Unit - 1

RISC vs CISC Architecture

RISC

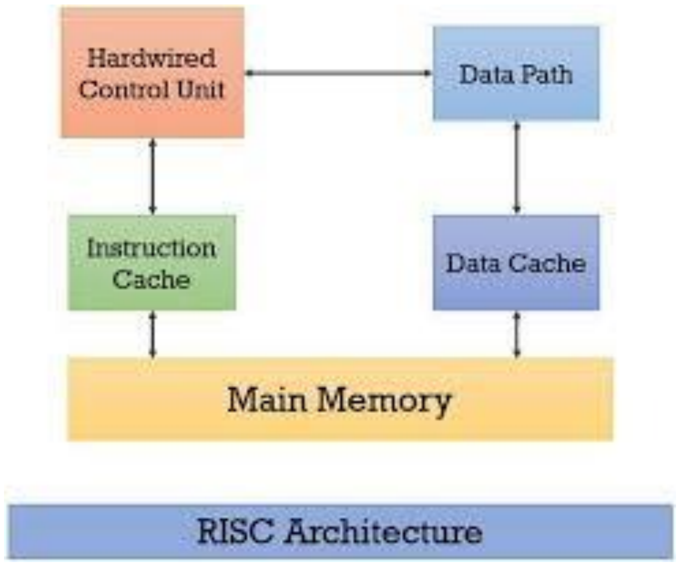
Reduced Instruction Set Computer architecture uses a smaller, more straightforward set of instructions to optimize performance. RISC processors are known for being efficient, scalable, and cost-effective. They are commonly used in modern microprocessors, embedded systems, and high-performance computing environments.

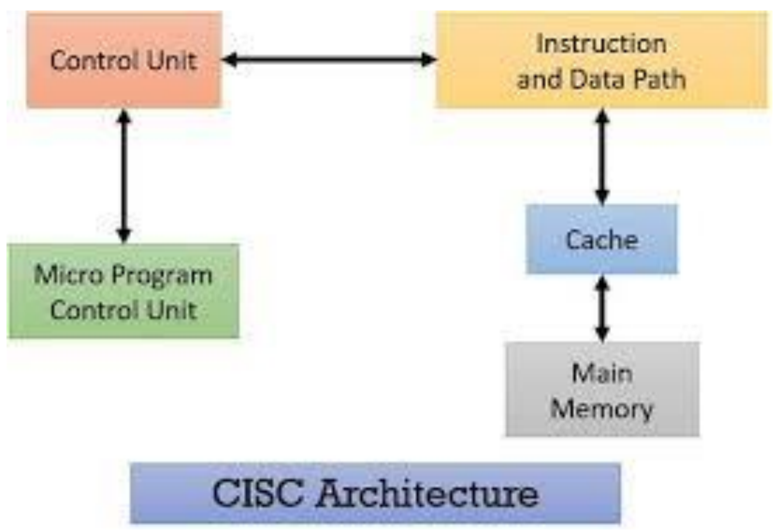
CISC

Complex Instruction Set Computer architecture uses a large set of complex instructions that can perform multiple operations in a single instruction. CISC processors are slower and less efficient than RISC processors because they use more memory references and have limited access to registers. Some examples of CISC processors include AMD, Intel x86, VAX, and System/360.

Basis	RISC (Reduced Instruction Set Computing)	CISC (Complex Instruction Set Computing)
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.

Basis	RISC (Reduced Instruction Set Computing)	CISC (Complex Instruction Set Computing)
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 branching and decisions.
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.
Example Architectures	Examples include **MIPS**, **ARM**, **and PowerPC**.	Examples include **x86 and Intel's IA-32 architecture**.





Data Representation

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.

1.2 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:

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 * 2 = 1.5 → Take the integer part 1.
  - 0.5 * 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.

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)

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"** i.e. 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**.

- **Single Precision:** 32 bits (1 sign bit, 8 bits for exponent, 23 bits for mantissa).
- **Double Precision:** 64 bits (1 sign bit, 11 bits for exponent, 52 bits for mantissa).

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):
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: $1.0111 * 2^2$
- 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

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.

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

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).

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.

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:

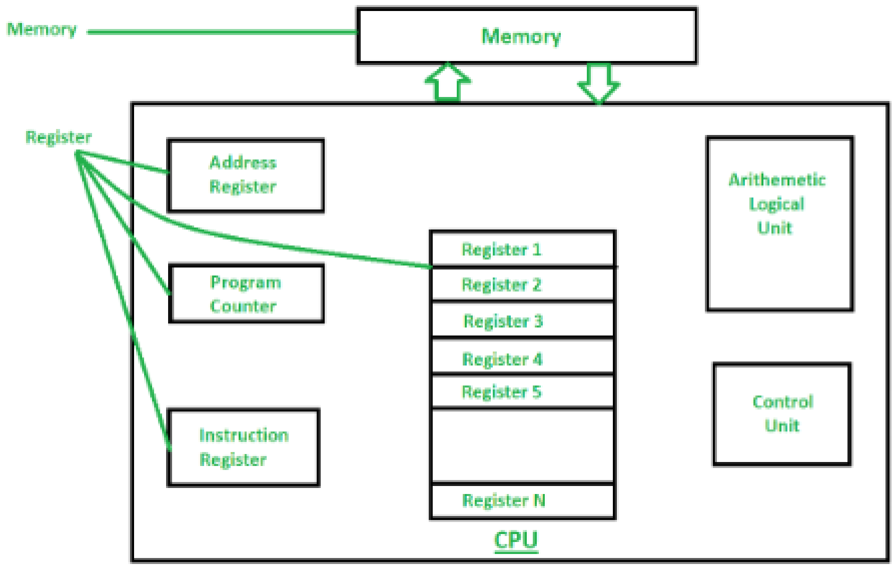
- **Adder**: A circuit that adds binary numbers.
- **Subtractor**: For subtraction, though subtraction can often be performed via addition using two's complement.
- **Shifter**: Shifts bits left or right, often used in multiplication and division by powers of two.
- **Comparator**: Compares two binary numbers and sets flags (zero flag, sign flag,etc.).

Registers:

A register is a small and temporary storage unit inside a computer's CPU. It plays a vital role in holding the data required by the CPU for immediate processing and is made up of **flip-flops**.

It usually holds a limited amount of data ranging from **8 to 64 bits**, depending on the processor architecture.

Registers act as intermediate storage for data during arithmetic logic and other processing operations.



Types of Registers:

- **Program Counter (PC):** Program Counter (PC) is used to keep the track of the execution of the program. PC points to the address of the next instruction to be fetched from the main memory when the previous instruction has been successfully completed. Program Counter (PC) **also** functions to **count** the number of instructions. In a 32-bit architecture, the PC gets incremented by 4 every time to fetch the next instruction.
- **Instruction Register (IR):** The IR holds the instruction which is just about to be executed. The instruction from the PC is fetched and stored in IR. As soon as the

instruction is placed in IR, the CPU starts executing the instruction, and the PC points to the next instruction to be executed.

- **Stack Pointer (SP):** The stack pointer is used in **stack-based memory** operations. It points to the top of the stack, which is a region of memory used for temporary storage of data and return addresses during function calls.
- **Flag Register:** A flag register(**status register/condition code register**) is used to indicate the status of the CPU or the outcome of various operations such as **Zero Flag, Carry flag, Sign Flag, Overflow Flag, Parity Flag, Auxiliary Carry Flag**,etc.
- **Accumulator**: An accumulator is a register that temporarily stores the results of intermediate arithmetic and logical operations. It functions as a **temporary storage location**. It is a general-purpose Register.
- **Memory Address Registers (MAR)**: It holds the address of the location to be accessed from memory. MAR and MDR together facilitate the communication of the CPU and the main memory.
- **Memory Data Registers (MDR)**: It contains data to be written into or to be read out from the addressed location.
- **Floating-Point Registers**: The Floating-point registers are specialized for handling floating-point numbers and performing floating-point arithmetic operations. These registers can store and manipulate floating-point numbers with **higher precision**.
- **Input Register (INPR):** Input Register is a register that stores the data from an input device. The computer's alphanumeric code determines the size of the input register.
- **Output Register (OUTR):** The Output Register stores the data that needs to be sent to an output device. Its size is determined by the alphanumeric code used by the computer.

Purpose of Registers

- **Storing Instruction**: Registers are used to store the instruction from programs before the CPU follows them. This helps the computer quickly find and follow the steps it needs to take.
- **Holding Answer**: When the computer does math calculations or other tasks, the register stores the temporary answer.
- **Quick Access to Important Stuff**: Registers are like the computer's quick-access shelves. They keep important things nearby, so the computer can grab them fast without going far away to get them.

1.2 CPU Performance Factors:

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 number 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
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**: Access memory if needed (e.g., for load/store instructions).
- **Write-back**: Write the result back to a register.

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.

Instruction Sets

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:-**

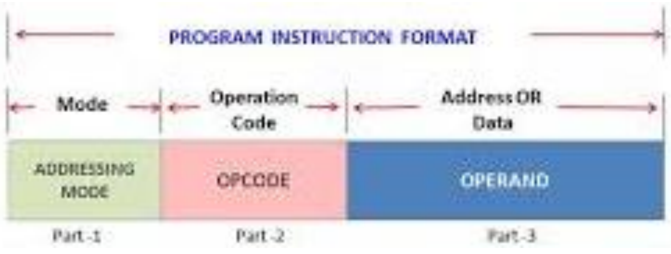
1. **Instruction Format:** The **structure** or **layout** of a machine instruction.
2. **Instruction Types:** The **categories of operations** the CPU can execute.
3. **Programming Considerations:** The impact of different instruction sets on software design and performance.

