Module 2

Machine Instructions, I/O Systems, and Memory Organization

Learning Objectives

By the end of this module, learners will be able to:

1. Explain various addressing modes used in computer architecture.
2. Interpret and write simple assembly language instructions.
3. Describe the principles of input and output operations in computer systems.
4. Understand subroutine structures and their role in program execution.
5. Explain instruction encoding techniques and their formats.
6. Analyze different methods of accessing I/O devices and handling interrupts.
7. Describe direct memory access (DMA) and its significance in I/O operations.
8. Compare standard I/O interfaces such as PCI, SCSI, and USB.
9. Understand the basic organization of memory systems including RAM and ROM.
10. Discuss memory hierarchy, speed, and cost trade-offs in computer design.

Structure

2.1 Addressing Modes

Addressing modes specify how the operand of an instruction is chosen or accessed during the execution of a program. These modes define whether the operand is part of the instruction itself, found in a register, or located at a specific memory address. Addressing modes provide **flexibility**, **efficiency**, and **compactness** to instruction design, making it easier to handle data structures, loops, branching, and dynamic memory. Understanding addressing modes is crucial for writing efficient low-level code and for designing instruction sets in processor architecture.

2.1.1 Immediate Addressing Mode

In immediate addressing mode, the operand is embedded **directly in the instruction** itself rather than being stored elsewhere. The instruction explicitly provides the constant value that is to be used in the operation, which the processor can use immediately without any memory lookup. This leads to **faster execution** and is frequently used for **initialization of variables**, **setting counters**, or **loading constants**.

Characteristics:

- Operand is a **constant value**, hard-coded in the instruction.
- No memory access is needed to retrieve the operand.
- Fast and efficient execution.
- Operand size is limited by instruction format.

Example:

```
MOV R1, #10      ; Load the constant value 10 directly into register R1
```

Use Cases:

- Assigning constant values
- Setting initial conditions in loops
- Loading small, known values during program startup

2.1.2 Direct Addressing Mode

In direct addressing mode, the **instruction contains the memory address** where the actual operand is located. The processor accesses this memory location to fetch the operand. This mode is **easy to understand and implement**, but the operand's location is fixed, which limits flexibility when working with dynamic data.

Characteristics:

- The **effective address** is explicitly stated in the instruction.
- Only **one memory access** is needed to retrieve the operand (after fetching the instruction).
- Simple but less adaptable for dynamic data structures.

Example:

```
MOV R1, [1000]      ; Load the value from memory address 1000 into register R1
```

Use Cases:

- Accessing global/static variables
- Reading from fixed I/O locations
- Suitable for small programs with limited memory requirements

Limitations:

- Hardcoding addresses makes the code less reusable or relocatable.
- Cannot support dynamic memory operations or pointer-based structures efficiently.

2.1.3 Indirect Addressing Mode

Indirect addressing mode involves an **intermediate reference**. The instruction contains a location (either a register or memory) which **holds the effective address** of the operand. The processor first fetches this effective address and then accesses the operand. This mode is **highly flexible** and supports **dynamic data handling**, **pointers**, and **complex data structures**.

Characteristics:

- Requires **two memory accesses**:
    1. First to get the effective address
    2. Second to get the operand from that address
- Supports variable data locations
- Facilitates pointers and dynamic memory operations

Example:

MOV R1, [[2000]] ; Memory location 2000 contains the address where the operand is stored

Use Cases:

- Pointer-based programming
- Linked list and dynamic data structures
- Array processing with base addresses stored in memory

Drawbacks:

- Slower due to additional memory access
- Slightly more complex instruction decoding

2.1.4 Register Addressing Mode

In register addressing mode, the operand is stored in one of the **CPU's internal registers**. The instruction specifies the **register name or number**, and the data is directly fetched from or written into the register. This addressing mode is **extremely fast** as registers are physically closest to the processing unit.

Characteristics:

- **No memory access**—entire operation occurs within the CPU.
- Fastest addressing mode.
- Efficient use of instruction bits, as registers can be encoded with fewer bits.

Example:

ADD R1, R2 ; Add the contents of R2 to R1

Use Cases:

- Temporary storage of intermediate results
- Fast execution loops
- Operations requiring frequent data access

Limitations:

- Limited number of available registers
- Not suitable for large data sets

2.1.5 Indexed and Relative Addressing Modes

These two addressing modes provide **versatile ways to compute effective addresses** dynamically, especially useful in handling **arrays, loops**, and **branching instructions**.

A. Indexed Addressing Mode

In indexed addressing, the effective address is calculated by **adding a constant (offset)** to the contents of an **index register**. This mode is widely used for **iterating through arrays**, where the index register is incremented or decremented to point to the next element.

Characteristics:

- Effective Address = Base Address + Index
- Supports sequential access to array elements
- Index can be adjusted during runtime

Example:

```
MOV R1, [BASE + R2]      ; R2 is the index register; BASE is a constant or memory location
```

Use Cases:

- Accessing elements of arrays or tables
- Loop operations over memory blocks

B. Relative Addressing Mode

In relative addressing, the operand's effective address is obtained by **adding a displacement (offset)** to the **current value of the Program Counter (PC)**. It is primarily used for **control flow**, especially in branch or jump instructions.

Characteristics:

- Effective Address = PC + Offset
- Used for **position-independent code**
- Ideal for forward and backward jumps

Example:

```
JMP +10      ; Jump to the instruction 10 bytes ahead of the current PC value
```

Use Cases:

- Conditional and unconditional branching
- Implementing loops and subroutine calls
- Creating relocatable code

2.2 Assembly Language Instructions

Assembly language is a low-level programming language that provides direct control over hardware using human-readable mnemonics. Unlike high-level languages, assembly closely maps to machine

language instructions, making it faster and more efficient but also hardware-specific. Understanding the structure and categories of assembly instructions is essential for system programming, compiler development, and embedded systems.

