VG

COMPUTER ARCHITECTURE

MCA-104

COMPUTER ARCHITECTURE

PREFACE

Dear Reader,

It is a privilege to present this comprehensive collection of **RTU MCA Semester Examination Notes**, designed to cover the complete syllabus with clarity and academic accuracy. Every concept, definition, explanation, and example has been organized to support thorough understanding and effective exam preparation.

The purpose of these notes is to simplify complex topics and provide a reliable study resource that can assist in both detailed learning and quick revision. Continuous effort has been made to ensure correctness and relevance; however, learning grows when readers engage, question, and explore further.

May these notes serve as a strong academic foundation and contribute meaningfully to your preparation and future growth.

Warm regards,
Virendra Goura
Author
www.virendragoura.com

DISCLAIMER

Unit-1

Basic Building Blocks: Gates, Boolean Functions and Expressions Designing Gate Networks, K-map simplification, Useful Combinational Parts, Programmable Combinational Parts, Timing and Control, Latches, Flip-flops, Registers and Counters, Sequential Circuits.

Arithmetic/Logic Unit: Numbers Representation, Arithmetic Operations, Floating-Point Arithmetic.

Unit-2

Register Transfer Language and Micro-operations: Concept of bus, data movement among registers, a language to represent conditional data transfer, data movement from/to memory. Design of Arithmetic & Logic Unit and Control Unit Control design hardwired control, micro programmed arithmetic and logical operations along with register transfer, timing in register.

Unit-3

Instruction and Addressing: A simple computer organization and instruction set, instruction formats, addressing modes, instruction cycle, instruction execution in terms of microinstructions, interrupt cycle, concepts of interrupt and simple 1/0 organization, Synchronous & Asynchronous data transfer, Data Transfer Mode: Program Controlled, Interrupt driven, DMA (Direct Memory Access). Implementation of processor using the building blocks.

Unit-4

Memory System Design: Memory Origination, Memory Hierarchy, Main Memory (RAM/ROM chips), Auxiliary memory, Associative memory, Cache Memory, Virtual Memory. Assembly Language Programs, Assembler Directives, Pseudo Instructions, Macroinstructions, Linking and Loading.

Unit-5

Vector and Array Processing: Shared-Memory, Multiprocessing, Distributed Mufti Computing.

Microprocessor Concepts: Pin Diagram of 8085, Architecture of 8085, Addressing Mode of 8085, functional block diagram of 8085 assembly language, instruction set of 8085.

UNIT 1– Basic Building Blocks & Arithmetic/Logic Unit

Basic Building Blocks: Gates, Boolean Functions and Expressions Designing Gate Networks, K-map simplification, Useful Combinational Parts, Programmable Combinational Parts, Timing and Control, Latches, Flip-flops, Registers and Counters, Sequential Circuits.

Arithmetic/Logic Unit: Numbers Representation, Arithmetic Operations, Floating-Point Arithmetic.

PART A – BASIC BUILDING BLOCKS

1. Introduction

Every computer system is built using **digital electronic circuits** that process data in the form of **binary signals** (0s and 1s).
These digital systems perform logical and arithmetic operations using **logic gates**, **combinational circuits**, and **sequential circuits**.

The goal of this unit is to understand how these fundamental building blocks work together to design and control a computer’s internal architecture.

2. Logic Gates

Logic gates are the most basic components of digital circuits.
They perform logical decisions based on binary inputs.
Each gate implements a **Boolean function**, which defines how output depends on inputs.

Binary System:

Binary has two logic levels:

- **Logic 0** → represents LOW voltage
- **Logic 1** → represents HIGH voltage

2.1 Types of Logic Gates

a) AND Gate

- Operation: Logical Multiplication
- Symbol: ·
- Expression: **Y = A · B**
- Truth Table:

A	B	Output Y
0	0	0
0	1	0
1	0	0

1	1	1

Explanation:
The output is high (1) only when both inputs are 1.

b) OR Gate

- Operation: Logical Addition
- Expression: **Y = A + B**
- Truth Table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Explanation:
The output becomes 1 if any input is 1.

c) NOT Gate (Inverter)

- Operation: Logical Complement
- Expression: **Y = $\bar{A}$**
- Truth Table:

A	Y
0	1
1	0

Explanation:
It inverts the input; if input is 1, output becomes 0.

d) NAND Gate

- Operation: NOT of AND
- Expression: **Y = (A · B)’**
- Truth Table:

A	B	Y
0	0	1
0	1	1
1	0	1

1	1	0

Explanation:
Output is 1 except when both inputs are 1.

e) NOR Gate

- Operation: NOT of OR
- Expression: **Y = (A + B)′**

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

f) XOR Gate (Exclusive OR)

- Expression: **Y = A ⊕ B = A′B + AB′**
- Truth Table:

A	**B**	**Y**
0	0	0
0	1	1
1	0	1
1	1	0

Explanation:
Output is 1 only if inputs are different.

g) XNOR Gate (Exclusive NOR)

- Expression: **Y = (A ⊕ B)′ = AB + A′B′**
- Truth Table:

A	**B**	**Y**
0	0	1
0	1	0
1	0	0
1	1	1

Explanation:
Output is 1 when both inputs are same.

2.2 Universal Gates

- **NAND and NOR** are called *universal gates* because any logic function can be constructed using only these gates.

3. Boolean Functions and Expressions

A **Boolean function** defines the logical relationship between binary input variables and the resulting output.
It uses Boolean algebraic operations (AND, OR, NOT).

Example:
F(A, B, C) = A′B + AC′ is a Boolean function.

A **Boolean expression** is a mathematical expression formed using **binary variables**, **logical operators**, and **constants (0 and 1)** to represent the behavior of a **digital logic circuit**.

3.1 Boolean Laws and Rules

1. **Commutative Laws**
   - A + B = B + A
   - A · B = B · A
2. **Associative Laws**
   - (A + B) + C = A + (B + C)
   - (A · B) · C = A · (B · C)
3. **Distributive Law**
   - A · (B + C) = (A · B) + (A · C)
4. **Complement Laws**
   - A + A′ = 1
   - A · A′ = 0
5. **Identity Laws**
   - A + 0 = A
   - A · 1 = A

3.2 Canonical Forms

(a) Sum of Products (SOP) – Expression written as sum (OR) of product terms (ANDs).
Example: F = A′B + AB′

(b) Product of Sums (POS) – Expression written as product (AND) of sum terms (ORs).
Example: F = (A + B)(A′ + B′)

4. Designing Gate Networks

Designing digital circuits involves:

1. Writing the Boolean function.
2. Simplifying it using Boolean laws or K-map.
3. Drawing circuit using gates.

Example:
F = A’B + AB’ → This is an XOR function.
Circuit: Uses two AND gates, two NOT gates, and one OR gate.

5. K-Map Simplification

A **Karnaugh Map (K-map)** is a **graphical technique** used to **simplify Boolean functions** in a systematic and visual manner. It reduces complex Boolean expressions into **minimal form without lengthy algebraic manipulation**.
K-map simplification helps in designing **efficient digital circuits** with **minimum number of logic gates**.

Purpose of K-Map

- Minimizes Boolean expressions
- Reduces number of logic gates
- Simplifies circuit design
- Decreases hardware cost and power consumption

Types of K-Maps

Number of Variables	K-Map Size
2 variables	2 × 2
3 variables	2 × 4
4 variables	4 × 4

Steps for K-Map Simplification

Step 1: Draw the K-Map

Draw the K-map according to the **number of variables** (2, 3, or 4).

Step 2: Mark Minterms

Fill **1s** in the cells corresponding to the given **minterms**.
All other cells contain **0**.

