academizz

Instructions

Course: Computer Architecture and Design

Study Materials

Prepared by Acadmizz

November 27, 2025

Contents

1 Hierarchy of Computer Languages **3**

2 The Computer Instruction **3**
2.1 Instruction Code 4
2.2 Stored Program Organization 4

3 The Instruction Cycle **5**
3.1 Basic Instruction Cycle 6
3.2 Instruction Cycle with Interrupts 6

4 Instruction Categories **7**
4.1 Data Transfer 7
4.2 Arithmetic and Logic 8
4.3 Branch (Control Transfer) 8
4.4 Input/Output (I/O) 8
4.5 Miscellaneous / System Control 8
4.6 Floating Point (FP) 8
4.7 BCD (Binary Coded Decimal) 8

5 Instruction Formats: Fixed vs. Variable **8**
5.1 Fixed-Length Format (RISC) 9
5.2 Variable-Length Format (CISC) 9

6 RISC vs. CISC Architectures **10**
6.1 Architecture Philosophies 10
6.1.1 CISC (Complex Instruction Set Computer) 10
6.1.2 RISC (Reduced Instruction Set Computer) 10

7 Instruction Formats & Addressing (MIPS Focus) 10
7.1 Three-Address Instructions 10
7.2 Two-Address Instructions 11
7.3 One-Address Instructions (Accumulator) 11
7.4 Zero-Address Instructions (Stack) 11

8 MIPS Instruction Formats (Fixed Length) 11
8.1 R-Type (Register) 11
8.2 I-Type (Immediate) 12
8.3 J-Type (Jump) 12

9 Theoretical Questions & Answers 12

10 Practice Problems & Solutions 14

1 Hierarchy of Computer Languages

Computers operate on multiple layers of abstraction, translating human intent into electrical signals.

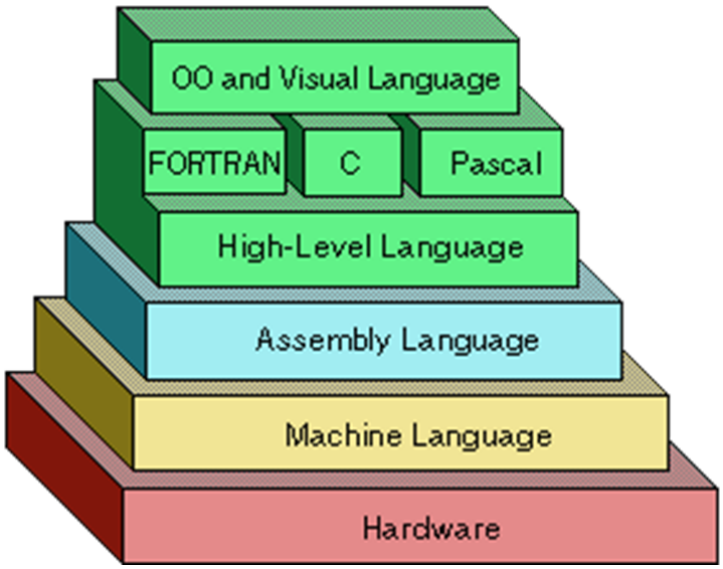

Figure 1: Hierarchy of Computer Languages

- **High-Level Languages (HLL):** Portable, human-readable code (e.g., C, Java). It uses constructs like if-else, loops, and functions.
  - Example: `int x = 42;`
- **Assembly Language:** Processor-specific mnemonic representation of machine instructions. It is a direct translation of machine code.
  - Example: `MOV R3, #15`
- **Machine Language:** Binary code (0s and 1s) directly executed by the hardware.
  - Example: `1100 1010 1011 0011`
- **System Hardware:** The physical logic gates and circuits that perform the operations.

2 The Computer Instruction

A **Computer Instruction** is a binary code that specifies a sequence of micro-operations for the computer.

- Instructions are read from memory and loaded into the control register.
- The control unit interprets the binary code and issues the corresponding control signals.