2.2.1 Structure of Assembly Language Instructions

An assembly instruction typically consists of **four main components**, arranged in a predefined format. These components provide all the necessary information for the processor to execute the instruction.

General Format:

[Label:] Mnemonic Operand1, Operand2 ; Comment

Components:

- **Label (Optional):**
  A symbolic name representing a memory location or instruction address, used for jumps or loops.

- **Mnemonic (Mandatory):**
  The operation code (opcode) in symbolic form, such as MOV, ADD, JMP, etc.

- **Operands (Required/Optional):**
  Specifies the source and destination for the operation. Can be registers, constants, or memory addresses.

- **Comment (Optional):**
  Begins with a semicolon ; and provides human-readable notes that are ignored by the assembler.

Example:

START: MOV AX, 05h ; Load immediate value 5 into AX register

Key Characteristics:

- One instruction per line
- Syntax varies slightly across different assembly languages (e.g., x86, ARM, MIPS)
- Operands may be in **source–destination** or **destination–source** order depending on architecture

2.2.2 Data Transfer Instructions

These instructions **move data** from one location to another without modifying it. They are fundamental to all assembly programs and are used extensively in all types of operations.

Types of Data Transfer Instructions:

- **Register to Register:**
  Moves data between CPU registers
  MOV AX, BX

- **Memory to Register / Register to Memory:**
  Transfers data between memory and registers
  MOV AX, [2000h] or MOV [2000h], AX

- **Immediate to Register/Memory:**
  Loads constant values
  MOV AX, 0x05

- **Stack-related Transfers:**
  PUSH, POP – Move data to/from the stack

- **String and Block Transfers:**
  Used to move blocks of data
  MOVSB, MOVSW, etc.

Key Points:

- Do **not perform arithmetic** or logical operations.
- Can use different addressing modes.
- Efficient use of these instructions is crucial for performance.

2.2.3 Arithmetic and Logic Instruction

These instructions perform mathematical and logical operations on data stored in registers or memory. The results are typically stored in registers and may update the processor's **status flags** (Zero, Sign, Carry, etc.).

Arithmetic Instructions:

- **Addition:** ADD AX, BX
- **Subtraction:** SUB AX, 1
- **Increment/Decrement:** INC AX, DEC AX
- **Multiplication:** MUL BX (unsigned), IMUL BX (signed)
- **Division:** DIV BX (unsigned), IDIV BX (signed)

Logical Instructions:

- **AND, OR, XOR:** AND AX, BX, OR AX, 1
- **NOT:** NOT AX
- **Shift/Rotate:** SHL AX, 1, ROR BX, 2

Comparison:

- CMP AX, BX – Subtracts BX from AX but doesn't store the result; updates flags only.

Key Points:

- Use **registers or memory** as operands.
- Affect **condition flags** used for branching.
- Crucial in implementing algorithms, decision-making, and bitwise operations.

2.2.4 Branch and Control Instructions

Branch and control instructions manage the **program flow**, allowing non-linear execution such as loops, decision-making, and subroutine handling.

Types of Branch Instructions:

- **Unconditional Jump:**
  JMP LABEL – Transfers control to a specified label.
- **Conditional Jump:**
  Depends on flag status after comparison:
  - JE, JZ (Jump if equal/zero)
  - JNE, JNZ (Jump if not equal/non-zero)

- JG, JL, JGE, JLE, etc.

- **Loop Control:**
  - LOOP LABEL – Decrements CX and jumps if not zero

Subroutine and Procedure Control:

- **CALL** – Jumps to a subroutine and saves return address
- **RET** – Returns from the subroutine to saved address

Interrupt Handling:

- INT n – Calls a system-defined interrupt handler

Key Points:

- Use **comparison results** or counter values
- Essential for **loops**, **if-else**, **function calls**
- Control logic and interrupt management depend on these

2.2.5 Input/Output and Stack Instructions

These instructions deal with **external communication** (I/O operations) and **temporary data storage** using the system stack.

A. Input/Output Instructions

Used for **communicating with peripheral devices** such as keyboards, displays, and storage.

- **IN Port, Register:** Reads data from an input port into a register
  IN AL, 60h
- **OUT Port, Register:** Sends data from register to an output port
  OUT 64h, AL

Characteristics:

- Primarily used in systems with **port-mapped I/O**
- Specific to architectures like x86
- Useful in embedded and hardware-level programming

B. Stack Instructions

The **stack** is a special area of memory used for storing temporary data like return addresses, local variables, and registers during subroutine calls.

- **PUSH Register/Memory:** Saves content onto the stack
  PUSH AX
- **POP Register/Memory:** Restores content from the stack
  POP AX
- **PUSHF / POPF:** Push and pop the flags register
- **CALL / RET:** Implicitly use the stack for storing and restoring return addresses

Key Points:

- Stack operates on **LIFO (Last-In-First-Out)** principle
- Stack pointer (SP) tracks the top of the stack

- Efficient for **nested function calls**, **interrupt handling**, and **temporary data storage**

2.3 Input and Output Operations

Input and output (I/O) operations are essential for a computer system to interact with the external environment. These operations handle the transfer of data between the processor (or memory) and peripheral devices such as keyboards, displays, storage drives, sensors, and network interfaces. I/O operations can be managed in several ways, each varying in terms of **complexity**, **speed**, **CPU involvement**, and **synchronization mechanisms**. Efficient I/O management is critical for system performance, particularly in real-time and embedded systems.

2.3.1 Program-Controlled I/O

Also known as **polling or synchronous I/O**, program-controlled I/O is the simplest method where the **CPU actively monitors** the I/O device and manages all data transfer manually through instructions.