Step 3: Group Adjacent 1s

- Group adjacent 1s in powers of **2** (1, 2, 4, 8…)
- Groups must be **rectangular**
- **Overlapping is allowed**
- **Wrapping around edges is allowed**
- Larger groups give **simpler expressions**

Step 4: Write the Simplified Expression

For each group:

- Identify variables that **remain constant**
- Eliminate changing variables
- Combine all group expressions using **OR**

<u>Example</u>

Given Boolean Function

F(A,B,C)=Σ(1,3,5,7)

Step 1: Draw 3-Variable K-Map

(Variables: A for rows, BC for columns)

	BC			
	00	01	11	10
A=0	0	1	1	0
A=1	0	1	1	0

Step 2: Mark Minterms

- Minterm 1 → (A=0, B=0, C=1)
- Minterm 3 → (A=0, B=1, C=1)
- Minterm 5 → (A=1, B=0, C=1)
- Minterm 7 → (A=1, B=1, C=1)

All these minterms correspond to **C = 1**.

Step 3: Group the 1s

All four 1s form **one group of 4**:

	BC			
	00	01	11	10
A=0	0	[1]	[1]	0
A=1	0	[1]	[1]	0

Step 4: Write the Simplified Expression

- Variable **C** remains constant (C = 1)
- Variables **A and B change**, so they are eliminated

Simplified Boolean Function

F = C

Verification Using Truth Table

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Advantages of K-Map Simplification

- Simple and visual method
- Eliminates algebraic complexity
- Produces minimal expressions
- Useful for up to **4 variables**

Limitations of K-Map

- Becomes complex for more than 4 variables
- Not suitable for automation
- Manual errors possible for large maps

6. Useful Combinational Circuits

Useful combinational circuits are digital circuits in which the **output depends only on the present input values**. These circuits **do not contain memory**, **do not require clock signals**, and are designed using **logic gates**. They are mainly used for **arithmetic operations, data selection, encoding, and decoding** in digital systems.

6.1 Half Adder

A Half Adder is a combinational circuit that adds two 1-bit binary numbers and produces two outputs:

- **Sum (S)**
- **Carry (C)**

It is called a *half* adder because it does not handle carry from a previous stage.

Inputs and Outputs

- Inputs: **A, B**
- Outputs: **Sum (S), Carry (C)**

Boolean Expressions

$S = A \oplus B$
$C = A \cdot B$

Truth Table

A	B	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Logic Diagram

```
A ─┐
   ├─ XOR ──> Sum
B ─┘

A ─┐
   ├─ AND ──> Carry
B ─┘
```

Limitation

Half adder cannot be used alone for multi-bit addition because it does not accept carry-in.

Applications

Basic arithmetic circuits, Used in the construction of full adders

6.2 Full Adder

A **Full Adder** is a combinational circuit that **adds three 1-bit binary inputs**:

- **A**
- **B**

- **Carry-in (Cin)**

It produces:

- **Sum (S)**
- **Carry-out (Cout)**

Inputs and Outputs

- Inputs: **A, B, Cin**
- Outputs: **Sum (S), Carry (Cout)**

Boolean Expressions

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + BC_{in} + AC_{in}$$

Truth Table

A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Logic Diagram (Using Two Half Adders)

```
        A ──┐
            ├─ Half Adder ── S1 ──┐
        B ──┘                     ├─ Half Adder ──> Sum
                                  │
        Cin ──────────────────────┘

Carry = C1 OR C2
```

Applications

Binary addition, ALU (Arithmetic Logic Unit), Processors and digital computers

<u>6.3 Subtractor</u>

A **Subtractor** is a combinational circuit that **performs binary subtraction**.

Half Subtractor

A Half Subtractor subtracts one 1-bit binary number from another and produces:

- Difference (D)
- Borrow (B)

Inputs and Outputs

- Inputs: A (minuend), B (subtrahend)
- Outputs: Difference, Borrow

Boolean Expressions

$$D = A \oplus B$$

$$Borrow = A'B$$

Truth Table

A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Logic Diagram

```
A ──┐
    ├── XOR ──> Difference
B ──┘

A ── NOT ──┐
           ├── AND ──> Borrow
B ─────────┘
```

6.4 Multiplexer (MUX)

A **Multiplexer (MUX)** is a combinational circuit that **selects one of many input lines** and sends it to a **single output line** based on **select inputs**.
It is also called a **data selector**.

4-to-1 Multiplexer

- Inputs: A0, A1, A2, A3
- Select lines: S1, S0
- Output: Y

Boolean Expression

$$Y = A_0 S_1' S_0' + A_1 S_1' S_0 + A_2 S_1 S_0' + A_3 S_1 S_0$$

Truth Table

S1	S0	Output
0	0	A0
0	1	A1
1	0	A2
1	1	A3

Block Diagram

```
A0 ─┐
A1 ─┤
A2 ─┤── MUX ──> Y
A3 ─┘

      ↑   ↑
      S1  S0
```

Applications

Data routing, CPU instruction selection, Signal switching

6.5 Demultiplexer (DEMUX)

A **Demultiplexer (DEMUX)** is a combinational circuit that **takes a single input** and distributes it to **one of many output lines** based on select inputs.
It is also known as a **data distributor**.

1-to-4 DEMUX

S1	S0	Active Output
0	0	Y0
0	1	Y1
1	0	Y2
1	1	Y3

Diagram

```
        ┌──> Y0
Input ──┤
        ├──> Y1
        ├──> Y2
        └──> Y3
          ↑  ↑
          S1 S0
```

Applications

Data distribution, Memory selection, Serial-to-parallel conversion

6.6 Encoder and Decoder

Encoder

An **Encoder** is a combinational circuit that **converts an active input line into a binary coded output**.

Example: 4-to-2 Encoder

Input Active	Output
D0	0
D1	1
D2	10
D3	11

Applications

Keyboard encoding, Interrupt systems, Data compression

Decoder

A **Decoder** is a combinational circuit that **converts binary input into one active output line**.

Example: 2-to-4 Decoder

A	B	Y0	Y1	Y2	Y3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Applications

Memory address decoding, Instruction decoding, Chip selection

7. Programmable Combinational Circuits

Programmable Combinational Circuits are digital circuits whose logic function can be **programmed by the user** rather than being fixed at design time. They implement **combinational logic only** (no memory elements like flip-flops).

Key Idea

Instead of building a circuit with fixed gates, programmable combinational circuits use arrays of logic elements (AND/OR/NOT) that can be configured to realize different Boolean functions.

Main Types

1. ROM (Read Only Memory)

- Uses a **fixed AND array** (decoder) and a **programmable OR array**
- Inputs act as address lines; outputs store logic values
- Can implement any combinational function
- **Disadvantage:** inefficient for large input sizes

2. PLA (Programmable Logic Array)

- **Programmable AND array**
- **Programmable OR array**
- Most flexible
- Can minimize hardware by sharing product terms
- **Disadvantage:** more complex and slower than PAL

3. PAL (Programmable Array Logic)

- **Programmable AND array**
- **Fixed OR array**
- Faster and cheaper than PLA
- Less flexible than PLA

4. GAL (Generic Array Logic)

- Improved version of PAL
- Reprogrammable (EEPROM based)
- Widely used in modern designs

Comparison Table

Device	AND Array	OR Array	Flexibility	Speed
ROM	Fixed	Programmable	Low	Medium
PLA	Programmable	Programmable	High	Low
PAL	Programmable	Fixed	Medium	High
GAL	Programmable	Fixed	Medium	High

Advantages

- Easy to modify logic

- Reduced design time
- Suitable for prototyping
- Fewer ICs required

Disadvantages

- Limited complexity
- Not suitable for sequential logic
- Higher cost than simple gates for small circuits

Applications

- Address decoding
- Code converters
- Arithmetic logic functions
- Control logic

8. Timing and Control

Timing and Control circuits coordinate the **sequence of operations** in a digital system by generating control signals at the **correct time**. They ensure that data flows correctly between system components such as registers, ALUs, and memory.

Timing

Timing defines **when** an operation occurs.

