MIPS reference sheet

Table 1: **SPIM system calls**

Call code ($v0)	Service	Arguments	Returns	Notes
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'
9	Allocate memory	$a0 = number of bytes	$v0 = address of first byte	-
10	Exit	-	-	ends simulation

Table 2: **General-purpose registers**

Number	Name	Purpose
R00	$zero	provides constant zero
R01	$at	reserved for assembler
R02, R03	$v0, $v1	system call code, return value
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	frame pointer
R31	$ra	return address

Table 3: **Assembler directives**

.data	assemble into data segment
.text	assemble into text (code) 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, unaligned space
.align n	align the next item to a 2^n-byte boundary
.ascii "string"	allocate ASCII string, do not terminate
.asciiz "string"	allocate ASCII string, terminate with '\0'

Table 4: **Function calling convention**

On function call:

Caller:	**Callee:**
saves temporary registers on stack	saves $ra and $fp on stack
passes arguments on stack	copies $sp to $fp
calls function using jal fn_label	allocates local variables on stack

On function return:

Callee:	**Caller:**
sets $v0 to return value	clears arguments off stack
clears local variables off stack	restores temporary registers off stack
restores saved $fp and $ra off stack	uses return value in $v0
returns to caller with jr $ra	

Table 5: **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 pseduoinstruction

Unsigned form (append 'u' to instruction name):
- : no unsigned form, or this is the unsigned form

Table 6: **MIPS instruction set**

Instruction format	Meaning	Operation	Immediate form	Unsigned form(u)
add Rdest, Rsrc1, Rsrc2	Add	Rdest = Rsrc1 + Rsrc2	addi	no overflow trap
sub Rdest, Rsrc1, Rsrc2	Subtract	Rdest = Rsrc1 - Rsrc2	⋆	no overflow trap
mul Rdest, Rsrc1, Rsrc2 ⋆	Multiply	Rdest = Rsrc1 * Rsrc2	⋆	unsigned operands
mulo Rdest, Rsrc1, Rsrc2 ⋆	Multiply (with 32-bit overflow)	Rdest = Rsrc1 * Rsrc2	⋆	unsigned operands
mult Rsrc1, Rsrc2	Multiply (machine instruction)	Hi:Lo = Rsrc1 * Rsrc2	-	unsigned operands
div Rdest, Rsrc1, Rsrc2 ⋆	Divide	Rdest=Rsrc1/Rsrc2	⋆	unsigned operands
div Rsrc1, Rsrc2	Divide (machine instruction)	Lo = Rsrc1/Rsrc2; Hi = Rsrc1 % Rsrc2	-	unsigned operands
rem Rdest, Rsrc1, Rsrc2 ⋆	Remainder	Rdest = Rsrc1 % Rsrc2	⋆	unsigned operands
neg Rdest, Rsrc ⋆	Negate	Rdest = -Rsrc1	-	no overflow trap
and Rdest, Rsrc1, Rsrc2	Bitwise AND	Rdest = Rsrc1 & Rsrc2	andi	-
or Rdest, Rsrc1, Rsrc2	Bitwise OR	Rdest = Rsrc1 | Rsrc2	ori	-
xor Rdest, Rsrc1, Rsrc2	Bitwise XOR	Rdest = Rsrc1 ∧ Rsrc2	xori	-
nor Rdest, Rsrc1, Rsrc2	Bitwise NOR	Rdest = ∼(Rsrc1 | Rsrc2)	⋆	-
not Rdest, Rsrc ⋆	Bitwise NOT	Rdest = ∼(Rsrc)	–	-
sll Rdest, Rsrc1, Rsrc2	Shift Left Logical	Rdest = Rsrc1 << Rsrc2	-	-
srl Rdest, Rsrc1, Rsrc2	Shift Right Logical	Rdest = Rsrc1 >> Rsrc2 (MSB=0)	-	-
sra Rdest, Rsrc1, Rsrc2	Shift Right Arithmetic	Rdest = Rsrc1 >> Rsrc2 (MSB preserved)	-	-
move Rdest, Rsrc ⋆	Move	Rdest=Rsrc	-	-
mfhi Rdest	Move from Hi	Rdest = Hi	-	-
mflo Rdest	Move from Lo	Rdest = Lo	-	-
li Rdest, Imm ⋆	Load immediate	Rdest=Imm	-	-
lui Rdest, Imm16	Load upper immediate	Rdest=Imm16 << Imm	-	-
la Rdest, Addr(or label) ⋆	Load Address	Rdest=Addr (or Rdest=label)	-	-
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]	-	-
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	-	-
beq Rsrc1, Rsrc2, label	Branch if equal	if (Rsrc1 == Rsrc2) PC = label	⋆	-
bne Rsrc1, Rsrc2, label	Branch if not equal	if (Rsrc1 != Rsrc2) PC = label	⋆	-
blt Rsrc1, Rsrc2, label ⋆	Branch if less than	if (Rsrc1 < Rsrc2) PC = label	⋆	unsigned operands
ble Rsrc1, Rsrc2, label ⋆	Branch if less than or equal	if (Rsrc1 <= Rsrc2) PC = label	⋆	unsigned operands
bgt Rsrc1, Rsrc2, label ⋆	Branch if greater than	if (Rsrc1 > Rsrc2) PC = label	⋆	unsigned operands
bge Rsrc1, Rsrc2, label ⋆	Branch if greater than or equal	if (Rsrc1 >= Rsrc2) PC = label	⋆	unsigned operands
slt Rdest, Rsrc1, Rsrc2	Set if less than	if (Rsrc1 < Rsrc2) Rdest=1 else Rdest=0	slti	unsigned operands
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`	-	-