How It Works:

- The CPU continuously checks (polls) the I/O device's status register.
- If the device is ready (e.g., data is available or buffer is empty), the CPU initiates the data transfer.
- CPU executes instructions to read/write data between the I/O device and memory or registers.

Characteristics:

- CPU remains busy checking I/O device status.
- Simple to implement and suitable for low-speed or infrequent I/O.
- Inefficient for high-speed devices or multitasking environments.

Example Use Case:

```
LOOP:   IN      AL, 60h          ; Read from keyboard port
        TEST AL, AL
        JZ      LOOP             ; Wait until a key is pressed
```

Advantages:

- Simple hardware and software implementation
- Useful in small embedded systems or teaching environments

Disadvantages:

- CPU time is wasted during polling
- Poor performance for systems requiring concurrent processing

2.3.2 Interrupt-Driven I/O

In this method, the I/O device **interrupts the CPU** when it is ready for communication, instead of the CPU continuously checking the device. This allows the CPU to perform other tasks and only respond when needed.

How It Works:

- The CPU executes regular instructions.
- When an I/O device becomes ready, it sends an **interrupt signal** to the CPU.
- The CPU temporarily halts its current task, saves its state, and jumps to the **Interrupt Service Routine (ISR)** to handle the I/O operation.
- After servicing, the CPU resumes its prior execution.

Characteristics:

- More efficient than polling.
- CPU only reacts when necessary, saving processing time.
- Requires interrupt handling hardware and software (ISR, vector tables, etc.).

Advantages:

- Better CPU utilization
- Suitable for **multitasking** and **real-time systems**
- Handles multiple devices efficiently

Disadvantages:

- More complex implementation
- Overhead from saving/restoring context and executing ISRs

2.3.3 DMA-Controlled I/O (Direct Memory Access)

DMA allows I/O devices to **directly transfer data to or from memory** without continuous CPU intervention, greatly improving performance, especially for **large data transfers** (e.g., disk, audio, video).

How It Works:

- A **DMA controller** manages the transfer.
- CPU initializes the DMA transfer (start address, data count, direction).
- DMA takes over the system bus and performs memory access independently.
- CPU is interrupted only at the **start and end** of the transfer.

Characteristics:

- Offloads repetitive data transfer work from the CPU.
- Transfers data at high speeds using block transfer mechanisms.
- Ideal for bulk I/O operations.

Example Use Case:

- Copying a 1 MB file from hard disk to RAM using DMA while CPU continues other tasks.

Advantages:

- Efficient for high-throughput operations
- Frees CPU for computation during I/O
- Reduces interrupt overhead

Disadvantages:

- Additional hardware (DMA controller) required
- Complex synchronization needed to avoid bus conflicts

2.3.4 Synchronization and Data Transfer Methods

Synchronization ensures that **data is transferred correctly and safely** between the processor and I/O devices, which often operate at different speeds. Both hardware and software techniques are used for timing coordination and preventing data loss or corruption.

A. Synchronization Methods:

1. **Handshaking:**
    - Involves a two-way signal exchange between CPU and I/O device.
    - Ensures that both sides are ready before data transfer occurs.
    - Common in serial communication.
2. **Busy Waiting (Polling):**
    - CPU waits in a loop until the device is ready.
    - Used in **program-controlled I/O**.
3. **Interrupts:**
    - Used in **interrupt-driven I/O**, where devices notify CPU when ready.
4. **Bus Arbitration:**
    - Ensures only one component (CPU, DMA, etc.) controls the bus at a time.

B. Data Transfer Methods:

1. **Serial Transfer:**
    - One bit at a time over a single line.
    - Used in serial ports (e.g., USB, UART).
2. **Parallel Transfer:**
    - Multiple bits transferred simultaneously over multiple lines.
    - Faster but more resource-intensive; used in internal buses.
3. **Programmed I/O:**
    - CPU manually handles each byte/word of data.
4. **Interrupt-Driven I/O:**
    - Device interrupts CPU only when ready.
5. **DMA Transfer:**
    - Device transfers blocks of data directly to/from memory.

Importance of Synchronization:

- Prevents **data corruption** due to mismatched timing.
- Maintains system stability in **concurrent I/O** operations.
- Ensures **device readiness** and **safe CPU handover** during transfers.

2.4 Subroutines

A **subroutine** (or procedure) is a block of code designed to perform a specific task and can be called from multiple places within a program. Subroutines improve code organization, promote reusability, and simplify complex programs by allowing **modular design**.

2.4.1 Concept of Subroutines and Procedures

Subroutines (also known as functions or procedures) are **self-contained code modules** that can be invoked by a main program or another subroutine. When a subroutine is called, control is transferred to it, and after execution, control returns to the calling program.

Key Features:

- Has a unique name or label
- May take input values (parameters)
- May return results
- Facilitates code reuse and modular programming

Benefits:

- Reduces code duplication
- Simplifies debugging and maintenance
- Enables code abstraction

Terminology:

- **Caller:** The program that invokes the subroutine
- **Callee:** The subroutine being called
- **Return Address:** The location in the caller to which control returns after execution

2.4.2 Calling and Returning Mechanisms

To execute a subroutine, the program must **transfer control** to it and later **return control** to the point following the call. This is managed by **CALL and RETURN instructions**, and usually involves saving the return address.

Calling Process:

- Save return address (current PC + instruction size) onto stack or register
- Transfer control to subroutine's starting address

Returning Process:

- Restore the saved return address
- Transfer control back to caller

Example (Assembly-style):

```
CALL SUB1          ; call subroutine
...
SUB1:
    ; perform task
    RET            ; return to calling point
```

Mechanisms Involved:

- Program Counter (PC) update
- Stack usage for saving return address
- Optional parameter and register saving

2.4.3 Parameter Passing Methods

Parameters (arguments) are the **input values or data** passed to subroutines. Efficient parameter passing ensures that subroutines can be reused with different inputs.

