← MIPS instruction cheatsheet

it's not actually cheating

Here are tables of common MIPS instructions and what they do. If you want some in-context examples of when you'd use them, see the **cookbook**.

Arithmetic and Bitwise Instructions

All arithmetic and bitwise instructions can be written in two ways:

1. `add t0, t1, t2`
   - adds two registers and puts the result in a third register.
   - this does `t0 = t1 + t2`
2. `add t0, t1, 4`
   - adds a register and a constant and puts the result in a second register.
   - this does `t0 = t1 + 4`

Sometimes for this second form, you will see it written like `addi` or `subi`. These are completely equivalent, just a different name for the same instruction.

The `i` means "immediate," since numbers inside instructions are called immediates.

Mnemonic	Operation	Description
`neg a, b`	`a = -b`	gives the negative of b.
`add a, b, c`	`a = b + c`	adds signed numbers.
`sub a, b, c`	`a = b - c`	subtracts signed numbers.
`mul a, b, c`	`a = b * c`	gives low 32 bits of signed multiplication.
`div a, b, c`	`a = b / c`	gives quotient of signed division.
`rem a, b, c`	`a = b % c`	gives remainder of signed division.

Mnemonic	Operation	Description
`addu a, b, c`	`a = b + c`	adds unsigned numbers.
`subu a, b, c`	`a = b - c`	subtracts unsigned numbers.
`mulu a, b, c`	`a = b * c`	gives low 32 bits of unsigned multiplication.
`divu a, b, c`	`a = b / c`	gives quotient of unsigned division.
`remu a, b, c`	`a = b % c`	gives remainder of unsigned division.
`mfhi a`	`a = HI`	after `mul`, gives high 32 bits. after `div`, gives remainder.
`mflo a`	`a = LO`	after `mul`, gives low 32 bits. after `div`, gives quotient.
`not a, b`	`a = ~b`	gives the bitwise complement of b (all bits flipped).
`and a, b, c`	`a = b & c`	bitwise ANDs numbers.
`or a, b, c`	`a = b | c`	bitwise ORs numbers.
`xor a, b, c`	`a = b ^ c`	bitwise XORs numbers.

Shift Instructions

MIPS decided to implement shifts a little differently than the rest of the arithmetic and bitwise instructions.

Mnemonic	Operation	Description
`sll a, b, imm`	`a = b << imm`	shift left by a constant amount.
`srl a, b, imm`	`a = b >>> imm`	shift right unsigned (logical) by a constant amount.
`sra a, b, imm`	`a = b >> imm`	shift right arithmetic by a constant amount.
`sllv a, b, reg`	`a = b << reg`	shift left by the amount in a register.

Mnemonic	Operation	Description
`srlv a, b, reg`	`a = b >>> reg`	shift right unsigned (logical) by the amount in a register.
`srav a, b, reg`	`a = b >> reg`	shift right arithmetic by the amount in a register.

Data Transfer Instructions

There are two “load” instructions **which do not access memory.** Also, `move` does not move, it copies. THAT’S LIFE.

Mnemonic	Operation	Description
`li a, imm`	`a = imm`	put a constant value into a register.
`la a, label`	`a = &label`	put the address that a label points to into a register.
`move a, b`	` a = b `	copy value from one register to another.

The rest of the load/store instructions **always access memory.** All of these instructions can be written in three different ways:

1. `lw t0, var`
    - copies a word (32-bit value) from the memory variable `var` into register `t0`
    - `var` must have been declared as something like:

    ```
    .data
    var: .word 0
    ```

2. `lw t0, (t1)`
    - copies a word from the memory address given by `t1` into register `t0`
3. `lw t0, 4(t1)`
    - copies a word from the memory address given by `t1 + 4` into register `t0`

> REMEMBER: stores copy values FROM registers TO memory. So FROM the left side TO the address on the right side.

Mnemonic	Operation	Description
`lw reg, addr`	`reg = MEM[addr]`	loads the 4 bytes at `addr` as a 32-bit value into `reg`.
`lh reg, addr`	`reg = sxt(MEM[addr])`	loads the 2 bytes at `addr` as a signed 16-bit value into `reg`.
`lb reg, addr`	`reg = sxt(MEM[addr])`	loads the 1 byte at `addr` as a signed 8-bit value into `reg`.
`lhu reg, addr`	`reg = zxt(MEM[addr])`	loads the 2 bytes at `addr` as an unsigned 16-bit value into `reg`.
`lbu reg, addr`	`reg = zxt(MEM[addr])`	loads the 1 byte at `addr` as an unsigned 8-bit value into `reg`.
`sw reg, addr`	`MEM[addr] = reg`	stores the value of `reg` into memory as 4 bytes starting at `addr`.
`sh reg, addr`	`MEM[addr] = lo16(reg)`	stores the low 16 bits of `reg` into memory as 2 bytes starting at `addr`.
`sb reg, addr`	`MEM[addr] = lo8(reg)`	stores the low 8 bits of `reg` into memory as 1 byte at `addr`.

Last, there are two stack (pseudo-)instructions which are used to save and restore values in functions:

Mnemonic	Operation	Description
`push reg`	`sp -= 4; MEM[sp] = reg`	pushes the value of `reg` onto the call stack
`pop reg`	`reg = MEM[sp]; sp += 4`	pops the top call stack value and puts it into `reg`

Unconditional Control Flow Instructions

These always change the PC to a new location.

Mnemonic	Operation	Description
`j label`	`PC = label`	goes to the instruction at `label`.
`jal label`	`ra = PC + 4; PC = label`	function call to `label`. stores return address in `ra`.
`jr reg`	`PC = reg`	goes to the instruction whose address is in `reg`, often `ra`.
`syscall`	`--->`	runs the system call function whose number is in `v0`.

Conditional Control Flow Instructions

All these instructions check the given condition, and if it's:

- **true,** goes to the given label
- **false,** goes to **the next instruction** (i.e. it does nothing)

Also, all of these instructions can be written two ways:

1. `blt t0, t1, label`
   - compares two registers (sees if `t0 < t1`)
2. `blt t0, 10, label`
   - compares a register to a constant (sees if `t0 < 10`)

Mnemonic	Operation	Description
`beq a, b, label`	`if(a == b) { PC = label }`	if `a` is equal to `b`, goes to `label`.
`bne a, b, label`	`if(a != b) { PC = label }`	if `a` is NOT equal to `b`, goes to `label`.
`blt a, b, label`	`if(a < b) { PC = label }`	if `a` is less than `b`, goes to `label`.
`ble a, b, label`	`if(a <= b) { PC = label }`	if `a` is less than or equal to `b`, goes to `label`.
`bgt a, b, label`	`if(a > b) { PC = label }`	if `a` is greater than `b`, goes to `label`.
`bge a, b, label`	`if(a >= b) { PC = label }`	if `a` is greater than or equal to `b`, goes to `label`.
`bltu a, b, label`	`if(a < b) { PC = label }`	same as `blt` but does an unsigned comparison.
`bleu a, b, label`	`if(a <= b) { PC = label }`	same as `ble` but does an unsigned comparison.
`bgtu a, b, label`	`if(a > b) { PC = label }`	same as `bgt` but does an unsigned comparison.
`bgeu a, b, label`	`if(a >= b) { PC = label }`	same as `bge` but does an unsigned comparison.