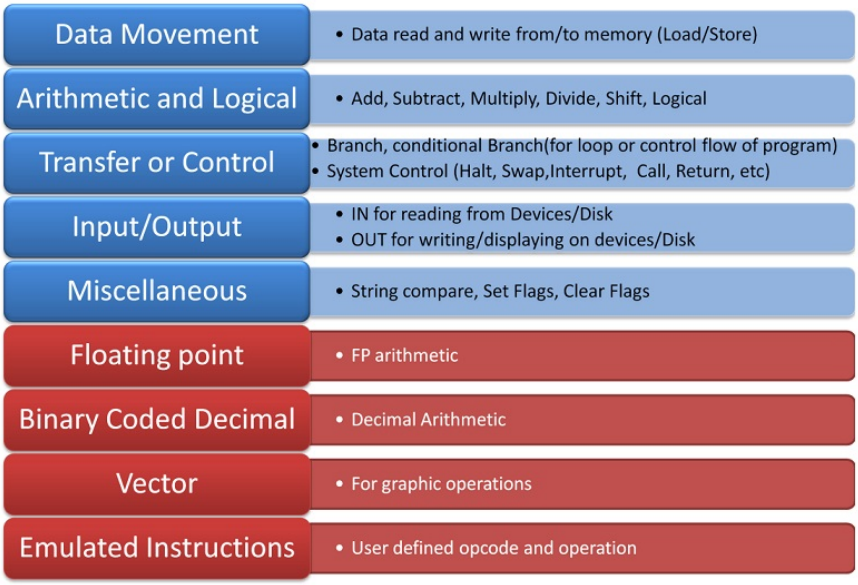

Figure 2: Data Movement: High Level to Machine Level

2.1 Instruction Code

An instruction code is a group of bits that directs the computer to perform a specific operation. It is divided into parts:

1. **Opcode (Operation Code):** Defines the type of operation (e.g., Add, Subtract, Shift).
    - If a system supports 2^n distinct operations, the opcode must be at least n bits long.
2. **Address/Operand:** Specifies the memory location or the data itself to be used by the operation.

2.2 Stored Program Organization

In the stored program concept, both **Instructions (Program)** and **Operands (Data)** are stored in the same main memory.

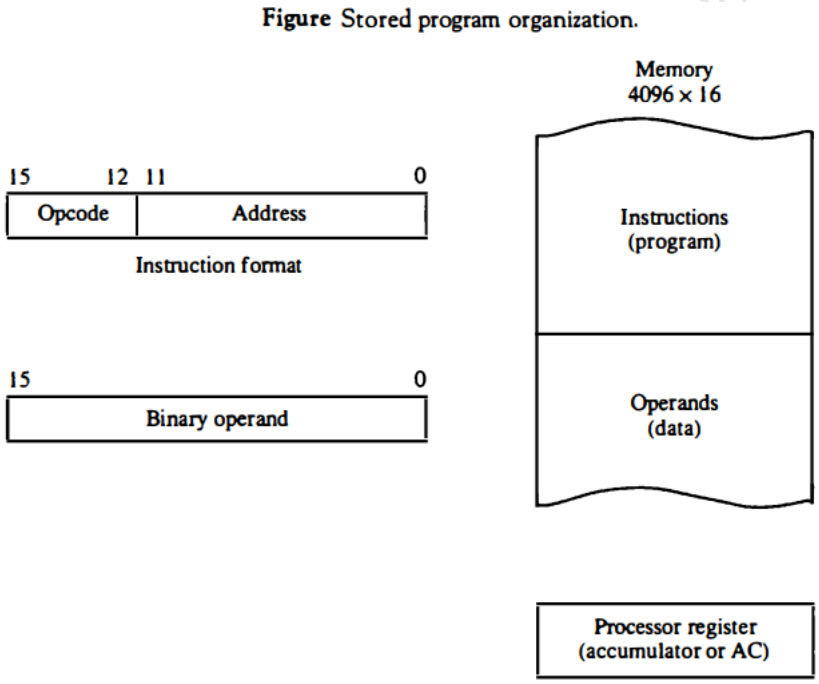

Figure 3: Stored Program Organization: Instructions and Data in Memory

- The CPU fetches instructions sequentially from memory.
- It uses the **Program Counter (PC)** to keep track of the next instruction address.
- Data is fetched into the **Accumulator (AC)** or general purpose registers for processing.

3 The Instruction Cycle

The processing required for a single instruction is called an instruction cycle.

