# MIPS Assembly Language II
# Notes for CSE220

**Amos Omondi**

November 13, 2023

# Chapter 3

# MIPS Instruction Set

This chapter is a summary of most of the MIPS instruction set. A few other instructions not included here are covered in subsequent chapters.[1]

We will group the instructions into four categories:

- arithmetic instructions,
- logical instructions,
- shift and rotate instructions,
- branch and jump instructions

We shall use the notation $[\cdots]$ for the contents of the $\cdots$, and $\Leftarrow$ means "becomes the value of". For example, if register 7 (i.e., $7) contains the value 42, then [$7] is 42. And the effect of

```
add $8, $5, $11
```

which is that the contents of registers 5 and 11 are added and the result written into register 8, may be expressed as

$$[\$8] \Leftarrow [\$5] + [\$11]$$

We will use *rs* and *rt* to denote source-operand registers and *rd* to denote a result-destination register. *imm16* will denote an "immediate" operand (i.e., integer constant) represented in 16 bits and which will be taken negative or positive, unless otherwise specified. Similarly, *imm5* will denote an immediate

[1] What is given is not the current MIPS32 instruction set. It is an older instruction set that is slighly different but which is more suitable for our purposes, as it what is implemented in the SPIM assembler and simulator.

operand that is a number represented in 5 bits. In the assembly-language code the representation of a number will be in decimal (taken as given) or hexadecimal (digits preceded by ”0x”).

## 3.1 Arithmetic instructions

The list of instructions is given in Table 3.1. Recall that there is a special 64-bit register, the Accumulator (here abbreviated ”ACC”), that consists of two 32-bit parts, High (HI) and Low (LO).

The instructions `add` and `sub` are for signed (two's complement) addition and subtraction. The operations are on operands in registers, and they trap on overflow. `addi` is similar, but with one operand an “immediate”. The corresponding instructions for unsigned arithmetic are `addu` and `addiu`. Likewise, `sub` and `subu` are the signed and unsigned instructions for subtraction.

| **Instruction** | **Description** |
|---|---|
| `add` *rd, rs, rt* | [*rd*] ⇐ [*rs*] + [*rt*] (signed) |
| `addu` *rd, rs, rt* | [*rd*] ⇐ [*rs*] + [*rt*] (signed) |
| `addi` *rd, rs, imm16* | [*rd*] ⇐ [*rs*] + *imm16* (signed) |
| `addiu` *rd, rs, imm16* | [*rd*] ⇐ [*rs*] + *imm16* |
| `sub` *rd, rs, rt* | [*rd*] ⇐ [*rs*] - [*rt*] (signed) |
| `subu` *rd, rs, rt* | [*rd*] ⇐ [*rs*] - [*rt*] |
| `mul` *rd, rs, rt* | [*rd*] ⇐ [*rs*] * [*rt*] |
| `mult` *rs, rt* | [ACC] ⇐ [*rs*] * [*rt*] (signed) |
| `multu` *rs, rt* | [ACC] ⇐ [*rs*] * [*rt*] |
| `div` *rs, rt* | [LO] ⇐ [*rs*] div [*rt*], [HI] ⇐ [*rs*] rem [*rt*] |
| `divu` *rs, rt* | [LO] ⇐ [*rs*] div [*rt*], [HI] ⇐ [*rs*] rem [*rt*] |
| `madd` *rs, rt* | [ACC] ⇐ [ACC] + [*rs*] * [*rt*] (signed) |
| `maddu` *rs, rt* | [ACC] ⇐ [ACC] + [*rs*] * [*rt*] |
| `msub` *rs, rt* | [ACC] ⇐ [ACC] - [*rs*] * [*rt*] |
| `msubu` *rs, rt* | [ACC] ⇐ [ACC] - [*rs*] * [*rt*] |

**Table 3.1:** Arithmetic instructions

`mul` is for multiplication that yields a 32-bit result from two 32-bit operands; the result is the lower 32 bits of what would be a full 64-bit result, with the corresponding upper 32 bits discarded. On the

other hand, `mult` gives the full 64-bit result of a signed multiplication of two 32-bit operands: the upper half of the result goes into the HI part of Accumulator, and the lower half goes into the LO part. The corresponding instruction for an unsigned operation is `multu`.

