**Unit - 1**

**RISC vs CISC Architecture**

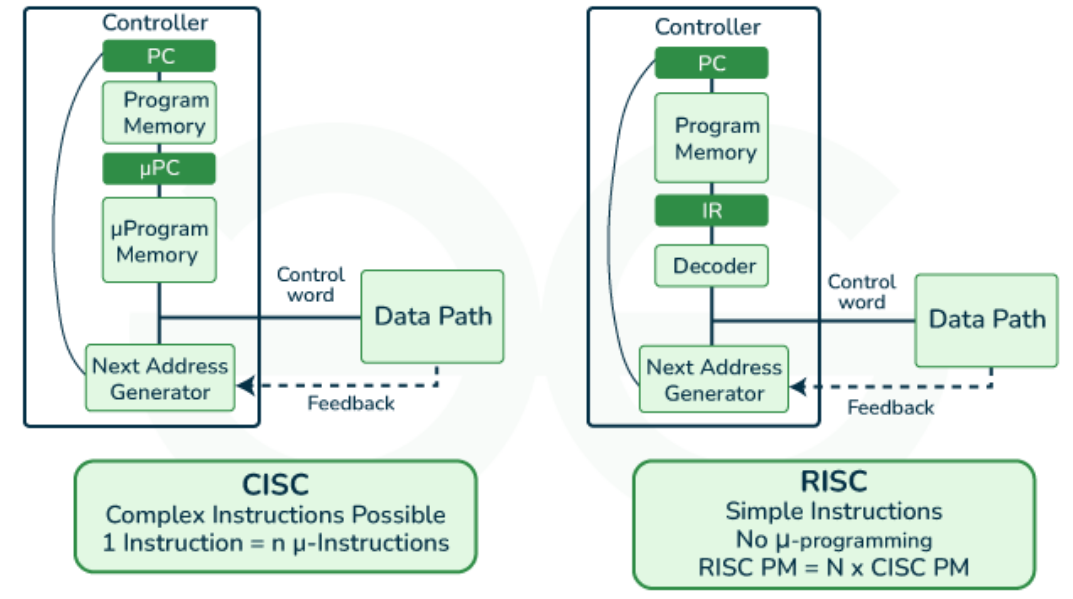
**RISC**

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. |
| **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**.

In scientific notation, a number is expressed as:



* **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 × 2exponent**
* **101.11** in binary can be normalized as: 1.0111 \* 22
* 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. |
| UseCase | 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.

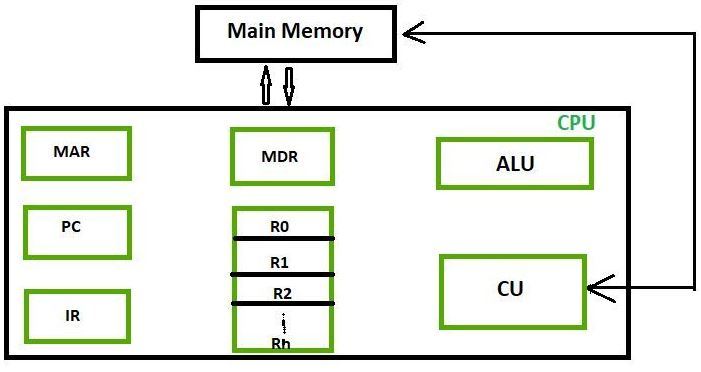
Components of the ALU:

* **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:**

Registers are very fast computer memory that is used to execute programs and operations efficiently. This is done by giving access to commonly used values, i.e., the values that are at the point of operation/execution at that time.

The sole purpose of having a register is fast retrieval of data for processing by CPU. Though accessing instructions from RAM is comparatively faster with a hard drive, but still it isn’t enough for the CPU.



