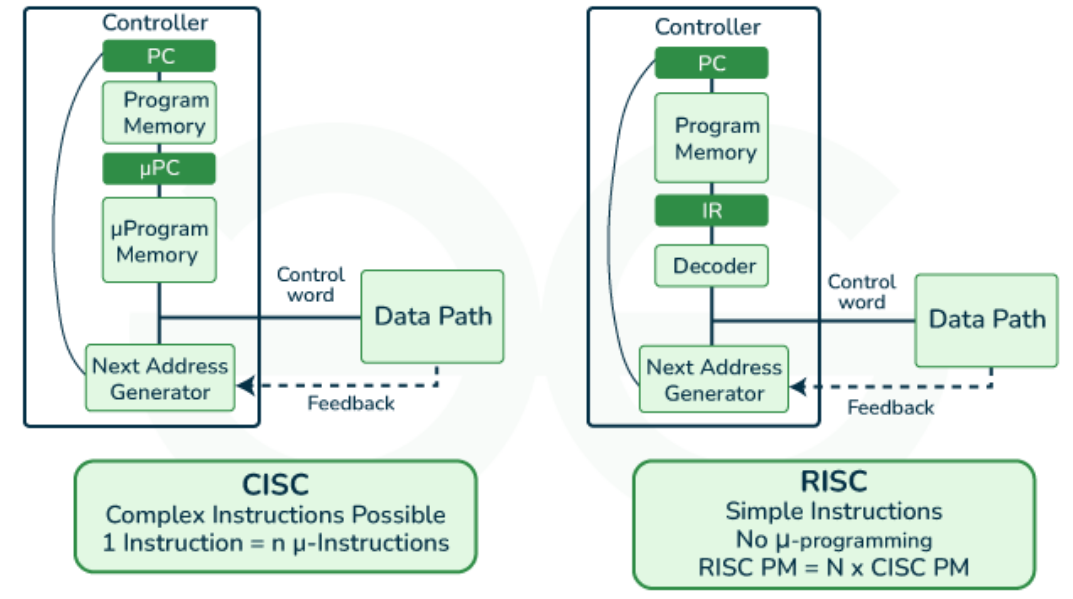
**Unit - 1**

**RISC vs CISC Architecture**

| **Basis** | **RISC (Reduced Instruction Set Computing) Apple** | **CISC (Complex Instruction Set Computing) Intel** |
| --- | --- | --- |
| **Focus** | Focuses on **software optimization**. | Focuses on **hardware optimization**. |
| **Control Unit** | Uses a **hardwired control unit** for fast instruction execution. | Uses both **hardwired** and **microprogrammed control units** for flexibility. |
| **Transistor Usage** | Transistors are used to **create** **more registers** to speed up instruction execution. | Transistors are used to **store** **complex instructions** and features like microcode. |
| **Instruction Size** | Instructions are **fixed in size**, typically 32 bits. | Instructions have **variable sizes**, depending on complexity. |
| **Arithmetic Operations** | Can perform **only register-to-register** operations. | Can perform **register-to-memory** and **memory-to-memory** operations. |
| **Registers** | Requires a **larger number of registers** to store intermediate data. | Requires **fewer registers** as it uses memory for many operations. |
| **Code Size** | **Larger code size** due to simpler instructions and more registers. | **Smaller code size** because complex instructions do more work per instruction. |
| **Instruction Execution** | Most instructions are executed in **one clock cycle**. | Many instructions take **multiple clock cycles** due to their complexity. |
| **Instruction Format** | Instructions are generally **one word long**. | Instructions are often **larger than one word**, depending on the task. |
| **Addressing Modes** | **Few and simple addressing modes** (e.g., immediate, register, base + offset). | **Many and complex addressing modes** (e.g., base, index, displacement). |
| **Pipeline Efficiency** | Pipelining is **very efficient** with most instructions completing in one cycle. | Pipelining is **less efficient** due to the variable length and complexity of instructions. |
| **Power Consumption** | Consumes **low power**, which is ideal for mobile and embedded devices. | Consumes **more power** due to complex operations and hardware overhead. |
| **RAM Requirement** | Requires **more RAM** to store larger code size and data. | Requires **less RAM** since the instructions do more work and are smaller. |
| **Operand Handling** | Operands are generally **restricted to registers** (e.g., register-register arithmetic). | Arithmetic and logical operations can be applied to **both memory and registers**. |
| **Condition Codes** | **No condition codes** are used for branching and decisions. | **Uses condition codes** (status flags) for decisions and branching. |
| **ISA Abstraction** | The internal working of the processor is **exposed** to machine-level programs. | The internal details are **abstracted**, with the ISA hiding the underlying hardware implementation. |
| **Array Support** | Does not have built-in support for arrays; relies on **software-level loops**. | Provides built-in support for **array handling** with dedicated instructions. |
| **RAM Usage** | **Heavy RAM usage** due to larger code and data requirements. | **More efficient use of RAM** since fewer instructions are needed to achieve complex tasks. |
| **Example Architectures** | Examples include **MIPS, ARM, and PowerPC**. | Examples include **x86 and Intel’s IA-32 architecture**. |