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.

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.



Instruction Types

1. **Data Transfer Instructions**: These instructions move data between memory and registers or between registers themselves. They are essential for accessing data stored in memory and for moving it into the processing unit for further computation. They **don't perform any computations** themselves.
   - **Load**: Transfers data from **memory to a register**.
   - **Store**: Transfers data from a **register back into memory**.
   - **Move**: Transfers data **between registers**.

2. **Arithmetic and Logic Instructions**: These instructions perform mathematical and logical operations. Arithmetic operations include addition, subtraction, multiplication, and division, while logical operations involve comparing or manipulating bits in registers.

- **Arithmetic Operations**:
  - **Addition**: Adds two data values and stores the result.
  - **Subtraction**: Subtracts one data value from another.
- **Logical Operations**:
  - **AND**: Compares two bits and sets the result to true if both bits are true.
  - **OR**: Compares two bits and sets the result to true if at least one bit is true.
  - **NOT**: Inverts the bits (changes 0 to 1 and vice versa).

3. **Control Flow Instructions**: These instructions manage the sequence in which instructions are executed, controlling the program's flow. They help implement loops, conditionals, and function calls.

- **Jump**: Directs the program to move to a different part of the code without any condition.
- **Branching**: Directs the program to another part of the code if a certain condition is met (e.g., if two values are equal).
- **Function Call**: Transfers control to a subroutine and saves the return address.
- **Return**: Transfers control back to the calling program after completing a function.

4. **Shift and Rotate Instructions**: These instructions manipulate the bits within a register by shifting or rotating them left or right. **Shifting** moves bits and fills the empty positions with zeros, while **Rotating** moves bits around in a circular fashion.

- **Shift Left (SHL)**: Moves the bits to the left and fills the vacant bits with zeros.
- **Shift Right (SHR)**: Moves the bits to the right and fills the vacant bits with zeros.
- **Rotate Left (ROL)**: Moves bits to the left, but the bits that are shifted out from the left end are placed at the right end (circular motion).
- **Rotate Right (ROR)**: Moves bits to the right, but the bits that are shifted out from the right end are placed at the left end (circular motion).

5. **Comparison Instructions**: These instructions compare the values stored in two registers or between a register and a memory location. They **do not store the result of the comparison directly but set certain flags in the CPU's status register** (such as zero, carry, sign, or overflow flags) that can be used in conjunction with control flow instructions.
   Comparison instructions are typically used with **branching** or **conditional** instructions to make decisions in a program (e.g., jump to another part of the code if two values are equal).

Key Functions:

- **Compare (CMP)**: Compares two operands and sets flags accordingly (e.g., zero flag if they are equal).
- **Test (TEST)**: Performs a bitwise AND on two operands but only sets flags based on the result without changing the values of the operands.

Examples of Flags Set by Comparison:

- **Zero Flag**: Set if the result of the comparison is zero (i.e., the operands are equal).
- **Carry Flag**: Set if there is a borrow/overflow in the comparison (for unsigned operations).
- **Sign Flag**: Set if the result of the comparison is negative.
- **Overflow Flag**: Set if there is an arithmetic overflow in signed operations.

6. **Stack Instructions:** The **stack** is a specialized data structure used to store data temporarily, often used for function calls, local variables, and return addresses. The stack operates on a **LIFO** (Last In, First Out) principle, meaning the last element added is the first one to be removed.
   Stack instructions are heavily used in **function calls**, **recursive algorithms**, and **managing local variables** in a program.

Key Operations:

- **PUSH**: Places a value onto the top of the stack.
- **POP**: Removes the value from the top of the stack.
- **CALL**: Pushes the return address onto the stack and jumps to a subroutine. When a function or subroutine is called, the **CALL** instruction pushes the current program counter (address of the next instruction) onto the stack.
- **RET (Return)**: Pops the return address from the stack and transfers control back to that address. The **RET** instruction retrieves this address, allowing the program to resume execution after the function finishes.

7. **Floating-Point Instructions:** These instructions perform arithmetic and logic operations on **floating-point** numbers, which are numbers that can have decimal points. Unlike integer operations, floating-point operations require special handling due to the way floating-point numbers are represented in memory (according to standards like IEEE 754).
   Floating-point instructions are widely used in scientific computations, graphics processing, machine learning, and any application where precise numerical calculations are essential.

Key Operations:

- **Floating-Point Addition (FADD)**: Adds two floating-point numbers.
- **Floating-Point Subtraction (FSUB)**: Subtracts one floating-point number from another.
- **Floating-Point Multiplication (FMUL)**: Multiplies two floating-point numbers.
- **Floating-Point Division (FDIV)**: Divides one floating-point number by another.

Programming Considerations

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

1. **Instruction Length and Complexity**
   - **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.
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.

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. **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.

Unit - 2

Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) is a core component of the central processing unit (CPU) in a computer. It performs both **arithmetic** (addition, subtraction) and **logical** (AND, OR, NOT, XOR) operations. The ALU is critical to the functioning of processors, as it executes the mathematical and logical tasks that form the foundation of all computational processes.

Key Functions of an ALU:

1. **Arithmetic Operations**:
   - **Addition**: Adds two binary numbers or values.
   - **Subtraction**: Subtracts one binary value from another.
   - **Multiplication and Division (in some ALUs):** More complex ALUs may handle these operations, though simpler ALUs leave these tasks to specialized units or use repetitive processes.
2. **Logical Operations**:
   - **AND, OR, XOR, NOT**: Perform bitwise logical comparisons between binary inputs.
3. **Shift Operations**:
   - Shifts the binary data left or right, often used in multiplication or division by powers of two.
4. **Comparison Operations**:
   - **Equality (==)**: Checks whether two values are equal.
   - **Less than (<) / Greater than (>)**: Determines relational comparison results between inputs.

5. **Bitwise Operations**:
    - The ALU can manipulate individual bits of a binary number (e.g., AND-ing specific bits or flipping them with a NOT operation).
6. **Increment and Decrement**:
    - Incrementing or decrementing a value by one.

<u>**Components of an ALU**</u>:

1. **Input and Output Registers**:

- ALUs receive inputs from registers, temporary storage locations inside the CPU.
- Output registers store the result of the operation performed by the ALU before passing it to other parts of the processor.

2. **Control Unit Interaction**: The control unit sends control signals to the ALU to specify whether to perform an arithmetic or logical operation.

3. **Flags and Status Registers**:

- **Flags** indicate the result of the operation (e.g., overflow, zero, carry bit).
- **Status registers** store these flags to help the CPU make decisions (e.g., conditional jumps based on comparison results).

<u>**Types of ALUs**</u>:

1. **Combinational ALU**: **Performs all operations in a single clock cycle** without storing intermediate results.

2. **Sequential ALU**: **Breaks operations into multiple steps**, processing them over several clock cycles.

3. **Fixed-Point vs Floating-Point ALU**:

- <u>**Fixed-Point ALU**</u>: Handles **standard integer operations**.
- <u>**Floating-Point ALU**</u>: Specialized ALUs for handling **real numbers** and **decimals**, useful in scientific computations and graphics. Operates on the **IEEE 754** standard for floating-point numbers.

COMBINATIONAL ALU VS SEQUENTIAL ALU

FEATURE	COMBINATIONAL ALU	SEQUENTIAL ALU
Definition	Performs operations in **a single clock cycle** without memory or storage. The output depends only on the current input.	Performs operations **over multiple clock cycles**. It uses registers to store intermediate results.
Clock Cycle Dependency	**Not-dependent on clock cycles.** All operations are completed in one cycle.	**Requires multiple clock cycles** for operations. Execution is step-by-step.
Operation Speed	**Faster**, as it performs all operations instantly in one clock cycle.	**Slower**, as it breaks operations into stages across multiple cycles.

FEATURE	COMBINATIONAL ALU	SEQUENTIAL ALU
Design Complexity	**Less** complex design, as the control units do not need to manage operations across cycles.	**More** complex design due to the need for control units to manage operations across cycles.
Use of Control Signals	**Minimal control signals** needed, as there is no state to track. Inputs directly determine the output.	Requires **extensive control signals** to sequence operations correctly and control registers.
Area and Resource Usage	**Uses more hardware resources** because all functional units are active at the same time, resulting in larger circuit size.	**Requires fewer hardware resources**, as operations are spread across time, leading to reduced area and resource usage.
Power Consumption	**Higher** power consumption since more components are active simultaneously.	**Lower** power consumption due to spreading the operations over time and using fewer active components at once.
Execution of Complex Operations	**Limited flexibility in handling very complex operations** within one clock cycle.	Can handle **more complex operations efficiently** by breaking them into simpler steps over multiple cycles.
Common Applications	Found in fast processors where **speed is prioritized**, such as in high-performance computing or data processing.	Used in low-power devices where **saving energy and minimizing area are critical**, like embedded systems or simple controllers.
Intermediate Results Storage	**No intermediate result storage**; everything is computed in one go.	Uses registers or memory to **store intermediate results between steps**.
Examples of Operations	**Basic arithmetic** (addition, subtraction) and **logic operations** (AND, OR) that can be performed quickly.	**More complex tasks** (like division, multiplication) that require several steps to complete.
Control Complexity	**Simpler control** as the ALU handles one operation at a time.	**Complex control** as control unit track the stages of operation and handle sequencing.

<u>Data Path Design</u>: **Data path design** refers to the architecture within a processor that facilitates the flow of data between components, such as registers, arithmetic logic units (ALUs), and memory. The **data path** is responsible for carrying out the core operations of a processor, including arithmetic, logical operations, and data movement between various storage elements.

