MIPS Instruction Set

An overview of the instruction set of the MIPS32 architecture as implemented by the mipsy and SPIM emulators. Adapted from reference documents from the University of Stuttgart and Drexel University, from material in the appendix of Patterson and Hennessey's *Computer Organization and Design*, and from the MIPS32 (r5.04) Instruction Set reference.

Registers

back to top

As implemented by mipsy, MIPS has 32× 32-bit general purpose registers as well as two special registers `Hi` and `Lo` for manipulating 64-bit integer quantities.

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

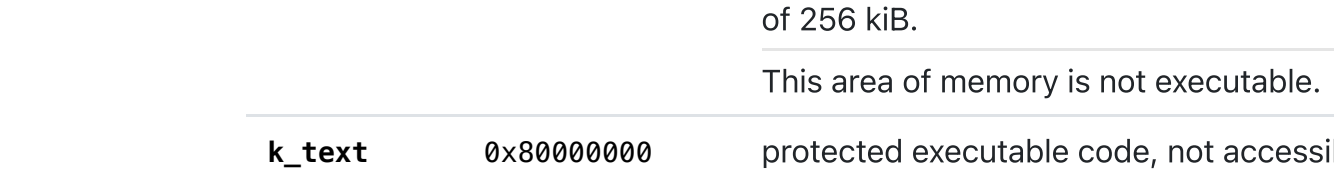
Regs	Names	Description
$0	$zero	the value **0**; writes are discarded
$1	$at	**a**ssembler **t**emporary; reserved for assembler use
$2 $3	$v0 $v1	**v**alue from expression evaluation or function return
$4 $5	$a0 $a1	first four **a**rguments to a function/subroutine
$6 $7	$a2 $a3	
$8 $9	$t0 $t1	**t**emporary; callers relying on their values must save them before calling subroutines as they may be overwritten
$10 $11	$t2 $t3	
$12 $13	$t4 $t5	
$14 $15	$t6 $t7	
$16 $17	$s0 $s1	**s**aved; subroutines must guarantee their values are unchanged (by, for example, restoring them)
$18 $19	$s2 $s3	
$20 $21	$s4 $s5	
$22 $23	$s6 $s7	
$24 $25	$t8 $t9	**t**emporary; callers relying on their values must save them before calling subroutines as they may be overwritten
$26 $27	$k0 $k1	for **k**ernel use; may change unexpectedly — avoid using in user programs
$28	$gp	**g**lobal **p**ointer (address of global area)
$29	$sp	**s**tack **p**ointer (top of stack)

Regs	Names	Description
$30	`$fp`	**f**rame **p**ointer (bottom of current stack frame); if not using a frame pointer, becomes a **s**ave register
$31	`$ra`	**r**eturn **a**ddress of most recent caller

Memory

back to top

mipsy's memory is partitioned as follows:



Segment	Base	Description
text	`0x00400000`	where user program code resides; In mipsy, it is the only area of memory where instructions are executable; its initial size is 256 kiB. This is the only area of memory where instructions are executable. In mipsy,this area of memory is also writeable. On a real system, this area of memory would generally be read-only.
data	`0x10000000`	where user data resides; its initial size is 256 kiB, but its size is not fixed, and can be changed with the *sbrk* syscall up to a maximum of 1 MiB. This area of memory is not executable.
stack	`0x7fffeff`	the function call stack; grows towards negative addresses. its initial size is 64 kiB, but it will grow as needed up to a maximum of 256 kiB. This area of memory is not executable.
k_text	`0x80000000`	protected executable code, not accessible in user mode; in a real system, the operating system kernel's code would be mapped here. In mipsy, the entry point is loaded here; its initial size is 64 kiB
k_data	`0x90000000`	protected data, not accessible in user mode; in a real system, the operating system's data would be mapped here. In mipsy, the entry point's data is loaded here; its initial size is 64 kiB; but it will grow as needed up to a maximum of 1 MiB.

Syntax

back to top

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

```
[label:]   opcode   [operand₁[, operand₂[, operand₃]]]   [# comment]
```

The number of operands for each instruction varies, but could be between zero and three. In the descriptions below, the following notation is used to describe instruction operands.