Types of Registers:

* **General-purpose Registers (GPRs):** These are numbered as R0, R1, R2 …. Rn-1, and used to store temporary data during any ongoing operation. Its content can be accessed by assembly programming. Modern CPU architectures tend to use more GPR so that register-to-register addressing can be used more, which is comparatively faster than other addressing modes.
* **Program Counter (PC):** Program Counter (PC) is used to keep the track of the execution of the program. It contains the memory address of the next instruction to be fetched. 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. The incrementation of PCs depends on the type of architecture being used. If we are using 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):** Points to the top of the stack, used in function calls and interrupts.
* **Flag Register:**A [flag register](https://www.geeksforgeeks.org/flag-register-8085-microprocessor/) , also known as a status register or condition code register, is a special type of register in a computer’s central processing unit (CPU) 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, and Interrupt Enable Flag.
* **Accumulator**: This is the most frequently used register used to store data taken from memory. It is in different numbers in different microprocessors.
* **Memory Address Registers (MAR)**: It holds the address of the location to be accessed from memory. MAR and MDR (Memory Data Register) 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.

**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. It’s like keeping your favourite tools on a shelf right next to you, instead of in a faraway closet.

**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.

**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 (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**

Arithmetic Logic Unit (ALU)

Definition:

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:

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.

Logical Operations:

AND, OR, XOR, NOT: Perform bitwise logical comparisons between binary inputs.

Shift Operations: Shifts the binary data left or right, often used in multiplication or division by powers of two.

Comparison Operations:

Equality (==): Checks whether two values are equal.

Less than (<) / Greater than (>): Determines relational comparison results between inputs.

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).

Increment and Decrement:

Incrementing or decrementing a value by one.

Components of an ALU:

Input and Output Registers:

ALUs receive inputs from registers, temporary storage locations inside the CPU. These registers hold the data that the ALU operates on.

Output registers store the result of the operation performed by the ALU before passing it to other parts of the processor.

Control Unit Interaction:

The control unit tells the ALU what operation to perform by sending specific control signals.

Control signals determine whether the ALU performs an arithmetic operation (like addition or subtraction) or a logical operation (like AND or OR).

Flags and Status Registers:

Flags in the ALU indicate the result of the operation (e.g., if there’s an overflow, if the result is zero, or if the carry bit is set after an addition).

Status registers store these flags and help the CPU make decisions (e.g., in conditional jumps based on comparison results).

Types of ALUs:

Combinational ALU:

Performs all operations in a single clock cycle without storing intermediate results (refer to earlier notes on combinational ALUs for detailed information).

Sequential ALU:

Breaks operations into multiple steps, processing them over several clock cycles (refer to earlier notes on sequential ALUs for detailed information).

Fixed-Point vs. Floating-Point ALU:

Fixed-Point ALU: Handles standard integer operations. Most basic ALUs are fixed-point.

Floating-Point ALU: Specialized ALUs used for handling real numbers and decimals, particularly useful in scientific computations and graphics. These ALUs are more complex and operate on the IEEE 754 standard for floating-point numbers.

Working of an ALU:

Input:

The ALU receives two inputs (often from registers) along with a control signal that specifies the type of operation (e.g., addition or AND).

Operation:

Depending on the control signal, the ALU will perform the required operation on the inputs.

For arithmetic operations (like addition), the ALU uses an adder circuit. For logical operations (like AND or OR), it uses logic gates.

Output:

The result is stored in the output register and then passed to other CPU components for further processing or stored in memory.

Flag Updates:

After the operation, the ALU updates flags (like carry, overflow, zero) in the status register, which can influence the next set of operations in the program.

ALU Design and Operations:

Simple ALU Design:

A simple ALU can be designed to perform basic operations like addition, subtraction, AND, OR, NOT.

Adder Circuit: The most critical component of an ALU is the adder, which handles addition operations and, through modification, can be used for subtraction (using two’s complement).

Complex ALU Design:

More complex ALUs have additional features, including:

Multiplier circuits for handling multiplication.

Floating-point units for dealing with decimal numbers.