<u>Components of Data Path Design:</u>

- **Buses:** Buses are communication pathways that transfer data between different components of the data path (e.g., registers, memory, and the ALU).

- **Registers:** Small, fast storage locations within the CPU that hold data temporarily during execution. Registers are used for holding operands for arithmetic/logical operations and storing intermediate results.
- **Arithmetic Logic Unit (ALU):** The ALU is the core component of the data path that performs arithmetic (addition, subtraction) and logical (AND, OR) operations.
- **Multiplexers (MUX):** Multiplexers are used to **select one input from multiple data sources.** They direct the appropriate input to the ALU or other processing units.
- **Control Unit:** The control unit generates **control signals** that dictate the behavior of various components within the data path. It controls the flow of data and the timing of operations.
- **Memory Elements:** Data paths are connected to memory units such as RAM where data is stored and retrieved for processing.

<u>**Data Path Design for Fixed-Point Arithmetic**</u>

Fixed-point arithmetic refers to numeric representation where numbers are stored with a fixed number of digits before and after the decimal point. This is used for operations that involve integers or numbers with a constant fractional part.

<u>**1. Addition and Subtraction in Fixed-Point Arithmetic:**</u>

- **Addition:**
- Fixed-point addition is straightforward, similar to standard binary addition. It involves adding two binary numbers, with carry propagation from one bit position to the next.
- Overflow can occur if the sum exceeds the fixed number of bits available to represent the result.
- **Subtraction:**
- Fixed-point subtraction is achieved by adding the two's complement of the number to be subtracted.
- Just like addition, subtraction may result in an overflow if the result exceeds the range that can be represented with the given number of bits.

Data Path for Addition/Subtraction:

- Two operands are retrieved from registers.
- They are fed into the ALU, which is configured for either addition or subtraction.
- The result is stored back into a destination register.
- Overflow detection logic is included to signal if the result is outside the representable range.

<u>**2. Multiplication in Fixed-Point Arithmetic:**</u>

- **Multiplication** in fixed-point arithmetic involves multiplying two binary numbers. The result of multiplying two n-bit numbers results in a 2n-bit product.
- Typically, the data path for multiplication consists of a **multiplier** unit that performs multiple additions based on the partial products.

Booth's Algorithm:

- A common technique used for multiplying binary numbers is **Booth's Algorithm**, which reduces the number of required additions by encoding the multiplier and performing fewer iterations.

Data Path for Multiplication:

- The operands are retrieved from registers and sent to the multiplier.
- The multiplication unit generates partial products and accumulates them over multiple cycles.
- The final product is stored in a destination register or in memory.

3. Division in Fixed-Point Arithmetic:

- **Division** in fixed-point arithmetic is more complex than multiplication, as it involves successive subtraction or a form of repeated estimation.
- The **restoring and non-restoring division algorithms** are common techniques used in binary division.

Restoring Division: This method involves comparing the dividend with the divisor, subtracting the divisor if the dividend is larger, and repeating the process until the result is obtained.

Non-Restoring Division: An optimized version of division that reduces the number of steps required by skipping unnecessary restore operations.

Data Path for Division:

- The dividend and divisor are retrieved from registers.
- The division algorithm is performed in multiple steps, generating a quotient and possibly a remainder.
- The final quotient is stored in the register.

BOOTH'S ALGORITHM

Booth's Algorithm is an efficient algorithm used for **multiplying two signed binary numbers in two's complement form**. It **reduces the number of required addition and subtraction operations** compared to traditional multiplication methods, making it useful in hardware implementations of multiplication.

Key Concept:

Booth's algorithm works by encoding a sequence of bits from the multiplier to reduce the number of additions/subtractions. It observes the transitions between bits (from 0 to 1 or from 1 to 0) to determine if addition, subtraction, or shifting should be performed. This reduces redundant operations and speeds up multiplication.

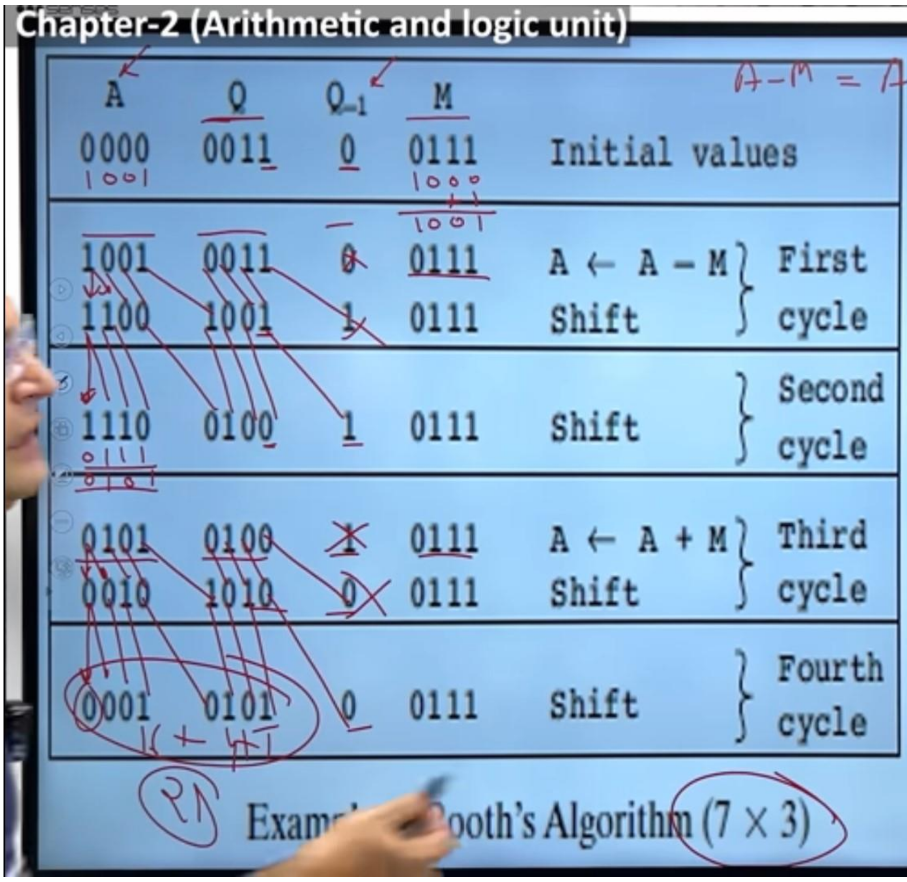
Advantages of Booth's Algorithm:

- **Efficiency**: Reduces the number of arithmetic operations (additions or subtractions) required for multiplication by encoding the multiplier more efficiently.
- **Handling Negative Numbers**: It automatically handles both positive and negative multiplications using two's complement representation.

- **Optimization for Runs of Zeros or Ones**: The algorithm skips redundant operations when there are sequences of 1s or 0s in the multiplier.

Disadvantages of Booth's Algorithm:

- **Complexity in Modern Systems**: Booth's algorithm can be more complex to implement in modern systems compared to other optimized algorithms.
- **Variable Execution Time**: The performance of Booth's algorithm can vary depending on the pattern of bits in the multiplier.

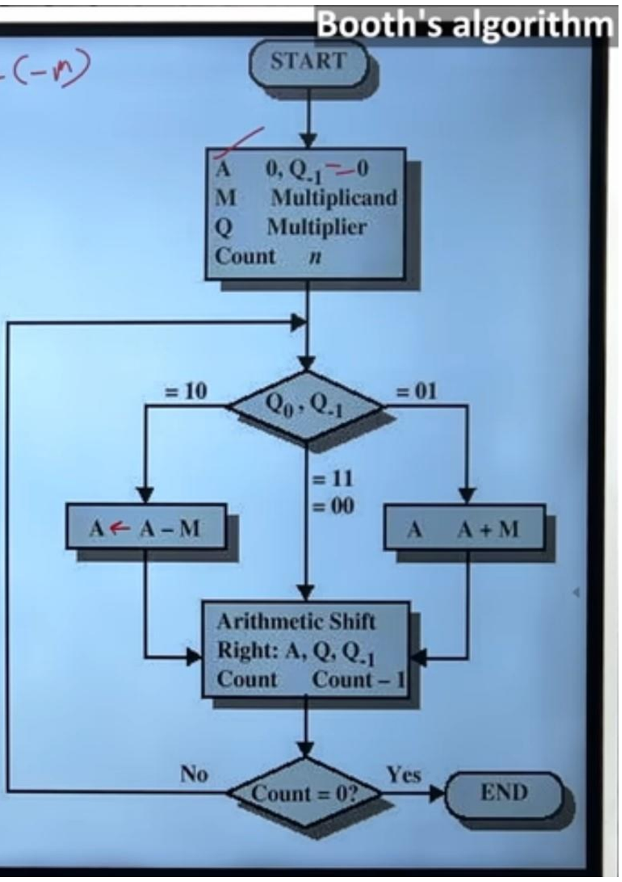


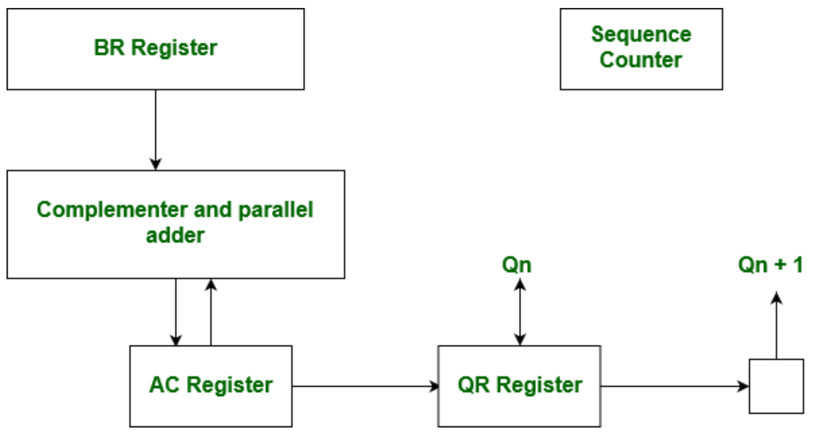
Chapter-2 (Arithmetic and logic unit)

$A-M = A+(-M)$

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First
1100	1001	1	0111	Shift	cycle
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third
0010	1010	0	0111	Shift	cycle
0001	0101	0	0111	Shift	Fourth cycle

Exam ooth's Algorithm (7 × 3)





Floating-Point Arithmetic: Floating-point arithmetic is used to represent real numbers that can have a wide range of values, including very large or very small numbers. Floating-point numbers are represented in a format similar to scientific notation, where a number is expressed as a *sign*, *mantissa* (significant digits), and *exponent*.