Key Elements

- **Clock Signal**
    - A periodic waveform (usually square wave)
    - Synchronizes all system operations
- **Clock Period (T)**: Time for one complete cycle
- **Clock Frequency (f)**: ($f = \frac{1}{T}$)

Types of Timing

1. **Synchronous Timing**
    - All operations triggered by a common clock
    - Predictable and reliable
2. **Asynchronous Timing**
    - No global clock

    - Operations triggered by events
    - Faster but more complex

<u>Control</u>

Control determines **what operation** is performed.

Control Signals

- Enable / Disable
- Load / Clear
- Read / Write
- Select signals

Control Unit

- Generates control signals based on:
    - Current instruction
    - Timing signals
    - Status flags

Timing and Control Unit

Combines timing and control functions.

Functions

- Generates **clock pulses**
- Produces **control signals**
- Sequences micro-operations
- Ensures proper data transfer

Micro-Operations

Basic operations performed on data:

- Register transfer
- Arithmetic operations
- Logical operations
- Shift operations

Example:

T1: MAR ← PC
T2: MDR ← Memory[MAR]
T3: IR ← MDR

Applications

- CPUs
- Microcontrollers
- Digital communication systems
- Control systems

Advantages

- Proper synchronization
- Prevents data corruption
- Improves system reliability

9. Latches

A **latch** is a basic memory element that stores one bit of information.
It is **level-triggered**.

Key Characteristics

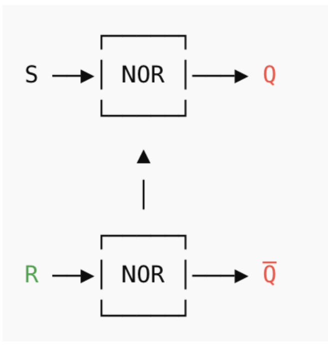
- Stores **1 bit**
- **Level-triggered** (not edge-triggered)
- Output changes **as long as enable is active**
- Basic memory element

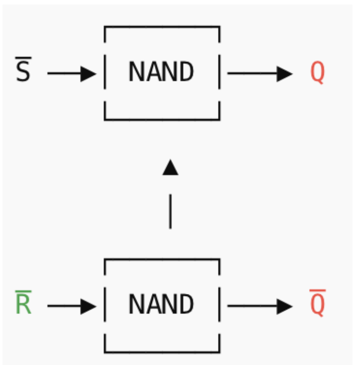
Types of Latches

1. SR (Set–Reset) Latch

Built using **NOR** or **NAND** gates.

NOR-based SR Latch



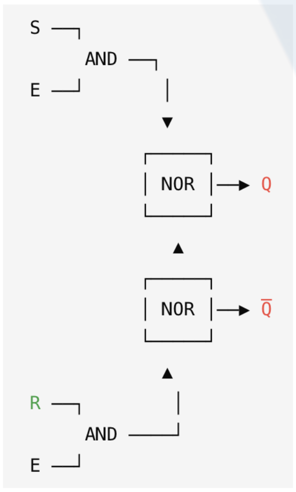


S	R	Q (Next State)	Operation
0	0	Q (no change)	Hold
1	0	1	Set
0	1	0	Reset
1	1	Invalid	Forbidden

⚠️ Invalid state causes unpredictable output.

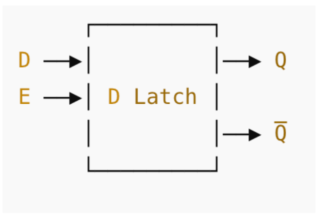
2. Gated SR Latch

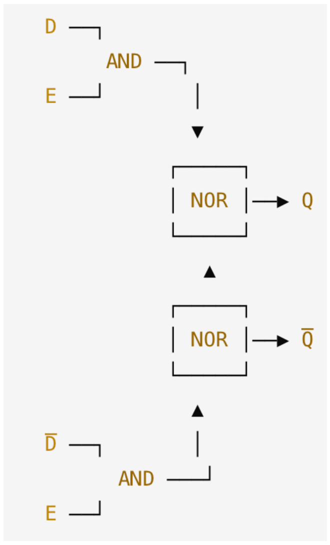
- Uses an **Enable (E)** signal
- Latch responds only when **E = 1**
- Prevents accidental state change



3. D (Data) Latch

- Eliminates invalid state
- Inputs: **D (data)** and **Enable**
- When **Enable = 1**, output follows D
- When **Enable = 0**, output holds





Enable	D	Q
0	X	Hold
1	0	0
1	1	1

4. JK Latch

- Improved SR latch
- No invalid state
- When **J = K = 1**, output toggles

J	K	Q (Next)
0	0	Hold
0	1	Reset
1	0	Set
1	1	Toggle

Latch vs Flip-Flop

Feature	Latch	Flip-Flop
Triggering	Level	Edge
Control	Enable	Clock
Speed	Faster	Slower
Stability	Less	More

Advantages

- Simple design
- Fast response
- Low power consumption

Disadvantages

- Glitches possible
- Not suitable for synchronous systems
- Timing issues

Applications

- Temporary data storage, Control circuits, Asynchronous systems ,Debouncing switches

10. Flip-Flops

A **flip-flop** is a **clock-controlled, bistable sequential logic circuit** capable of storing **one bit of binary information (0 or 1)**.
It changes its output **only at a specific moment of the clock signal (edge-triggered)** and retains its state until the next clock pulse.

Flip-flops are the **basic memory elements** used in **registers, counters, memory units, and control circuits**.

Characteristics of Flip-Flops

Feature	Description
Storage Capacity	Stores 1 bit
Triggering	Edge-triggered
Memory	Yes
Clock Dependency	Required
Feedback	Present

Difference Between Latch and Flip-Flop

Feature	Latch	Flip-Flop
Sensitivity	Level-sensitive	Edge-triggered
Clock	Not mandatory	Mandatory
Output Change	Any time during enable	Only at clock edge
Reliability	Less	More

Types of Flip-Flops

1. SR (Set-Reset) Flip-Flop

SR flip-flop has **two inputs**:

- **S (Set)**
- **R (Reset)**

It is used to **set or reset the output**.

Truth Table (Clock Enabled)

S	R	Q(n+1)	Operation
0	0	Q(n)	No Change
0	1	0	Reset
1	0	1	Set

1	1	Invalid	Not Allowed

Characteristic Equation

$$Q(n+1) = S + \overline{R}Q(n)$$

Limitation

The condition **S = R = 1** produces an **invalid state**, which makes SR flip-flop unreliable.

2. JK Flip-Flop

JK flip-flop is an **improved version of SR flip-flop** where the invalid condition is removed.

Inputs:

- **J (Set)**
- **K (Reset)**

Truth Table

J	K	Q(n+1)	Operation
0	0	Q(n)	No Change
0	1	0	Reset
1	0	1	Set
1	1	Toggle	Complement

Characteristic Equation

$$Q(n+1) = J\overline{Q(n)} + \overline{K}Q(n)$$

Advantage

Eliminates invalid state and allows **toggle operation**, making it suitable for **counters**.

Disadvantage

At high clock speeds, **race around condition** may occur.

3. D (Data / Delay) Flip-Flop

D flip-flop has **a single input (D)** and stores the value present at input **D** at the **clock edge**.

Truth Table

D	Q(n+1)
0	0

1	1

Characteristic Equation

$$Q(n+1) = D$$

Advantages

- No invalid condition
- Simple operation
- Widely used in **registers and memory**

Example

If **D = 1** at the rising edge of clock, output **Q becomes 1** and remains unchanged until next clock pulse.

4. T (Toggle) Flip-Flop

T flip-flop toggles the output when **T = 1** and holds the state when **T = 0**.

Truth Table

T	Q(n+1)
0	Q(n)
1	$\overline{Q(n)}$

Characteristic Equation

$$Q(n+1) = T \oplus Q(n)$$

Application

Used in **binary counters** and $\overline{Q(n)}$ frequency division.

Edge Triggering in Flip-Flops

Types

1. **Positive Edge Triggered** (↑)
2. **Negative Edge Triggered** (↓)

Flip-flop changes state **only at the edge**, not throughout the clock level.

