01	\$at	reserved: assembler translation of pseudos
R02 - R03	\$v0 - \$v1	system call code, return values
R04 - R07	\$a0 - \$a3	system call and function arguments
R08 - R15	\$t0 - \$t7	temporary storage (caller-saved)
R16 - R23	\$s0 - \$s7	temporary storage (callee-saved)
R24 - R25	\$t8 - \$t9	temporary storage (caller-saved)
R26 - R27	\$k0 - \$k1	reserved for kernel code
R28	\$gp	pointer to global area
R29	\$sp	stack pointer
R30	\$fp (or \$s8)	frame pointer
R31	\$ra	return address

Full function calling convention

Caller: <ul style="list-style-type: none"> saves temporary registers on stack passes arguments on stack calls function by: <code>jal fn_label</code> 		a) Function call: 	Callee: <ul style="list-style-type: none"> saves value of \$ra on stack saves value of \$fp on stack copies \$sp to \$fp allocates local variables on stack
Caller: <ul style="list-style-type: none"> clears arguments off stack restores temporary registers off stack uses return value in \$v0 		b) Function return: 	Callee: <ul style="list-style-type: none"> sets \$v0 to return value clears local variables off stack restores saved \$fp off stack restores saved \$ra off stack returns to caller by: <code>jr \$ra</code>

Assembler directives

.data	assemble into data segment
.text	assemble into text segment
.byte b1[, b2, ...]	allocate byte(s), with initial value(s)
.half h1[, h2, ...]	allocate halfword(s), with initial value(s)
.word w1[, w2, ...]	allocate word(s), with initial value(s)
.space n	allocate n bytes of uninitialized and unaligned space
.align n	align the next item to a 2 ⁿ byte boundary
.ascii "a string"	allocate ASCII string, do not terminate
.asciiz "a string"	allocate ASCII string, terminate with '\0'

Instruction Set

A partial instruction set is on the next page. The following conventions apply:

Instruction format:

- Rsrc, Rsrc1, Rsrc2:** source operand(s), - must be a register value(s)
- Src2:** source operand, - may be an immediate value or a register value
- Rdest:** destination, - must be a register
- Imm:** Immediate value, may be 32 or 16 bits
- Imm16:** Immediate 16-bit value
- Addr:** Address in the form: `offset(Rsrc)` ie. absolute address = `Rsrc + offset`
- Label:** label of an instruction
- ★:** pseudoinstruction
- Immediate form:**
 - : no immediate form, or this *is* the immediate form
 - ★ : immediate form synthesized as pseudoinstruction
- Unsigned form** (append 'u' to instruction name):
 - : no unsigned form, or this *is* the unsigned form

	Instruction format	Meaning	Operation	Immediate form	Unsigned form (u)
Arithmetical operations	add Rdest, Rsrc1, Src2	Add	Rdest = Rsrc1 + Src2	addi	no overflow trap
	sub Rdest, Rsrc1, Src2	Subtract	Rdest = Rsrc1 - Src2	★	no overflow trap
	mul Rdest, Rsrc1, Src2 ★	Multiply	Rdest = Rsrc1 * Src2	★	unsigned operands
	mulo Rdest, Rsrc1, Src2 ★	Multiply (with 32-bit overflow)	Rdest = Rsrc1 * Src2	★	unsigned operands
	mult Rsrc1, Rsrc2	Multiply (machine instruction)	Hi:Lo = Rsrc1 * Rsrc2	-	unsigned operands
	div Rdest, Rsrc1, Src2 ★	Divide	Rdest = Rsrc1 / Src2	★	unsigned operands
	div Rsrc1, Rsrc2	Divide (machine instruction)	Lo = Rsrc1 / Rsrc2; Hi = Rsrc1 % Rsrc2	-	unsigned operands
	rem Rdest, Rsrc1, Src2 ★	Remainder	Rdest = Rsrc1 % Src2	★	unsigned operands
	neg Rdest, Rsrc ★	Negate	Rdest = -Rsrc	-	no overflow trap
Bitwise Logic	and Rdest, Rsrc1, Src2	Bitwise AND	Rdest = Rsrc1 & Src2	andi	-
	or Rdest, Rsrc1, Src2	Bitwise OR	Rdest = Rsrc1 Src2	ori	-
	xor Rdest, Rsrc1, Src2	Bitwise Exclusive OR	Rdest = Rsrc1 ^ Src2	xori	-
	nor Rdest, Rsrc1, Src2	Bitwise NOR	Rdest = ~(Rsrc1 Src2)	★	-
	not Rdest, Rsrc ★	Bitwise NOT	Rdest = ~(Rsrc)	-	-
Shifts	sll Rdest, Rsrc1, Src2	Shift Left Logical	Rdest = Rsrc1 << Src2	-	-
	srl Rdest, Rsrc1, Src2	Shift Right Logical	Rdest = Rsrc1 >> Src2 (MSB=0)	-	-
	sra Rdest, Rsrc1, Src2	Shift Right Arithmetic	Rdest = Rsrc1 >> Src2 (MSB preserved)	-	-
Reg Moves	move Rdest, Rsrc ★	Move	Rdest = Rsrc	-	-
	mfhi Rdest	Move from Hi	Rdest = Hi	-	-
	mflo Rdest	Move from Lo	Rdest = Lo	-	-
Ld Const	li Rdest, Imm ★	Load immediate	Rdest = Imm	-	-
	lui Rdest, Imm16	Load upper immediate	Rdest = Imm16 << 16	-	-
	la Rdest, Addr (or label) ★	Load address	Rdest = Addr (or Rdest=label)	-	-
Ld Data	lb Rdest, Addr (or label ★)	Load byte	Rdest = mem8[Addr]	-	zero-extends data
	lh Rdest, Addr (or label ★)	Load halfword	Rdest = mem16[Addr]	-	zero-extends data
	lw Rdest, Addr (or label ★)	Load word	Rdest = mem32[Addr]	-	-
Store	sb Rsrc2, Addr (or label ★)	Store byte	mem8[Addr] = Rsrc2	-	-
	sh Rsrc2, Addr (or label ★)	Store halfword	mem16[Addr] = Rsrc2	-	-
	sw Rsrc2, Addr (or label ★)	Store word	mem32[Addr] = Rsrc2	-	-
Conditional Branches	beq Rsrc1, Src2, label	Branch if equal	if (Rsrc1 == Src2), PC = label	★	-
	bne Rsrc1, Src2, label	Branch if not equal	if (Rsrc1 != Src2), PC = label	★	-
	blt Rsrc1, Src2, label ★	Branch if less than	if (Rsrc1 < Src2), PC = label	★	unsigned operands
	ble Rsrc1, Src2, label ★	Branch if less than or equal	if (Rsrc1 <= Src2), PC = label	★	unsigned operands
	bgt Rsrc1, Src2, label ★	Branch if greter than	if (Rsrc1 > Src2), PC = label	★	unsigned operands
	bge Rsrc1, Src2, label ★	Branch if greter than or equal	if (Rsrc1 >= Src2), PC = label	★	unsigned operands
	slt Rdest, Rsrc1, Src2	Set if less than	if (Rsrc1 < Src2), Rdest = 1 else Rdest = 0	slti	unsigned operands
Jumps/Calls	j label	Jump	PC = label	-	-
	jal label	Jump and link	\$ra = PC + 4 ; PC = label	-	-
	jr Rsrc	Jump register	PC = Rsrc	-	-
	jalr Rsrc	Jump and link register	\$ra = PC + 4 ; PC = Rsrc	-	-
	syscall	System call	depends on call code in \$v0	-	-