IEEE 754 Standard: The most widely used standard for representing floating-point numbers in modern processors.

Single Precision: 32 bits (1 sign bit, 8 bits for exponent, 23 bits for mantissa).

Double Precision: 64 bits (1 sign bit, 11 bits for exponent, 52 bits for mantissa).

Floating-Point Addition/Subtraction:

Alignment of exponents is crucial in floating-point arithmetic. To add or subtract two floating-point numbers, their **exponents must first be aligned**, and **then the mantissas can be added or subtracted**.

Normalization: After the operation, the result is normalized to fit within the available range for the exponent.

Floating-Point Multiplication/Division:

For **multiplication**, the mantissas are multiplied, and the exponents are added.

For **division**, the mantissas are divided, and the exponents are subtracted.

Data Path for Floating-Point Arithmetic:

The **floating-point unit (FPU)** is **responsible for handling floating-point arithmetic**.

The FPU typically includes **specialized circuits** for **normalizing**, **rounding**, and **aligning exponents**.

Operations are more complex than fixed-point and require additional control logic for handling special cases like overflow, underflow, and NaN (Not a Number).

Pipeline Processing:

Pipeline Processing is a technique used in computer architecture to **improve instruction throughput by overlapping the execution of multiple instructions**. It involves breaking down the process of executing an instruction into multiple stages, where each stage handles a part of the instruction. By doing this, **a new instruction can begin processing before the previous one has completed**, leading to **more efficient use of CPU resources** and **increased overall performance**.

Stages of a Pipeline:

A typical pipeline divides instruction execution into multiple stages, such as:

- **Fetch:** Retrieve the instruction from memory.
- **Decode:** Interpret/Understand the instruction and prepare for execution.
- **Execute:** Perform the actual operation (using the ALU for arithmetic/logic operations).
- **Memory Access:** Read or write data from/to memory if required.
- **Write Back:** Store the result in the destination register.

- **Instruction-Level Parallelism:** By overlapping the execution of multiple instructions, pipelines enable **instruction-level parallelism (ILP)** which improves performance by maximizing resource utilization.
- **Hazards in Pipelining:**

  Hazards in Pipelining refer to **situations that cause delays or incorrect execution of instructions in a pipelined CPU**. These hazards arise when multiple instructions are in different stages of execution, and there is some form of conflict, leading to inefficiencies or errors in instruction processing.

Types of Hazards in Pipelining:

1. **Data Hazards:**

- Data hazards occur **when instructions in a pipeline depend on the data generated by previous instructions, but that data is not yet available**.
- **For example:** if one instruction is still in progress and a subsequent instruction needs its result, the pipeline may stall until the first instruction completes.
- Data hazards are typically classified into three types:
    - ***RAW (Read After Write)*:** A later instruction tries to read data before a previous instruction writes the required data.
    - ***WAR (Write After Read)*:** An instruction writes a value before a previous instruction reads it.
    - ***WAW (Write After Write)*:** Two instructions write to the same location, and the order of writes matters for correctness.

2. **Control Hazards:**

- Control hazards occur **when the pipeline needs to make a decision based on the outcome of a conditional instruction (such as a branch or jump)**.
- The delay in determining the correct path for program execution causes stalls because the pipeline may have to discard partially executed instructions.
- For instance, in branch instructions, the pipeline may have already fetched the next instruction before determining whether the branch is taken or not, leading to incorrect instruction execution.

3. **Structural Hazards:**

- Structural hazards arise **when hardware resources (such as memory, registers, or functional units) are insufficient to support all the instructions in the pipeline**.
- If two or more instructions need access to the same hardware resource at the same time, a conflict occurs, leading to **delays or the stalling** of instructions.
- These hazards can be caused by insufficient hardware units (like having a single memory or arithmetic unit), making it impossible for multiple instructions to execute concurrently.

Handling Hazards:

Handling Hazards is essential in ensuring the smooth and efficient operation of pipelined processors. There are various techniques employed to handle or mitigate the effects of data, control, and structural hazards, enabling the pipeline to function effectively even when conflicts arise.

Techniques for Handling Hazards:

Forwarding (Data Forwarding):

- Forwarding, also called **bypassing**, is a technique used to **resolve data hazards** by directly sending results from one pipeline stage to another without waiting for the result to be written back to the register file.

- It prevents pipeline stalls by allowing dependent instructions to use the result of a previous instruction immediately after it's computed, rather than waiting for the register write-back.
- For example, the result of an arithmetic operation can be forwarded from the execution stage to the input of the next instruction if needed.
- Although forwarding adds complexity with additional hardware like multiplexers and control logic, it significantly improves pipeline performance by reducing delays.

Branch Prediction:

- Branch prediction **addresses control hazards** that occur when the next instruction depends on the outcome of a branch or conditional jump.
- Instead of waiting for the branch to resolve, branch prediction **guesses the outcome and speculatively fetches and executes instructions along the predicted path**.
- If the **prediction is correct**, execution continues without interruption. If **incorrect**, the pipeline is **flushed**, and the correct instructions are fetched.
- Advanced branch prediction algorithms dynamically learn from **past patterns** to improve accuracy, reducing performance penalties from frequent branches and ensuring smoother instruction execution.

Pipeline Stalling (Inserting NOPs):

- **Stalling** is a method used when no other techniques can resolve a hazard immediately. In this approach, the **pipeline temporarily halts the progression of instructions to allow the hazard to be resolved**.
- **For example:** If a data hazard occurs because a required value is not yet available, the pipeline inserts a no-operation (NOP) instruction into the pipeline stages, effectively delaying the dependent instruction until the needed data becomes available.
- Stalling prevents incorrect results but also reduces pipeline efficiency, as the pipeline is **underutilized** during the stall period.

Unit – 3

Control Design

Control Design refers to the mechanisms responsible for directing the operations of the processor. It involves **generating the control signals** needed to coordinate the activities of various components (such as the ALU, registers, and memory) in order to execute instructions efficiently.

Control design is crucial in ensuring that the CPU operates correctly by issuing control signals that dictate data movement, arithmetic operations, and overall coordination between components.

1. Hardwired Control: **Hardwired Control** refers to a control design approach where control signals are generated through fixed logic circuits. These circuits use combinational and sequential logic to directly produce the signals needed to control the CPU.

Key Characteristics:

- **Fast Execution:** Since control signals are generated using simple logic gates and flip-flops, hardwired control units can produce control signals very quickly.
- **Fixed Control Logic:** The control unit is designed with fixed logic, meaning it cannot be easily modified without changing the underlying circuitry.
- **Difficult to Modify:** Any change to the control signals requires redesigning the logic circuit, making it less flexible compared to microprogrammed control.

Working of Hardwired Control:

- The control unit decodes the instruction and generates the appropriate signals through a combinational logic circuit.
- The logic is predefined for each instruction, and signals are sent directly to components like the ALU, registers, and memory.
- **Sequential circuits (e.g. counters or finite state machines)** control the timing and sequence of operations.

Advantages:

- **High Speed:** Hardwired control is faster than microprogrammed control due to its direct and minimalistic design.
- **Efficient for Simple CPUs:** Hardwired control works best for simple instruction sets, where the control logic doesn't need to be overly complex.

Disadvantages:

- **Difficult to Modify or Upgrade:** Since the control logic is fixed, changes or upgrades to the instruction set or control signals require a redesign of the entire control unit.
- **Complexity for Large Instruction Sets:** As the instruction set grows, the control logic becomes more complicated, making it harder to manage and design.

2. Microprogrammed Control: **Microprogrammed Control** is a control design technique where control signals are generated by executing a sequence of microinstructions stored in a control memory (called a **control store**). These microinstructions define the control signals for each step of the instruction execution.

Key Characteristics:

- **Control Store:** Contains microinstructions that dictate the control signals for every machine instruction. Each machine instruction maps to a sequence of microinstructions that control the processor's operations.
- **Flexible Design:** Microprogrammed control allows for easier modification and expansion of the instruction set by updating the microinstructions stored in memory.
- **Simpler Design for Complex Instructions:** For processors with large and complex instruction sets, microprogramming simplifies control design by breaking down complex instructions into smaller microinstructions.

Working of Microprogrammed Control:

- Each machine-level instruction corresponds to a **microprogram** (a series of microinstructions).
- The **Control Unit** reads the appropriate microinstructions from the control store.

