academizz

The Language of the Computer and Introduction to MIPS Assembly Language

Course: Computer Architecture and Design
Study Materials

Prepared by Acadmizz

November 27, 2025

Contents

1 The Language Hierarchy

Computers operate on layers of abstraction. As computer scientists, we navigate between human logic and machine execution.

- **High-Level Language (HLL):** Portable, human-readable (e.g., C, Java, Python).
  - *Example:* `a = b + c`
- **Assembly Language:** Processor-specific, mnemonic representation of machine code.
  - *Example:* `add $t0, $s1, $s2`
- **Machine Language:** Binary (0s and 1s) that hardware executes.
  - *Example:* `000000 10001 10010 01000 00000 100000`

2 MIPS Architecture Fundamentals

MIPS (Microprocessor without Interlocked Pipeline Stages) is a **RISC** (Reduced Instruction Set Computer) architecture.

2.1 Key Design Principles

1. **Simplicity favors regularity:** Fixed 32-bit instruction length simplifies hardware decoding.
2. **Smaller is faster:** A limited number of registers (32) is faster to access than a large number.
3. **Make the common case fast:** Optimize for frequently used arithmetic and logical operations.
4. **Good design demands good compromises:** We trade flexibility for performance (e.g., fixed instruction formats).

2.2 The Register File

MIPS is a **load-store architecture**: arithmetic happens *only* in registers. Data must be loaded from memory to registers, processed, and stored back.

Register	Name	Usage
`$zero`	`$0`	Constant value 0. Cannot be overwritten.
`$v0-$v1`	`$2-$3`	Values returned from functions.
`$a0-$a3`	`$4-$7`	Arguments passed to functions.
`$t0-$t9`	`$8-$15, $24-$25`	Temporary variables (not preserved across calls).
`$s0-$s7`	`$16-$23`	Saved variables (must be preserved across calls).

Table 1: Key MIPS Registers

3 MIPS Instruction Formats

Every MIPS instruction is exactly **32 bits** long.

3.1 R-Type (Register)

Used for arithmetic (`add`, `sub`) and logical (`and`, `or`, `sll`) operations.

op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
Opcode	Source 1	Source 2	Dest	Shift Amt	Function

Example: `add $t0, $s1, $s2`

- `op`: 0 (R-type)
- `rs`: `$s1` (17) → `10001`
- `rt`: `$s2` (18) → `10010`
- `rd`: `$t0` (8) → `01000`
- `shamt`: 0
- `funct`: 32 (add) → `100000`

3.2 I-Type (Immediate)

Used for constants (`addi`), loads (`lw`), stores (`sw`), and branches (`beq`).

op (6)	rs (5)	rt (5)	constant / address (16)
Opcode	Base/Src	Dest/Src	16-bit Immediate Value

Example: `lw $t0, 32($s3)`

- `op`: 35 (lw)
- `rs`: `$s3` (base address)
- `rt`: `$t0` (destination)
- `constant`: 32 (offset)

3.3 J-Type (Jump)

Used for jumps (`j`, `jal`).

op (6)	address (26)
Opcode	Pseudo-direct target address

4 Translating C to MIPS (Theory & Examples)

4.1 Arithmetic

Formula: $f = (g + h) - (i + j)$
Assume f, g, h, i, j are in $s0 through $s4.

```
add $t0, $s1, $s2    # t0 = g + h
add $t1, $s3, $s4    # t1 = i + j
sub $s0, $t0, $t1    # f = t0 - t1
```

4.2 Memory Access (Arrays)

Formula: $g = h + A[8]$
Assume g, h are in $s1, $s2 and base of A is in $s3.
Note: MIPS uses byte addressing. Index 8 of an integer array means an offset of $8 \times 4 = 32$ bytes.

```
lw  $t0, 32($s3)     # Load A[8] into temp reg $t0
add $s1, $s2, $t0    # g = h + A[8]
```

4.3 Control Flow (If-Else)

C Code:

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

Assume f, g, h, i, j are in $s0 through $s4.

MIPS Assembly:

```
      bne $s3, $s4, Else    # If i != j, go to Else
      add $s0, $s1, $s2     # f = g + h (if block)
      j Exit                # Jump over Else block
Else: sub $s0, $s1, $s2     # f = g - h (else block)
Exit:
```

4.4 Loops (While)

C Code:

```
while (save[i] == k)
    i += 1;
```

Assume i is in $s3, k in $s5, base of save in $s6.

MIPS Assembly:

```
Loop: sll  $t1, $s3, 2      # t1 = i * 4 (byte offset)
      add  $t1, $t1, $s6    # t1 = address of save[i]
      lw   $t0, 0($t1)      # t0 = save[i]
      bne  $t0, $s5, Exit   # if save[i] != k, exit loop
      addi $s3, $s3, 1      # i = i + 1
      j    Loop             # Jump back to start
Exit:
```

5 Addressing Modes (How We Find Data)

1. **Register Addressing:** Operand is a register (fastest).
   - Example: `add $t0, $s1, $s2`
2. **Immediate Addressing:** Operand is a constant in the instruction.
   - Example: `addi $s1, $s2, 20`
3. **Base (Displacement) Addressing:** Address = Register + Offset (16-bit).
   - Example: `lw $s1, 20($s2)` $\rightarrow$ Addr = `$s2` + 20.
4. **PC-Relative Addressing:** Branch Target = $PC + 4 + (offset \times 4)$.
   - Used for `beq`, `bne`.
5. **Pseudo-direct Addressing:** Jump Target = $PC[31:28] \mid (address[25:0] \ll 2)$.
   - Used for `j`.