Shifters for logical and arithmetic shifts (important for tasks like bitwise operations or multiplication by powers of two).

Advantages of an ALU:

Speed of Operations:

The ALU performs operations quickly, with basic operations like addition or logical comparisons happening in a single clock cycle (for a combinational ALU).

Integration with Control Unit:

The tight coupling between the ALU and the control unit allows for efficient instruction execution within the CPU.

Ability to Handle Multiple Operations:

ALUs are versatile, capable of executing multiple types of operations (arithmetic, logical, bitwise), which are the building blocks for complex computations.

Disadvantages of an ALU:

Hardware Cost and Complexity:

A more powerful ALU (e.g., one that supports floating-point arithmetic or complex operations) requires more hardware, which increases the size and power consumption of the CPU.

Limited by Clock Speed:

ALU performance is bound by the clock speed of the processor. Increasing the clock speed to improve performance can result in higher power consumption and heat.

Complexity in Advanced Operations:

While ALUs are efficient at basic arithmetic and logic tasks, operations like division, floating-point arithmetic, or complex bit manipulations require additional circuits and may take multiple clock cycles to complete.

Applications of ALUs:

General-Purpose CPUs:

The ALU is a core component of every CPU, handling all the arithmetic and logical operations that form the foundation of computing tasks.

Graphics Processing Units (GPUs):

ALUs in GPUs are designed for parallel processing of arithmetic tasks, such as those needed in image rendering and video processing.

Embedded Systems:

In microcontrollers and embedded processors, the ALU handles the arithmetic and logic needed to control devices, from simple consumer electronics to complex industrial machines.

Scientific Computing:

ALUs, especially floating-point ALUs, are critical in scientific and engineering applications where high-precision calculations are required, such as in simulations, weather forecasting, and data modeling.

ALU Enhancements:

Parallelism and Pipelining:

In modern processors, pipelining is often used to improve ALU performance, allowing multiple instructions to be processed at different stages simultaneously.

Superscalar architectures use multiple ALUs to execute several instructions in parallel, significantly speeding up computation.

Specialized ALUs:

Many modern CPUs include specialized ALUs for tasks like vector processing (SIMD - Single Instruction, Multiple Data) used in multimedia, encryption, and deep learning.

Summary:

The Arithmetic Logic Unit (ALU) is a crucial part of a CPU, performing essential arithmetic and logical operations that underpin all computer processing. From basic integer arithmetic to complex floating-point calculations, the ALU plays a pivotal role in data manipulation, enabling the CPU to execute programs efficiently. Understanding how the ALU works, its design, and its applications is fundamental to grasping computer architecture and organization.

**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 internal storage (registers) to store intermediate results. |
| **Clock Cycle Dependency** | Non-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. |
| **Design Complexity** | Less complex design, as the entire operation happens at once without control units to track states. | 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 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. |
| **Design Example** | A 4-bit adder ALU that instantly adds two numbers. | A sequential multiplier that multiplies two numbers over multiple cycles using partial products. |
| **Control Complexity** | Simpler control as the ALU handles one operation at a time. | Requires a control unit to track the stages of operation and handle sequencing. |

**Data Path Design** : **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.

**Components of Data Path Design:**

1. **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.

1. **Arithmetic Logic Unit (ALU):**

* The ALU is the core component of the data path that performs arithmetic (addition, subtraction) and logical (AND, OR) operations.

1. **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.

1. **Buses:**

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

1. **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.

1. **Memory Elements:**

* Data paths are connected to memory units, such as RAM, where data is stored and retrieved for processing.

**Data Path Design for Fixed-Point Arithmetic**

**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.

**1. Addition and Subtraction in Fixed-Point Arithmetic:**

* **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.

**2. Multiplication in Fixed-Point Arithmetic:**

* **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.

**Advanced Topics in Data Path Design**

**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 in Data Path Design:**

**Pipeline processing** is an advanced technique used in modern processors to improve the throughput of instruction execution. It allows multiple instructions to be in different stages of execution simultaneously, which increases the overall performance.