- The microinstructions generate the control signals that direct data movement, ALU operations, and memory access.
- A **microprogram counter** keeps track of which microinstruction to execute next.

Advantages:

- **Ease of Modification:** Changing the control signals or adding new instructions is relatively easy by updating the microinstructions in the control memory.
- **Simplicity in Complex CPUs:** For complex CPUs with many instructions, microprogrammed control simplifies the design by making control generation programmable.

Disadvantages:

- **Slower Execution:** Microprogrammed control is generally slower than hardwired control because fetching microinstructions adds extra overhead to the instruction execution process.
- **More Memory Required:** The control store needs additional memory to hold the microinstructions, which adds to the overall system complexity.

3. Design Examples:

Multiplier Control Unit:

- In both hardwired and microprogrammed control, specialized control units can be designed for specific tasks like multiplication.
- **Hardwired Example:** The control signals for a sequential multiplier would be generated using a finite state machine (FSM), controlling when to load operands, perform partial products, and accumulate results.
- **Microprogrammed Example:** A microprogram could implement multiplication by fetching a series of microinstructions that direct the ALU to perform addition and shifting in multiple steps, storing partial results until the final product is obtained.

CPU Control Unit:

- The **control unit** in a CPU ensures that every instruction executes correctly by generating control signals based on the instruction being executed.
- **In a hardwired control CPU**, each instruction is broken down into a fixed sequence of control signals generated through combinational logic.
- **In a microprogrammed CPU**, the control unit accesses the control store to fetch the appropriate microinstructions for each instruction and generates control signals accordingly.

Pipeline Control: **Pipelining** is a technique used in modern processors to increase instruction throughput by executing multiple instructions in different stages of execution simultaneously. **Pipeline control** ensures that each stage of the pipeline operates correctly and efficiently, handling multiple instructions at different stages without conflicts.

1. Instruction Pipelines: An **instruction pipeline** is **a series of stages through which an instruction passes during execution**. Each stage performs part of the instruction's execution.

Key Stages of a Pipeline:

1. **Instruction Fetch (IF):** The instruction is fetched from memory.
2. **Instruction Decode (ID):** The fetched instruction is decoded to understand the operation.
3. **Execution (EX):** The actual operation (e.g., addition or logical operation) is performed.
4. **Memory Access (MEM):** Memory operations (load or store) are performed if required.
5. **Write-Back (WB):** The result of the operation is written back to a register or memory.

Advantages:

- **Increased Throughput:** Multiple instructions can be processed simultaneously, as different stages of multiple instructions are executed in parallel.
- **Better Resource Utilization:** Pipelines improve the overall efficiency of CPU resources by keeping various components (ALU, memory units, etc.) busy.

Disadvantages:

- **Hazards:** Pipeline control must handle several types of hazards that can disrupt the smooth flow of instructions:
    - **Data Hazards:** Occur when instructions depend on the results of previous instructions.
    - **Control Hazards:** Result from branch instructions where the next instruction to be executed depends on the outcome of a conditional operation.
    - **Structural Hazards:** Arise when multiple instructions compete for the same hardware resources (e.g., ALU or memory).

2. Pipeline Performance: Pipeline performance is typically measured by the number of instructions completed per cycle (called **throughput**).

- **Ideal Performance:** In an ideal pipeline, every stage of the pipeline is filled with instructions, allowing a new instruction to be completed every cycle.
- **Pipeline Latency:** The total time to execute an individual instruction is called latency. While latency remains roughly the same, throughput improves because multiple instructions are completed simultaneously.
- **Stalling:** When a hazard occurs, the pipeline must temporarily halt or "stall" until the hazard is resolved, which negatively impacts performance.

Performance Improvement Techniques:

- **Forwarding/Bypassing:** Data from one stage of the pipeline is passed directly to another stage that needs it, reducing data hazards.
- **Branch Prediction:** Used to minimize control hazards by guessing the outcome of branch instructions and continuing pipeline execution along the predicted path.
- **Superscalar Execution:** Multiple pipelines are used to execute multiple instructions in parallel, further improving performance.

3. Super-Scalar Processing: **Super-scalar processing** refers to the **ability of a CPU to issue and execute more than one instruction per clock cycle by having multiple execution units operating in parallel**. This is an **extension of pipelining**, where **instead of a single instruction pipeline, there are multiple pipelines working concurrently**.

Key Characteristics:

- **Multiple Functional Units:** The processor includes multiple ALUs, FPUs (floating-point units), and other execution units to allow simultaneous instruction execution.
- **Out-of-Order Execution:** Instructions can be executed out of order, provided that data dependencies are maintained and results are stored correctly.
- **Instruction-Level Parallelism (ILP):** The degree to which a processor can exploit parallelism at the instruction level, improving performance by executing independent instructions simultaneously.

Advantages:

- **High Performance:** Superscalar processors significantly increase throughput by executing multiple instructions simultaneously in parallel pipelines.
- **Efficient Resource Utilization:** Multiple execution units are kept busy, ensuring that CPU resources are fully utilized.

Disadvantages:

- **Complexity:** Managing multiple pipelines and resolving instruction dependencies introduces complexity in control design.
- **Increased Power and Area:** Additional execution units and control logic increase the chip's power consumption and physical area.

HARDWIRED CONTROL UNIT VS MICROPROGRAMMED CONTROL UNIT

Basis	Hardwired Control Unit	Microprogrammed Control Unit
Implementation	Implemented using **fixed hardware circuitry and logic gates**.	Implemented using **Microinstructions** stored in control memory.
Control Signal Generation	Control signals are generated using **hardware circuits**.	Control signals are generated using **microinstructions**.
Speed	**Faster** due to direct hardware signal generation.	**Slower** due to microinstruction-based signal generation.
Flexibility	Difficult to modify due to **fixed hardware**.	Easier to modify by **updating microinstructions**.
Cost	More expensive due to hardware-based implementation.	Less expensive as it relies on software (microinstructions).
Complex Instruction Handling	**Cannot handle** complex instructions efficiently; **circuit becomes complex**.	Can **handle** complex instructions efficiently.
Instruction Set	**Limited number of instructions** can be **implemented**.	Can implement a **larger set of instructions**.

Basis	Hardwired Control Unit	Microprogrammed Control Unit
Usage	Typically used in **RISC** (Reduced Instruction Set Computers).	Typically used in **CISC** (Complex Instruction Set Computers).
Modifications	**Hardware changes** are required for updates.	Modifications can be done at the **instruction level**.
Overall Performance	**Faster but less flexible**; suitable for **simple operations**.	**Slower but more flexible**; suitable for **complex operations**.

Unit – 4

Memory Organization: Memory Organization refers to the way data is stored, managed, and accessed within a computer system's memory hierarchy. It encompasses the structure and design of various memory types used to efficiently store data and instructions for processing. Proper memory organization ensures that data is accessed quickly and efficiently, balancing speed, capacity, and cost in modern computing systems.

Memory Technology

Memory technology refers to the various types of memory used in computer systems, each with unique characteristics suited for specific functions. Memory is essential for storing both data and instructions used by the processor.

1. Memory Device Characteristics

Key Characteristics of Memory Devices:

- **Capacity (Size):** Refers to the amount of data that a memory device can store, measured in bytes (e.g., MB, GB).
- **Access Time:** The time it takes to read or write data to memory. **Access time** is critical in determining how fast memory can provide data to the processor.
- **Cycle Time:** The total time between successive accesses to memory. It is generally longer than access time because memory needs some idle time before it can be accessed again.
- **Volatile Memory:** Data is lost when the power is turned off (e.g., **RAM**).
- **Non-volatile Memory:** Retains data even when the power is turned off (e.g. **ROM**, **flash memory**).
- **Power Consumption:** Different types of memory consume varying amounts of power. Lower power consumption is essential in battery-powered devices like smartphones and laptops.
- **Cost per Bit:** Memory technologies differ in terms of cost. Fast and high-capacity memory is often more expensive. There is typically a trade-off between speed, capacity, and cost.
- **Physical Form Factor:** Memory devices come in various physical forms (e.g., chips, modules, cards) based on their use in different systems like desktops, laptops, or embedded systems.

Random-Access Memory (RAM)

Random-Access Memory (RAM) is a type of memory where **any data location can be accessed directly and in the same amount of time**. RAM is used for **temporary data storage while programs are running**.

Types of RAM:

A. **Dynamic RAM (DRAM):**

- **Definition:** DRAM stores each bit of data in a **tiny capacitor**, which gradually loses its charge. Because of this, DRAM needs to be **refreshed** thousands of times per second to maintain data integrity.
- **Characteristics:** DRAM is **slower** and **less expensive** than static RAM but **can store more data per unit area**, making it suitable for **main memory**.
- **Applications:** Used primarily as **system memory** (main memory) in computers.

B. **Static RAM (SRAM):**

- **Definition:** SRAM stores data using **flip-flops**, which **do not need constant refreshing**, making it **faster than DRAM**.
- **Characteristics:** SRAM is **faster** and **more expensive** than DRAM and has a **lower capacity**. It **consumes less power when idle** but **more power when actively accessed**.
- **Applications:** Used in **cache memory** and as **buffers** in CPUs and GPUs due to its high speed.