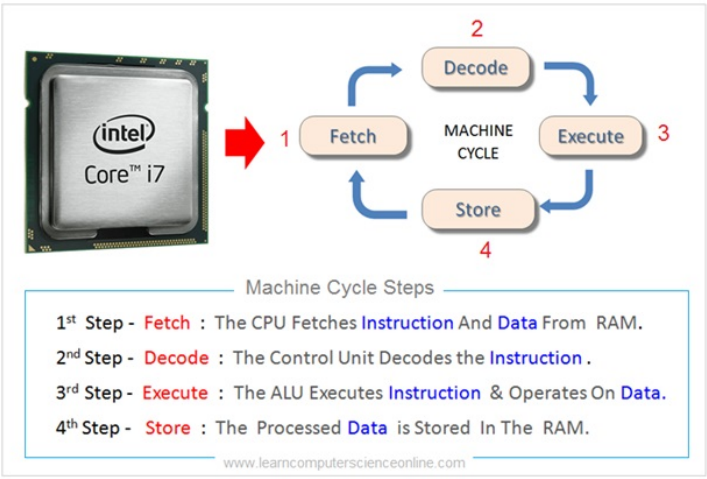

Figure 4: Generic CPU Instruction Cycle

3.1 Basic Instruction Cycle

The cycle consists of two main phases: 1. **Fetch Cycle:** Read the instruction from memory into the CPU. 2. **Execute Cycle:** Decode the opcode and perform the operation.

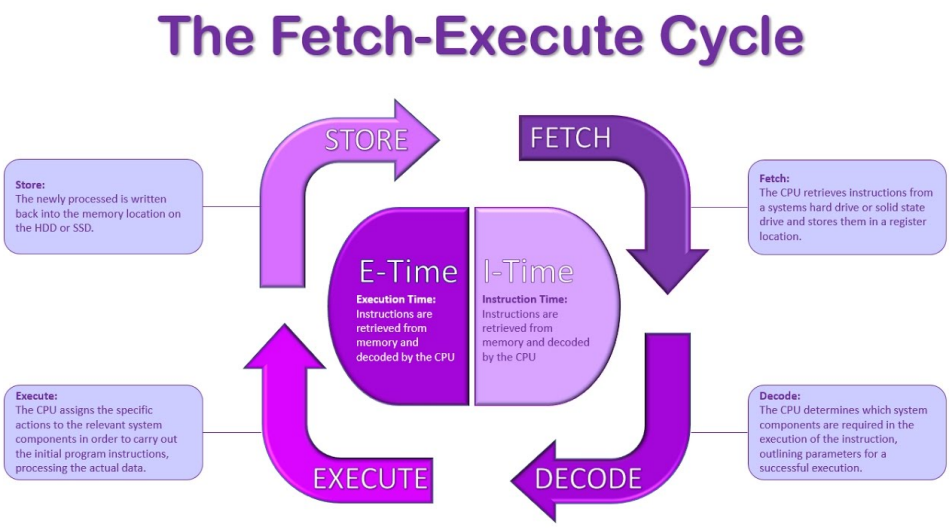

Figure 5: Basic Fetch-Execute Cycle

3.2 Instruction Cycle with Interrupts

To handle external events (I/O, errors), an **Interrupt Cycle** is added.

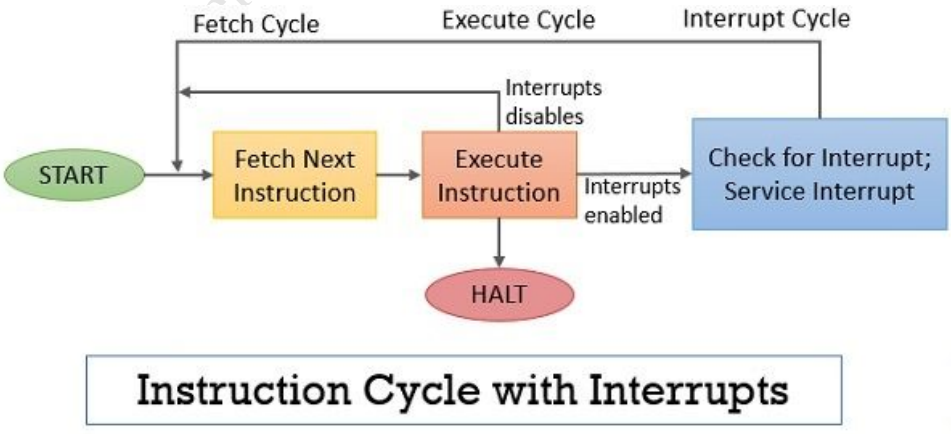

Figure 6: Instruction Cycle with Interrupts

- After executing an instruction, the CPU checks for interrupts.
- **If Interrupt Pending:** The CPU suspends the current program, saves its context, and jumps to an **Interrupt Handling Program**.
- **If No Interrupt:** The CPU fetches the next instruction of the current program.