Operand	Description
R_n	a register — commonly, R_s and R_t are sources, and R_d is a destination; registers may be specified either by a numeric name (`$0` to `$31`), or by a symbolic name (`$sN` , `$tN` , etc.)

Operand	Description
Imm	a literal constant value, or "immediate": may be specified as an octal, decimal, hexadecimal, or character literal; if followed by a number (e.g., Imm_{16}) that specifies the width in bits and implies the range of the value.
Label	a symbolic name which is associated with a memory address
Addr	a memory address, in one of the formats described below

Many instructions have an address operand; these may be written in a number of formats:

Useful for accessing arrays.

Format	Address
Label	a symbolic name which is associated with a memory address
(*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 Useful for accessing the stack.
***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* Useful for accessing structs.
***Label* ± *Imm*(*R_n*)**	the sum of *Label*'s address and *Imm* and the value stored in register R_n Useful for accessing arrays of structs.

Instructions

back to top

The mipsy emulator implements instructions from the MIPS32 instruction set, as well as *pseudo-instructions* (which look like MIPS instructions, but which aren't provided on real hardware). Real MIPS instructions are marked with a ✓. All other instructions are pseudo-instructions. Operators in expressions have the same meaning as their C counterparts.

	Instruction		Description		Encoding
	CPU Arithmetic Instructions				back to top
✓	ADD	R_d, R_s, R_t	R_d = R_s + R_t	INTEGER OVERFLOW	000000ssssstttttddddd00000100000
✓	ADDI	R_t, R_s, Imm_{16}	R_t = R_s + Imm_{16}	INTEGER OVERFLOW	001000ssssstttttIIIIIIIIIIIIIIII
✓	ADDU	R_d, R_s, R_t	R_d = R_s + R_t		000000ssssstttttddddd00000100001
✓	ADDIU	R_t, R_s, Imm_{16}	R_t = R_s + Imm_{16}		001001ssssstttttIIIIIIIIIIIIIIII
✓	SUB	R_d, R_s, R_t	R_d = R_s − R_t	INTEGER OVERFLOW	000000ssssstttttddddd00000100010
✓	SUBU	R_d, R_s, R_t	R_d = R_s − R_t		000000ssssstttttddddd00000100011
✓	MUL	R_d, R_s, R_t	R_d = R_s * R_t		011100ssssstttttddddd00000000010
✓	MULT	R_s, R_t	(Hi,Lo) = R_s * R_t		000000ssssstttttt0000000000011000
✓	MULTU	R_s, R_t	(Hi,Lo) = R_s * R_t		000000ssssstttttt0000000000011001
✓	MADD	R_s, R_t	(Hi,Lo) += R_s * R_t		011100ssssstttttt0000000000000000
✓	MADDU	R_s, R_t	(Hi,Lo) += R_s * R_t		011100ssssstttttt0000000000000001
✓	MSUB	R_s, R_t	(Hi,Lo) -= R_s * R_t		011100ssssstttttt0000000000000100
✓	MSUBU	R_s, R_t	(Hi,Lo) -= R_s * R_t		011100ssssstttttt0000000000000101
✓	DIV	R_s, R_t	Lo = R_s / R_t; Hi = R_s % R_t		000000ssssstttttt0000000000011010
✓	DIVU	R_s, R_t	Lo = R_s / R_t; Hi = R_s % R_t		000000ssssstttttt0000000000011011
	DIV	R_d, R_s, R_t	R_d = R_s / R_t		pseudo-instruction
	DIVU	R_d, R_s, R_t	R_d = R_s / R_t		pseudo-instruction