**Data Representation: Basic Formats**

**1.1 Basics of Data Representation:**

Computers represent all types of data in **binary format** because the fundamental electronic components (transistors) in computers only recognize two states: **on (1)** or **off (0)**. This binary system is a base-2 numbering system that uses **bits** (binary digits) to store and process data.

* **Bit (binary digit):** The smallest unit of data in a computer. It can be either 0 or 1.
* **Byte:** A group of 8 bits. Most data in modern systems are stored in multiples of bytes.

**Common Formats for Representing Data:**

1. **Unsigned Integers:** Represent only positive whole numbers (e.g., 0, 1, 2, 3, etc.).
2. **Signed Integers (Two’s Complement):** Can represent both positive and negative numbers.
3. **Fixed-point Numbers:** Used for representing fractional numbers with a fixed position for the decimal point.
4. **Floating-point Numbers (IEEE 754):** Used for representing very large or very small numbers with high precision, similar to scientific notation.

Now let's explore **fixed-point numbers** and **floating-point numbers** in depth with examples.

**1.2 Fixed-Point Numbers**

**What Are Fixed-Point Numbers?**

Fixed-point representation is used to represent numbers that have both integer and fractional parts (decimal numbers), where the **decimal point is fixed** at a certain position in the binary representation. The position of the decimal point is agreed upon before using the system, hence the term "fixed-point."

Fixed-point representation is simpler but limited because the range and precision depend on the number of bits allocated for the integer and fractional parts.

**Steps for Representing Fixed-Point Numbers:**

Example:

Consider representing the number **5.75** in fixed-point format using 8 bits, where 4 bits are for the integer part and 4 bits are for the fractional part.

**Step 1: Convert the Integer Part to Binary**

* The integer part of **5** in binary is 101. Using 4 bits, it becomes 0101.

**Step 2: Convert the Fractional Part to Binary**

* The fractional part of **0.75** can be converted to binary by multiplying it by 2:
  + ( 0.75 \times 2 = 1.5 ) → Take the integer part (1).
  + ( 0.5 \times 2 = 1.0 ) → Take the integer part (1).
  + Therefore, **0.75** in binary is 11.
* Pad the fractional part to 4 bits: 1100.

**Step 3: Combine the Integer and Fractional Parts**

* The integer part is 0101, and the fractional part is 1100.
* The **fixed-point representation of 5.75** in an 8-bit system with 4 integer and 4 fractional bits is: [ 0101.1100 ]
* This binary value corresponds to **5.75** in decimal.