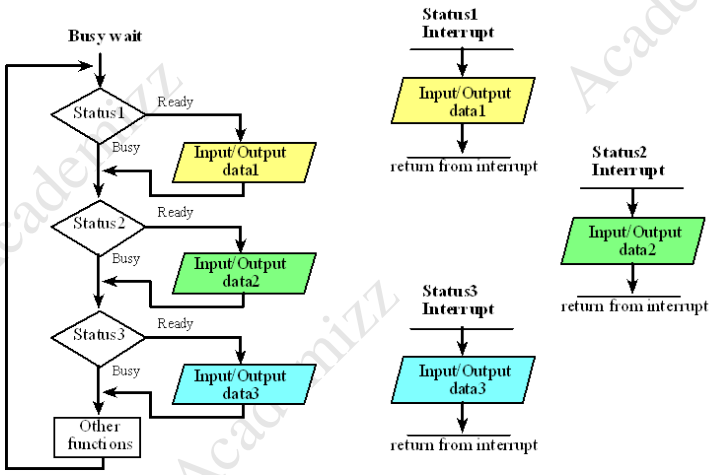

Figure 7: Interrupt Operation Flow

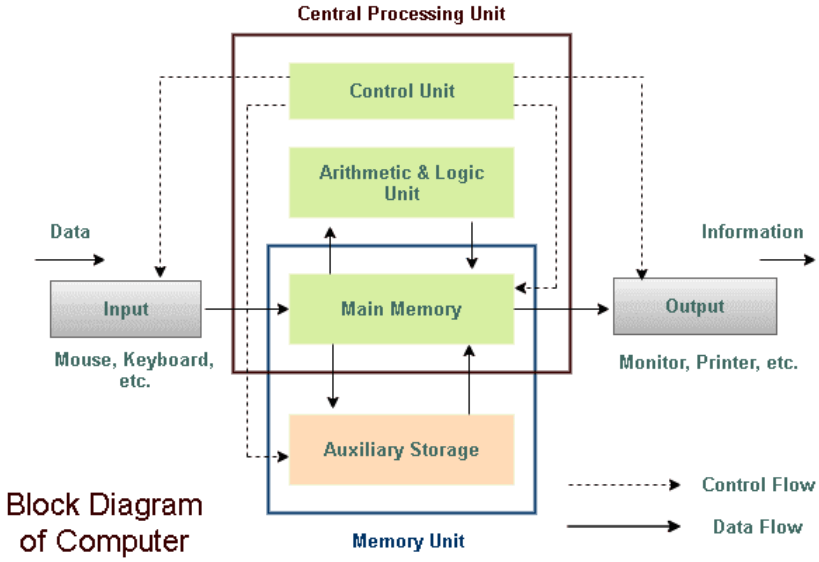

Figure 8: Detailed CPU Behavior Flowchart

4 Instruction Categories

Instructions are classified based on their function. A complete instruction set includes:

4.1 Data Transfer

Moves data between registers, memory, and I/O devices without changing the data content.

- `MOV`: Transfer data (e.g., `MOV R1, A`).
- `LOAD`: Load from memory to register (e.g., `LOAD R1, A`).
- `STORE`: Store register to memory (e.g., `STORE R1, A`).

4.2 Arithmetic and Logic

Performs computations on data.

- `ADD`, `SUB`, `MUL`, `DIV`.
- `AND`, `OR`, `XOR`, `NOT`.
- `SHIFT`, `ROTATE`.

4.3 Branch (Control Transfer)

Alters the sequence of execution (Program Counter manipulation).

- **Conditional:** `BZ` (Branch if Zero), `BNE` (Branch Not Equal). Used for loops and if-else.
- **Unconditional:** `JMP` (Jump), `CALL` (Call subroutine), `RET` (Return).

