

MIPS Instruction Reference

This is a description of the MIPS instruction set, their meanings, syntax, semantics, and bit encodings. The syntax given for each instruction refers to the assembly language syntax supported by the MIPS assembler. Hyphens in the encoding indicate "don't care" bits which are not considered when an instruction is being decoded.

General purpose registers (GPRs) are indicated with a dollar sign (\$).

Note: ALL arithmetic immediate values are sign-extended. After that, they are handled as signed or unsigned 32 bit numbers, depending upon the instruction. The only difference between signed and unsigned instructions is that signed instructions can generate an overflow exception and unsigned instructions cannot.

Note: Instruction mnemonics are shown in upper case. In writing a program use only lower case.

ADD – Add (with overflow)

Description: Adds two registers and stores the result in a register

Operation: $\$d = \$s + \$t$; advance pc(4)

Syntax: add \$d, \$s, \$t

ADD.S – Add

Description: Adds two floating point registers and stores the result in a floating point register

Operation: $\$fd = \$fs + \$ft$; advance pc(4)

Syntax: add.s \$fd, \$fs, \$ft

ADDI -- Add immediate (with overflow)

Description: Adds a register and a sign-extended immediate value and stores the result in a register

Operation: $\$t = \$s + \text{imm}$; advance pc (4);

Syntax: addi \$t, \$s, imm

ADDIU -- Add immediate unsigned (no overflow)

Description: Adds a register and a sign-extended immediate value and stores the result in a register

Operation: $\$t = \$s + \text{imm}$; advance pc (4);

Syntax: addiu \$t, \$s, imm

ADDU -- Add unsigned (no overflow)

Description: Adds two registers and stores the result in a register

Operation: $\$d = \$s + \$t$; advance_pc (4);

Syntax: addu \$d, \$s, \$t
Encoding: 0000 00ss ssst tttt dddd d000 0010 0001

AND -- Bitwise and

Description: Bitwise ands two registers and stores the result in a register
Operation: \$d = \$s & \$t; advance_pc (4);
Syntax: and \$d, \$s, \$t

ANDI -- Bitwise and immediate

Description: Bitwise ands a register and an immediate value and stores the result in a register
Operation: \$t = \$s & imm; advance pc (4);
Syntax: andi \$t, \$s, imm

BEQ -- Branch on equal

Description: Branches if the two registers are equal
Operation: if \$s == \$t advance pc (offset << 2)); else advance pc (4);
Syntax: beq \$s, \$t, offset

BGEZ -- Branch on greater than or equal to zero

Description: Branches if the register is greater than or equal to zero
Operation: if \$s >= 0 advance pc (offset << 2)); else advance pc (4);
Syntax: bgez \$s, offset

BGTZ -- Branch on greater than zero

Description: Branches if the register is greater than zero
Operation: if \$s > 0 advance pc (offset << 2)); else advance pc (4);
Syntax: bgtz \$s, offset

BLEZ -- Branch on less than or equal to zero

Description: Branches if the register is less than or equal to zero
Operation: if \$s <= 0 advance pc (offset << 2)); else advance pc (4);
Syntax: blez \$s, offset

BLTZ -- Branch on less than zero

Description: Branches if the register is less than zero
Operation: if \$s < 0 advance pc (offset << 2)); else advance pc (4);
Syntax: bltz \$s, offset

BNE -- Branch on not equal

Description: Branches if the two registers are not equal
Operation: if \$s != \$t advance pc (offset << 2)); else advance pc (4);
Syntax: bne \$s, \$t, offset
Pseudo Instructions for branches

The MIPS environment includes a set of instructions that the assembler recognizes and translates into appropriate sequences of MIPS instructions. In the conditionals, they allow one to compare two registers for more than equal or not equal. Their format is

b??? \$s, \$d, offset

They are

| | |
|-----|-----------|
| blt | branch < |
| bgt | branch > |
| bge | branch >= |
| ble | branch <= |

C.REL.S – compare floating point REL

Description: Set the condition flag to true (1) if REL is true
Operation: if \$fs .REL. \$ft is true, set condition flag to true
Syntax: c.eq.s \$fs, \$ft

REL values are:

| | |
|------|--------------------|
| .eq. | equal |
| .lt. | less than |
| .le. | less than or equal |