	Instruction		Description		Encoding
	REM	R_d, R_s, R_t	R_d = R_s % R_t		pseudo-instruction
	REMU	R_d, R_s, R_t	R_d = R_s % R_t		pseudo-instruction
✓	CLO	R_d, R_s	R_d = count_leading_ones(R_s)		011100ssssstttttddddd00000100001
✓	CLZ	R_d, R_s	R_d = count_leading_zeroes(R_s)		011100ssssstttttddddd00000100000
✓	SEB	R_d, R_s	R_d = sign_extend (R_s & 0x000000ff)		01111100000tttttddddd10000100000
✓	SEH	R_d, R_s	R_d = sign_extend (R_s & 0x0000ffff)		01111100000tttttddddd11000100000
	SEQ	R_d, R_s, R_t	R_d = R_s == R_t		pseudo-instruction
	SNE	R_d, R_s, R_t	R_d = R_s != R_t		pseudo-instruction
	SLE	R_d, R_s, R_t	R_d = R_s <= R_t		pseudo-instruction
	SLEU	R_d, R_s, R_t	R_d = R_s <= R_t		pseudo-instruction
✓	SLT	R_d, R_s, R_t	R_d = R_s < R_t		000000ssssstttttddddd00000101010
✓	SLTU	R_d, R_s, R_t	R_d = R_s < R_t		000000ssssstttttddddd00000101011
	SGT	R_d, R_s, R_t	R_d = R_s > R_t		pseudo-instruction
	SGTU	R_d, R_s, R_t	R_d = R_s > R_t		pseudo-instruction
	SGE	R_d, R_s, R_t	R_d = R_s >= R_t		pseudo-instruction
	SGEU	R_d, R_s, R_t	R_d = R_s >= R_t		pseudo-instruction
✓	SLTI	R_t, R_s, Imm_{16}	R_t = R_s < Imm_{16}		001010ssssstttttIIIIIIIIIIIIIIII
✓	SLTIU	R_t, R_s, Imm_{16}	R_t = R_s < Imm_{16}		001011ssssstttttIIIIIIIIIIIIIIII
	ABS	R_t, R_s	R_t = |R_s|		pseudo-instruction
	NEG	R_t, R_s	R_t = −R_s	INTEGER OVERFLOW	SUB R_t, $0, R_s
	NEGU	R_t, R_s	R_t = −R_s		SUBU R_t, $0, R_s
	CPU Logical Instructions				back to top
✓	AND	R_d, R_s, R_t	R_d = R_s & R_t		000000ssssstttttddddd00000100100
✓	ANDI	R_t, R_s, Imm_{16}	R_t = R_s & Imm_{16}		001100ssssstttttIIIIIIIIIIIIIIII
✓	OR	R_d, R_s, R_t	R_d = R_s | R_t		000000ssssstttttddddd00000100101
✓	ORI	R_t, R_s, Imm_{16}	R_t = R_s | Imm_{16}		001101ssssstttttIIIIIIIIIIIIIIII
✓	NOR	R_d, R_s, R_t	R_d = ~(R_s | R_t)		000000ssssstttttddddd00000100111
✓	XOR	R_d, R_s, R_t	R_d = R_s ^ R_t		000000ssssstttttddddd00000100110
✓	XORI	R_t, R_s, Imm_{16}	R_t = R_s ^ Imm_{16}		001110ssssstttttIIIIIIIIIIIIIIII
	NOT	R_t, R_s	R_t = ~R_s		NOR R_t, R_s, $0
	CPU Shift Instructions				back to top
	ROL	R_d, R_t, R_s	R_d = R_t rot < R_s		pseudo-instruction
	ROR	R_d, R_t, R_s	R_d = R_t rot > R_s		pseudo-instruction
✓	ROTR	R_d, R_t, a	R_d = R_t rot > a		00000000001tttttddddaaaaa000010
✓	ROTRV	R_d, R_t, R_s	R_d = R_t rot > R_s		000000ssssstttttddddd00001000110
✓	SLL	R_d, R_t, a	R_d = R_t << a		00000000000tttttddddaaaaa000000
✓	SLLV	R_d, R_t, R_s	R_d = R_t << R_s		000000ssssstttttddddd00000000100
✓	SRA	R_d, R_t, a	R_d = R_t >> a	SIGN EXTENDED	00000000000tttttddddaaaaa000011
✓	SRAV	R_d, R_t, R_s	R_d = R_t >> R_s	SIGN EXTENDED	000000ssssstttttddddd00000000111
✓	SRL	R_d, R_t, a	R_d = R_t >> a		00000000000tttttddddaaaaa000010
✓	SRLV	R_d, R_t, R_s	R_d = R_t >> R_s		000000ssssstttttddddd00000000110
	CPU Load, Store, and Memory Control Instructions				back to top