Summary of Fixed-Point Representation:

**Advantages:** Simple to implement, especially in hardware like embedded systems.

**Disadvantages:** Limited range and precision since both the integer and fractional parts have a fixed number of bits.

Example: If we only have 4 bits for the integer part, we can represent numbers from **-8** to **+7** (in two’s complement). Any number larger or smaller cannot be represented.

**1.3 Floating-Point Numbers (IEEE 754 Standard)**

**What Are Floating-Point Numbers?**

Floating-point representation is more complex than fixed-point. It is designed to handle a much larger range of numbers, including very small and very large values. This is achieved by allowing the **decimal point to "float"**; that is, the position of the decimal point can change based on the value of the number.

Floating-point numbers are commonly represented using the **IEEE 754 standard**, which defines how numbers are encoded in binary using **scientific notation**.

In scientific notation, a number is expressed as: [ \text{Number} = (-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{\text{exponent} - \text{bias}} ]

* **Sign Bit (1 bit):** Determines whether the number is positive or negative (0 for positive, 1 for negative).
* **Exponent (8 bits for single-precision):** Encodes the exponent, adjusted by a bias.
* **Mantissa (23 bits for single-precision):** Represents the significant digits of the number.

**Steps for Representing Floating-Point Numbers (Single Precision, 32 bits):**

Example:

Let's represent **5.75** as a floating-point number using IEEE 754 single-precision (32 bits).

**Step 1: Convert the Number to Binary**

* **5.75** in decimal is equal to 101.11 in binary.

**Step 2: Normalize the Binary Number**

* To normalize the binary number, we need it in the form of **1.mantissa × 2^exponent**.
* **101.11** in binary can be normalized as: [ 1.0111 \times 2^2 ]
* Here, the **mantissa** is 0111, and the **exponent** is 2.

**Step 3: Encode the Sign Bit**

* Since **5.75** is positive, the sign bit is 0.

**Step 4: Encode the Exponent**

* In IEEE 754 single-precision, the exponent is represented using 8 bits with a bias of **127**.
* The actual exponent is **2**, so the biased exponent is: [ 2 + 127 = 129 ]
* **129** in binary is 10000001.

**Step 5: Encode the Mantissa**

* The mantissa (or fractional part) is the binary digits after the decimal point in the normalized number (1.0111), so the mantissa is 01110000000000000000000 (padded to 23 bits).

**Step 6: Combine the Sign, Exponent, and Mantissa**

* Sign bit: 0
* Exponent: 10000001
* Mantissa: 01110000000000000000000

The final **32-bit floating-point representation** of **5.75** is:

[ 0 10000001 01110000000000000000000 ]

**Interpreting the Floating-Point Number:**

If you decode this 32-bit value back to decimal, you would get **5.75**.

Summary of Floating-Point Representation:

**Advantages:**

* Can represent a wide range of values, from very small to very large.
* Essential for scientific calculations where high precision and large range are required.

**Disadvantages:**

* More complex to implement than fixed-point.
* Subject to **rounding errors** and precision limits because not all real numbers can be represented exactly in binary.

| **Feature** | **Fixed-Point Representation** | **Floating-Point Representation (IEEE 754)** |
| --- | --- | --- |
| **Precision** | Limited precision, determined by the number of bits. | High precision, with dynamic adjustment of the decimal point. |
| **Range** | Small range due to fixed integer and fractional bits. | Large range (can represent very small and very large numbers). |
| **Performance** | Fast and simple, especially in hardware implementations. | More complex and slower due to exponent and normalization. |
| **Use Case** | Embedded systems, simple applications. | Scientific computing, graphics, high-precision applications. |
| **Implementation** | Easier to implement in hardware. | More complex hardware needed, especially for normalization. |

**Conclusion:**

In summary, **fixed-point representation** is useful when simplicity and speed are prioritized, but it has limited range and precision. **Floating-point representation**, following the IEEE 754 standard, is much more versatile, capable of representing very large or very small numbers with high precision, though it is more complex and prone to rounding errors. Understanding both formats is essential for tasks like embedded systems development, scientific computing, and real-time applications.