1. **Stages of a Pipeline:**

* A typical pipeline divides instruction execution into multiple stages, such as:
  + **Fetch:** Retrieve the instruction from memory.
  + **Decode:** Interpret 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.

1. **Instruction-Level Parallelism:**

* By overlapping the execution of multiple instructions, pipelines enable **instruction-level parallelism (ILP)**, which improves performance by maximizing resource utilization.

1. **Hazards in Pipelining:**

* **Data Hazards:** Occur when instructions depend on the results of previous instructions that have not yet completed.
* **Control Hazards:** Result from branch instructions where the next instruction to execute depends on the outcome of a conditional operation.
* **Structural Hazards:** Arise when two or more instructions need the same hardware resource at the same time.

1. **Data Path for Pipeline Processing:**

* The data path in a pipelined processor is segmented into multiple stages, with **pipeline registers** between each stage to hold intermediate results.
* Each stage operates on a different instruction simultaneously, with the pipeline registers passing results to the next stage in each clock cycle.

1. **Handling Hazards:**

* **Forwarding (Data Forwarding):** Helps to resolve data hazards by feeding the result of an operation directly to a subsequent instruction before it is written back to the register.
* **Branch Prediction:** Used to reduce the impact of control hazards by guessing the outcome of branch instructions and continuing execution along the predicted path.

**Summary of Data Path Design Concepts:**

* **Fixed-Point Arithmetic:** Handles integer operations using basic addition, subtraction, multiplication, and division circuits. These operations are simple and fast but limited to a fixed range of numbers.
* **Floating-Point Arithmetic:** Manages real numbers with larger dynamic ranges and more precision, but requires more complex circuitry and control logic due to normalization, rounding, and handling of special cases.
* **Pipeline Processing:** Significantly improves the performance of processors by overlapping the execution of multiple instructions. Pipelines require careful management of hazards and additional control to ensure correct execution.

**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**

**Definition:**

* **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.

**Applications:**

* Hardwired control is typically used in **small, simple processors**, where speed is critical, and flexibility is not a primary concern (e.g., in **embedded systems** or **special-purpose processors**).

**2. Microprogrammed Control**

**Definition:**

* **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.

**Applications:**

* Used in **general-purpose processors** like **CISC (Complex Instruction Set Computing)** architectures, where the instruction set is large and requires flexible control logic (e.g., in **mainframes** or **general-purpose CPUs**).

**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**

**Definition:**

* **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:**

**Definition:**

* An **instruction pipeline** is a series of stages through which an instruction passes during execution. Each stage performs part of the instruction’s execution (e.g., fetch, decode, execute, memory access, write-back).

**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).

**Applications:**

* Instruction pipelines are a fundamental part of modern **RISC (Reduced Instruction Set Computer)** processors like **ARM** and **MIPS**, improving instruction throughput and efficiency.

**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:**

**Definition:**

* **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.

**Applications:**

* Super-scalar processing is used in modern high-performance processors like

**Intel's Core** and **AMD Ryzen** processors, where maximizing instruction throughput is critical for performance.

**Summary of Control Design Concepts**

* **Hardwired Control:** Fast and efficient for small systems, but difficult to modify or scale. Suitable for simple processors.
* **Microprogrammed Control:** Flexible and easier to modify, making it ideal for complex instruction sets. Slower than hardwired control due to the overhead of fetching microinstructions.
* **Pipeline Control:** Improves CPU performance by overlapping the execution of multiple instructions. Hazards and stalling can reduce performance, but techniques like forwarding and branch prediction help mitigate these issues.
* **Super-scalar Processing:** Extends pipeline control by allowing multiple instructions to be executed simultaneously using parallel pipelines, significantly increasing throughput and performance.

**Unit – 4**

**Memory Organization**

**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.

**2. 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:**

1. **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.

1. **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.

