MIPS assembly language

These are assembly language instructions that have direct hardware implementation, as opposed to *pseudoinstructions* which are translated into multiple real instructions before being assembled.

- In the following, the register letters *d*, *t*, and *s* are placeholders for (register) numbers or register names.
- *C* denotes a constant (*immediate*).
- All the following instructions are native instructions.
- Opcodes and funct codes are in hexadecimal.
- The MIPS32 Instruction Set states that the word *unsigned* as part of Add and Subtract instructions, is a *misnomer*. The difference between *signed* and *unsigned* versions of commands is not a sign extension (or lack thereof) of the operands, but controls whether a trap is executed on overflow *(e.g. Add)* or an overflow is ignored *(Add unsigned)*. An immediate operand CONST to these instructions is always sign-extended.

Integer

MIPS has 32 integer registers. Data must be in registers to perform arithmetic. Register $0 always holds 0 and register $1 is normally reserved for the assembler (for handling pseudo instructions and large constants).

The encoding shows which bits correspond to which parts of the instruction. A hyphen (-) is used to indicate don't cares.

Category	Name	Instruction syntax	Meaning	Format/opcode/funct			Notes/Encoding
Arithmetic	Add	`add $d,$s,$t`	`$d = $s + $t`	R	0	20_{16}	adds two registers, executes a trap on overflow `000000ss ssstttt ddddd--- --100000`
	Add unsigned	`addu $d,$s,$t`	`$d = $s + $t`	R	0	21_{16}	as above but ignores an overflow `000000ss ssstttt ddddd--- --100001`
	Subtract	`sub $d,$s,$t`	`$d = $s - $t`	R	0	22_{16}	subtracts two registers, executes a trap on overflow `000000ss ssstttt ddddd--- --100010`
	Subtract unsigned	`subu $d,$s,$t`	`$d = $s - $t`	R	0	23_{16}	as above but ignores an overflow `000000ss ssstttt ddddd000 00100011`
	Add immediate	`addi $t,$s,C`	`$t = $s + C (signed)`	I	8_{16}	-	Used to add sign-extended constants (and also to copy one register to another: addi $1, $2, 0), executes a trap on overflow `001000ss ssstttt CCCCCCCC CCCCCCCC`
	Add immediate unsigned	`addiu $t,$s,C`	`$t = $s + C (signed)`	I	9_{16}	-	as above but ignores an overflow `001001ss ssstttt CCCCCCCC CCCCCCCC`
	Multiply	`mult $s,$t`	`LO = (($s * $t) << 32) >> 32;` `HI = ($s * $t) >> 32;`	R	0	18_{16}	Multiplies two registers and puts the 64-bit result in two special memory spots - LO and HI. Alternatively, one could say the result of this operation is: `(int HI,int LO) = (64-bit) $s * $t` . See mfhi and mflo for accessing LO and HI regs.
	Multiply unsigned	`multu $s,$t`	`LO = (($s * $t) << 32) >> 32;` `HI = ($s * $t) >> 32;`	R	0	19_{16}	Multiplies two registers and puts the 64-bit result in two special memory spots - LO and HI. Alternatively, one could say the result of this operation is: `(int HI,int LO) = (64-bit) $s * $t` . See mfhi and mflo for accessing LO and HI regs.
	Divide	`div $s, $t`	`LO = $s / $t  HI = $s % $t`	R	0	$1A_{16}$	Divides two registers and puts the 32-bit integer result in LO and the remainder in HI.[36]
	Divide unsigned	`divu $s, $t`	`LO = $s / $t  HI = $s % $t`	R	0	$1B_{16}$	Divides two registers and puts the 32-bit integer result in LO and the remainder in HI.
	Load word	`lw $t,C($s)`	`$t = Memory[$s + C]`	I	23_{16}	-	loads the word stored from: MEM[$s+C] and the following 3 bytes.
	Load halfword	`lh $t,C($s)`	`$t = Memory[$s + C] (signed)`	I	21_{16}	-	loads the halfword stored from: MEM[$s+C] and the following byte. Sign is extended to width of register.
	Load halfword unsigned	`lhu $t,C($s)`	`$t = Memory[$s + C] (unsigned)`	I	25_{16}	-	As above without sign extension.
							loads the byte stored from:

Data Transfer	Load byte	`lb $t,C($s)`	`$t = Memory[$s + C] (signed)`	I	20_{16}	-	MEM[$s+C].
	Load byte unsigned	`lbu $t,C($s)`	`$t = Memory[$s + C] (unsigned)`	I	24_{16}	-	As above without sign extension.
	Store word	`sw $t,C($s)`	`Memory[$s + C] = $t`	I	$2B_{16}$	-	stores a word into: MEM[$s+C] and the following 3 bytes. The order of the operands is a large source of confusion.
	Store half	`sh $t,C($s)`	`Memory[$s + C] = $t`	I	29_{16}	-	stores the least-significant 16-bit of a register (a halfword) into: MEM[$s+C].
	Store byte	`sb $t,C($s)`	`Memory[$s + C] = $t`	I	28_{16}	-	stores the least-significant 8-bit of a register (a byte) into: MEM[$s+C].
	Load upper immediate	`lui $t,C`	`$t = C << 16`	I	F_{16}	-	loads a 16-bit immediate operand into the upper 16-bits of the register specified. Maximum value of constant is 2^{16}-1
	Move from high	`mfhi $d`	`$d = HI`	R	0	10_{16}	Moves a value from HI to a register. Do not use a multiply or a divide instruction within two instructions of mfhi (that action is undefined because of the MIPS pipeline).
	Move from low	`mflo $d`	`$d = LO`	R	0	12_{16}	Moves a value from LO to a register. Do not use a multiply or a divide instruction within two instructions of mflo (that action is undefined because of the MIPS pipeline).
	Move from Control Register	`mfcZ $t, $d`	`$t = Coprocessor[Z].ControlRegister[$d]`	R	0		Moves a 4 byte value from Coprocessor Z Control register to a general purpose register. Sign extension.
	Move to Control Register	`mtcZ $t, $d`	`Coprocessor[Z].ControlRegister[$d] = $t`	R	0		Moves a 4 byte value from a general purpose register to a Coprocessor Z Control register. Sign extension.
Logical	And	`and $d,$s,$t`	`$d = $s & $t`	R	0	24_{16}	Bitwise and `000000ss ssstttttt ddddd--- --100100`
	And immediate	`andi $t,$s,C`	`$t = $s & C`	I	C_{16}	-	Leftmost 16 bits are padded with 0s `001100ss ssstttttt CCCCCCCC CCCCCCCC`
	Or	`or $d,$s,$t`	`$d = $s | $t`	R	0	25_{16}	Bitwise or
	Or immediate	`ori $t,$s,C`	`$t = $s | C`	I	D_{16}	-	Leftmost 16 bits are padded with 0s
	Exclusive or	`xor $d,$s,$t`	`$d = $s ^ $t`	R	0	26_{16}	Bitwise exclusive or
	Exclusive or immediate	`xori $t,$s,C`	`$t = $s ^ C`	I	E_{16}	-	Leftmost 16 bits are padded with 0s
	Nor	`nor $d,$s,$t`	`$d = ~ ($s | $t)`	R	0	27_{16}	Bitwise nor
	Set on less than	`slt $d,$s,$t`	`$d = ($s < $t)`	R	0	$2A_{16}$	Tests if one register is less than another.
	Set on less than unsigned	`sltu $d,$s,$t`	`$d = ($s < $t)`	R	0	$2B_{16}$	Tests if unsigned integer in one register is less than another.
	Set on less than immediate	`slti $t,$s,C`	`$t = ($s < C)`	I	A_{16}	-	Tests if one register is less than a constant.
	Shift left logical immediate	`sll $d,$t,shamt`	`$d = $t << shamt`	R	0	0	shifts `shamt` number of bits to the left (multiplies by 2^{shamt})
	Shift right	`srl`	`$d = $t >> shamt`				shifts `shamt` number of bits to the right - zeros are shifted in (divides by 2^{shamt}). Note that this