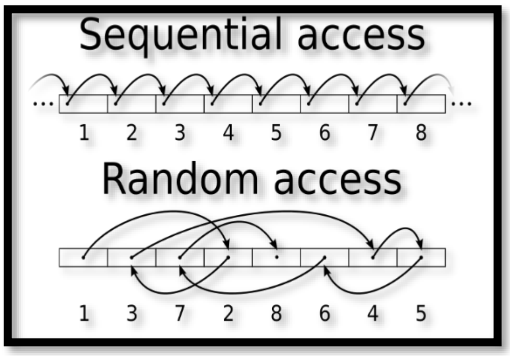
Serial-Access Memory (SAM):

It is a **type of memory where data is accessed sequentially**. To retrieve data, the system has to scan through other data elements in sequence until the desired data is reached.

Types of SAM:

1. **Shift Registers:** A type of serial-access memory where **data moves in and out one bit at a time in a linear sequence**. Often used in simple circuits where **speed is less critical**.
2. **Magnetic Tapes:** Magnetic tapes store data sequentially and are mainly used for **archival storage**. Data is written and read in a linear fashion, which makes it slower than RAM. Primarily used in **backup systems** and environments **where large volumes of data need to be stored over a long period**.

Advantages of SAM:



- **High Data Density:** Serial-access memory like **tapes** can **store large amounts of data per unit of physical space**, making them ideal for **archiving**.
- **Low Cost:** SAM tends to be cheaper than RAM, making it suitable for tasks where fast access is not a priority.

Disadvantages of SAM:

- **Slow Access Time:** Data retrieval is **slower** compared to RAM since it requires sequential searching through data elements.

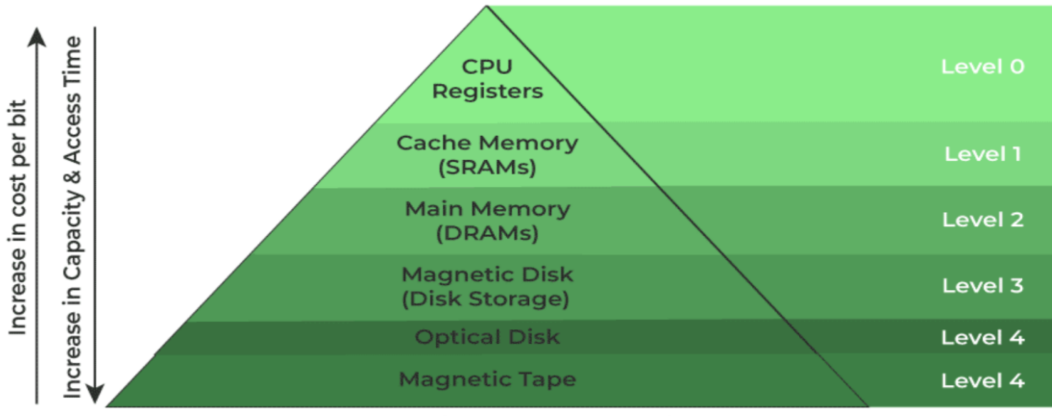
Memory Systems

Memory systems are designed to efficiently manage memory resources in a way that balances speed, capacity, and cost. Modern computer systems employ several layers of memory, each serving a different role to optimize performance and storage.

1. **Multilevel Memories:** **Multilevel Memory Systems** use a hierarchy of memory types with varying speeds and capacities to optimize system performance and cost. Faster, more expensive memory types are used close to the CPU, while slower, cheaper memory types provide larger storage capacity.

Memory Hierarchy:

1. **Registers:** The fastest, smallest memory located inside the CPU, used for holding temporary data and instructions currently being executed.
2. **Cache Memory (SRAM):**
   - **L1 Cache:** Located directly inside the CPU, closest to the execution units, with the fastest access time.
   - **L2/L3 Cache:** Larger but slightly slower caches located either inside or just outside the CPU.
3. **Main Memory (DRAM):** Larger than cache but slower. This is where **active programs and data are stored for fast access while running**.
4. **Secondary Storage (HDD/SSD):**
   - **Hard Disk Drives (HDDs):** Slower but offer large capacity at a lower cost.
   - **Solid-State Drives (SSDs):** Faster than HDDs, but still much slower than DRAM. SSDs are non-volatile and have no moving parts.
5. **Tertiary Storage (Magnetic Tape, Optical Disks):** Very large capacity but extremely slow access time, mainly used for long-term storage and backups.



Memory Hierarchy Design

Benefits of Multilevel Memory:

- **Speed vs Cost Trade-Off:** The system leverages the speed of smaller, faster memory (registers, caches) and the large capacity of slower memory (HDD/SSDs) to optimize performance.
- **Cost Efficiency:** Multilevel memory design allows the system to use high-speed memory in small amounts and large, slower memory for bulk storage, keeping costs manageable.

Address Translation: **Address Translation** is the process of converting virtual memory addresses generated by a program into physical addresses that the system's hardware can use. This allows a computer to provide a large, continuous virtual address space, even if physical memory is smaller.

Virtual Memory: Virtual memory extends the physical memory by using disk space (typically on the hard drive or SSD) as additional RAM. When a program needs more memory than is physically available, parts of the program are stored temporarily in the **swap space** on disk.

Translation Lookaside Buffer (TLB): The **TLB** is a **small, fast cache used by the CPU to speed up the translation of virtual addresses to physical addresses**. It **stores recent address mappings to avoid recalculating address translations repeatedly**.

Working of Address Translation:

- Programs generate **virtual addresses**, which are translated into **physical addresses** by the **Memory Management Unit (MMU)**, with the help of the **page tables**.
- **Page tables** map virtual addresses to physical addresses in memory. However, because accessing page tables can be slow, the **Translation Lookaside Buffer (TLB)** is used to cache recent translations, speeding up the process.
- If a **TLB miss** occurs (the translation is not in the TLB), the page table is accessed to retrieve the mapping, which is then added to the TLB for future use.

Advantages:

- **Efficient Memory Usage:** Virtual memory allows systems to run programs larger than physical memory by using disk space to store inactive parts of the program.
- **Memory Protection:** Each program runs in its own virtual memory space, protecting it from interference by other programs.

Disadvantages:

- **Overhead:** Address translation introduces overhead, especially if there are frequent TLB misses, which can slow down the system.
- **Page Faults:** When a program tries to access data not currently in physical memory (but in virtual memory), a **page fault** occurs, requiring the system to load the page from disk, which can be time-consuming.

Memory Allocation: **Memory Allocation** is the process by which the **system assigns portions of memory to various programs and processes**. Efficient memory allocation is crucial for **ensuring optimal system performance** and **preventing memory leaks or fragmentation**.

Types of Memory Allocation:

1. Static Memory Allocation:
   - Memory is allocated at **compile time**, and the **size of the memory block remains fixed** throughout the program's execution.

- **Advantages:** Simple and fast.
- **Disadvantages:** Inefficient if the program's memory needs change during execution.

2. **Dynamic Memory Allocation:**
   - Memory is allocated at **runtime** based on the program's needs. This is typically done using functions like **malloc() in C**.
   - **Advantages: More flexible**, allows programs to request memory as needed.
   - **Disadvantages:** Requires **careful management** to avoid **memory leaks** (when memory is allocated but not freed) and **fragmentation**.

Fragmentation:

- **External Fragmentation:** Occurs when there are **enough free memory spaces**, but **they are not contiguous**, making it impossible to allocate large blocks of memory.
- **Internal Fragmentation:** Happens **when memory is allocated in fixed-size blocks**, and the **allocated memory is larger than what is needed**, leading to wasted space.

Caches: A **cache** is a **small, high-speed memory used to store frequently accessed data temporarily**. Caches act as **intermediaries between the CPU and main memory** (DRAM), reducing the time required to access data from slower main memory.

Key Characteristics:

- **Small Size:** Cache memory is **much smaller than main memory**, but it's faster because it's made of **SRAM** (Static RAM).
- **Fast Access:** Cache is located close to the CPU, ensuring low latency and fast data access.
- **Temporal Locality:** Data that has been accessed recently is likely to be accessed again soon, so it is stored in the cache for quick retrieval.
- **Spatial Locality:** When data at a particular memory address is accessed, nearby data is likely to be accessed soon, so it is loaded into the cache as well.

Address Mapping in Caches: **Address Mapping** refers to the method used to map data from main memory to a specific location in cache memory. It is the process of determining where to store or retrieve data from main memory in the cache. It is crucial for improving memory access performance, as the cache acts as a smaller, faster memory layer between the main memory and the processor. Proper address mapping ensures efficient data retrieval and reduces cache misses, which occur when data requested by the processor is not found in the cache.

Types of Cache Mapping Techniques

1. **Direct-Mapped Cache**: In a direct-mapped cache, each block of main memory is mapped to exactly one specific location in the cache. The address of the block determines the cache location using a modulo operation, dividing the main memory into equal-sized blocks.

- **Advantages**:
    - Direct-mapped caches are simple in design, as each memory block can be stored only in one specific cache location.
    - They are fast because the cache controller knows exactly where to find or place the data. There is no need to search multiple cache locations, leading to faster memory access times.
    - Because of their simplicity, direct-mapped caches are cheaper to implement and require fewer hardware resources.
- **Disadvantages**:
    - **Cache Conflicts**: One of the primary drawbacks of direct-mapped caches is the potential for frequent cache misses when multiple memory blocks map to the same cache location. This results in **cache conflicts**, where repeatedly accessed blocks evict each other, causing more cache misses.
    - **Thrashing**: If multiple frequently accessed blocks map to the same cache line, the system may experience **cache thrashing**, where the blocks continuously overwrite each other, leading to poor performance.

2. **Fully Associative Cache**: In a fully associative cache, any block of main memory can be placed in any location within the cache. There are no restrictions on where a block can be stored, unlike in a direct-mapped cache.