Timing Parameters of Flip-Flops

Parameter	Description
Setup Time	Data stable before clock

Hold Time	Data stable after clock
Propagation Delay	Clock to output delay

Applications of Flip-Flops

- Registers, Counters, Memory Units, Shift Registers, Control Circuits, Digital Clocks

Comparison of Flip-Flops

Feature	SR	JK	D	T
Inputs	2	2	1	1
Invalid State	Yes	No	No	No
Toggle	No	Yes	No	Yes
Complexity	Low	Medium	Low	Low

11. Registers

A **register** is a **group of flip-flops** used to store **binary information (multi-bit data)**.
Each flip-flop stores one bit (0 or 1).
Thus, a **n-bit register** can store **n bits of data**.

Registers are one of the most important elements of the CPU as they provide **temporary storage** and **fast access** to data during processing.

Purpose of Registers

- To **store binary data temporarily** during computation.
- To **hold operands**, **intermediate results**, and **addresses** during instruction execution.
- To act as **buffers** between CPU and memory.
- To **shift or manipulate data** using shift registers.

Structure of a Register

Each register is made up of **flip-flops connected in parallel** so that each bit is stored simultaneously.

Example: 4-bit Register

A 4-bit register consists of **4 flip-flops (FF0, FF1, FF2, FF3)**, each capable of storing one bit.

Flip-Flop	Bit Stored	Output
FF0	Least Significant Bit (LSB)	Q0
FF1	Next Bit	Q1

FF2	Next Bit	Q2
FF3	Most Significant Bit (MSB)	Q3

Binary Number Stored: Q3 Q2 Q1 Q0

If Q3Q2Q1Q0 = 1010 → Decimal 10

Types of Registers

Registers are classified according to how data is entered or removed.

Type	Full Form	Description	Example Application
SISO	Serial Input Serial Output	Data enters and leaves one bit at a time	Serial communication
SIPO	Serial Input Parallel Output	Data enters serially and appears on all outputs simultaneously	Serial-to-parallel conversion
PISO	Parallel Input Serial Output	Data enters simultaneously and exits one bit at a time	Parallel-to-serial conversion
PIPO	Parallel Input Parallel Output	Data enters and leaves simultaneously	High-speed data transfer

A. Serial-In Serial-Out (SISO) Register

- Data bits are entered **one at a time** with each clock pulse.
- Output bits appear **one at a time** as well.

Clock Pulse	Input	Output
1	1	-
2	0	1
3	1	0
4	0	1

Application: Serial communication systems (UART).

B. Serial-In Parallel-Out (SIPO) Register

- Data enters serially (bit-by-bit).
- After n clock pulses, all bits are available in parallel.

Example:

Clock	Input	Q3	Q2	Q1	Q0

1	1	0	0	0	1
2	0	0	0	1	0
3	1	0	1	0	1
4	0	1	0	1	0

Use: Converts serial input data to parallel output.

C. Parallel-In Serial-Out (PISO) Register

- All bits are entered **simultaneously (parallel)**.
- Data is then shifted **out serially (bit-by-bit)**.

Step	Parallel Input	Serial Output
Load	1101	-
Shift 1	-	1
Shift 2	-	0
Shift 3	-	1
Shift 4	-	1

Use: Parallel to serial conversion (e.g., printer communication).

D. Parallel-In Parallel-Out (PIPO) Register

- All bits are entered and retrieved simultaneously.
- Fastest data transfer method.

Clock	Input (ABCD)	Output (Q3Q2Q1Q0)
1	1011	1011

Use: Inside CPU for storing and transferring full words (e.g., 8-bit accumulator).

Register Applications

1. **Temporary data storage**
2. **Data shifting and rotation**
3. **Serial/parallel data conversion**
4. **Instruction registers** in processors
5. **Buffer registers** in I/O devices

12. Counters

A **counter** is a **sequential circuit** that **counts clock pulses** or **events**.
It is made of **flip-flops connected in sequence**, where each flip-flop represents one bit of the count.

Counters are essential for time measurement, event tracking, digital clocks, and control applications.

Types of Counters

1. Asynchronous Counter (Ripple Counter)

- Each flip-flop is **triggered by the output of the previous one**.
- The clock signal is applied only to the first flip-flop.
- Propagation delay accumulates as count progresses.

Example: 3-bit Ripple Counter

Clock Pulse	Q2	Q1	Q0	Decimal
0	0	0	0	0
1	0	0	1	1
2	0	1	0	2
3	0	1	1	3
4	1	0	0	4
5	1	0	1	5
6	1	1	0	6
7	1	1	1	7

Explanation:
Each flip-flop toggles when its previous output changes from 1 to 0.

2. Synchronous Counter

- All flip-flops are triggered **simultaneously** by the same clock pulse.
- Count changes occur at the same instant → faster and more reliable.

Clock	Q2	Q1	Q0	Decimal
1	0	0	1	1
2	0	1	0	2
3	0	1	1	3

4	1	0	0	4
5	1	0	1	5

3. Up Counter

Counts from **000 → 111** (binary increment).
Used in timers, event counting, etc.

4. Down Counter

Counts in **reverse order**, from **111 → 000**.
Used in countdown timers.

5. Up/Down Counter

Can operate in both up and down modes depending on control input.

6. Ring Counter

A circular shift register where the output of the last flip-flop is fed into the first one.

Example:
For a 4-bit ring counter → Sequence: 1000, 0100, 0010, 0001

7. Johnson Counter

Modified ring counter where complement of last flip-flop is fed to first input.

Example:
4-bit Johnson counter → 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001

Applications of Counters

- Digital clocks and timers
- Frequency dividers
- Memory address counting
- Event counters in processors
- Pulse and signal generation

13. Sequential Circuits

Sequential circuits are **digital circuits** whose **output depends on both current inputs and past outputs (previous states)**.
Unlike combinational circuits, they have **memory elements (flip-flops)** that store the state of the system.

Types of Sequential Circuits

Type	Trigger	Description	Example
Synchronous	Clock-based	All flip-flops triggered simultaneously by a clock pulse	Registers, Synchronous Counters
Asynchronous	Input-based	Flip-flops change state as soon as input changes	Ripple Counters, Timers

Components of Sequential Circuits

1. **Flip-Flops:** Memory elements to store binary information.
2. **Logic Gates:** Implement combinational logic to determine next state.
3. **Clock:** Provides timing synchronization.
4. **Feedback Paths:** Allow previous output to affect next state.

Basic Model

$$\begin{tikzcd}
\text{INPUT} \arrow[r] & \text{COMBINATIONAL CIRCUIT} \arrow[r] \arrow[d] & \text{OUTPUT} \\
& \text{MEMORY ELEMENTS} &
\end{tikzcd}$$

Sequential Circuit Operation Example

Let’s consider a simple **2-bit binary counter** (synchronous).

Clock Pulse	State (Q1Q0)	Decimal Output
0	0	0
1	1	1
2	10	2
3	11	3
4	0	0 (Repeat)

Explanation:

- On each clock pulse, output advances by 1.
- Flip-flops retain previous state to determine next output.

State Table

Present State	Input	Next State	Output
0	1	1	0
1	1	10	0
10	1	11	0
11	1	0	1

State Diagram

Represented as circles (states) and arrows (transitions).

```
(00) → (01) → (10) → (11)
 ↑             ↓
 └──────────────────────┘
```

Applications of Sequential Circuits

- Counters and timers
- Traffic light controllers
- Elevator control systems
- Microprocessor control units
- Digital communication protocols

Summary

Concept	Description	Examples
Register	Stores multi-bit data using flip-flops	SISO, SIPO, PISO, PIPO
Counter	Counts clock pulses or events	Asynchronous, Synchronous, Johnson
Sequential Circuit	Output depends on current and past input	Counters, Registers, Control Logic

PART B – ARITHMETIC/LOGIC UNIT (ALU)

The **Arithmetic Logic Unit (ALU)** is a core component of the CPU that performs all arithmetic and logical operations.