`div` is for signed integer division. The result of the integer division goes into the LO register, and the remainder goes into the HI register. `divu` is for the corresponding unsigned division.

`madd` and `maddu` are for signed and unsigned Multiply-Add (Multiply-Accumulate). The result of a multiplication is added to the contents of the Accumulator. `msub` and `msubu` are similar, but with subtraction-from instead of addition-to.

| **Instruction** | **Description** |
|---|---|
| `and` *rd, rs, rt* | $[rd] \Leftarrow [rs]$ & $[rt]$ |
| `andi` *rd, rs, imm16* | $[rd] \Leftarrow [rs]$ & *imm16* |
| | |
| `or` *rd, rs, rt* | $[rd] \Leftarrow [rs] \mid [rt]$ |
| `ori` *rd, rs, imm16* | $[rd] \Leftarrow [rs] \mid$ *imm16* |
| `nor` *rd, rs, rt* | $[rd] \Leftarrow \neg([rs] \mid [rt])$ |
| | |
| `xor` *rd, rs, rt* | $[rd] \Leftarrow [rs] \oplus [rt]$ |
| `xori` *rd, rs, imm* | $[rd] \Leftarrow [rs] \oplus$ *imm16* |
| | |
| `not` *rd, rs* | $[rd] \Leftarrow \neg([rs])$ (***pseudo-instruction***) |

**Table 3.2:** Logical instructions

## 3.2 Logical instructions

These instructions perform bitwise operations on bit pairs of the operands; that is, the logical operation is on bit *i* of one operand and bit *i* of the other operand, for all 32 bits of the operands. For example,

```
0011 AND 1001 = 0001
0011  OR 1001 = 1011
```

The MIPS logical instructions are given in Table 3.2, in which & denotes AND, $\mid$ denotes OR, $\oplus$ denotes exclusive-OR, and $\neg$ denotes negation (NOT).

The pseudo-instruction `not`, for logical negation, is effected through the instruction `nor`, with $0 as one operand, on the basis that for any *a* and *b* :

$$\begin{aligned} \neg(a|b) &= (\neg a)\,\&\,(\neg b) \\ &= \neg a \quad \text{if } b = 0 \end{aligned}$$

## 3.3 Shift and rotate instructions

The instructions are given in Table 3.3. The notation $a \ll b$ means that the bits of $a$ are shifted to the left by $b$ bit positions; similarly, $a \gg b$ means that the bits of $a$ are shifted to the right by $b$ bit positions. In both cases the bits shifted out at the least significant end or most significant end are discarded.

The shifting distance is given as a 5-bit immediate operand or as *variable* whose value is represented in least significant 5 bits of the contents of a register. In the latter case the notation $[\cdots]_{4:0}$ denotes the least significant 5 bits of $[\cdots]$.

There are two types of shift operations: *logical shift* and *arithmetic shift*. In a logical left shift, 0s are inserted at the least significant end, and in a logical right shift they are inserted at the most significant end. In a logical shift no particular interpretation is made of the bits to be shifted, whereas in an arithmetic shift the bits are taken as the (two's complement here) representation of of a signed number. Recall that in such representation the sign is a truncation of an infinite string of 0s (for a positive number) or of 1s (for a negative number); so the representation of the sign should be appropriately extended. Thus in a right shift if the most significant bit is a 0, then 0s are inserted at the most significant end; and if that bit is a 1, then 1s are inserted. And in a left shift 0s are inserted at the least significant end, regardless of the sign bit. Therefore, logical and arithmetic shifts to the left have the same effect, but there is a difference between logical and arithmetic shifts to the right.