CPU Organization: Fundamentals and Additional Features

1.1 CPU Basics:

The Central Processing Unit (CPU) is a key component in any computer system, responsible for executing instructions from programs and coordinating operations across the computer. The CPU consists of several key components:

Control Unit (CU):

Acts as the coordinator of the CPU's operations.

It manages the fetch-decode-execute cycle, where it:

Fetches an instruction from memory.

Decodes the instruction to understand what operation is to be performed.

Executes the operation by sending the necessary signals to other components (like the ALU or memory).

The CU interacts with various parts of the CPU and memory, ensuring data flows correctly between registers, ALU, and memory.

Types of Control Units:

Hardwired Control: Uses fixed logic circuits to control signals, making it fast but less flexible.

Microprogrammed Control: Uses a sequence of instructions (microcode) stored in memory to generate control signals. This allows for more complex instruction sets (common in CISC architectures).

Arithmetic Logic Unit (ALU):

The ALU performs all arithmetic and logical operations. It is capable of:

Arithmetic operations: Addition, subtraction, multiplication, division.

Logical operations: AND, OR, XOR, NOT, and comparisons like greater than, less than, equal to.

Components of the ALU:

Adder: A circuit that adds binary numbers.

Subtractor: For subtraction, though subtraction can often be performed via addition using two’s complement.

Shifter: Shifts bits left or right, often used in multiplication and division by powers of two.

Comparator: Compares two binary numbers and sets flags (zero flag, sign flag, etc.).

Registers:

Registers are small, fast storage elements within the CPU. Unlike memory (which is large but slow), registers allow for very quick access to frequently used data.

Types of Registers:

General-purpose Registers (GPRs): Used for temporary storage during program execution. For example, in the MIPS architecture, there are 32 registers (e.g., $t0, $t1, $s0, etc.).

Special-purpose Registers:

Program Counter (PC): Holds the address of the next instruction to execute. The control unit updates this after each instruction.

Instruction Register (IR): Holds the current instruction being executed.

Stack Pointer (SP): Points to the top of the stack, used in function calls and interrupts.

Status Register: Holds flags that indicate the result of the last operation (e.g., carry flag, zero flag, overflow flag).

1.2 CPU Performance Factors:

Understanding how the CPU’s performance is measured and optimized is essential for computer architecture.

Clock Speed:

The clock speed (measured in Hz) defines the number of cycles the CPU performs per second. Modern processors operate at speeds like 2-4 GHz (billion cycles per second).

A higher clock speed increases the number of instructions executed per second, but this is only one part of the performance equation. It doesn't directly translate to faster performance if the CPU is stalled waiting for data from memory.

Cycles Per Instruction (CPI):

Different instructions take different numbers of clock cycles to execute. Simple instructions like register-to-register additions may take one cycle, while complex instructions like memory access may take several cycles.

Formula to calculate CPU time:

CPU Time

=

Instruction Count

×

CPI

×

Clock Cycle Time

CPU Time=Instruction Count×CPI×Clock Cycle Time

Clock Cycle Time

=

1

Clock Speed (in Hz)

Clock Cycle Time=

Clock Speed (in Hz)

1

​

Reducing CPI or the instruction count (with more efficient instructions) leads to better performance.

Pipelining:

Pipelining allows overlapping of instructions by dividing the execution of an instruction into several stages (fetch, decode, execute, memory access, write-back). Each stage performs a part of the instruction execution in parallel with other stages. A typical 5-stage pipeline would be:

Fetch: Fetch the instruction from memory.

Decode: Decode the instruction to understand the operation and operands.

Execute: Perform the operation (e.g., addition in the ALU).

Memory: Access memory if needed (e.g., for load/store instructions).

Write-back: Write the result back to a register.

Pipeline Hazards: Pipelining increases throughput but can introduce hazards:

Data hazards: When one instruction depends on the result of a previous instruction that hasn’t finished.