1. Number Representation

Computers use **binary numbers** to represent and process data.

1.1 Number Systems

System	Base	Digits	Example (Decimal 25)
Binary	2	0,1	11001
Octal	8	0–7	31
Decimal	10	0–9	25
Hexadecimal	16	0–9, A–F	19

2. Binary Arithmetic

Addition Rules

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Subtraction Rules

A	B	Borrow	Difference
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Multiplication Example

101 × 11 = 1111

```
   101
×   11
--------
   101   ← (101 × 1)
+ 1010   ← (101 × 1, shifted left by 1 position)
--------
  1111
```

```
        1 1 0 1 1
    x 1 1 1 1 0
      1 0 0 0 0 0
    1 1 1 0 1 1
    1 1 0 1 1
  1 0 1 0 0 0 1 0
```

Division Example

$$0 \div 0 = \text{Undefined}$$

$$0 \div 1 = 0 \text{ with Remainder } = 0$$

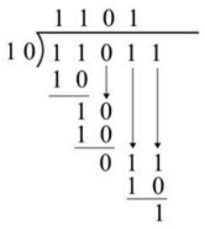
$$1 \div 0 = \text{Undefined}$$

$$1 \div 1 = 1 \text{ with Remainder } = 0$$

11011 ÷ 10 = 1101

The binary division of 11011 by 10 is explained below −

11011 10 = 1101



In this example, the quotient is 1101 and the remainder is 1.

3. Floating-Point Arithmetic

Floating-point arithmetic is a method used by computers to represent and process **real numbers (numbers with fractional parts)** in binary form.
It allows representation of **very large and very small numbers** that cannot be handled efficiently using fixed-point representation.

Need for Floating-Point Representation

Fixed-point representation has:

- Limited range
- Fixed precision

Floating-point representation overcomes these limitations by storing numbers in **scientific notation** form.

General Floating-Point Representation

A floating-point number is represented as: $\text{Number} = (-1)^S \times M \times 2^E$

Where:

- **S** = Sign bit (0 → positive, 1 → negative)
- **M** = Mantissa (significand)
- **E** = Exponent

General Format

Sign	Exponent	Mantissa
1 bit	k bits	n bits

IEEE-754 Floating-Point Standard

The **IEEE-754 standard** defines how floating-point numbers are represented and processed in computers.

Single Precision (32-bit)

Field	Bits
Sign	1
Exponent	8
Mantissa	23

Double Precision (64-bit)

Field	Bits
Sign	1
Exponent	11
Mantissa	52

Exponent Bias

IEEE-754 uses a **biased exponent** to represent both positive and negative exponents.

Precision	Bias
Single	127
Double	1023

Actual Exponent = Stored Exponent - Bias

Representation of a Floating-Point Number (Example)

Represent **+5.75** in IEEE-754 single precision.

Step 1: Convert to Binary

$5.75_{10} = 101.11_2$

Step 2: Normalize

$101.11 = 1.0111 \times 2^2$

Step 3: Identify Components

- Sign bit (S) = 0
- Mantissa (M) = 01110000000000000000000
- Exponent = 2 + 127 = 129 → 10000001

Final Representation

0 | 10000001 | 01110000000000000000000

4. Floating-Point Arithmetic Operations

4.1 Floating-Point Addition

Steps

1. Compare exponents
2. Align mantissas (shift smaller exponent)
3. Add or subtract mantissas
4. Normalize result
5. Round if required

Example

2.5 + 1.25

$2.5 = 1.01 \times 2^1$
$1.25 = 1.01 \times 2^0$

Align exponents:

1.01×2^1
0.101×2^1

Add mantissas:

```
1.01
+0.101
------
```
1.111×2^1

4.2 Floating-Point Subtraction

Performed same as addition, except mantissas are **subtracted** after exponent alignment.

4.3 Floating-Point Multiplication

Steps

1. Multiply mantissas

2. Add exponents
3. Normalize result
4. Adjust sign

Example

(1.5) × (2.0)

$1.5 = 1.1 \times 2^0$
$2.0 = 1.0 \times 2^1$

Multiply mantissas:

$1.1 \times 1.0 = 1.1$

Add exponents:

$0 + 1 = 1$

Final result:

$1.1 \times 2^1 = 3.0$

4.4 Floating-Point Division

Divide: 6.0 ÷ 1.5

Step 1: Convert to Normalized Binary Form

$6.0 = 1.1 \times 2^2$
$1.5 = 1.1 \times 2^0$

Step 2: Divide Mantissas

$1.1 \div 1.1 = 1.0$

Step 3: Subtract Exponents

$2 - 0 = 2$

Step 4: Normalize Result

1.0×2^2 (already normalized)

Final Result

$6.0 \div 1.5 = 4.0$

4.5. Normalization

Normalization is the process of adjusting the mantissa and exponent of a floating-point number so that the mantissa lies in the standard form:

[1.xxxxx]

where only one non-zero digit appears to the left of the binary point.

4.6. Rounding Modes

Mode	Description
Round to Nearest	Default, most accurate
Round toward Zero	Truncation
Round toward +∞	Ceiling
Round toward −∞	Floor

4.7. Special Floating-Point Values

Value	Meaning
+0, −0	Signed zero
+∞, −∞	Overflow
NaN	Not a Number
Denormalized	Very small numbers

4.8. Errors in Floating-Point Arithmetic

Rounding Error

A rounding error occurs when a real number cannot be represented exactly in the floating-point format and must be approximated to the nearest representable value.

Overflow

Overflow occurs when the result of a floating-point operation is too large to be represented within the available exponent range.

Underflow

Underflow occurs when the result of a floating-point operation is too small (close to zero) to be represented in normalized form.

UNIT 2– Register Transfer Language & Micro-operations

Register Transfer Language and Micro-operations: Concept of bus, data movement among registers, a language to represent conditional data transfer, data movement from/to memory. Design of Arithmetic & Logic Unit and Control Unit Control design hardwired control, micro programmed arithmetic and logical operations along with register transfer, timing in register.

1. Introduction

A digital computer executes instructions by performing a sequence of simple internal operations. These internal operations occur at the hardware level and involve registers, buses, arithmetic circuits, and control signals. To describe such operations precisely, **Register Transfer Language (RTL)** is used. RTL provides a symbolic and systematic method to represent how data is transferred and processed inside the CPU.

Micro-operations are the smallest functional steps carried out by the processor. Each micro-operation is synchronized with the system clock and forms the building block of instruction execution.

2. Register Transfer Language (RTL)

Register Transfer Language is a notation used to describe the transfer of data between registers and the execution of arithmetic or logical operations on that data.

Key Characteristics of RTL

- Describes operations at register level
- Independent of programming language
- Closely related to hardware structure
- Used for datapath and control unit design

Basic RTL Statement

Destination ← Source
Example

R1 ← R2
This statement means that the contents of register R2 are transferred into register R1 at the next clock pulse.

3. Micro-operations

Micro-operations are elementary operations performed on data stored in registers. An instruction is executed as a sequence of micro-operations.

Types of Micro-operations

1. Register transfer micro-operations
2. Arithmetic micro-operations

3. Logical micro-operations

4. Shift micro-operations

Example (Instruction Decomposition)

Instruction: ADD R1, R2

Micro-operations:

T1: A ← R1
T2: B ← R2
T3: R1 ← A + B
Each step represents a single micro-operation executed in one clock cycle.

4. Concept of Bus

A bus is a group of parallel wires that provides a shared communication path between registers and other components of the CPU.

Types of Buses in a Computer System

A computer system generally uses three types of buses:

1. **Data Bus**
   The data bus carries actual data between registers, the ALU, and memory. Its width determines how much data can be transferred at one time.

2. **Address Bus**
   The address bus carries memory addresses from the CPU to the memory unit. It is unidirectional and determines the maximum addressable memory size.