**3. Serial-Access Memory (SAM)**

**Serial-Access Memory (SAM)** 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.

1. **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**

**Definition:**

* **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.

1. **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.

1. **Main Memory (DRAM):**

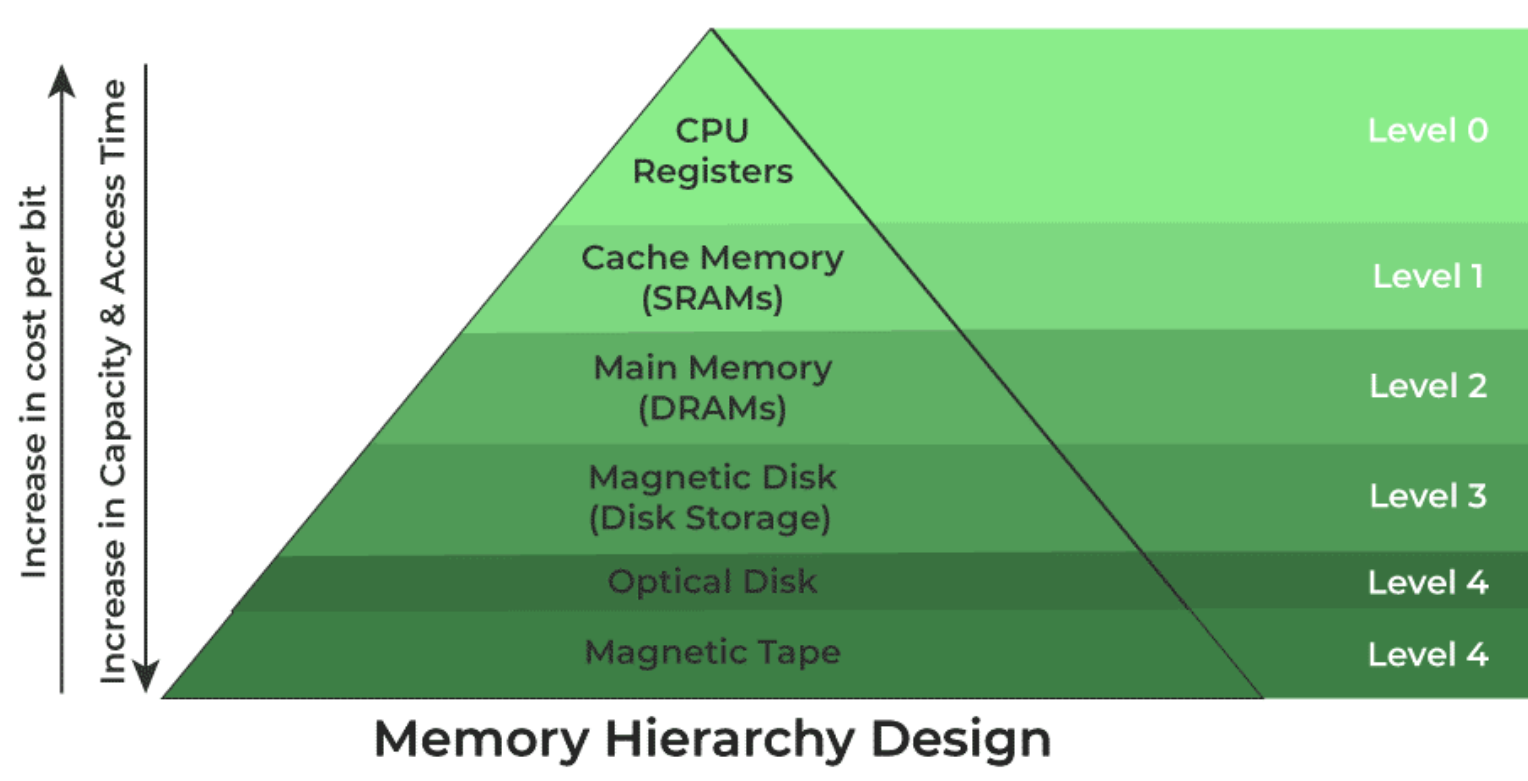
* Larger than cache but slower. This is where active programs and data are stored for fast access while running.