	Instruction		Description		Encoding
	LI	R_t, *Imm*	R_t = *Imm*		pseudo-instruction
	LA	R_t, *Label*	R_t = *Label*		pseudo-instruction
✓	LUI	R_t, Imm_{16}	R_t = Imm_{16} << 16		00111100000ttttIIIIIIIIIIIIIIII
✓	LB	R_t, $Offset_{16}(R_b)$	R_t = RAM[R_b + $Offset_{16}$]	ADDRESS ERROR SIGN EXTENDED	100000bbbbbttttt0000000000000000
✓	LBU	R_t, $Offset_{16}(R_b)$	R_t = RAM[R_b + $Offset_{16}$]	ADDRESS ERROR	100100bbbbbttttt0000000000000000
✓	LH	R_t, $Offset_{16}(R_b)$	R_t = RAM[R_b + $Offset_{16}$]	ADDRESS ERROR SIGN EXTENDED	100001bbbbbttttt0000000000000000
✓	LHU	R_t, $Offset_{16}(R_b)$	R_t = RAM[R_b + $Offset_{16}$]	ADDRESS ERROR	100101bbbbbttttt0000000000000000
✓	LW	R_t, $Offset_{16}(R_b)$	R_t = RAM[R_b + $Offset_{16}$]	ADDRESS ERROR	100011bbbbbttttt0000000000000000
✓	SB	R_t, $Offset_{16}(R_b)$	RAM[R_b + $Offset_{16}$] = R_t	ADDRESS ERROR	101000bbbbbttttt0000000000000000
✓	SH	R_t, $Offset_{16}(R_b)$	RAM[R_b + $Offset_{16}$] = R_t	ADDRESS ERROR	101001bbbbbttttt0000000000000000
✓	SW	R_t, $Offset_{16}(R_b)$	RAM[R_b + $Offset_{16}$] = R_t	ADDRESS ERROR	101011bbbbbttttt0000000000000000
	PUSH	R_s	$sp -= 4 RAM[$sp] = R_s		pseudo-instruction (mipsy only)
	POP	R_s	R_s = RAM[$sp] $sp += 4		pseudo-instruction (mipsy only)
	BEGIN		$sp -= 4 RAM[$sp] = $fp $fp = $sp		pseudo-instruction (mipsy only)
	END		$sp = $fp + 4 $fp = RAM[$fp]		pseudo-instruction (mipsy only)
	CPU Move Instructions				back to top
✓	MFHI	R_d	R_d = HI		00000000000000000ddddd00000010000
✓	MFLO	R_d	R_d = LO		00000000000000000ddddd00000010010
✓	MTHI	R_d	HI = R_d		000000sssss00000000000000010001
✓	MTLO	R_d	LO = R_d		000000sssss00000000000000010011
	MOVE	R_t, R_s	R_t = R_s		ADDU R_t, $0, R_s
✓	MOVZ	R_d, R_s, R_t	IF R_t == 0 THEN R_d = R_s		000000ssssstttttddddd00000001010
✓	MOVN	R_d, R_s, R_t	IF R_t != 0 THEN R_d = R_s		000000ssssstttttddddd00000001011
	CPU Branch and Jump Instructions				back to top
	B	$Offset_{16}$	PC += $Offset_{16}$ << 2		00010000000000000000000000000000 BEQ $t0, $t0, $Offset_{16}$
✓	BEQ	R_s, R_t, $Offset_{16}$	IF R_s == R_t THEN PC += $Offset_{16}$ << 2		000100ssssstttttt0000000000000000
	BEQ	R_s, *Imm*, $Offset_{16}$	IF R_s == *Imm* THEN PC += $Offset_{16}$ << 2		pseudo-instruction
	BEQZ	R_s, $Offset_{16}$	IF R_s == 0 THEN PC += $Offset_{16}$ << 2		BEQ $0, R_s, $Offset_{16}$
✓	BNE	R_s, R_t, $Offset_{16}$	IF R_s != R_t THEN PC += $Offset_{16}$ << 2		000101ssssstttttt0000000000000000
	BNE	R_s, *Imm*, $Offset_{16}$	IF R_s != *Imm* THEN PC += $Offset_{16}$ << 2		pseudo-instruction
	BNEZ	R_s, $Offset_{16}$	IF R_s != 0 THEN PC += $Offset_{16}$ << 2		BNE $0, R_s, $Offset_{16}$
	BGE	R_s, R_t, $Offset_{16}$	IF R_s >= R_t THEN PC += $Offset_{16}$ << 2		SLT $at, R_s, R_t BEQ $0, $at, $Offset_{16}$