Common Methods:

1. **Pass by Register:**
    - Parameters stored in CPU registers
    - Fast, but limited by register availability
2. **Pass by Stack:**
    - Parameters pushed onto the stack
    - Scalable and flexible
3. **Pass by Memory:**
    - Parameters stored in fixed memory locations
    - Less efficient but simple
4. **Pass by Reference:**
    - Passes memory addresses instead of actual values
    - Allows modification of caller's data

Choosing a Method:

- Small data: use registers
- Large data or multiple arguments: use stack or memory
- Shared access: use reference passing

2.4.4 Stack Implementation in Subroutines

The **stack** plays a critical role in managing subroutine calls, especially when nested or recursive calls are used.

Functions of Stack in Subroutines:

- Stores **return address**
- Saves **register contents**
- Passes **parameters**
- Holds **local variables**

Call Stack Behavior:

- **PUSH:** Save data (return address, parameters)
- **POP:** Restore data (after subroutine completes)

Example:

```
CALL SUB1          ; Save return address on stack
...
SUB1:
    PUSH AX        ; Save register state
    ...
```

```
POP AX          ; Restore state
RET             ; Return to caller
```

Advantages of Stack-Based Subroutines:

- Supports recursion
- Enables local scope for variables
- Handles dynamic function calls

2.5 Instruction Encoding

Instruction encoding refers to how a computer represents machine instructions in binary form. It is a key component of **instruction set architecture (ISA)** design and directly influences CPU complexity, performance, and memory usage.

2.5.1 Need for Instruction Encoding

Instruction encoding provides a **systematic way to represent instructions** in binary format, allowing the CPU to decode and execute operations accurately and efficiently.

Why Encoding is Necessary:

- Enables compact representation of instructions
- Allows the CPU to interpret instructions correctly
- Distinguishes between different operations and operand types
- Ensures compatibility within the instruction set

Goals of Instruction Encoding:

- Maximize **efficiency**
- Minimize **instruction size**
- Support **variety of operations**
- Ensure **ease of decoding**

2.5.2 Opcode and Operand Fields

An instruction is divided into **fields**, each serving a specific function.

Common Fields in an Instruction:

1. **Opcode (Operation Code):**
    - Specifies the operation to be performed (e.g., ADD, MOV, JMP)
    - Each opcode has a unique binary code
2. **Operand(s):**
    - Specifies source and/or destination data
    - Can refer to registers, memory addresses, or immediate values
3. **Addressing Mode Bits (optional):**
    - Indicates how to interpret the operand (immediate, direct, etc.)
4. **Miscellaneous Fields:**
    - May include flags, condition codes, or instruction length indicators

Example Format (Hypothetical 16-bit instruction):
| 4 bits | 4 bits | 8 bits |
| Opcode | Reg No | Address/Immediate |

2.5.3 Fixed-Length and Variable-Length Encoding
Fixed-Length Encoding:

- Every instruction has the **same size** (e.g., 32 bits).
- Easier to decode, aligns well with pipelines.
- May waste space for simple instructions.

Advantages:

- Simplifies instruction fetching and decoding
- Ideal for RISC architectures

Disadvantages:

- Inefficient for small/simple instructions

Variable-Length Encoding:

- Instruction size varies based on opcode and operands.
- More compact but decoding is complex.

Advantages:

- Saves memory
- Suitable for dense instruction sets (CISC)

Disadvantages:

- Increases decoding complexity
- Difficult to align instructions for pipelining

2.5.4 Examples of Instruction Encoding
Example 1: Fixed-Length (RISC Style)
32-bit instruction format:
[6-bit opcode][5-bit reg1][5-bit reg2][16-bit immediate]
ADD R1, R2 → 000001 00001 00010 0000000000000000
Example 2: Variable-Length (x86 Style)
Instruction: MOV AX, 2000h
Encoding: B8 00 20
- B8 is opcode for MOV to AX
- 00 20 is the immediate 16-bit data (2000h)
Example 3: Stack-Based Architecture
Instruction: PUSH 5
Encoding: 1101 0101 00000101
- Opcode: PUSH
- Operand: Immediate value 5
These examples highlight how different ISAs use different encoding schemes to balance between performance, compactness, and decoding simplicity.
2.6 Accessing I/O Devices
To interact with the external world, computers use **Input/Output (I/O) devices**. Accessing these devices involves transferring data between the CPU/memory and peripherals such as keyboards,

printers, displays, sensors, or network cards. Various architectural mechanisms exist to organize and control this interaction efficiently and reliably.

2.6.1 I/O Port Concept

An **I/O port** is a hardware interface through which data is transferred between the CPU and an external device. Each port typically corresponds to a particular device or a specific function of a device.

Types of I/O Ports:

- **Input Ports:** Used to read data from external devices (e.g., keyboard input)
- **Output Ports:** Used to send data to devices (e.g., sending characters to a printer)
- **Bidirectional Ports:** Support both input and output (e.g., USB ports)

How Ports Are Accessed:

- CPU sends or receives data using special **IN** and **OUT** instructions.
- Port numbers or addresses are assigned to each device.
- I/O ports are usually addressed using **8-bit or 16-bit** values.

Use Cases:

- Reading a keystroke from keyboard
- Sending display data to monitor
- Controlling motors, LEDs in embedded systems

2.6.2 Memory-Mapped I/O vs. Isolated I/O

There are **two primary methods** for accessing I/O ports: **Memory-Mapped I/O** and **Isolated I/O**. Both differ in how the CPU communicates with I/O devices and how the I/O space is managed.

Memory-Mapped I/O:

- I/O devices share the **same address space as memory**.
- Standard data transfer instructions (MOV, LOAD, STORE) are used.
- Each I/O port is assigned a **unique memory address**.