- **Advantages**:
    - **Reduced Cache Misses**: Fully associative caches significantly reduce cache misses because there are no constraints on where data can be placed. This flexibility makes it easier to store data efficiently in the cache without conflicts.
    - **Optimal Cache Utilization**: Because any block can go into any cache location, fully associative caches make better use of the available cache space, minimizing the likelihood of thrashing.
    - **Improved Performance**: In systems with high cache conflict rates (like direct-mapped caches), fully associative caches can greatly improve performance by allowing more flexible storage and retrieval of data.
- **Disadvantages**:
    - **Complexity**: Fully associative caches require complex hardware to search the entire cache for a block, making them more difficult and expensive to implement.
    - **Slower Access Times**: Since the cache controller must search through all cache lines to find the correct block, fully associative caches can have slower access times compared to direct-mapped caches. This search process is often done using **Content-Addressable Memory (CAM)**, which adds to the cost and complexity.

3. Set-Associative Cache: Set-associative cache is a hybrid between direct-mapped and fully associative caches. Memory blocks are mapped to a specific set of cache lines, but within each set, the block can be placed in any line. For example, in a **2-way set-associative cache**, each block can be placed in one of two cache lines within a set.

- **Advantages**:
    - **Reduced Cache Misses**: Set-associative caches offer a good compromise between direct-mapped and fully associative caches. By allowing blocks to be placed in more than one cache location within a set, they reduce cache misses caused by conflicts in direct-mapped caches.
    - **Flexibility**: Set-associative caches allow for more flexible data placement compared to direct-mapped caches, leading to fewer cache misses, especially in programs with many conflicting memory accesses.
    - **Performance Balance**: They strike a balance between the complexity of fully associative caches and the simplicity of direct-mapped caches, making them a popular choice in modern computer systems. For example, a 4-way set-associative cache allows four blocks to share a set, improving performance without the high cost of full associativity.
- **Disadvantages**:
    - **Moderate Complexity**: Set-associative caches are more complex to implement than direct-mapped caches because the system needs to search a set of cache lines for a block rather than a single location. This increases hardware complexity and cost, although not as much as in fully associative caches.
    - **Moderate Access Time**: Although set-associative caches are faster than fully associative caches, they are slower than direct-mapped caches due to the need to search through multiple cache lines within a set.

Cache Mapping Algorithms

1. **Replacement Policies**: When the cache is full and a new block needs to be loaded, the system must decide which block to evict. Different replacement policies are used depending on the cache mapping strategy:
    - **Least Recently Used (LRU)**: The block that has not been used for the longest period is replaced.
    - **First-In-First-Out (FIFO)**: The oldest block in the set is replaced.
    - **Random Replacement**: A random block is evicted to make room for a new one.
2. **Tag and Index Bits**:

- Cache mapping also involves dividing the memory address into different parts, typically including **tag**, **index**, and **offset** bits:
    - **Index** bits determine which set (or line in a direct-mapped cache) the block is placed in.
    - **Tag** bits are used to uniquely identify blocks within the cache, ensuring that the correct data is retrieved during cache lookups.

    - **Offset** bits determine the exact byte within the block that needs to be accessed.

Cache Structure vs Performance

Cache Structure:

- **Multilevel Caches:** Modern processors often use a hierarchy of caches (L1, L2, L3) to balance speed and capacity:
- **L1 Cache:** Located closest to the CPU cores, very fast but small.
- **L2 Cache:** Larger but slightly slower, shared between cores in some designs.
- **L3 Cache:** Even larger, shared by all cores, slower but still faster than main memory.

Cache Performance Metrics:

1. **Hit Rate:** The percentage of memory accesses that are found in the cache. A higher hit rate means better performance.
2. **Miss Penalty:** The additional time required to fetch data from main memory when it is not found in the cache (cache miss).
3. **Hit Time:** The time it takes to retrieve data from the cache.

Improving Cache Performance:

- **Increasing Cache Size:** Larger caches can store more data, reducing miss rates, but they are also more expensive and slower.
- **Improving Cache Associativity:** Higher associativity reduces conflicts between memory blocks mapping to the same cache location.
- **Using Multilevel Caches:** A combination of L1, L2, and L3 caches helps balance speed and storage capacity.

CACHE COHERENCE

Cache coherence is the process of ensuring that data stored in different caches in a multiprocessor system is consistent and synchronized. This is important because if data is not coherent, it can lead to data corruption and incorrect program behavior.

Cache coherence problem occurs when multiple caches store copies of the same data, and one cache is modified. If the caches are not kept in sync, processors may have an inconsistent view of memory.

There are various **Cache Coherence Protocols** in multiprocessor system:-

- MSI protocol (Modified, Shared, Invalid)
- MOSI protocol (Modified, Owned, Shared, Invalid)
- MESI protocol (Modified, Exclusive, Shared, Invalid)
- MOESI protocol (Modified, Owned, Exclusive, Shared, Invalid)

Unit - 5

System Organization

I/O and System Control

Input/Output (I/O) refers to the communication between a computer system and the external world, which includes peripherals like keyboards, printers, disks, etc. Efficient I/O control is critical for system performance, as it manages how data is transferred between the CPU, memory, and external devices.

1. Programmed I/O (Polling): Programmed I/O is an I/O control method where the **CPU is directly responsible** for managing data transfer between the system and the I/O device. It **continually checks (polls) the status of the I/O device** to see if it is ready for data transfer.

Key Characteristics:

- The CPU waits for the I/O device to be ready, checking its status repeatedly, which is known as **polling**.
- Once the device is ready, the CPU transfers data to or from the I/O device.
- It is simple to implement but can be **inefficient** since the CPU spends time waiting and polling rather than performing useful tasks.

Advantages:

- **Simplicity:** Easy to implement, as the CPU has direct control over I/O operations.
- **No Interrupt Handling:** No need for interrupt mechanisms, reducing complexity.

Disadvantages:

- **CPU Wasting Time:** The CPU remains idle while polling the device, leading to inefficiency, especially for slower I/O devices.
- **Not Scalable:** The CPU becomes a bottleneck as it needs to manage every I/O request manually.

Applications: Typically used in **small embedded systems** or simple applications where I/O operations are infrequent and low-latency isn't a requirement.

2. Direct Memory Access (DMA): Direct Memory Access (DMA) is a technique that allows I/O devices to transfer data directly to or from memory **without continuous involvement from the CPU**. A separate DMA controller manages the transfer, freeing up the CPU to perform other tasks.

Key Characteristics:

- The **DMA controller** takes control of the system buses to move data between I/O devices and memory, bypassing the CPU.
- Once the transfer is set up by the CPU, the DMA controller handles the data transfer autonomously.
- After completion, the DMA controller can interrupt the CPU to signal that the transfer is complete.

Advantages:

- **Improved Efficiency:** Since the CPU is free to execute other instructions while data transfer is ongoing, overall system performance improves.

- **Faster Data Transfer:** DMA is faster than programmed I/O because it eliminates the need for constant CPU intervention.

Disadvantages:

- **Hardware Complexity:** Requires additional hardware (DMA controller), which adds to the system's complexity and cost.
- **Bus Contention:** The DMA controller and CPU share the same bus, which can lead to contention if both try to access memory simultaneously.

Applications: Widely used in **high-performance systems** like **hard disk controllers**, **network interface cards**, and **graphics processors**, where large volumes of data need to be transferred quickly.

3. Interrupt-Driven I/O: **Interrupt-Driven I/O** uses interrupts to signal the CPU when an I/O device is ready for data transfer. Instead of continuously polling the device, the CPU can perform other tasks and respond only when interrupted by the device.

Key Characteristics:

- The I/O device sends an interrupt signal to the CPU when it's ready to send or receive data.
- The CPU temporarily stops its current task, services the I/O interrupt (through an **Interrupt Service Routine (ISR)**), and then resumes its previous task.
- Interrupts allow the CPU to be more efficient, as it does not need to waste time polling.

Advantages:

- **Increased Efficiency:** The CPU can continue performing other tasks while waiting for an I/O device to become ready.
- **Responsive to I/O Devices:** The CPU reacts immediately when the I/O device is ready, minimizing latency.

Disadvantages:

- **Interrupt Overhead:** Handling frequent interrupts can introduce overhead and slow down system performance.
- **Complexity:** More complex to implement than programmed I/O, as it requires **interrupt handling and priority management**.

4. I/O Processors (IOPs): An **I/O Processor (IOP)** is a **specialized processor** that manages all I/O operations, taking over from the CPU. It handles tasks like data transfer, device management, and interrupt processing, offloading these responsibilities from the main processor.

Key Characteristics:

- IOPs operate independently of the CPU, executing I/O instructions and managing data transfers.
- They may have their own memory and operate concurrently with the main CPU.

Advantages:

- **Offloading CPU:** The main processor is freed from managing I/O, allowing it to focus on executing programs.
- **Improved System Throughput:** IOPs improve overall system performance by parallelizing I/O operations and CPU tasks.

Disadvantages:

- **Increased Cost and Complexity:** Additional hardware in the form of dedicated IOPs adds to system cost and design complexity.

Applications: Used in **mainframes** and **high-performance systems**, where I/O performance is crucial for overall system efficiency.

<u>**Parallel Processing:**</u> Parallel processing is a method used to enhance system performance by executing multiple instructions or tasks simultaneously. It improves processing power and speed by dividing work among multiple processing units.

<u>**1. Processor-Level Parallelism (PLP):**</u> **Processor-Level Parallelism** refers to the parallel execution of instructions at the processor level, enabling multiple tasks or instructions to be carried out simultaneously.