Control hazards: Caused by branch instructions, where the next instruction may not be known until the branch decision is made.

Structural hazards: When hardware resources are insufficient to handle multiple instructions simultaneously (e.g., both fetching and writing at the same time).

Parallelism:

Instruction-Level Parallelism (ILP): The ability to execute multiple instructions simultaneously. This is achieved by techniques like superscalar architecture, where multiple execution units work in parallel, and out-of-order execution, where instructions are dynamically reordered to reduce stalls.

Thread-Level Parallelism (TLP): Refers to executing multiple threads in parallel on different cores or through Simultaneous Multithreading (SMT), such as Intel’s Hyper-Threading Technology, which allows multiple threads to run on the same core by sharing resources like registers and caches.

1.3 Additional CPU Features:

Cache Memory:

Caches are small, fast memory blocks located near the CPU, designed to store frequently accessed data to reduce the time spent accessing slower main memory.

Cache Hierarchy:

L1 Cache: Typically 32KB to 128KB, closest to the CPU core, very fast but small.

L2 Cache: Larger (256KB to 2MB) but slower than L1.

L3 Cache: Shared among cores in multi-core processors, can be 4MB to 64MB.

Cache Mapping:

Direct-mapped cache: Each memory block maps to one specific cache line.

Fully associative cache: Any memory block can be placed in any cache line.

Set associative cache: A compromise between direct-mapped and fully associative, where a block can be placed in any line within a set.

Branch Prediction:

Static Branch Prediction: A simple approach where branches are always predicted to go one way (e.g., always predict "taken").

Dynamic Branch Prediction: Uses historical information to predict the direction of branches. Modern CPUs use branch prediction buffers to store the outcomes of recently executed branches, improving accuracy.

Multithreading and Hyperthreading:

Multithreading: A CPU core can switch between multiple threads to keep its execution units busy. This is especially useful when one thread is stalled waiting for data from memory.

Hyperthreading (HT): Intel's proprietary SMT technology, where each physical core appears as two logical cores to the operating system, allowing it to execute two threads simultaneously by sharing resources like caches and execution units.

Multi-core Processors:

Multi-core CPUs contain multiple processing units (cores) on a single chip. Each core can execute its own thread or program, leading to greater parallelism and faster overall performance in multi-threaded applications.

Inter-core Communication: Cores share resources like L3 cache and memory but can communicate through a bus or mesh interconnect to coordinate task execution.

**1. Instruction Sets Overview**

An **Instruction Set Architecture (ISA)** defines the collection of machine-level instructions that a CPU can execute. Each instruction performs a specific operation, such as data transfer, arithmetic, logic operations, or control flow (branching).

The **ISA** provides a crucial interface between hardware (CPU) and software (programs). Different processors can have different instruction sets, but they all follow the same fundamental principles.

**Key Concepts:**

1. **Instruction Format:** The structure or layout of a machine instruction.
2. **Instruction Types:** The categories of operations the CPU can execute (e.g., arithmetic, logic, control).
3. **Programming Considerations:** The impact of different instruction sets on software design and performance.

Now, let's break down each of these in detail.

**2. Instruction Formats**

**Instruction Format** refers to how instructions are laid out in binary form, dictating how the CPU interprets them. Every instruction has several fields, and these fields vary based on the instruction set used by the processor.

**2.1 Components of an Instruction Format**

* **Opcode (Operation Code):** Specifies the operation to be performed (e.g., ADD, SUB, LOAD).
* **Operands:** The data or locations (registers, memory addresses) on which the operation is performed.
* **Addressing Modes:** Defines how the CPU identifies the operands (e.g., immediate, direct, indirect).
* **Instruction Length:** Instructions are typically 16-bit, 32-bit, or 64-bit depending on the architecture.

**2.2 Common Instruction Formats**

Different architectures like **MIPS**, **x86**, and **ARM** use different instruction formats, but let's discuss two general categories:

1. **R-Type (Register Type) Format:**

* Used for **arithmetic** and **logic** instructions that operate on registers.
* Operands are stored in registers (fast access).