A useful aspect of shifting is that a left shift doubles the magnitude of an integer—if the bits are taken as a representation of such—and a right shift halves it. (The latter is a floor function.)

There are two instructions for rotation: `rotr` and `rotrv`, which are somewhat similar to the shift instructions `srl` and `srlv`. (n Table 3.3, we use the notation $a \hookrightarrow b$ for the rotation to the right of the bits of $a$ by $b$ bit positions.) In the shift instructions, bits shifted out at the least significant end are discarded; on the other hand, in the rotate instructions they are shifted into the most significant end.

| Instruction | Description | |
|---|---|---|
| `sll` *rd, rs, imm5* | $[rd] \Leftarrow [rs] \ll imm5$ | (logical shift) |
| `sllv` *rd, rs, rt* | $[rd] \Leftarrow [rs] \ll [rt]_{4:0}$ | (logical shift) |
| `srl` *rd, rs, imm5* | $[rd] \Leftarrow [rs] \gg imm5$ | (logical shift) |
| `srlv` *rd, rs, rt* | $[rd] \Leftarrow [rs] \gg [rt]_{4:0}$ | (logical shift) |
| `sra` *rd, rs, imm5* | $[rd] \Leftarrow [rs] \gg b\ imm5$ | (arithmetic shift) |
| `srav` *rd, rs, rt* | $[rd] \Leftarrow [rs] \gg [rt]_{4:0}$ | (arithmetic shift) |
| `rotr` *rd, rs, imm5* | $[rd] \Leftarrow [rs] \hookrightarrow imm5$ | (rotate) |
| `rotrv` *rd, rs, rt* | $[rd] \Leftarrow [rs] \hookrightarrow [rt]_{4:0}$ | (rotate) |

**Table 3.3:** Shift and rotate instructions

## 3.4 Comparison instructions

The general aspect of a comparison instruction is "set-if-less-than" (slt): two values, one in a register and the other also in a register or an immediate operand, are compared; the value in a result register is set to 1 (TRUE) if the condition is satisfied and 0 (FALSE) otherwise. Note that the latter result corresponds to "greater-than-or-equal-to".

The four comparison instructions are given in Table 3.4. The comparisons in `slt` and `slti` are signed operations; the others are unsigned.

| Instruction | Description |
|---|---|
| `slt` *rd, rs, rt* | If $[rs] < [rt]$, then $[rd] \Leftarrow 1$]; else, $[rd] \Leftarrow 0$ |
| `slti` *rd, rs, imm16* | If $[rs] < imm16$, then $[rd] \Leftarrow 1$; else, $[rd] \Leftarrow 0$ |
| `sltu` *rd, rs, rt* | If $[rs] < [rt]$, then $[rd] \Leftarrow 1$; else, $[rd] \Leftarrow 0$ |
| `sltiu` *rd, rs, imm16* | If $[rs] < imm16$, $[rd] \Leftarrow 1$; else, $[rd] \Leftarrow 0$ |

**Table 3.4:** Comparison instructions

## 3.5 Branch and jump instructions

| Instruction | Description |
|---|---|
| `beq` *rs, rt, label* | If [*rs*] = [*rd*], go to instruction at *label* |
| `bne` *rs, rt, label* | If [*rs*] $\neq$ [*rd*], go to instruction at *label* (***pseudo-instruction***) |
| `blt` *rs, rt, label* | If [*rs*] $<$ [*rd*], go to instruction at *label* |
| `bgt` *rs, rt, label* | If [*rs*] $>$ [*rd*], go to instruction at *label* |
| `ble` *rs, rt, label* | If [*rs*] $\leq$ [*rd*], go to instruction at *label* |
| `bge` *rs, rt, label* | If [*rs*] $\geq$ [*rd*], go to instruction at *label* |
| `bgez` *rs, label* | If [*rs*] $\geq 0$, go to instruction at *label* |
| `bgtz` *rs, label* | If [*rs*] $> 0$, go to instruction at *label* |
| `blez` *rs, label* | If [*rs*] $\leq 0$, go to instruction at *label* |
| `bltz` *rs, label* | If [*rs*] $< 0$, go to instruction at *label* |
| `bgezal` *rs, label* | If [*rs*] $\geq 0$, set [$31] to address of next instruction and go to instruction at *label* |
| `bltzal` *rs, label* | If [*rs*] $< 0$, set [$31] to address of following instruction and go to instruction at *label* |
| `j` *label* | Go to instruction at *label* |
| `jal` *label* | Set [$31] to address of next instruction and go to instruction at *label* |
| `jalr` *rd, rs* | Set [*rd*] to address of following instruction and o to instruction at address [*rs*] |
| `jr` *rs* | Go to instruction at address [*rs*] |