	Instruction		Description		Encoding
	BGE	R_s, *Imm*, *Offset*$_{16}$	IF R_s >= *Imm* THEN PC += *Offset*$_{16}$ << 2		pseudo-instruction
	BGEU	R_s, R_t, *Offset*$_{16}$	IF R_s >= R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_s, R_t BEQ $0, $at, *Offset*$_{16}$
	BGEU	R_s, *Imm*, *Offset*$_{16}$	IF R_s >= *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BGEZ	R_s, *Offset*$_{16}$	IF R_s >= 0 THEN PC += *Offset*$_{16}$ << 2		000001sssss0000100000000000000000
	BGT	R_s, R_t, *Offset*$_{16}$	IF R_s > R_t THEN PC += *Offset*$_{16}$ << 2		SLT $at, R_t, R_s BNE $0, $at, *Offset*$_{16}$
	BGT	R_s, *Imm*, *Offset*$_{16}$	IF R_s > *Imm* THEN PC += *Offset*$_{16}$ << 2		pseudo-instruction
	BGTU	R_s, R_t, *Offset*$_{16}$	IF R_s > R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_t, R_s BNE $0, $at, *Offset*$_{16}$
	BGTU	R_s, *Imm*, *Offset*$_{16}$	IF R_s > *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BGTZ	R_s, *Offset*$_{16}$	IF R_s > 0 THEN PC += *Offset*$_{16}$ << 2		000111sssss0000000000000000000000
	BLT	R_s, R_t, *Offset*$_{16}$	IF R_s < R_t THEN PC += *Offset*$_{16}$ << 2		SLT $at, R_s, R_t BNE $0, $at, *Offset*$_{16}$
	BLT	R_s, *Imm*, *Offset*$_{16}$	IF R_s < *Imm* THEN PC += *Offset*$_{16}$ << 2		pseudo-instruction
	BLTU	R_s, R_t, *Offset*$_{16}$	IF R_s < R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_s, R_t BNE $0, $at, *Offset*$_{16}$
	BLTU	R_s, *Imm*, *Offset*$_{16}$	IF R_s < *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BLTZ	R_s, *Offset*$_{16}$	IF R_s < 0 THEN PC += *Offset*$_{16}$ << 2		000001sssss0000000000000000000000
	BLE	R_s, R_t, *Offset*$_{16}$	IF R_s <= R_t THEN PC += *Offset*$_{16}$ << 2		SLT $at, R_t, R_s BEQ $0, $at, *Offset*$_{16}$
	BLE	R_s, *Imm*, *Offset*$_{16}$	IF R_s <= *Imm* THEN PC += *Offset*$_{16}$ << 2		pseudo-instruction
	BLEU	R_s, R_t, *Offset*$_{16}$	IF R_s <= R_t THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	SLTU $at, R_t, R_s BEQ $0, $at, *Offset*$_{16}$
	BLEU	R_s, *Imm*, *Offset*$_{16}$	IF R_s <= *Imm* THEN PC += *Offset*$_{16}$ << 2	UNSIGNED COMPARISON	pseudo-instruction
✓	BLEZ	R_s, *Offset*$_{16}$	IF R_s <= 0 THEN PC += *Offset*$_{16}$ << 2		000110sssss0000000000000000000000
✓	J	*Address*$_{26}$	PC = PC[31–28] && *Address*$_{26}$ << 2		000010AAAAAAAAAAAAAAAAAAAAAAAA
✓	JAL	*Address*$_{26}$	$ra = PC + 4 PC = PC[31–28] && *Address*$_{26}$ << 2		000011AAAAAAAAAAAAAAAAAAAAAAAA
✓	JR	R_s	PC = R_s		000000sssss0000000000hhhhh001000
✓	JALR	R_s	$ra = PC + 4 PC = R_s		000000sssss0000011111hhhhh001001
✓	JALR	R_d, R_s	R_d = PC + 4 PC = R_s		000000sssss00000dddddhhhhh001001
	CPU Trap Instructions				back to top
✓	SYSCALL		perform a system call		000000cccccccccccccccccccc001100
✓	BREAK		trigger a breakpoint		000000cccccccccccccccccccc001101