4.4 Input/Output (I/O)

Handles communication with external devices.

- `IN`: Read from a port.
- `OUT`: Write to a port.

4.5 Miscellaneous / System Control

- `NOP`: No Operation (used for timing delays).
- `HLT`: Halt execution.
- `WAIT`: Wait for an event/interrupt.

4.6 Floating Point (FP)

Specialized arithmetic for decimal numbers (requires FPU).

- `FADD`, `FSUB`, `FSQRT`.

4.7 BCD (Binary Coded Decimal)

Instructions to handle decimal arithmetic directly.

- `DAA` (Decimal Adjust Accumulator): Corrects the result of binary addition to valid BCD.

5 Instruction Formats: Fixed vs. Variable

The format defines the layout of the instruction bits (Opcode + Operands).

5.1 Fixed-Length Format (RISC)

All instructions are the same size (e.g., 32 bits).

- **Pros:** Simple decoding, fast execution (pipelining).
- **Cons:** May waste space (padding bits).
- **Example:** MIPS, ARM.

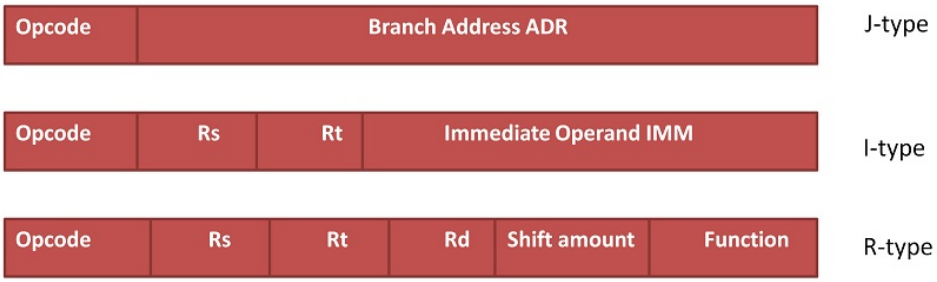

Figure 9: Fixed Length Instruction Format (MIPS R-Type)

5.2 Variable-Length Format (CISC)

Instructions vary in size (e.g., 1 to 15 bytes) depending on the operation and addressing mode.

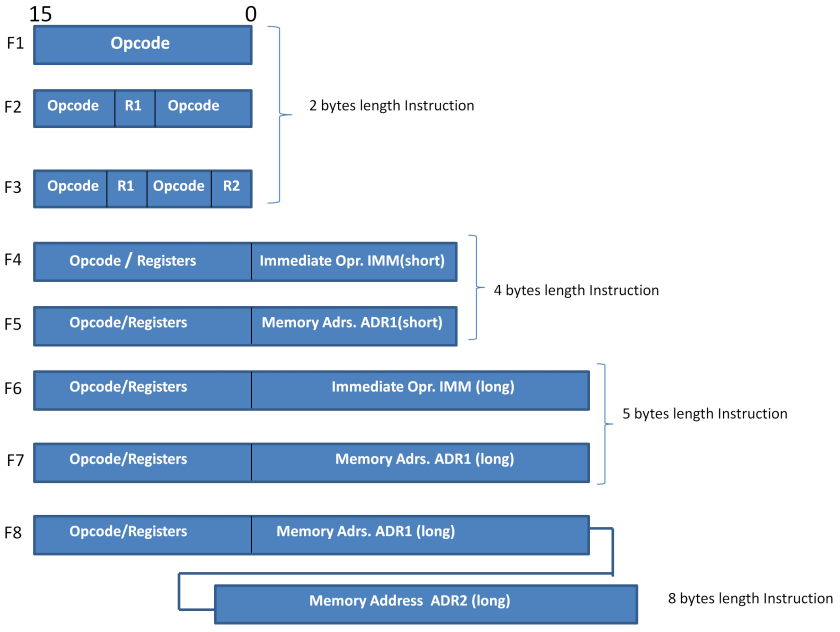

Figure 10: Variable Length Instruction Formats

- **Pros:** Memory efficient (common instructions are short).
- **Cons:** Complex decoding hardware required.

- **Example:** Intel x86.