Advantages:

- Uniformity in instructions (no separate I/O instructions)
- Easier to integrate with memory-based operations
- Can use all addressing modes

Disadvantages:

- Reduces available memory address space
- Requires careful memory management to avoid conflicts

Isolated I/O (Port-Mapped I/O):

- I/O ports are given a **separate address space**.
- Uses **special I/O instructions** (IN, OUT) for communication.
- Common in x86 architecture.

Advantages:

- Full memory space remains dedicated to memory
- Simple and efficient for systems with limited I/O

Disadvantages:

- Requires separate instruction set
- Less flexible in addressing modes

2.6.3 I/O Device Control and Status Registers

Most I/O devices use internal registers to manage data flow and device state. These include:

1. Control Registers:

- Receive commands from the CPU (e.g., start operation, reset device)
- Configure operational modes (e.g., baud rate for serial ports)

2. Status Registers:

- Indicate device conditions (e.g., ready, busy, error)
- Polled by CPU or checked by interrupt handlers

3. Data Registers:

- Temporary holding places for input or output data

How They Work Together:

- CPU writes to **control register** to initiate an operation
- CPU checks **status register** to verify readiness
- Data is read from or written to the **data register**

2.6.4 Handshaking and Data Transfer

Handshaking is a mechanism used to **synchronize** communication between the CPU and I/O devices, especially when devices operate at different speeds.

Basic Handshaking Process:

- Device raises a **ready/busy** signal.
- CPU checks the signal before initiating data transfer.
- Transfer occurs when both sender and receiver are ready.

Signals Involved:

- **Ready:** Device is ready to send/receive data.
- **Acknowledge (ACK):** Confirmation signal sent by receiver.

Types of Data Transfer Techniques:

- **Synchronous Transfer:** Requires both systems to operate on the same clock or time schedule.
- **Asynchronous Transfer:** Uses handshaking to coordinate timing between sender and receiver.

Applications:

- Serial communication (UART)

- Parallel buses with slow peripherals
- Communication in embedded microcontrollers

2.7 Interrupt Mechanism and Hardware
Interrupts are signals that **temporarily halt the current program** so the processor can execute a **more urgent task**. Once the task is handled, the processor resumes normal operation. Interrupts are critical for **real-time responsiveness** and **multi-device management**.

2.7.1 Interrupt Concepts and Types
What Is an Interrupt?
An interrupt is a **hardware or software signal** that causes the CPU to stop its current task and execute a special routine called the **Interrupt Service Routine (ISR)**.

Types of Interrupts:
1. **Hardware Interrupts:**
    - Triggered by external devices (e.g., keyboard, mouse, disk controller)
    - Examples: IRQ0 for timer, IRQ1 for keyboard in x86
2. **Software Interrupts:**
    - Generated by executing specific instructions like INT n
    - Used for system calls and debugging
3. **Maskable Interrupts:**
    - Can be disabled or ignored by CPU
    - Controlled by interrupt mask settings
4. **Non-Maskable Interrupts (NMI):**
    - Cannot be ignored
    - Used for critical events like power failure or hardware errors
5. **Vectored vs. Non-Vectored:**
    - **Vectored:** CPU jumps to a predefined ISR address
    - **Non-Vectored:** ISR address must be supplied by device or memory

2.7.2 Interrupt Cycle and Processing
When an interrupt is received, the CPU enters an **interrupt cycle** instead of continuing with the next instruction.
Steps in Interrupt Cycle:
1. **Interrupt Request:** Device sends a signal to the CPU.
2. **Acknowledgment:** CPU accepts and confirms the interrupt.
3. **Save Context:** Current PC and flags are saved (typically on stack).
4. **Transfer Control:** Jump to the ISR address.
5. **Execute ISR:** Perform necessary actions (read data, reset device, etc.).

6. **Restore Context:** Pop saved data from stack.
7. **Resume Execution:** Continue with interrupted program.

Important Points:

- Interrupts improve **CPU efficiency** by avoiding busy-waiting.
- Multiple interrupts may be handled by **priority** and **vectoring**.

2.7.3 Interrupt Priority and Vectoring

Interrupt Priority:

Used when **multiple devices** request attention simultaneously.

- Devices are assigned **priority levels**
- Higher-priority devices are serviced first
- CPU may disable lower-priority interrupts during ISR execution

Nested Interrupts:

- Higher-priority interrupts can interrupt the ISR of a lower-priority device.
- Requires careful context saving and stack management.

Interrupt Vectoring:

- Mechanism by which the CPU **identifies the ISR address**.
- In **vectored interrupts**, each device has a unique vector/address.
- In **non-vectored interrupts**, the device must supply the ISR address during the interrupt request.

Vector Table:

- A reserved memory region containing addresses of all ISRs
- Indexed by interrupt numbers (e.g., INT 21h in x86 for DOS services)

2.7.4 Hardware for Interrupt Handling

Efficient interrupt handling requires specific **hardware components** to detect, prioritize, and signal interrupts to the CPU.

Key Hardware Components:

1. **Interrupt Request Lines (IRQ):**
    - Dedicated lines for signaling interrupt requests to the processor
2. **Interrupt Controller:**
    - Manages multiple interrupt sources
    - Examples: **Programmable Interrupt Controller (PIC)** like Intel 8259A
3. **Interrupt Mask Register:**
    - Allows enabling/disabling specific interrupts
4. **Vector Generator:**
    - Supplies the ISR address in vectored systems

5. **Nested Interrupt Support:**
    - Stack-based saving of multiple return addresses

Modern Systems:

- Use **Advanced PICs**, **APICs (Advanced Programmable Interrupt Controllers)**, and **interrupt remapping** for multicore and virtualized systems

2.8 Direct Memory Access (DMA)

Direct Memory Access (DMA) is a technique that allows peripheral devices to transfer data directly to or from memory without involving the CPU in every transaction. It is especially useful for high-speed devices such as disks, network cards, and audio systems.