Bitwise shift	logical immediate	`$d,$t,shamt`		R	0	2_{16}	instruction only works as division of a two's complement number if the value is positive.
	Shift right arithmetic immediate	`sra $d,$t,shamt`	$\$d=\$t>>shamt+\left(\sum_{n=1}^{shamt} 2^{32-n}\right)\cdot(\$t>>31)$	R	0	3_{16}	shifts `shamt` number of bits - the sign bit is shifted in (divides a positive or even 2's complement number by 2^{shamt})
	Shift left logical	`sllv $d,$t,$s`	`$d = $t << $s`	R	0	4_{16}	shifts `$s` number of bits to the left (multiplies by $2^{\$s}$)
	Shift right logical	`srlv $d,$t,$s`	`$d = $t >> $s`	R	0	6_{16}	shifts `$s` number of bits to the right - zeros are shifted in (divides by $2^{\$s}$). Note that this instruction only works as division of a two's complement number if the value is positive.
	Shift right arithmetic	`srav $d,$t,$s`	$\$d=\$t>>\$s+\left(\sum_{n=1}^{\$s} 2^{32-n}\right)\cdot(\$t>>31)$	R	0	7_{16}	shifts `$s` number of bits - the sign bit is shifted in (divides a positive or even 2's complement number by $2^{\$s}$)
Conditional branch	Branch on equal	`beq $s,$t,C`	`if ($s == $t) go to PC+4+4*C`	I	4_{16}	-	Goes to the instruction at the specified address if two registers are equal. `000100ss ssstttt CCCCCCCC CCCCCCCC`
	Branch on not equal	`bne $s,$t,C`	`if ($s != $t) go to PC+4+4*C`	I	5_{16}	-	Goes to the instruction at the specified address if two registers are *not* equal.
Unconditional jump	Jump	`j C`	`PC = PC+4[31:28] . C*4`	J	2_{16}	-	Unconditionally jumps to the instruction at the specified address.
	Jump register	`jr $s`	goto address `$s`	R	0	8_{16}	Jumps to the address contained in the specified register
	Jump and link	`jal C`	`$31 = PC + 4; PC = PC+4[31:28] . C*4`	J	3_{16}	-	For procedure call - used to call a subroutine, $31 holds the return address; returning from a subroutine is done by: jr $31. Return address is PC + 8, not PC + 4 due to the use of a branch delay slot which forces the instruction after the jump to be executed

Note: In MIPS assembly code, the offset for branching instructions can be represented by a label elsewhere in the code.

Note: There is no corresponding *load lower immediate* instruction; this can be done ori (or immediate) with the register $0 (whose value is always zero). For example, both `addi $1, $0, 100` and `ori $1, $0, 100` load the decimal value 100 into register $1. However, if you are trying to create a 32-bit value with lui (load upper immediate) followed by a "load lower immediate", it is wise to use `ori $1, $0, 100`. The instruction addi will sign extend the most significant bit and potentially overwrite the upper 16 bits when adding negative values.

Note: Subtracting an immediate can be done with adding the negation of that value as the immediate.

Floating point

MIPS has 32 floating-point registers. Two registers are paired for double precision numbers. Odd numbered registers cannot be used for arithmetic or branching, just as part of a double precision register pair.

Category	Name	Instruction syntax	Meaning	Format	opcode	funct	Notes/Encoding
Arithmetic	FP add single	`add.s $x,$y,$z`	`$x = $y + $z`				Floating-Point add (single precision)
	FP subtract single	`sub.s $x,$y,$z`	`$x = $y - $z`				Floating-Point subtract (single precision)
	FP multiply single	`mul.s $x,$y,$z`	`$x = $y * $z`				Floating-Point multiply (single precision)
	FP divide single	`div.s $x,$y,$z`	`$x = $y / $z`				Floating-Point divide (single precision)
	FP add double	`add.d $x,$y,$z`	`$x = $y + $z`				Floating-Point add (double precision)
	FP subtract double	`sub.d $x,$y,$z`	`$x = $y - $z`				Floating-Point subtract (double precision)
	FP multiply double	`mul.d $x,$y,$z`	`$x = $y * $z`				Floating-Point multiply (double precision)
	FP divide double	`div.d $x,$y,$z`	`$x = $y / $z`				Floating-Point divide (double precision)
Data Transfer	Load word coprocessor	`lwcZ $x,CONST ($y)`	`Coprocessor[Z].DataRegister[$x] = Memory[$y + CONST]`	I			Loads the 4 byte word stored from: MEM[$y+CONST] into a Coprocessor data register. Sign extension.
	Store word coprocessor	`swcZ $x,CONST ($y)`	`Memory[$y + CONST] = Coprocessor[Z].DataRegister[$x]`	I			Stores the 4 byte word held by a Coprocessor data register into: MEM[$y+CONST]. Sign extension.
Logical	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	`cond = ($f2 < $f4)`				Floating-point compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	`cond = ($f2 < $f4)`				Floating-point compare less than double precision
Branch	branch on FP true	`bc1t 100`	`if (cond) goto PC+4+100;`				PC relative branch if FP condition
	branch on FP false	`bc1f 100`	`if (cond) goto PC+4+100;`				PC relative branch if not condition