3. **Control Bus**
   The control bus carries control signals such as read, write, and clock signals. These signals coordinate and control the operation of all components connected to the bus.

Role of Bus in Register Transfer

In register transfer operations, the bus acts as an intermediate path. When a register transfer such as

R2 ← R1
is executed, register R1 places its contents on the bus, and register R2 loads the data from the bus on the next clock edge. This mechanism ensures reliable and synchronized data transfer.

Importance of Bus System

The bus system:

- Reduces hardware complexity
- Improves scalability of the processor
- Enables efficient data communication

- Plays a key role in datapath and control unit design

5. Data Movement Among Registers

Data movement among registers is a fundamental operation in a computer system and forms the basis of instruction execution. Registers are high-speed storage elements located within the CPU, and efficient transfer of data between them is essential for achieving high performance.

Register-to-register data transfer is typically carried out using a **common internal bus**. When a data transfer operation is required, the control unit selects one register as the source and enables it to place its contents on the bus. At the same time, the destination register is enabled through a load control signal. On the active edge of the system clock, the destination register captures the data present on the bus.

This process ensures that data transfer is **synchronous**, reliable, and free from conflicts. To avoid ambiguity and data corruption, only one register is allowed to place its contents on the bus at any given time. The control unit strictly regulates this process using selection and enable signals.

Register Transfer Language Representation

Register-to-register data movement is expressed using Register Transfer Language (RTL). A typical RTL statement is:

$R2 \leftarrow R1$
This statement indicates that the contents of register R1 are transferred into register R2 during a clock cycle.

Control Signals in Register Transfer

The movement of data among registers is governed by several control signals generated by the control unit:

- **Source Select Signal** determines which register places its data on the bus.
- **Load Enable Signal** allows the destination register to accept data.
- **Clock Signal** synchronizes the data transfer operation.

These signals work together to ensure that the transfer occurs correctly and at the appropriate time.

Example of Register-to-Register Data Transfer

Consider the operation:

$R3 \leftarrow R1$
The control unit first selects register R1 as the source and enables it to place its contents on the bus. Simultaneously, the load signal of register R3 is activated. When the clock pulse occurs, register R3 captures the data from the bus, completing the transfer.

Significance of Data Movement Among Registers

Data movement among registers plays a critical role in processor operation:

- It reduces the need for frequent memory access.
- It improves instruction execution speed.
- It enables arithmetic and logical operations within the ALU.
- It supports efficient instruction sequencing and control.

6. Conditional Data Transfer

Conditional data transfer is a register transfer operation that takes place only when a specified condition is satisfied. Unlike unconditional register transfers, conditional transfers depend on the outcome of previous operations and are essential for decision-making and control flow in a computer system.

In a processor, conditions are generally derived from **status flags** generated by the Arithmetic and Logic Unit (ALU). These flags indicate specific characteristics of the result of an operation, such as whether the result is zero, whether a carry was generated, or whether an overflow occurred. The control unit evaluates these flags to determine whether a particular data transfer should occur.

Register Transfer Language Representation

Conditional data transfer is expressed in Register Transfer Language (RTL) using the following notation:

Condition : Destination ← Source
This notation means that the data transfer from the source register to the destination register will occur only if the specified condition evaluates to true.

Example

(Z = 1) : PC ← AR
This statement indicates that the contents of the Address Register (AR) are transferred to the Program Counter (PC) only if the Zero flag is set.

Role of Status Flags in Conditional Transfer

Status flags are binary indicators that reflect the outcome of arithmetic or logical operations. Common status flags include:

- **Zero Flag (Z):** Set when the result of an operation is zero
- **Carry Flag (C):** Set when a carry is generated in arithmetic operations
- **Sign Flag (S):** Indicates the sign of the result
- **Overflow Flag (V):** Indicates signed arithmetic overflow

These flags are stored in a flag register and are continuously monitored by the control unit to enable or disable conditional data transfers.

Working of Conditional Data Transfer

When a conditional transfer instruction is executed, the control unit performs the following steps:

1. Evaluates the specified condition based on the status flags.
2. If the condition is true, enables the source register to place its data on the bus.
3. Activates the load signal of the destination register.
4. Transfers the data on the next clock pulse.

If the condition is false, the transfer is skipped, and the processor proceeds to the next instruction.

Applications of Conditional Data Transfer

Conditional data transfer is widely used in processor operations such as:

- Branch and jump instructions
- Loop control mechanisms
- Decision-making in programs
- Error and exception handling

Without conditional data transfer, it would not be possible to implement control structures such as if-else statements and loops at the hardware level.

Importance in Instruction Execution

Conditional data transfer enables the processor to modify the sequence of instruction execution based on computed results. This capability is fundamental to program control flow and efficient execution of algorithms.

7. Data Transfer Between Memory and Registers

Data movement between the CPU and main memory is an essential part of instruction execution in a computer system. Unlike register-to-register transfers, memory access operations involve additional control signals and require more time due to the relatively slower speed of main memory. These operations are performed using specific registers and follow a well-defined sequence of micro-operations.

The CPU communicates with memory through two special-purpose registers: the **Address Register (AR)** and the **Data Register (DR)**. The Address Register holds the memory address of the data to be accessed, while the Data Register temporarily stores the data being transferred between the CPU and memory.

Memory Read Operation

A memory read operation transfers data from a memory location into a CPU register. This operation is required during instruction fetch and when an instruction needs data stored in memory.

The Register Transfer Language (RTL) representation of a memory read operation is:

$DR \leftarrow M[AR]$
This statement indicates that the contents of the memory location specified by the address in AR are transferred into the Data Register (DR).

Sequence of Operations in Memory Read

During a memory read operation, the following steps occur:

1. The memory address is loaded into the Address Register (AR).
2. The control unit activates the memory read control signal.
3. The memory places the data stored at the specified address onto the data bus.
4. The data is loaded into the Data Register (DR) on the active clock edge.

This sequence ensures reliable and synchronized data transfer from memory to the CPU.

Memory Write Operation

A memory write operation transfers data from the CPU to a memory location. This operation is used when the result of a computation must be stored in memory.

The RTL representation of a memory write operation is:

$M[AR] \leftarrow DR$

This statement indicates that the contents of the Data Register (DR) are written into the memory location specified by the address in AR.

Sequence of Operations in Memory Write

The memory write operation proceeds as follows:

1. The target memory address is placed in the Address Register (AR).
2. The data to be stored is loaded into the Data Register (DR).
3. The control unit activates the memory write control signal.
4. The data in DR is written to the specified memory location.

Control Signals in Memory Transfer

Memory access operations are controlled by specific signals generated by the control unit:

- **Read Signal** indicates a memory read operation.
- **Write Signal** indicates a memory write operation.
- **Clock Signal** synchronizes the operation.

These signals ensure that memory operations occur correctly and without data corruption.

Difference Between Register Transfer and Memory Transfer

Register-to-register transfers are faster because registers are located within the CPU and operate at high speed. Memory transfers are slower due to memory access delays and therefore require careful timing and control.

Importance of Memory Data Transfer

Data movement from and to memory:

- Enables instruction fetch and execution
- Allows storage of program data and results
- Supports large data handling beyond register capacity

8. Arithmetic and Logic Unit (ALU)

The Arithmetic and Logic Unit (ALU) is a fundamental component of the Central Processing Unit (CPU) responsible for performing all arithmetic and logical operations required during instruction execution. It acts as the computational core of the processor and plays a crucial role in data processing and decision-making.

The ALU is a **combinational logic circuit**, meaning that its output depends solely on its current inputs and control signals, without requiring any internal memory. It receives input data from registers, performs the specified operation, and produces the result, which is then stored back into a register.

Functions of the ALU

The ALU performs two primary categories of operations:

Arithmetic Operations

Arithmetic operations involve numerical calculations on binary data. Common arithmetic operations include:

- Addition
- Subtraction
- Increment
- Decrement

These operations are essential for calculations, address computation, and program control.