6 RISC vs. CISC Architectures

Feature	RISC (Reduced)	CISC (Complex)
Instruction Size	Small, Simple	Large, Complex
Length	Fixed (e.g., 32 bits)	Variable (1-15 bytes)
Execution	Single cycle mostly	Multi-cycle often
Control Unit	Hardwired (Fast)	Microprogrammed
Memory Usage	Less efficient (more code)	Efficient (compact code)
Examples	MIPS, ARM	Intel x86, VAX

Table 1: Comparison of RISC and CISC Architectures

6.1 Architecture Philosophies

6.1.1 CISC (Complex Instruction Set Computer)

Goal: Minimize the number of instructions per program.

- **Philosophy:** Hardware is expensive (historically), so make each instruction do a lot of work (e.g., "Add value from memory location A to memory location B").
- **Outcome:** Code is very compact (small memory footprint), but the processor (hardware) must be very complex to decode and execute variable-length, powerful instructions.

6.1.2 RISC (Reduced Instruction Set Computer)

Goal: Minimize the cycles per instruction.

- **Philosophy:** Simplify the hardware to run very fast. Only simple instructions (like "Add two registers") are allowed. Complex tasks are broken down into multiple simple instructions by the **Compiler**.
- **Outcome:** The hardware is simpler and easier to optimize (pipelining), but the code size is larger.

7 Instruction Formats & Addressing (MIPS Focus)

The number of address fields in an instruction dictates how expressions are evaluated.
Example Expression: $X = (A + B) \times (C + D)$

7.1 Three-Address Instructions

Specifies two source operands and one destination.

```
ADD R1, A, B       # R1 <- M[A] + M[B]
ADD R2, C, D       # R2 <- M[C] + M[D]
MUL X, R1, R2      # M[X] <- R1 * R2
```

7.2 Two-Address Instructions

One operand serves as both source and destination.

```
MOV R1, A          # R1 <- M[A]
ADD R1, B          # R1 <- R1 + M[B]
MOV R2, C          # R2 <- M[C]
ADD R2, D          # R2 <- R2 + M[D]
MUL R1, R2         # R1 <- R1 * R2
MOV X, R1          # M[X] <- R1
```

7.3 One-Address Instructions (Accumulator)

Uses an implicit Accumulator (AC) register.

```
LOAD A             # AC <- M[A]
ADD B              # AC <- AC + M[B]
STORE T            # M[T] <- AC
LOAD C             # AC <- M[C]
ADD D              # AC <- AC + M[D]
MUL T              # AC <- AC * M[T]
STORE X            # M[X] <- AC
```

7.4 Zero-Address Instructions (Stack)

Uses a Stack (Last-In, First-Out). Operands are popped, result is pushed.

```
PUSH A
PUSH B
ADD                # Pop A, B; Push (A+B)
PUSH C
PUSH D
ADD                # Pop C, D; Push (C+D)
MUL                # Pop (A+B), (C+D); Push Result
POP X              # Store Top of Stack to X
```

8 MIPS Instruction Formats (Fixed Length)

MIPS is a **RISC** architecture with fixed 32-bit instructions.

8.1 R-Type (Register)

Used for arithmetic instructions (e.g., `add`, `sub`).

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, `funct`=32 (for add)
- `rs`=$s1, `rt`=$s2, `rd`=$t0

8.2 I-Type (Immediate)

Used for data transfer and constants (e.g., `lw`, `addi`).

op (6)	rs (5)	rt (5)	address/immediate (16)
Opcode	Base	Dest	Offset / Constant

Example: `lw $s1, 100($s2)`

- `op`=35 (for lw)
- `rs`=$s2 (base), `rt`=$s1 (dest), `addr`=100

8.3 J-Type (Jump)

Used for branching/jumps (e.g., `j`).

op (6)	target address (26)
Opcode	Jump Target Address

Example: `j LABEL`

9 Theoretical Questions & Answers

Q1: Architecture vs. Organization

Question: Explain the difference between ”Architecture” and ”Organization” in the context of computer systems.

Answer:

- **Computer Architecture** refers to the attributes of a system visible to the programmer (e.g., instruction set, data types, addressing modes). It answers *“What does the system do?”*.
- **Computer Organization** refers to the operational units and their interconnections (e.g., control signals, memory technology, clock frequency). It answers *“How does the system do it?”*.

Q2: Fixed vs. Variable Length Instructions

Question: Why does MIPS use a fixed-length instruction format (32 bits), whereas x86 uses variable-length instructions? What is the trade-off?