1. **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.

1. **Tertiary Storage (Magnetic Tape, Optical Disks):**

* Very large capacity but extremely slow access time, mainly used for long-term storage and backups.



**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.

**2. Address Translation**

**Definition:**

* **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.

**3. Memory Allocation**

**Definition:**

* **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.

1. **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**

**Definition:**

* 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.

**1. Main Features of Cache 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.

**2. Address Mapping in Caches**

**Definition:**

* **Address Mapping** refers to the method used to map data from main memory to a specific location in cache memory.

**Types of Cache Mapping Techniques:**

1. **Direct-Mapped Cache:**

* Each block of main memory maps to exactly one location in the cache.
* **Advantages:** Simple and fast.
* **Disadvantages:** Can cause frequent **cache misses** if multiple blocks compete for the same cache location.

1. **Fully Associative Cache:**

* Any block of main memory can be loaded into any cache location.
* **Advantages:** Reduces cache misses by providing more flexibility.
* **Disadvantages:** More complex and slower than direct-mapped due to the need for searching the entire cache.

1. **Set-Associative Cache:**

* A compromise between direct-mapped and fully associative caches. Memory blocks are mapped to a set of locations (e.g., 2-way, 4-way associative), and within each set, any block can be loaded.
* **Advantages:** Provides a good balance between complexity and performance, reducing the likelihood of cache misses compared to direct-mapped caches.

**3. 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.

**Summary of Memory Organization Concepts**

* **Memory Technology:** Memory devices are categorized based on characteristics like access time, volatility, power consumption, and form factor. RAM (volatile) and storage devices like SSDs (non-volatile) serve different roles in a computer’s memory hierarchy.
* **Multilevel Memory Systems:** Use a hierarchy of registers, cache, RAM, and secondary storage to balance speed, capacity, and cost, ensuring efficient data management and fast access.
* **Address Translation:** Virtual memory provides the illusion of a larger memory space by translating virtual addresses to physical addresses with the help of the TLB.
* **Memory Allocation:** Efficient memory allocation is key to optimizing system performance, while poorly managed allocation can lead to fragmentation and memory leaks.
* **Caches:** Small, fast memories that reduce the time it takes to access frequently used data. Caches improve overall performance by reducing the number of times data is fetched from slower main memory.

These comprehensive notes cover key aspects of **memory organization** for your exams, ensuring you understand both the theoretical concepts and their practical applications.

**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)**

**Definition:**

* **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)**

**Definition:**

* **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**

**Definition:**

* **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.

**Applications:**

* Common in systems with **real-time constraints**, like **network communication** or **multitasking operating systems**, where quick response to I/O events is critical.

**4. I/O Processors (IOPs)**

**Definition:**

* 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.

**Parallel Processing**

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.

**1. Processor-Level Parallelism (PLP)**

**Definition:**

* **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.

**2. Multiprocessors**

**Definition:**

* **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.

**Types of Multiprocessor Systems:**

* **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.

**Summary of System Organization Concepts**

* **Programmed I/O**: The CPU actively polls devices for their status, but this can waste CPU time.
* **DMA**: Offloads data transfer from the CPU, allowing for faster, more efficient I/O.
* **Interrupt-Driven I/O**: Interrupts the CPU when devices are ready, improving efficiency by reducing idle time.
* **IO Processors**: Specialized processors that handle I/O tasks, improving system performance by offloading the I/O work from the main CPU.
* **Processor-Level Parallelism**: Techniques like pipelining, superscalar execution, and out-of-order execution increase the number of instructions a processor can execute simultaneously.
* **Multiprocessors**: Systems that use multiple processors to share tasks and improve performance, commonly used in servers and high-performance computing.