Pseudo instructions

These instructions are accepted by the MIPS assembler, although they are not real instructions within the MIPS instruction set. Instead, the assembler translates them into sequences of real instructions.

Name	instruction syntax	Real instruction translation	meaning
Move	`move $rt,$rs`	`add $rt,$rs,$zero`	`R[rt]=R[rs]`
Clear	`clear $rt`	`add $rt,$zero,$zero`	`R[rt]=0`
Not	`not $rt, $rs`	`nor $rt, $rs, $zero`	`R[rt]=~R[rs]`
Load Address	`la $rd, LabelAddr`	`lui $rd, LabelAddr[31:16]` `ori $rd,$rd, LabelAddr[15:0]`	`$rd` = Label Address
Load Immediate	`li $rd, IMMED[31:0]`	`lui $rd, IMMED[31:16]` `ori $rd,$rd, IMMED[15:0]`	`$rd` = 32 bit Immediate value
Branch unconditionally	`b Label`	`beq $zero,$zero,Label`	`PC=Label`
Branch and link	`bal Label`	`bgezal $zero,Label`	`R[31]=PC+8;` `PC=Label;`
Branch if greater than	`bgt $rs,$rt,Label`	`slt $at,$rt,$rs` `bne $at,$zero,Label`	`if (R[rs]>R[rt])` `  PC=Label`
Branch if less than	`blt $rs,$rt,Label`	`slt $at,$rs,$rt` `bne $at,$zero,Label`	`if (R[rs]<R[rt])` `  PC=Label`
Branch if greater than or equal	`bge $rs,$rt,Label`	`slt $at,$rs,$rt` `beq $at,$zero,Label`	`if (R[rs]>=R[rt])` `  PC=Label`
Branch if less than or equal	`ble $rs,$rt,Label`	`slt $at,$rt,$rs` `beq $at,$zero,Label`	`if (R[rs]<=R[rt])` `  PC=Label`
Branch if less than or equal to zero	`blez $rs,Label`	`slt $at,$zero,$rs` `beq $at,$zero,Label`	`if (R[rs]<=0)` `  PC=Label`
Branch if greater than unsigned	`bgtu $rs,$rt,Label`	`sltu $at,$rt,$rs` `bne $at,$zero,Label`	`if (R[rs]>R[rt])` `  PC=Label`
Branch if greater than zero	`bgtz $rs,Label`	`slt $at,$zero,$rs` `bne $at,$zero,Label`	`if (R[rs]>0)` `  PC=Label`
Branch if equal to zero	`beqz $rs,Label`	`beq $rs,$zero,Label`	`if (R[rs]==0)` `  PC=Label`
Branch if not equal to zero	`bnez $rs,Label`	`bne $rs,$zero,Label`	`if (R[rs]!=0)` `  PC=Label`
Multiplies and returns only first 32 bits	`mul $d, $s, $t`	`mult $s, $t` `mflo $d`	`$d = $s * $t`
Divides and returns quotient	`div $d, $s, $t`	`div $s, $t` `mflo $d`	`$d = $s / $t`
Divides and returns remainder	`rem $d, $s, $t`	`div $s, $t` `mfhi $d`	`$d = $s % $t`

Other instructions

- NOP (no operation) (machine code 0x00000000, interpreted by CPU as `sll $0,$0,0`)
- break (breaks the program, used by debuggers)
- syscall (used for system calls to the operating system)

Many other pseudoinstructions and floating-point instructions present in MIPS R2000 are given in Appendix B.10 of *Computer Organization and Design, Fourth Edition* by Patterson and Hennessy.