**Table 3.5:** Branch and jump instructions

## 3.6 Data-movement instructions

We use the notation **Mem**$[\cdots]\{n\}$ to denote the (first) $n$ bits of contents of the memory location whose address is $\cdots$. A double arrow indicates the direction of movement of data, i.e., from the source to the destination. And, as usual [*rs*] and [*rd*] denote the contents of the specified registers. Thus, for example, if $11 contains 3467, then `lw $13, 45($11)` is `[$13]` $\Leftarrow$ **Mem**$[3467+45]\{32\}$, which means the contents of $13 become the 32-bit value at memory location 3512.

| Instruction | Description |
|---|---|
| `lb` *rd, imm16*(*rs*) | [*rd*] $\Leftarrow$ **Mem**[[*rs*]+*imm16*]{8} (signed) |
| `lbu` *rd, imm16*(*rs*) | [*rd*] $\Leftarrow$ **Mem**[[*rs*]+*imm16*]{8} |
| `lh` *rd, imm16*(*rs*) | [*rd*] $\Leftarrow$ **Mem**[[*rs*]+*imm16*]{16} (signed) |
| `lhu` *rd, imm16*(*rs*) | [*rd*] $\Leftarrow$ **Mem**[[*rs*]+*imm16*]{16} |
| `lw` *rd, imm16*(*rs*) | [*rd*] $\Leftarrow$ **Mem**[[*rs*]+*imm16*]{32} (signed) |
| | |
| `sb` *rs, imm16*(*rd*) | **Mem**[[*rd*]+*imm16*]{8}$\Leftarrow$ [*rs*] |
| `sh` *rs, imm16*(*rd*) | **Mem**[[*rd*]+*imm16*]{16}$\Leftarrow$ [*rs*] |
| `sw` *rs, imm16*(*rd*) | **Mem**[[*rd*]+*imm16*]{32}$\Leftarrow$ [*rs*] |
| | |
| `movn` *rd, rs, rt* | If [*rt*] $\neq 0$, [*rd*] $\Leftarrow$ [*rs*] |
| `movz` *rd, rs, rt* | If [*rt*] $= 0$, [*rd*] $\Leftarrow$ [*rs*] |
| `move` *rd, rs* | [*rd*] $\Leftarrow$ [*rs*] (***pseudo-instruction***) |
| | |
| `mfhi` *rd* | [*rd*] $\Leftarrow$ [HI] |
| `mflo` *rd* | [*rd*] $\Leftarrow$ [LO] |
| `mthi` *rs* | [HI] $\Leftarrow$ [*rs*] |
| `mtlo` *rs* | [LO] $\Leftarrow$ [*rs*] |
| | |
| `la` *rd, label* | [*rd*] $\Leftarrow$ address corresponding to *label* (***pseudo-instruction***) |
| `lui` *rd, imm16* | [*rd*] $\Leftarrow$ unsigned 16-bit immediate *imm16* |
| `li` *rd, imm32* | [*rd*] $\Leftarrow$ 32-bit immediate, *imm32* (***pseudo-instruction***) |

**Table 3.6:** Data-movement instructions