2.8.1 DMA Operation and Working Principle

DMA enables **fast and efficient data transfer** between memory and I/O devices. When a DMA operation is initiated:

- The **CPU configures the DMA controller** with transfer parameters: source address, destination address, size, and direction.
- Once started, the **DMA controller takes over the system buses** (data, address, and control).
- The controller **transfers data directly** from I/O device to memory (or vice versa) without further CPU intervention.
- When the transfer is complete, the DMA controller may **generate an interrupt** to notify the CPU.

Key Characteristics:

- CPU is involved only at the **start and end** of the transfer.
- The DMA controller **temporarily suspends** the CPU's access to the memory bus.
- Allows **block transfers**, ideal for large data sets.

2.8.2 DMA Controller and Registers

The DMA controller is a dedicated hardware module responsible for managing DMA transfers.

Main Components of a DMA Controller:

1. **Address Register:**
    - Holds the memory address for reading/writing data.
2. **Count Register:**
    - Specifies the number of bytes/words to transfer.
3. **Control Register:**
    - Defines the direction (read/write), mode, and status flags.
4. **Status Register:**
    - Indicates current state: busy, error, complete.
5. **Request and Acknowledge Lines:**
    - Used for handshaking between CPU, DMA controller, and peripheral.

DMA Modes:

- **Burst Mode:** Transfers entire block without releasing bus.
- **Cycle Stealing:** Takes control of bus one cycle at a time.
- **Transparent Mode:** DMA works only when CPU is idle.

2.8.3 Advantages of DMA over Interrupt I/O

DMA offers several benefits over traditional interrupt-driven data transfers, especially in high-speed or high-volume scenarios.

DMA Advantages:

- **Reduced CPU Overhead:**
  - CPU doesn't manage each byte transfer, freeing it for other tasks.
- **Faster Data Transfer:**
  - Transfers are direct and occur in bulk.
- **Efficient Bus Utilization:**
  - DMA can take over bus access with minimal delay.
- **Lower Interrupt Rate:**
  - Only one interrupt at end of transfer, compared to one per byte in interrupt I/O.

Limitations of Interrupt I/O:

- CPU is interrupted repeatedly.
- More context switching.
- Slower performance for large data volumes.

2.8.4 DMA and System Performance

DMA significantly enhances overall system performance, especially when dealing with large or continuous data streams.

Performance Benefits:

- **Parallel Processing:**
  - CPU continues executing while DMA handles data transfer.
- **High Throughput:**
  - Suitable for disk-to-memory, memory-to-memory transfers.
- **Improved Responsiveness:**
  - Ideal for real-time systems needing quick I/O response.

Performance Considerations:

- DMA introduces **bus contention**; must coordinate with CPU.
- **Bus arbitration** mechanisms resolve access conflicts.
- Best used with **priority control** to prevent delays in time-critical tasks.

2.9 Standard I/O Interfaces

I/O interfaces serve as standardized ways to connect peripheral devices to a computer system, ensuring compatibility and efficient communication.

2.9.1 Peripheral Component Interconnect (PCI)

PCI is a high-speed **parallel bus standard** for connecting internal peripherals (sound cards, network cards, etc.) to the motherboard.

Key Features:

- Supports **32- or 64-bit data buses**.
- Plug-and-play functionality.
- Operates at 33 MHz or 66 MHz.
- Uses **bus mastering** to allow devices to control the bus.

Applications:

- Internal expansion cards
- Communication and multimedia devices

2.9.2 Small Computer System Interface (SCSI)

SCSI is a **parallel interface** used for connecting a variety of peripheral devices, especially **storage devices**, to a computer system.

Key Features:

- Allows daisy-chaining of multiple devices (up to 16).
- Supports hard disks, scanners, tape drives, etc.
- Provides high-speed, reliable data transfer.
- Operates independently of CPU using **command queueing**.

Advantages:

- Fast, flexible interface
- Ideal for enterprise storage systems

2.9.3 Universal Serial Bus (USB)

USB is a widely adopted **serial bus standard** for connecting external devices such as keyboards, flash drives, and mobile devices.

Key Features:

- Hot-swappable and plug-and-play
- Supports multiple device types
- Uses a tiered star topology (via hubs)
- USB versions:
    - USB 1.1: 12 Mbps
    - USB 2.0: 480 Mbps
    - USB 3.0/3.1: up to 10 Gbps

Advantages:

- Universally supported

- Simplified connector and cable management
- Power supply through the port

2.9.4 Comparison of I/O Interface Standards

Feature	PCI	SCSI	USB
Type	Parallel	Parallel	Serial
Device Support	Internal	Storage, others	External, all types
Hot-Plugging	No	Limited	Yes
Max Devices	5–10	Up to 16	Up to 127
Speed	Moderate	High	Varies (Low–High)
Usage	Expansion Cards	Enterprise Storage	General Use

2.10 Basic Memory Organization

Memory organization refers to how data is structured and accessed within the memory system of a computer. Understanding it is crucial for hardware design, OS memory management, and performance tuning.

2.10.1 Memory Cell and Word Organization

Memory Cell:

- The smallest unit of memory that stores **1 bit** (0 or 1).
- Composed of a **flip-flop or capacitor** in RAM.

Word Organization:

- A group of **N bits** (e.g., 8, 16, 32, 64) forming a "word".
- Each word has a **unique address**.
- Memory can be **byte-addressable** or **word-addressable**.

Example:

- A 32-bit word = 4 bytes = addressable as a single unit in word-addressable systems.

2.10.2 Random Access Memory (RAM)

RAM is **volatile memory** used for temporary data and program storage during operation.