Key Characteristics:

- **Superscalar Architecture:** A superscalar CPU can issue and execute more than one instruction during a single clock cycle by having multiple execution units (ALUs, FPUs, etc.).
- **Pipelining:** Instructions are broken down into stages (fetch, decode, execute, etc.) and executed in an overlapping manner across multiple pipeline stages.
- **Out-of-Order Execution:** Instructions can be executed out of order as long as dependencies are maintained, allowing better utilization of the processor's resources.

Advantages:

- **Increased Throughput:** By executing multiple instructions simultaneously, system performance and throughput are improved.
- **Efficient Resource Utilization:** The processor makes better use of its execution units, reducing idle time.

Disadvantages:

- **Complexity:** Managing parallelism at the processor level requires complex control logic and careful management of dependencies and hazards.
- **Power Consumption:** More resources being active simultaneously leads to increased power consumption and heat generation.

Applications: Found in **modern processors**, especially those used in **servers**, **supercomputers**, and **high-end consumer devices** like gaming consoles and smartphones.

<u>**2. Multiprocessors:**</u> **Multiprocessors** refer to **computer systems with two or more processors that work together to perform tasks**. These processors **share the same memory and work in parallel** to enhance computational power.

<u>Types of Multiprocessor Systems:</u>

- **Symmetric Multiprocessing (SMP):**
    - In SMP, all processors share the same memory and have equal access to I/O devices.
    - All processors run the same operating system and can execute tasks simultaneously.
- **Asymmetric Multiprocessing (AMP):**
    - In AMP, each processor is assigned specific tasks or operates independently, often running separate operating systems.
    - One processor may act as the master, managing other processors (slaves).

Key Characteristics:

- **Shared Memory:** Multiprocessor systems typically share a common memory space, allowing processors to communicate and coordinate with each other.
- **Parallel Execution:** Tasks are divided among multiple processors, which execute them concurrently, speeding up processing time.
- **Scalability:** As more processors are added, the system's computational power increases, allowing it to handle more tasks simultaneously.

Advantages:

- **Improved Performance:** By distributing tasks across multiple processors, performance improves significantly, especially for **multithreaded applications**.
- **Reliability and Fault Tolerance:** Multiprocessor systems can continue operating even if one processor fails, as other processors can take over the failed processor's workload.

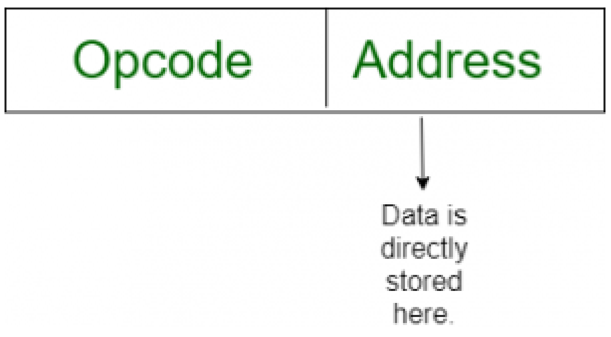
Disadvantages:

- **Synchronization Overhead:** Managing shared memory and synchronizing processors can introduce significant overhead and complexity.
- **Resource Contention:** Multiple processors competing for the same resources (e.g., memory, I/O) can lead to contention and bottlenecks.

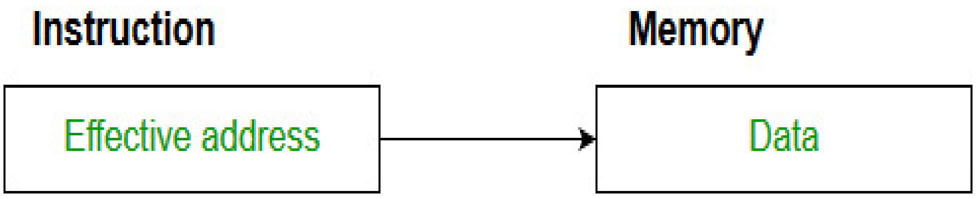
Applications: Widely used in **servers**, **supercomputers**, and **parallel processing systems** where tasks are distributed to enhance processing speed and computational capacity.

Addressing Modes

1. **Immediate addressing mode:** In this mode data is present in address field of instruction. Designed like one address instruction format.
   Note: Limitation in the immediate mode is that the range of constants are restricted by size of address field.



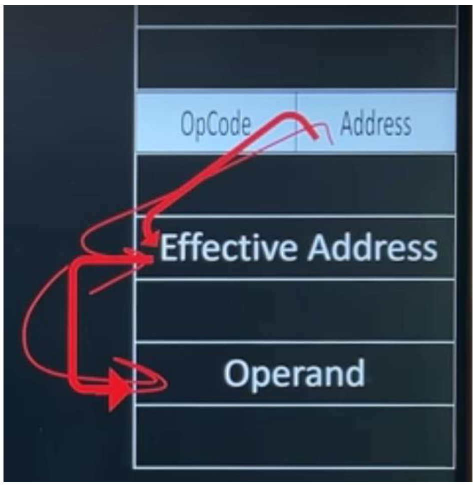
2. **Direct addressing/ Absolute addressing Mode:** The operand's offset is given in the instruction as an 8 bit or 16 bit displacement element. In this addressing mode the 16 bit effective address of the data is the part of the instruction.
   Here only one memory reference operation is required to access the data.



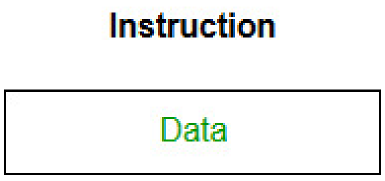
3. **Indirect addressing Mode:** In this mode address field of instruction contains the address of effective address. Here two references are required.
   1st reference to get effective address.
   2nd reference to access the data.

Based on the availability of Effective address, Indirect mode is of two kind:

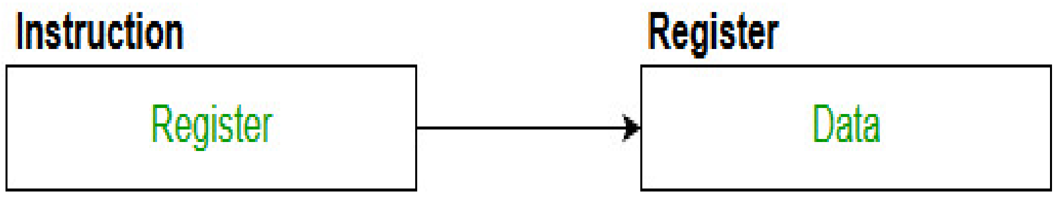
a. Register Indirect: In this mode effective address is in the register, and corresponding register name will be maintained in the address field of an instruction.
   Here one register reference, one memory reference is required to access the data.
b. Memory Indirect: In this mode effective address is in the memory, and corresponding memory address will be maintained in the address field of an instruction.
   Here two memory reference is required to access the data.



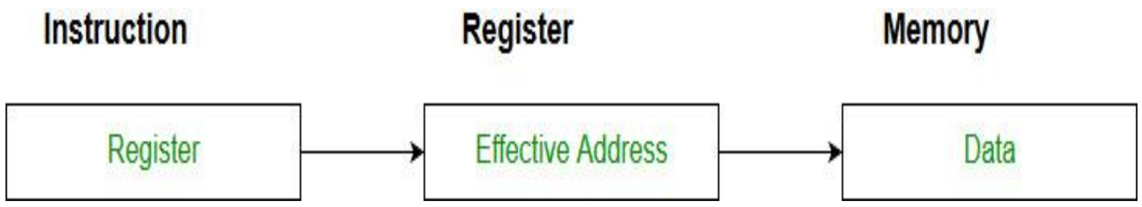
4. **Implied mode:** In implied addressing the operand is specified in the instruction itself. In this mode the data is 8 bits or 16 bits long and data is the part of instruction.Zero address instruction are designed with implied addressing mode.



5. **Register mode:** In register addressing the operand is placed in one of 8 bit or 16 bit general purpose registers. The data is in the register that is specified by the instruction.
   Here one register reference is required to access the data.

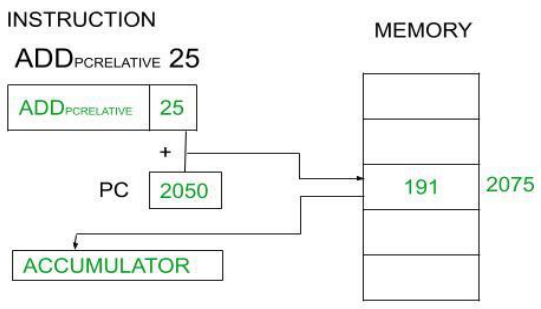


6. **Register Indirect mode**: In this addressing the operand's offset is placed in any one of the registers BX, BP, SI, DI as specified in the instruction. The effective address of the data is in the base register or an index register that is specified by the instruction.
   Here two register reference is required to access the data.



7. **Base register addressing mode:** Base register addressing mode is used to implement inter segment transfer of control. In this mode effective address is obtained by adding base register value to address field value.
   EA= Base register + Address field value.
   PC= Base register + Relative value.

8. **Relative Address mode:** In this mode, the Effective Address (EA) of the operand is calculated by adding the content of the CPU register and the address part of the instruction word. The effective address is calculated by adding displacement (immediate value given in the instruction) and the register value. The address part of the instruction is usually a signed number, either positive or negative. The effective address thus calculated is relative to the address of the next instruction.
   EA = CPU Register + Displacement


INSTRUCTION
ADDPCRELATIVE 25
ADDPCRELATIVE
25
+
PC
2050
ACCUMULATOR
MEMORY
191
2075