Logical Operations

Logical operations manipulate data at the bit level. Common logical operations include:

- AND
- OR
- XOR
- NOT

Logical operations are widely used for masking, comparison, and decision-making processes.

Inputs to the ALU

The ALU typically receives the following inputs:

- **Operand A** from a register
- **Operand B** from another register or constant
- **Control signals** specifying the operation to be performed

The control signals are generated by the control unit based on the instruction being executed.

Outputs of the ALU

The ALU produces:

- **Result of the operation**, which is sent to a destination register
- **Status flags**, which indicate specific conditions resulting from the operation

Status Flags Generated by the ALU

Status flags are single-bit indicators stored in a flag register. Common flags include:

- **Zero Flag (Z):** Set if the result of the operation is zero
- **Carry Flag (C):** Set if a carry is generated in arithmetic operations
- **Sign Flag (S):** Indicates the sign of the result
- **Overflow Flag (V):** Indicates signed arithmetic overflow

These flags are essential for conditional operations and control flow.

ALU Control and Operation Selection

The specific operation performed by the ALU is determined by control signals from the control unit. These signals select the appropriate arithmetic or logical function within the ALU.

For example:

- An ADD control signal activates the adder circuit
- An AND control signal activates the logical AND circuit

Only one operation is performed at a time, based on the active control signal.

Role of ALU in Register Transfer Operations

The ALU works closely with registers during instruction execution. Data is transferred from registers to the ALU, processed, and then transferred back to a register.

Example RTL representation:

$R3 \leftarrow R1 + R2$

This statement indicates that the contents of registers R1 and R2 are added in the ALU, and the result is stored in register R3.

9. Control Unit

The Control Unit is a vital component of the Central Processing Unit (CPU) that directs and coordinates the activities of all other hardware components. While the Arithmetic and Logic Unit performs computations and registers store data, the Control Unit ensures that each operation occurs in the correct sequence and at the correct time.

The Control Unit does not process data directly. Instead, it generates control signals that regulate data movement, arithmetic and logical operations, and memory access during instruction execution.

Functions of the Control Unit

The primary functions of the Control Unit include:

- Fetching instructions from memory
- Decoding the instruction to determine the required operation
- Generating control signals for registers, ALU, and memory
- Sequencing micro-operations in synchronization with the system clock

Through these functions, the Control Unit orchestrates the entire instruction execution process.

Inputs to the Control Unit

The Control Unit receives several inputs that influence its operation:

- **Instruction Opcode** from the Instruction Register (IR)
- **Status Flags** from the ALU
- **Clock Signal** to synchronize operations
- **Interrupt Signals**, if supported by the system

These inputs allow the Control Unit to make decisions regarding the execution path of instructions.

Outputs of the Control Unit

Based on its inputs, the Control Unit generates output signals such as:

- Register load and enable signals
- ALU operation select signals
- Memory read and write signals
- Bus control signals

These signals ensure proper coordination between different components of the CPU.

Role of the Control Unit in the Instruction Cycle

The Control Unit manages all phases of the instruction cycle, including:

1. **Instruction Fetch** – retrieving the instruction from memory
2. **Instruction Decode** – interpreting the opcode
3. **Instruction Execute** – performing the required operation
4. **Result Storage** – storing the result in a register or memory

Each phase is implemented through a sequence of micro-operations controlled by the Control Unit.

Types of Control Unit Design

Control units are broadly classified into two types based on their design approach:

- Hardwired Control Unit
- Microprogrammed Control Unit

Each type has its own advantages and limitations, which are discussed in subsequent sections.

Importance of the Control Unit

The Control Unit is essential for correct processor operation because:

- It ensures proper timing and sequencing of operations
- It enables coordination among registers, ALU, and memory
- It supports conditional and branching instructions

Without the Control Unit, the CPU components would operate independently and fail to execute instructions correctly.

10. Hardwired Control Unit

A **Hardwired Control Unit** is a type of control unit in which control signals are generated using fixed hardware logic such as logic gates, decoders, multiplexers, and flip-flops. In this approach, the control logic is directly implemented in hardware and is designed to produce the required control signals for each micro-operation.

The behavior of a hardwired control unit is determined at the time of hardware design and cannot be easily modified after implementation. It is commonly used in processors where high-speed operation is a primary requirement.

Working Principle of Hardwired Control Unit

The hardwired control unit operates as a **finite state machine (FSM)**. Each state of the FSM corresponds to a specific step in the instruction execution cycle. Transitions between states are governed by the system clock and the opcode of the instruction being executed.

The instruction opcode is decoded using combinational logic circuits. Based on the decoded opcode and the current timing step, the control unit generates the appropriate control signals required to activate specific micro-operations.

Components Used in Hardwired Control

The hardwired control unit consists of the following major components:

- **Instruction Decoder**, which decodes the opcode
- **Timing and Control Logic**, which generates timing signals
- **Logic Gates and Flip-Flops**, which implement the control logic

These components work together to produce control signals at precise clock intervals.

Advantages of Hardwired Control Unit

The hardwired control unit offers several advantages:

- Very fast operation due to direct hardware implementation
- Low execution delay
- Suitable for simple and frequently executed instruction sets

Limitations of Hardwired Control Unit

Despite its speed, the hardwired control unit has some limitations:

- Lack of flexibility
- Difficult to modify or extend the instruction set
- Complex design for large and complex processors

Any change in the instruction set or control logic requires redesigning the hardware.

Applications of Hardwired Control Unit

Hardwired control units are commonly used in:

- Reduced Instruction Set Computers (RISC)
- High-performance processors
- Systems where speed is more important than flexibility

Importance in Processor Design

The hardwired control unit plays a critical role in achieving high execution speed. By generating control signals directly through hardware logic, it minimizes control delays and ensures efficient instruction execution.

11. Microprogrammed Control Unit

A **Microprogrammed Control Unit** is a type of control unit in which the control signals required for instruction execution are generated by executing a sequence of **microinstructions** stored in a special memory called **control memory**. Unlike the hardwired control unit, this approach uses a software-like method to control hardware operations.

Each instruction in the machine language is implemented as a sequence of microinstructions, collectively known as a **microprogram**. This method provides greater flexibility and ease of modification in processor design.

Control Memory

Control memory is a read-only memory (ROM) or writable control store that contains microinstructions. Each microinstruction specifies the control signals needed to perform one or more micro-operations.

The contents of control memory define the behavior of the processor at the micro-operation level. Modifying the microprogram allows changes in instruction behavior without altering hardware circuitry.

Microinstruction

A microinstruction is a binary word that contains information required to generate control signals and determine the next microinstruction to be executed.

A typical microinstruction consists of:

- A **control field** that specifies control signals
- A **condition field** that allows conditional branching
- A **next address field** that specifies the address of the next microinstruction

Working Principle of Microprogrammed Control Unit

The microprogrammed control unit operates by fetching microinstructions from control memory using a **microprogram counter**. Each fetched microinstruction is decoded to generate the required control signals.

After execution of a microinstruction, the next microinstruction address is determined based on sequencing logic and condition evaluation. This process continues until the execution of the current machine instruction is complete.

Advantages of Microprogrammed Control Unit

The microprogrammed control unit offers several advantages:

- High flexibility

- Easy modification and expansion of instruction set
- Simpler design for complex instruction sets

Limitations of Microprogrammed Control Unit

Despite its flexibility, the microprogrammed control unit has some disadvantages:

- Slower operation due to control memory access
- Additional memory requirement
- Lower performance compared to hardwired control units

Applications of Microprogrammed Control Unit

Microprogrammed control units are widely used in:

- Complex Instruction Set Computers (CISC)
- General-purpose processors
- Systems requiring frequent instruction updates

Comparison with Hardwired Control Unit

While hardwired control units emphasize speed, microprogrammed control units emphasize flexibility. Modern processors often use a hybrid approach, combining the advantages of both techniques.