Example: **MIPS R-Type Instruction Format** [ \text{opcode (6 bits)} \ | \ \text{rs (5 bits)} \ | \ \text{rt (5 bits)} \ | \ \text{rd (5 bits)} \ | \ \text{shamt (5 bits)} \ | \ \text{funct (6 bits)} ]

* **Opcode:** Specifies the operation.
* **rs, rt:** Source registers.
* **rd:** Destination register.
* **shamt:** Shift amount (used in shift operations).
* **funct:** Provides additional information to specify the exact operation (e.g., ADD, SUB).

**Example (MIPS)**:

add $t0, $t1, $t2 # Add the values in registers $t1 and $t2, store result in $t0

* **Opcode:** 000000 (ADD instruction)
* **rs:** $t1
* **rt:** $t2
* **rd:** $t0
* **shamt:** 00000 (no shift here)
* **funct:** 100000 (ADD function)

2. **I-Type (Immediate Type) Format:**

* Used for **data transfer** (e.g., load/store) and instructions that involve **immediate values** (constants).

Example: **MIPS I-Type Instruction Format** [ \text{opcode (6 bits)} \ | \ \text{rs (5 bits)} \ | \ \text{rt (5 bits)} \ | \ \text{immediate (16 bits)} ]

* **Immediate:** Represents a constant value or offset.

**Example (MIPS)**:

addi $t0, $t1, 5 # Add immediate value 5 to $t1, store result in $t0

* **Opcode:** 001000 (ADDI instruction)
* **rs:** $t1
* **rt:** $t0
* **Immediate:** 0000000000000101 (5)

3. **J-Type (Jump Type) Format:**

* Used for **control flow instructions** like jumps and procedure calls.

Example: **MIPS J-Type Instruction Format** [ \text{opcode (6 bits)} \ | \ \text{address (26 bits)} ]

* **Address:** 26-bit address for jumping to the destination.

**Example (MIPS)**:

j 10000 # Jump to instruction located at address 10000

* **Opcode:** 000010 (Jump instruction)
* **Address:** 10000

**3. Instruction Types**

Instructions are classified based on their function, and most architectures support a few fundamental types of instructions:

**3.1 Data Transfer Instructions**

These instructions move data between memory and registers or between registers themselves. They don’t perform any computation but are essential for accessing and storing data.

* **Load (LD, LW):** Load data from memory into a register.
* Example (MIPS):
* lw $t0, 0($t1) # Load word from memory address stored in $t1, store it in $t0
* **Store (ST, SW):** Store data from a register into memory.
* Example (MIPS):
* sw $t0, 4($t1) # Store the value in $t0 into memory at the address $t1 + 4
* **Move (MOV):** Transfer data from one register to another.
* Example (ARM): assembly MOV R0, R1 # Move the contents of R1 into R0

**3.2 Arithmetic and Logic Instructions**

These instructions perform arithmetic (add, subtract, multiply, divide) and logical (AND, OR, NOT, XOR) operations on data stored in registers or memory.

* **Addition (ADD):** Adds two registers and stores the result in a destination register.
* Example (MIPS):
* add $t0, $t1, $t2 # Add $t1 and $t2, store result in $t0
* **Subtraction (SUB):** Subtracts the second register from the first and stores the result.
* Example (MIPS):
* sub $t0, $t1, $t2 # Subtract $t2 from $t1, store result in $t0
* **Logical AND (AND):** Performs bitwise AND between two registers.
* Example (MIPS):
* and $t0, $t1, $t2 # Bitwise AND of $t1 and $t2, store result in $t0
* **Logical OR (OR):** Performs bitwise OR between two registers.
* Example (MIPS): mips or $t0, $t1, $t2 # Bitwise OR of $t1 and $t2, store result in $t0

**3.3 Control Flow Instructions**

These instructions modify the flow of execution of a program, allowing for loops, conditionals, and function calls.