CVT.S.W – convert from word (integer) to float

Description: Convert the word in integer register \$fs to a single precision and store in \$fd
Operation: \$fd = convert to IEEE 754 single precision(\$fs)
Syntax: cvt.s.w \$fd,\$fs

CVT.W.S – convert from float to integer

Description: Convert the float in integer register \$fs to an and store in \$fd
Operation: \$fd = truncate(\$fs)
Syntax: cvt.w.s \$fd,\$fs

DIV

Description: Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI
Operation: \$LO = \$s / \$t; \$HI = \$s % \$t; advance_pc (4);

Syntax: div \$s, \$t

DIVU -- Divide unsigned

Description: Divides \$s by \$t and stores the quotient in \$LO and the remainder in \$HI

Operation: \$LO = \$s / \$t; \$HI = \$s % \$t; advance_pc (4);

Syntax: divu \$s, \$t

J -- Jump

Description: Jumps to the calculated address

Operation: PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);

Syntax: j target

JAL -- Jump and link

Description: Jumps to the calculated address and stores the return address in \$31

Operation: \$31 = PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);

Syntax: jal target

JR -- Jump register

Description: Jump to the address contained in register \$s

Operation: PC = nPC; nPC = \$s;

Syntax: jr \$s

LA -- Load address

Description: The address specified by the operand is loaded

Operation: \$t = ADDRESS of X; advance pc (4);

Syntax: la \$t, address

Example la \$5, somewhere # the address somewhere is loaded into register \$5

LB -- Load byte

Description: A byte is loaded into a register from the specified address. Bit 7 of the byte is treated as the sign and is thus repeated in the upper 24 bits (8-31) – called sign extend

Operation: \$t = MEM[\$s + offset]; advance pc (4);

Syntax: lb \$t, offset(\$s)

LI -- Load immediate

Description: The immediate value of the operand is loaded into the register

Operation: \$t = imm; advance pc (4);

Syntax: li \$t, imm

Example: li \$a0, -23 # register \$a0 = -23

LBU -- Load byte unsigned

Description: A byte is loaded into a register from the specified address. Bits 8-31 are set to 0
Operation: $\$t = \text{MEM}[\$s + \text{offset}]$; advance pc (4);
Syntax: `lbu $t, offset($s)`

LH -- Load half word

Description: A half word is loaded into a register from the specified address sign extend
Operation: $\$t = \text{MEM}[\$s + \text{offset}]$; advance pc (4);
Syntax: `lh $t, offset($s)`

LHU -- Load half word unsigned

Description: A half word is loaded into a register from the specified address. Bits 16-31 are set to 0
Operation: $\$t = \text{MEM}[\$s + \text{offset}]$; advance pc (4);
Syntax: `lhu $t, offset($s)`

LUI -- Load upper immediate

Description: The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation: $\$t = (\text{imm} \ll 16)$; advance pc (4);
Syntax: `lui $t, imm`

LW -- Load word

Description: A word is loaded into a register from the specified address.
Operation: $\$t = \text{MEM}[\$s + \text{offset}]$; advance pc (4);
Syntax: `lw $t, offset($s)`

MFC1 -- Move \$fs (floating point register) to \$d (integer register)

Description: The contents of register \$fs are moved to the specified register \$d.
Operation: $\$d = \fs ; advance_pc (4);
Syntax: `mfc1 $d, $fs`
Example: `mfc1 $a0, $f0 $a0=$f0`

MFHI -- Move from HI

Description: The contents of register HI are moved to the specified register.
Operation: $\$d = \HI ; advance_pc (4);

Syntax: mfhi \$d

MFLO -- Move from LO

Description: The contents of register LO are moved to the specified register.

Operation: \$d = \$LO; advance_pc (4);

Syntax: mflo \$d

MOVE -- Move \$s to \$d equivalent to add \$s, \$d, \$zero

Description: The contents of register \$s are moved to the specified register.

Operation: \$d = \$s; advance_pc (4);

Syntax: move \$s, \$d

MOV.S -- Move \$fs to \$fd

Description: The contents of register \$fs are moved to the specified register \$fd.

Operation: \$Fd=\$fs; advance_pc (4);

Syntax: mov.s \$fd, \$fs NOTE: this is opposite of move for fs and fd

MTC1 -- Move \$fs (floating point register) to \$d (integer register)

Description: The contents of register \$fs are moved to the specified register \$d.