Answer:

- **MIPS (Fixed):** Simplifies instruction decoding, allowing for faster execution (often single-cycle).
- **x86 (Variable):** More memory efficient as complex instructions are encoded compactly.
- **Trade-off:** MIPS prioritizes speed and hardware simplicity. x86 prioritizes code density and backward compatibility.

Q3: Program Counter (PC) Role

Question: Describe the role of the Program Counter (PC) during the "Fetch" phase.

Answer: The PC holds the memory address of the *next* instruction. During fetch:

1. CPU reads the address in the PC.
2. Fetches the instruction from that address.
3. Increments the PC (typically by 4) to point to the next instruction.

Q4: Interrupt Cycle

Question: What is the purpose of the "Interrupt Cycle"?

Answer: It allows the CPU to handle external events (I/O, errors) immediately.

- Checks for interrupts after executing an instruction.
- If pending, suspends the current program, saves state, and jumps to an Interrupt Service Routine (ISR).

10 Practice Problems & Solutions

Problem 1: Stack vs. Accumulator

Question: Write the assembly code to evaluate $Z = (X \times Y) + (W \times U)$ using:

1. Zero-Address (Stack) Machine
2. One-Address (Accumulator) Machine

Solution: 1. Stack Machine:

```
PUSH X
PUSH Y
MUL        # Top = X*Y
PUSH W
PUSH U
MUL        # Top = W*U, Next = X*Y
ADD        # Top = (X*Y) + (W*U)
POP Z
```

2. Accumulator Machine:

```
LOAD X     # AC = X
MUL Y      # AC = X * Y
STORE T    # T = X * Y (Temp storage)
LOAD W     # AC = W
MUL U      # AC = W * U
ADD T      # AC = (W * U) + (X * Y)
STORE Z    # Z = Result
```

Problem 2: MIPS Instruction Encoding

Question: Encode the instruction `sub $t0, $s1, $s2` into binary.
Given: Opcode=0, Funct=34, $t0=8, $s1=17, $s2=18.

Solution:

1. **Format (R-Type):** `op | rs | rt | rd | shamt | funct`
2. **Mapping:**
   - `op`: 000000
   - `rs` ($s1=17): 10001
   - `rt` ($s2=18): 10010
   - `rd` ($t0=8): 01000
   - `shamt`: 00000
   - `funct`: 100010
3. **Binary:** `000000 10001 10010 01000 00000 100010`

Problem 3: Instruction Count

Question: A CISC architecture executes a task in 5 instructions, each taking 10 cycles. A RISC architecture executes the same task in 15 instructions, each taking 2 cycles. Which is faster?

Solution: Formula: Total Cycles = Instruction Count $\times$ Cycles Per Instruction (CPI)
CISC:

$$5 \text{ instr} \times 10 \text{ cycles/instr} = 50 \text{ cycles}$$

RISC:

$$15 \text{ instr} \times 2 \text{ cycles/instr} = 30 \text{ cycles}$$

Answer: The RISC architecture is faster (30 cycles vs 50 cycles).

Problem 4: Hex to Assembly (Decoding)

Question: Decode the hex instruction `0x8C090014` into MIPS assembly.
Hint: Opcode for `lw` is 35 (binary 100011).

Solution: 1. Convert Hex to Binary: `100011 00000 01001 0000000000010100`
2. Group by Fields (I-Type):

- **Opcode:** 100011 (35) $\rightarrow$ `lw`
- **rs (Base):** 00000 (0) $\rightarrow$ `$zero` (or `$0`)
- **rt (Dest):** 01009 (9) $\rightarrow$ `$t1`
- **Immediate:** 0...10100 (20 decimal)

Assembly: `lw $t1, 20($zero)`

Problem 5: Calculating Effective Addresses

Question: Calculate the target address for the branch instruction:
`bne $s0, $s1, Label`
Assume the instruction is at address `0x4000`, and the offset (immediate field) is `0x0004`.

Solution: Formula: Target = $(PC + 4) + (\text{Offset} \times 4)$

- **PC + 4:** $0x4000 + 4 = 0x4004$
- **Shift Offset:** $4 \times 4 = 16$ bytes ($0x10$)
- **Total:** $0x4004 + 0x10 = 0x4014$

Target Address: `0x4014`