* **Jump (J):** Unconditionally jumps to a specified address.
* Example (MIPS):
* j target\_label # Jump to the instruction at 'target\_label'
* **Branch if Equal (BEQ):** Branches to a specified address if two registers are equal.
* Example (MIPS):
* beq $t0, $t1, target\_label # Branch to 'target\_label' if $t0 equals $t1
* **Branch if Not Equal (BNE):** Branches to a specified address if two registers are not equal.
* Example (MIPS):
* bne $t0, $t1, target\_label # Branch to 'target\_label' if $t0 is not equal to $t1
* **Function Call (CALL):** Jumps to a subroutine and saves the return address for when the function finishes.
* Example (x86):
* CALL myFunction # Call subroutine 'myFunction'
* **Return (RET):** Returns control to the calling procedure after a function call.
* Example (x86): assembly RET # Return from subroutine to the caller

**3.4 Shift and Rotate Instructions**

These instructions shift or rotate the bits in a register to the left or right.

* **Shift Left Logical (SLL):** Shifts the bits of a register left, filling the empty positions with 0s.
* Example (MIPS):
* sll $t0, $t1, 2 # Shift $t1 left by 2 bits, store result in $t0
* **Shift Right Logical (SRL):** Shifts the bits of a register right, filling the empty positions with 0s.
* Example (MIPS):
* srl $t0, $t1, 2 # Shift $t1 right by 2 bits, store result in $t0
* **Rotate Left (ROL):** Rotates the bits of a register to the

left.

* Example (ARM): assembly ROL R0, #1 # Rotate the bits of R0 left by 1 bit

**4. Programming Considerations**

When writing assembly or machine code, there are several important considerations based on the instruction set used by the CPU.

**4.1 Instruction Length and Complexity**

* **Fixed vs. Variable Length:**
* **RISC (e.g., MIPS, ARM)**: All instructions are typically of **fixed length** (e.g., 32 bits). This makes instruction decoding simpler and faster.
* **CISC (e.g., x86)**: Instructions are of **variable length**, which allows for more complex operations but makes decoding more difficult.

**4.2 Memory Access and Register Use**

* **Load/Store Architecture (RISC):**
* In architectures like **MIPS**, arithmetic and logical operations can only be performed on registers. Memory is accessed through **load/store instructions**.
* This architecture is **fast** because operations on registers are quicker than memory access.
* **Memory-to-Memory Operations (CISC):**
* In **CISC** architectures, operations can be performed directly on memory without loading the data into registers first.
* While this might reduce the number of instructions, it can slow down execution since memory access is slower than register access.

**4.3 Pipelining and Parallelism**

* **RISC architectures** are typically optimized for **pipelining**, meaning multiple instructions can be overlapped in execution to improve performance. Each instruction takes exactly one clock cycle.
* **CISC architectures**, with their more complex and variable-length instructions, are generally **harder to pipeline**.

**4.4 Code Size**

* **RISC (Reduced Instruction Set Computing):**
* Typically results in **larger code size** because it uses simple instructions, meaning more instructions may be required to perform complex tasks.
* **CISC (Complex Instruction Set Computing):**
* **Smaller code size** since fewer instructions can accomplish more complex operations.

**Summary**

* **Instruction Formats** define the structure of machine-level instructions and typically consist of fields like the opcode, operands, and addressing mode.
* **Instruction Types** fall into categories such as data transfer, arithmetic/logic, control flow, and bit manipulation (shift/rotate). These instructions are the building blocks of programs, controlling how the CPU operates on data.
* **Programming Considerations** involve understanding how the architecture (RISC vs. CISC) influences factors like instruction length, memory access, pipelining, and code size. Each architecture has trade-offs that affect performance and software design.

Understanding the instruction set of the CPU you're working with is crucial for optimizing both hardware and software performance.

**Unit - 2**