Operation: \$fd=\$s; advance_pc (4);

Syntax: mtc1 \$s, \$fd

Example: mtc1 \$a0,\$f0 \$f0=\$a0

MULT -- Multiply

Description: Multiplies \$s by \$t and stores the result in \$LO.

Operation: \$LO = \$s * \$t; advance_pc (4);

Syntax: mult \$s, \$t

MULTU -- Multiply unsigned

Description: Multiplies \$s by \$t and stores the result in \$LO.

Operation: \$LO = \$s * \$t; advance_pc (4);

Syntax: multu \$s, \$t

NOOP -- no operation

Description: Performs no operation.

Operation: advance_pc (4);

Syntax: noop

Note: The encoding for a NOOP represents the instruction SLL \$0, \$0, 0 which has no side

effects. In fact, nearly every instruction that has \$0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR -- Bitwise or

Description: Bitwise logical ors two registers and stores the result in a register
Operation: $\$d = \$s \mid \$t$; advance_pc (4);
Syntax: or \$d, \$s, \$t

ORI -- Bitwise or immediate

Description: Bitwise ors a register and an immediate value and stores the result in a register
Operation: $\$t = \$s \mid \text{imm}$; advance_pc (4);
Syntax: ori \$t, \$s, imm

SB -- Store byte

Description: The least significant byte of \$t is stored at the specified address.
Operation: $\text{MEM}[\$s + \text{offset}] = (0\text{xff} \ \& \ \$t)$; advance_pc (4);
Syntax: sb \$t, offset(\$s)

SLL -- Shift left logical

Description: Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation: $\$d = \$t \ll h$; advance_pc (4);
Syntax: sll \$d, \$t, h

SLLV -- Shift left logical variable

Description: Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation: $\$d = \$t \ll \$s$; advance_pc (4);
Syntax: sllv \$d, \$t, \$s

SLT -- Set on less than (signed)

Description: If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation: if $\$s < \t $\$d = 1$; advance_pc (4); else $\$d = 0$; advance_pc (4);
Syntax: slt \$d, \$s, \$t

SLTI -- Set on less than immediate (signed)

Description: If \$s is less than immediate, \$t is set to one. It gets zero otherwise.
Operation: if $\$s < \text{imm}$ $\$t = 1$; advance_pc (4); else $\$t = 0$; advance_pc (4);
Syntax: slti \$t, \$s, imm

SLTIU -- Set on less than immediate unsigned

Description: If \$s is less than the unsigned immediate, \$t is set to one. It gets zero otherwise.
Operation: if $\$s < \text{imm}$ $\$t = 1$; advance pc (4); else $\$t = 0$; advance pc (4);
Syntax: sltiu \$t, \$s, imm

SLTU -- Set on less than unsigned

Description: If \$s is less than \$t, \$d is set to one. It gets zero otherwise.
Operation: if $\$s < \t $\$d = 1$; advance_pc (4); else $\$d = 0$; advance_pc (4);
Syntax: sltu \$d, \$s, \$t

SRA -- Shift right arithmetic

Description: Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation: $\$d = \$t \gg h$; advance_pc (4);
Syntax: sra \$d, \$t, h

SRL -- Shift right logical

Description: Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation: $\$d = \$t \gg h$; advance_pc (4);
Syntax: srl \$d, \$t, h

SRLV -- Shift right logical variable

Description: Shifts a register value right by the amount specified in \$s and places the value in the destination register. Zeroes are shifted in.
Operation: $\$d = \$t \gg \$s$; advance pc (4);
Syntax: srlv \$d, \$t, \$s

SUB -- Subtract

Description: Subtracts two registers and stores the result in a register
Operation: $\$d = \$s - \$t$; advance_pc (4);
Syntax: sub \$d, \$s, \$t

SUBU -- Subtract unsigned

Description: Subtracts two registers and stores the result in a register
Operation: $\$d = \$s - \$t$; advance_pc (4);
Syntax: subu \$d, \$s, \$t

SW -- Store word

Description: The contents of \$t is stored at the specified address
Operation: MEM[\$s + offset] = \$t; advance pc (4);
Syntax: sw \$t,offset(\$s)

SYSCALL -- System call

Description: Generates a software interrupt.
Operation: advance pc (4);
Syntax: syscall

The syscall instruction is described in more detail on the System Calls page.