6 Practice Problems & Solutions

Problem 1: Machine Code Translation (R-Type)

Question: Translate `sub $t0, $s1, $s2` to hexadecimal machine code.
Registers: `$t0`=8, `$s1`=17, `$s2`=18.
Opcode/Funct: `sub` opcode=0, funct=34 (0x22).

Solution:

1. **Format:** `op | rs | rt | rd | shamt | funct`
2. **Values:** 0 | 17 | 18 | 8 | 0 | 34
3. **Binary:**
   - 0 $\rightarrow$ `000000`
   - 17 $\rightarrow$ `10001`
   - 18 $\rightarrow$ `10010`
   - 8 $\rightarrow$ `01000`
   - 0 $\rightarrow$ `00000`
   - 34 $\rightarrow$ `100010`
4. **Concatenate:** `000000 10001 10010 01000 00000 100010`
5. **Group by 4:** `0000 0010 0011 0010 0100 0000 0010 0010`
6. **Hex:** `0x02324022`

Problem 2: Branch Target Calculation

Question: Current instruction is at address `0x00400014`: `beq $s0, $s1, Label`. The `Label` is at address `0x00400028`. What is the 16-bit immediate field value stored in the machine code?

Solution:

1. **Formula:** Target Addr = $(PC + 4) + (Immediate \times 4)$
2. **Values:**
   - Target = `0x00400028`
   - $PC + 4$ = `0x00400018` (PC increments *before* calculation)
3. **Calculate Byte Distance:**
   - Distance = Target - $(PC + 4)$
   - `0x00400028` - `0x00400018` = `0x10` (16 decimal) bytes.
4. **Convert to Words (Immediate value):**
   - Immediate = Distance / 4
   - 16 / 4 = 4.
5. **Answer:** The immediate field is **4** (or `0000 0000 0000 0100` binary).

Problem 3: Jump Address Calculation

Question: Instruction `j Label` is at memory address `0x00400050`. The `Label` is at `0x004000C0`. What is the 26-bit field in the instruction?

Solution:

1. **Constraint:** Jump target must be word-aligned (divisible by 4).
2. **Extract Address:** `0x004000C0`.
3. **Divide by 4 (Shift Right 2):**
   - `0x004000C0` / 4 = `0x00100030`
4. **Remove Top 4 bits:**
   - The top 4 bits of the PC (`0000`) are assumed.
   - We keep the lower 28 bits (which became 26 bits after dividing by 4).
   - `0x00100030` in binary (28 bits): `0000 0000 0001 0000 0000 0000 0011 0000`
   - Take the lower 26 bits: `00 0001 0000 0000 0000 0011 0000`
5. **Answer (Hex):** `0x0100030` (This 26-bit value is put into the J-Type instruction).

Problem 4: C to Assembly Logic

Question: Translate `A[i] = h + A[i];`
h in $s2, i in $s4, Base of A in $s3.

Solution:

```
sll  $t0, $s4, 2      # $t0 = i * 4 (byte offset)
add  $t0, $s3, $t0    # $t0 = address of A[i] (Base + offset)
lw   $t1, 0($t0)      # $t1 = value of A[i]
add  $t1, $s2, $t1    # $t1 = h + A[i]
sw   $t1, 0($t0)      # Store result back into A[i] at address $t0
```

Problem 5: I-Type Machine Code Translation

Question: Translate `addi $s1, $s2, -50` to hexadecimal machine code.
Registers: `$s1`=17, `$s2`=18.
Opcode: `addi` opcode=8.

Solution:

1. **Format:** `op | rs | rt | immediate`
2. **Values:** 8 | 18 | 17 | -50
3. **Binary Conversion:**
   - Opcode (8): `001000`
   - rs ($s2 = 18): `10010`
   - rt ($s1 = 17): `10001`
   - Immediate (-50): Convert to 16-bit 2's complement.
     - 50 = 0000 0000 0011 0010
     - Invert: 1111 1111 1100 1101
     - Add 1: 1111 1111 1100 1110 (`0xFFCE`)
4. **Concatenate:** `001000 10010 10001 1111111111001110`
5. **Group by 4:** `0010 0010 0101 0001 1111 1111 1100 1110`
6. **Hex:** `0x2251FFCE`

Problem 6: Base Addressing Calculation

Question: A C code statement is `B[8] = A[i] + A[j]`.
Assume base addresses of A and B are in `$s0` and `$s1`, and `i, j` are in `$s2, $s3`.
Write the MIPS code.

Solution:

```
sll   $t0, $s2, 2       # t0 = i * 4 (offset for A[i])
add   $t0, $s0, $t0     # t0 = address of A[i]
lw    $t1, 0($t0)       # t1 = value of A[i]

sll   $t2, $s3, 2       # t2 = j * 4 (offset for A[j])
add   $t2, $s0, $t2     # t2 = address of A[j]
lw    $t3, 0($t2)       # t3 = value of A[j]

add   $t4, $t1, $t3     # t4 = A[i] + A[j]
sw    $t4, 32($s1)      # Store result in B[8] (8 * 4 = 32 offset)
```