Types of RAM:

- **Static RAM (SRAM):**
    - Faster, used in cache
    - No need for refreshing
    - Expensive and lower density
- **Dynamic RAM (DRAM):**
    - Slower, needs refreshing
    - Used in main memory
    - Higher density, lower cost

Features:

- Read/write access

- Contents lost on power off
- Fast access time

2.10.3 Read Only Memory (ROM)
ROM is **non-volatile memory** used to store **firmware** and permanent instructions.
Types of ROM:
- **PROM:** Programmable once
- **EPROM:** Erasable with UV light
- **EEPROM/Flash ROM:** Electrically erasable

Features:
- Data is prewritten and fixed
- Used for BIOS, boot loaders
- Retains contents even when powered off

2.10.4 Memory Address Decoding
Memory address decoding is the process of **identifying which memory chip or location** should respond to a CPU request.
Techniques:
- **Full/Absolute Decoding:**
    - Unique address assigned to every memory location
    - Requires more hardware (decoders)
- **Partial Decoding:**
    - Fewer lines used; multiple locations may respond to same address
    - Can cause **address aliasing**

Decoding Components:
- **Address Bus:** Carries address signals
- **Decoder Circuit:** Translates address to chip-enable signals

Importance:
- Ensures correct memory access
- Avoids conflicts between memory and I/O devices

2.11 Memory Hierarchy Principles
The **memory hierarchy** is a layered structure that organizes storage components based on **speed, cost, size**, and **proximity to the CPU**. It ensures efficient data access and optimizes system performance while managing cost and capacity constraints.

2.11.1 Hierarchical Memory Concept
The hierarchical memory model places different types of memory at various levels depending on their access time and cost per bit. As we move from top to bottom in the hierarchy:
- **Speed decreases**
- **Capacity increases**

- **Cost per bit decreases**

Memory Levels (Top to Bottom):

1. **CPU Registers**
2. **Cache (L1, L2, L3)**
3. **Main Memory (RAM)**
4. **Secondary Storage (HDD/SSD)**
5. **Tertiary/Backup Storage (Optical Discs, Tapes, Cloud)**

Purpose:

- Bring frequently used data closer to the CPU
- Reduce average memory access time
- Balance **performance and cost**

Access Characteristics:

- Frequently accessed data is placed in **higher levels** (faster memory)
- Less frequently accessed or bulk data resides in **lower levels**

2.11.2 Cache Memory and Locality of Reference

Cache memory is a small, high-speed memory located between the CPU and main memory. It stores **frequently used instructions and data** to reduce access time.

Types of Cache:

- **L1:** Closest to CPU core, very fast, smallest
- **L2:** Slightly slower, larger, may be shared across cores
- **L3:** Shared among multiple cores, even larger

Locality of Reference Principle:

1. **Temporal Locality:** Recently accessed data is likely to be accessed again soon.
2. **Spatial Locality:** Data near recently accessed locations is likely to be used soon.

Cache Operation:

- On memory access, CPU checks the cache.
- If found (cache hit), data is read quickly.
- If not (cache miss), data is fetched from RAM and possibly stored in cache for future access.

Benefits:

- Reduces average memory latency
- Improves execution speed
- Essential in pipelined and multi-core processors

2.11.3 Secondary Storage Devices

Secondary storage holds data permanently and is non-volatile, unlike RAM. It provides high-capacity, long-term storage at a lower cost.

Types:

1. **Hard Disk Drives (HDD):**
    - Magnetic storage
    - Large capacity, lower speed
2. **Solid State Drives (SSD):**
    - Flash-based storage
    - Faster than HDDs, more expensive per GB
3. **Optical Drives:**
    - CDs, DVDs, Blu-rays
    - Low cost, mainly used for backup or distribution
4. **Flash Drives & SD Cards:**
    - Portable, solid-state
    - Suitable for removable storage

Role in Hierarchy:

- Stores the OS, applications, user files
- Feeds data into RAM as needed
- Interface with virtual memory systems (e.g., paging)

2.11.4 Virtual Memory

Virtual memory is a technique that gives an application the illusion of a large, continuous memory space by using **disk space to extend RAM**.

Key Concepts:

- **Paging:** Memory is divided into fixed-size pages; disk holds the pages not currently in use.
- **Page Table:** Maps virtual addresses to physical locations
- **Page Fault:** Occurs when a program accesses data not currently in RAM; OS fetches it from disk

Benefits:

- Enables execution of large programs with limited RAM
- Isolates processes, improving security and stability
- Supports multitasking and memory protection

Trade-offs:

- Slower access due to disk latency
- Can cause performance degradation if paging is excessive (thrashing)

2.12 Speed and Cost Trade-Offs

Designing a computer's memory system involves balancing **speed**, **cost**, and **capacity**. Fast memory is expensive and small, while large memory is slower and cheaper.

2.12.1 Relationship between Speed, Cost, and Capacity

There exists an **inverse relationship** among the three:

Factor	Speed	Cost per Bit	Capacity
Registers	Very High	Very High	Very Low
Cache	High	High	Low
RAM	Medium	Moderate	Medium
SSD/HDD	Low	Low	High
Tape	Very Low	Very Low	Very High

Observations:

- Faster memory (e.g., cache) is more expensive and limited.
- Bulk storage (e.g., HDD) is cheap but slow.
- Systems use layered memory to optimize performance and cost.

2.12.2 Performance Optimization Techniques

To bridge the gap between speed and capacity, designers employ several optimization techniques.

1. Caching:

- Frequently accessed data stored in fast memory

2. Memory Interleaving:

- Spreads memory addresses across multiple modules to parallelize access

3. Pipelining:

- Overlaps stages of instruction execution to improve throughput

4. Prefetching:

- Anticipates future memory accesses and loads data ahead of time

5. Write Buffers and Write-Back Caches:

- Temporarily store writes to avoid blocking CPU

6. Use of DRAM with Row Buffers:

- Reduces access latency within rows

2.12.3 Cost-Effective Memory Design Strategies

Effective memory system design aims to **maximize performance within budget constraints**.