	Instruction		Description		Encoding
✓	TEQ	R_s, R_t	IF R_s == R_t THEN trigger a breakpoint		000000ssssstttttcccccccccc110100
✓	TEQI	R_s, Imm_{16}	IF R_s == Imm_{16} THEN trigger a breakpoint		000001sssss01100IIIIIIIIIIIIIIII
✓	TNE	R_s, R_t	IF R_s != R_t THEN trigger a breakpoint		000000ssssstttttcccccccccc110110
✓	TNEI	R_s, Imm_{16}	IF R_s != Imm_{16} THEN trigger a breakpoint		000001sssss01110IIIIIIIIIIIIIIII
✓	TGE	R_s, R_t	IF R_s >= R_t THEN trigger a breakpoint		000000ssssstttttcccccccccc110000
✓	TGEU	R_s, R_t	IF R_s >= R_t THEN trigger a breakpoint	UNSIGNED COMPARISON	000000ssssstttttcccccccccc110001
✓	TGEI	R_s, Imm_{16}	IF R_s >= Imm_{16} THEN trigger a breakpoint		000001sssss01000IIIIIIIIIIIIIIII
✓	TGEIU	R_s, Imm_{16}	IF R_s >= Imm_{16} THEN trigger a breakpoint	UNSIGNED COMPARISON	000001sssss01001IIIIIIIIIIIIIIII
	TGT	R_s, R_t	IF R_s > R_t THEN trigger a breakpoint		pseudo-instruction
	TGTU	R_s, R_t	IF R_s > R_t THEN trigger a breakpoint	UNSIGNED COMPARISON	pseudo-instruction
	TGTI	R_s, Imm_{16}	IF R_s > Imm_{16} THEN trigger a breakpoint		pseudo-instruction
	TGTIU	R_s, Imm_{16}	IF R_s > Imm_{16} THEN trigger a breakpoint	UNSIGNED COMPARISON	pseudo-instruction
✓	TLT	R_s, R_t	IF R_s < R_t THEN trigger a breakpoint		000000ssssstttttcccccccccc110010
✓	TLTU	R_s, R_t	IF R_s < R_t THEN trigger a breakpoint	UNSIGNED COMPARISON	000000ssssstttttcccccccccc110011
✓	TLTI	R_s, Imm_{16}	IF R_s < Imm_{16} THEN trigger a breakpoint		000001sssss01010IIIIIIIIIIIIIIII
✓	TLTIU	R_s, Imm_{16}	IF R_s < Imm_{16} THEN trigger a breakpoint	UNSIGNED COMPARISON	000001sssss01011IIIIIIIIIIIIIIII
	TLE	R_s, R_t	IF R_s <= R_t THEN trigger a breakpoint		pseudo-instruction
	TLEU	R_s, R_t	IF R_s <= R_t THEN trigger a breakpoint	UNSIGNED COMPARISON	pseudo-instruction
	TLEI	R_s, Imm_{16}	IF R_s <= Imm_{16} THEN trigger a breakpoint		pseudo-instruction
	TLEIU	R_s, Imm_{16}	IF R_s <= Imm_{16} THEN trigger a breakpoint	UNSIGNED COMPARISON	pseudo-instruction

CPU Control Instructions

back to top

NOP		do nothing	00000000000000000000000000000000 SLL $0, $0, 0

System Services

back to top

The mipsy emulator provides a number of mechanisms for interacting with the host system, to provide input and output, file operations, and other miscellaneous services, which we refer to as "system calls" or "syscalls". These are invoked via the `syscall` instruction after storing the service code in the register `$v0`.