Example code

The following sample code implements the Euler's totient function in MIPS assembly language:

```
.text
.globl main
     main:
         la $a0, query                  #First the query
         li $v0, 4
         syscall
         li $v0, 5                   #Read the input
         syscall
         move $t0, $v0                   #store the value in a temporary variable
                                                 #store the base values in $t1, $t2
                                # $t1 iterates from m-1 to 1
                                # $t2 maintains a counter of the number of coprimes less than m

        sub $t1, $t0, 1
        li $t2, 0

    tot:
        blez $t1, done                  #termination condition
        move $a0, $t0                   #Argument passing
        move $a1, $t1                   #Argument passing
        jal gcd                         #to GCD function
        sub $t3, $v0, 1
        beqz $t3, inc                   #checking if gcd is one
        addi $t1, $t1, -1               #decrementing the iterator
        b tot

     inc:
         addi $t2, $t2, 1               #incrementing the counter
         addi $t1, $t1, -1              #decrementing the iterator
         b tot

     gcd:                           #recursive definition
            addi $sp, $sp, -12
            sw $a1, 8($sp)
        sw $a0, 4($sp)
                sw $ra, 0($sp)
        move $v0, $a0
        beqz $a1, gcd_return                    #termination condition
        move $t4, $a0                   #computing GCD
        move $a0, $a1
        remu $a1, $t4, $a1
        jal gcd
        lw $a1, 8($sp)
        lw $a0, 4($sp)

    gcd_return:
        lw $ra, 0($sp)
        addi $sp, $sp, 12
        jr $ra

    done:                           #print the result
                                #first the message
        la $a0, result_msg
        li $v0, 4
        syscall
                                #then the value
        move $a0, $t2
        li $v0, 1
        syscall
                                #exit
        li $v0, 10
        syscall

 .data
    query: .asciiz "Input m =  "
    result_msg: .asciiz "Totient(m) =  "
```

Compiler register usage

The hardware architecture specifies that:

- General purpose register $0 always returns a value of 0.
- General purpose register $31 is used as the link register for jump and link instructions.
- HI and LO are used to access the multiplier/divider results, accessed by the mfhi (move from high) and mflo commands.

These are the only hardware restrictions on the usage of the general purpose registers.

The various MIPS tool-chains implement specific calling conventions that further restrict how the registers are used. These calling conventions are totally maintained by the tool-chain software and are not required by the hardware.

Registers for O32 Calling Convention

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0–$v1	$2–$3	values for function returns and expression evaluation	No
$a0–$a3	$4–$7	function arguments	No
$t0–$t7	$8–$15	temporaries	No
$s0–$s7	$16–$23	saved temporaries	Yes
$t8–$t9	$24–$25	temporaries	No
$k0–$k1	$26–$27	reserved for OS kernel	N/A
$gp	$28	global pointer	Yes (except PIC code)
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

Registers for N32 and N64 Calling Conventions[38]

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0–$v1	$2–$3	values for function returns and expression evaluation	No
$a0–$a7	$4–$11	function arguments	No
$t4–$t7	$12–$15	temporaries	No
$s0–$s7	$16–$23	saved temporaries	Yes
$t8–$t9	$24–$25	temporaries	No
$k0–$k1	$26–$27	reserved for OS kernel	N/A
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$s8	$30	frame pointer	Yes
$ra	$31	return address	N/A

Registers that are preserved across a call are registers that (by convention) will not be changed by a system call or procedure (function) call. For example, $s-registers must be saved to the stack by a procedure that needs to use them, and $sp and $fp are always incremented by constants, and decremented back after the procedure is done with them (and the memory they point to). By contrast, $ra is changed automatically by any normal function call (ones that use jal), and $t-registers must be saved by the program before any procedure call (if the program needs the values inside them after the call).