Strategies Include:

- **Multilevel Cache Hierarchy:**
    - Using multiple cache levels to balance speed and cost
- **Hybrid Memory Systems:**
    - Combining DRAM and SSD for faster storage
- **Compression and Deduplication:**
    - Reducing data size to save space and improve speed
- **Smart Replacement Policies:**
    - Using LRU (Least Recently Used) or LFU (Least Frequently Used) for efficient cache usage

- **Dynamic Memory Allocation:**
    - Allocating memory only when needed to avoid waste

Design Goals:

- Maintain high hit rates in caches
- Minimize average memory access time
- Control costs by using slower memory for infrequently used data

2.13 Summary

Unit 2 provides an in-depth understanding of the **fundamental concepts and mechanisms** that govern computer instruction execution, memory operations, and I/O management at the architectural level.

The unit begins by introducing **addressing modes**, which define how operands are accessed in memory or registers. Various modes such as immediate, direct, indirect, register, indexed, and relative addressing enable flexible and efficient instruction formulation, especially in low-level programming and instruction decoding.

It then explores **assembly language instructions**, including their structure and categories such as data transfer, arithmetic, logical, control, and I/O operations. Understanding how instructions interact with registers, memory, and devices is critical for system-level programming and CPU design. The instruction cycle — fetch, decode, execute — is central to this discussion, detailing how the processor processes one instruction at a time.

The unit also discusses **I/O operations**, emphasizing the differences between **program-controlled I/O**, **interrupt-driven I/O**, and **DMA-controlled I/O**. These methods differ in how much they engage the CPU during data transfers. While program-controlled I/O relies on constant CPU polling, DMA allows high-speed transfers with minimal CPU involvement, thus improving overall efficiency.

The concept of **subroutines** (procedures) is covered next, highlighting how modular code segments are called, executed, and returned from using stack-based mechanisms. Parameter passing methods (by register, stack, reference) are discussed along with stack-based implementation for recursion and nested subroutines.

A major portion of the unit is dedicated to **memory systems**, including basic memory organization, memory hierarchy, address decoding, and virtual memory. It emphasizes how memory is structured in cells and words, and how access is managed using RAM and ROM types. Memory address decoding is explained in terms of absolute and partial decoding techniques, which are vital for chip selection and hardware integration.

Modern computer systems use **hierarchical memory models** to balance speed, cost, and capacity. At the top are fast but expensive registers and caches; in the middle is main memory; and at the bottom, cost-effective but slow secondary and tertiary storage. **Cache memory** and the principle of **locality of reference** are discussed in detail, along with virtual memory's ability to simulate large memory through disk paging.

The unit concludes with an analysis of **I/O interface standards** such as PCI, SCSI, and USB, comparing their architecture, use cases, and performance characteristics. It also presents the trade-offs between speed, cost, and capacity, leading to practical **memory design strategies**. Techniques like caching, memory interleaving, pipelining, and compression are identified as essential to building high-performance, cost-effective systems.

Overall, this unit bridges the gap between **hardware-level operations** and **system-level performance**, providing a foundational understanding of how modern computers execute instructions, manage data, interact with devices, and maintain performance under resource constraints.

2.14 Keywords

1. **Addressing Mode** – A method used to determine the location of operands during instruction execution.
2. **DMA (Direct Memory Access)** – A technique allowing I/O devices to transfer data to/from memory without CPU intervention.

3. **Interrupt** – A signal that temporarily halts CPU execution to address an urgent task or event.
4. **Cache Memory** – High-speed memory that stores frequently accessed data for faster CPU access.
5. **Subroutine** – A reusable code block that performs a specific task and can be invoked using CALL/RET instructions.
6. **Virtual Memory** – A memory management technique that uses disk space to extend available RAM and run larger programs.

2.15 Self-Assessment Questions

1. Differentiate between memory-mapped I/O and isolated I/O with examples.
2. Explain the fetch-decode-execute cycle in the context of instruction processing.
3. What is the role of cache memory in the memory hierarchy, and how does it enhance system performance?
4. Describe the working principle of DMA and its advantages over interrupt-driven I/O.
5. List and explain any three addressing modes with examples.
6. How does virtual memory improve the performance and capability of a computer system?

2.16 Case Study

Case Study: Data Acquisition System Using DMA and Hierarchical Memory

A real-time industrial data acquisition system is designed to monitor temperature and pressure across various units in a manufacturing plant. It uses **sensors** interfaced with **microcontrollers** which log data continuously to RAM. To reduce CPU load, the system employs **DMA** to transfer large sensor data blocks directly to memory. The system also includes **cache memory** for high-speed operations and **virtual memory** to handle temporary data overflow. Critical alerts (e.g., pressure spikes) are sent via **interrupts** to the CPU for immediate response. The data is archived on SSDs using **memory-mapped I/O**, while USB interfaces allow for manual data extraction.

Questions:

1. How does the use of DMA and cache memory improve the performance of the data acquisition system described?
2. Explain the role of interrupts and memory-mapped I/O in the system's real-time responsiveness and data handling.

2.17 References

1. M. Morris Mano, *Computer System Architecture*, Pearson Education.
2. William Stallings, *Computer Organization and Architecture: Designing for Performance*, Pearson.
3. Carl Hamacher, Zvonko Vranesic, Safwat Zaky, *Computer Organization*, McGraw-Hill.
4. Andrew S. Tanenbaum, *Structured Computer Organization*, Pearson.
5. David A. Patterson, John L. Hennessy, *Computer Organization and Design*, Morgan Kaufmann.
6. Intel Corporation, *Intel® 8259A Programmable Interrupt Controller Datasheet*.

7. IEEE Standard 1003.1, *Information Technology – Portable Operating System Interface (POSIX)*.