`$v0` =	Arguments	Result	Description
Printing Values			back to top
1	`$a0 : int`		*print_int*: Print the integer in `$a0` to the console as a signed decimal.
2	`$f12 : float`		*print_float*: Print the float in `$f12` to the console as a `%.8f`.
3	`$f12 / $f13 : double`		*print_double*: Print the double in `$f12 / $f13` to the console as a `%.18g`
4	`$a0 : char *`		*print_string*: Print the nul-terminated array of bytes referenced by `$a0` to the console as an ASCII string.
11	`$a0 : char`		*print_character*: Print the character in `$a0`, analogous to *putchar*.
Reading Values			back to top
5		`$v0 : int`	*read_int*: Read an integral value from the console, with *atol*'s semantics, into register `$v0`
6		`$f0 : float`	*read_float*: Read a floating-point value from the console, with *atof*'s semantics, into register `$f0`
7		`$f0 / $f1 : double`	*read_double*: Read a double-precision floating-point value from the console, with *atof*'s semantics, into registers `$f0 / $f1`
8	`$a0 : char *;` `$a1 : int`		*read_string*: Read a string into the provided buffer (referenced by `$a0`); up to *size* (given in `$a1`) bytes are read, and the result is nul-terminated.
12		`$v0 : char`	*read_character*: Read the next character from the console into register `$v0`; analogous to *getchar*
File Manipulation			back to top

13	`$a0 : char *;`
	`$a1 : int;`
	`$a2 : mode_t`
		`$v0 : fd`	*open*: Open the file specified by *name* (referenced by `$a0`) in a particular access mode as specified by *flags* (given by `$a1`), and, if it is to be created, with mode *mode* (given by `$a2`). Returns a *file descriptor*, a small non-negative `int`. Effectively, *open*.

14	`$a0 : fd;`
	`$a1 : void *;`
	`$a2 : int`
		`$v0 : int`	*read*: On the file given by the file descriptor *fd* (given in `$a0`), read *len* bytes (given by `$a2`) into *buffer* (given by `$a1`). Returns the number of bytes read, or -1 if an error occurred. Effectively, *read*.

15	`$a0 : fd;`
	`$a1 : void *;`
	`$a2 : int`
		`$v0 : int`	*write*: On the file given by the file descriptor *fd* (given in `$a0`), write *len* bytes (given by `$a2`) from *buffer* (given by `$a1`). Returns the number of bytes written, or -1 if an error occurred. Effectively, *write*.

16	`$a0 : fd`	`$v0 : int`	*close*: Close the file given by the file descriptor *fd* (given in `$a0`). Returns 0 if successful, or -1 if an error occurred. Effectively, *close*.
Process Services			back to top
9	`$a0 : int`		*sbrk*: Extend the `.data` segment by adding `$a0` bytes; a primitive useful for, e.g., implementing *malloc*
10			*exit*: The program exits with code 0.
17	`$a0 : int`		*exit2*: The program exits with *code* (given in `$a0`).

Directives

back to top

The mipsy 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
`.ktext`	the instructions following this directive are placed in the `kernel text segment` of memory
`.kdata`	the data defined following this directive is placed in the `kernel data segment` of memory
`.align N`	arrange that the next datum is stored with appropriate alignment (that the lower *N* bits of its address are set to zero) by inserting enough padding — nearly always automatically done; a half word requires `.align 1` (for two bytes), a word requires `.align 2` (for four bytes), and a double requires `.align 3` (for eight bytes).
`.ascii "string"`	store an ASCII string *without* a '\0'-terminator at the next location(s) in the current data segment. nearly always not what you want; use `.asciiz` instead!
`.asciiz "string"`	store a '\0'-terminated ASCII string at the next location(s) in the current data segment
`.space n`	allocate *n* uninitialised bytes of space at the next location in the current segment
`.byte val [, ...]`	store values in successive byte(s) at the next location(s) in the current segment
`.half val [, ...]`	store values in successive half word(s) at the next location(s) in the current segment
`.word val [, ...]`	store values in successive word(s) at the next location(s) in the current segment
`.float val [, ...]`	store values in successive float(s) at the next location(s) in the current segment
`.double val [, ...]`	store values in successive double(s) at the next location(s) in the current segment
`.globl label [, ...]`	Declare the listed label(s) as global to enable referencing from other files

Further documentation

back to top

- MIPS32 Quick Reference
- MIPS32 Instruction Reference

COMP1521 24T2: Computer Systems Fundamentals is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs1521@cse.unsw.edu.au
CRICOS Provider 00098G