XOR -- Bitwise exclusive or

Description: Exclusive ors two registers and stores the result in a register
Operation: \$d = \$s XOR \$t; advance pc (4);
Syntax: xor \$d, \$s, \$t

XORI -- Bitwise exclusive or immediate

Description: Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation: \$t = \$s XOR imm; advance pc (4);
Syntax: xori \$t, \$s, imm

SYSCALL functions available in MARS

Introduction

A number of system services, mainly for input and output, are available for use by your MIPS program.

How to use SYSCALL system services

- Step 1. Load the service number in register \$v0.
- Step 2. Load argument values, if any, in \$a0, \$a1, \$a2, or \$f12 as specified.
- Step 3. Issue the SYSCALL instruction.
- Step 4. Retrieve return values, if any, from result registers as specified.

Example: display the value stored in \$t0 on the console

```
li $v0, 1          # service 1 is print integer
add $a0, $t0, $zero # load desired value into argument register $a0, using pseudo-op
syscall
```

Table of Available Services for SYSCALL

ServiceCode is placed in \$v0

| | \$v0 | Arguments and Result |
|-----------------|------|---|
| print integer | 1 | \$a0 = integer to print |
| print float | 2 | \$f12 = float to print |
| print double | 3 | \$f12 = double to print |
| print string | 4 | \$a0 = address of null- terminated string to print |
| read integer | 5 | \$v0 contains integer read |
| read float | 6 | \$f0 contains float read |
| read double | 7 | \$f0 contains double read |
| read string | 8 | \$a0 = address of input buffer \$a1 = maximum number of characters to read |
| exit | 10 | terminate execution |
| print character | 11 | \$a0 = character to print |
| read character | 12 | \$v0 = character read |

Memory Allocation (MIPS Directives)

```
.word 10, 0x25, -3    # allocates a word of value 10, hex 25, and -3 in consecutive memory
                      # locations

.byte 5, 0xff, 17     # allocates bytes rather than words

.float 10.5, 1.2, -0.5 # allocates floating point values (IEEE)

.ascii "a test"       # allocates an ASCII string

.asciiz "a test"      # allocates a null terminated ASCII string

.asciiz "\t\n"        # allocates a null terminated ASCII string of a tab followed by a newline
                      # character

.space 100            # allocates 100 bytes (25 words) of memory, values unknown
```

| Register name | Number | Usage |
|---------------|--------|---|
| \$zero | 0 | constant 0 |
| \$at | 1 | reserved for assembler |
| \$v0 | 2 | expression evaluation and results of a function |
| \$v1 | 3 | expression evaluation and results of a function |
| \$a0 | 4 | argument 1 |
| \$a1 | 5 | argument 2 |
| \$a2 | 6 | argument 3 |
| \$a3 | 7 | argument 4 |
| \$t0 | 8 | temporary (not preserved across call) |
| \$t1 | 9 | temporary (not preserved across call) |
| \$t2 | 10 | temporary (not preserved across call) |
| \$t3 | 11 | temporary (not preserved across call) |
| \$t4 | 12 | temporary (not preserved across call) |
| \$t5 | 13 | temporary (not preserved across call) |
| \$t6 | 14 | temporary (not preserved across call) |
| \$t7 | 15 | temporary (not preserved across call) |
| \$s0 | 16 | saved temporary (preserved across call) |
| \$s1 | 17 | saved temporary (preserved across call) |
| \$s2 | 18 | saved temporary (preserved across call) |
| \$s3 | 19 | saved temporary (preserved across call) |
| \$s4 | 20 | saved temporary (preserved across call) |
| \$s5 | 21 | saved temporary (preserved across call) |
| \$s6 | 22 | saved temporary (preserved across call) |
| \$s7 | 23 | saved temporary (preserved across call) |
| \$t8 | 24 | temporary (not preserved across call) |
| \$t9 | 25 | temporary (not preserved across call) |
| \$k0 | 26 | reserved for OS kernel |
| \$k1 | 27 | reserved for OS kernel |
| \$gp | 28 | pointer to global area |
| \$sp | 29 | stack pointer |
| \$fp | 30 | frame pointer |
| \$ra | 31 | return address (used by function call) |

FIGURE A.6.1 MIPS registers and usage convention.