12. Arithmetic and Logical Operations with Register Transfer

Arithmetic and logical operations with register transfer describe how data is moved between registers and processed by the Arithmetic and Logic Unit (ALU) during instruction execution. These operations form the core of computation in a digital computer and are implemented using a combination of **register transfers** and **ALU micro-operations**.

In this process, operands are first transferred from registers to the ALU inputs, the required arithmetic or logical operation is performed, and the result is then transferred back to a destination register. All these actions are controlled and synchronized by the control unit.

Arithmetic Operations with Register Transfer

Arithmetic operations involve numerical manipulation of binary data stored in registers. Common arithmetic operations include addition, subtraction, increment, and decrement.

An arithmetic operation using register transfer can be expressed in Register Transfer Language (RTL) as:

$R3 \leftarrow R1 + R2$
This statement indicates that the contents of registers R1 and R2 are transferred to the ALU, added together, and the result is stored in register R3.

Sequence of Arithmetic Register Transfer

The execution of an arithmetic operation generally follows these steps:

1. Transfer the first operand from a register to the ALU input.
2. Transfer the second operand from another register to the ALU input.
3. Perform the arithmetic operation inside the ALU.
4. Transfer the result back to the destination register.

Each step corresponds to one or more micro-operations controlled by the control unit.

Logical Operations with Register Transfer

Logical operations perform bit-level manipulation on register contents. These operations are essential for decision-making, masking, and comparison tasks.

Logical operations using register transfer can be represented as:

R1 ← R1 AND R2
This statement means that a bitwise AND operation is performed between the contents of registers R1 and R2, and the result is stored back in R1.

Common Logical Operations

- AND: Used for masking selected bits
- OR: Used for setting bits
- XOR: Used for bit comparison
- NOT: Used for bit inversion

Role of Status Flags

During arithmetic and logical operations, the ALU generates status flags such as Zero, Carry, Sign, and Overflow. These flags reflect the outcome of the operation and are stored in a flag register.

The control unit uses these flags to:

- Enable conditional data transfer
- Perform branch operations
- Control program flow

Synchronization with Register Transfer

All arithmetic and logical operations are synchronized using the system clock. Register transfers occur at specific clock edges, ensuring that data is stable and correctly processed.

The control unit ensures that:

- Source registers are selected correctly

- ALU control signals are activated at the right time
- Destination registers load the result without conflict

Importance of Arithmetic and Logical Register Transfers

Arithmetic and logical operations with register transfer are fundamental because:

- They enable execution of computational instructions
- They form the basis of higher-level program operations
- They integrate data movement and processing efficiently

13. Timing in Register Transfer

Timing in register transfer refers to the coordination and sequencing of register transfer operations and micro-operations with respect to the system clock. Since a digital computer operates as a synchronous system, all internal operations must occur in a well-defined order to ensure correct data processing and reliable system behavior.

Each register transfer and micro-operation is executed during a specific **clock cycle**, and the timing of these operations is controlled by the control unit.

Role of the System Clock

The system clock generates a series of uniform pulses that synchronize all operations within the CPU. These clock pulses define the timing boundaries within which data transfers and processing occur.

Registers are typically **edge-triggered**, meaning they load data only at a specific clock transition, such as the rising or falling edge. This ensures that data is stable before and after the transfer, preventing incorrect data capture.

Timing Steps in Register Transfer

The execution of an instruction is divided into a sequence of timing steps, commonly denoted as T0, T1, T2, and so on. Each timing step corresponds to one clock cycle.

During each timing step, one or more micro-operations may be performed. The control unit activates the required control signals for the specific timing step, ensuring that the correct micro-operations are executed in sequence.

Example of Timed Register Transfer

Consider a simple instruction fetch operation. The register transfer sequence may be expressed as:

T0: AR ← PC
T1: DR ← M[AR]
T2: IR ← DR
T3: PC ← PC + 1

Each register transfer occurs in a separate timing step, synchronized with the system clock. This sequence ensures that data flows correctly from memory to the instruction register and that the program counter is updated properly.

Control Unit and Timing

The control unit plays a central role in managing timing. It uses a timing generator, often implemented as a counter or sequence generator, to produce timing signals.

These timing signals are combined with instruction decoding logic to determine which control signals should be activated during each clock cycle. This mechanism ensures orderly execution of micro-operations.

Importance of Proper Timing

Proper timing in register transfer is critical for several reasons:

- It prevents data hazards and race conditions
- It ensures data stability during transfer
- It guarantees correct sequencing of micro-operations
- It enables reliable instruction execution

Incorrect timing can lead to data corruption, incorrect computation, and system malfunction.

Synchronous Nature of Register Transfer

Register transfer operations are synchronous in nature, meaning that all components operate under the control of a common clock. This synchronization simplifies processor design and ensures predictable behavior.

By aligning register transfers with clock pulses, the processor can execute complex instructions reliably and efficiently.

UNIT 3– Instruction & Addressing:

Instruction and Addressing: A simple computer organization and instruction set, instruction formats, addressing modes, instruction cycle, instruction execution in terms of microinstructions, interrupt cycle, concepts of interrupt and simple 1/0 organization, Synchronous & Asynchronous data transfer, Data Transfer Mode: Program Controlled, Interrupt driven, DMA (Direct Memory Access). Implementation of processor using the building blocks.

1. A Simple Computer Organization and Instruction Set

A simple computer organization is an abstract model designed to explain the internal functioning of a digital computer in a clear and systematic manner. It helps in understanding how instructions are stored, fetched, decoded, and executed using basic hardware components. Although simplified, this model captures the essential features of real computer systems and forms the foundation for studying advanced computer architectures.

A simple computer typically consists of a Central Processing Unit (CPU), main memory, and input/output units. The CPU includes a small set of registers, an Arithmetic and Logic Unit (ALU), and a control unit. The instruction set of a simple computer is limited but sufficient to perform data transfer, arithmetic, logical, and control operations.

Instruction Set of a Simple Computer

An instruction set is the complete collection of machine-level instructions that a processor can execute. In a simple computer, the instruction set is deliberately kept small to simplify hardware design and instruction exe1.cution.

Each instruction is represented in binary form and generally consists of two main parts:

- An **opcode**, which specifies the operation to be performed
- An **operand or address field**, which specifies the data or its location

The instruction set typically includes memory reference instructions, register reference instructions, and input/output instructions.

2. Instruction Formats

An **instruction format** defines the layout of bits within an instruction. It specifies how many bits are allocated for the opcode, addressing information, and operand fields. The design of instruction formats directly affects instruction decoding complexity and execution speed.

In a simple computer, instructions usually have a fixed length and a uniform format. This simplifies instruction decoding and control unit design. A common instruction format includes:

- Opcode field
- Address field
- Mode or control bits (if required)

Fixed-length instruction formats make hardware implementation easier but may reduce flexibility.

3. Addressing Modes

An **addressing mode** specifies how the effective address of an operand is determined during instruction execution. Addressing modes provide flexibility in accessing operands and support efficient program execution.

Common addressing modes used in a simple computer include:

- **Immediate Addressing Mode**, where the operand is part of the instruction itself
- **Direct Addressing Mode**, where the address field specifies the memory location of the operand
- **Indirect Addressing Mode**, where the address field points to a memory location that contains the actual operand address
- **Register Addressing Mode**, where the operand is located in a register

Addressing modes influence instruction length, execution time, and programming convenience.

4. Instruction Cycle

The **instruction cycle** is the complete sequence of steps required to fetch and execute a single instruction. Every instruction executed by the CPU follows this cycle.

The instruction cycle consists of the following phases:

1. **Fetch Cycle** – The instruction is fetched from memory using the Program Counter.
2. **Decode Cycle** – The instruction opcode is decoded to determine the required operation.
3. **Execute Cycle** – The specified operation is performed using registers, ALU, or memory.
4. **Interrupt Check** – The processor checks whether an interrupt has occurred.

This cycle repeats continuously as long as the computer is running.

5. Instruction Execution in Terms of Microinstructions

Instruction execution at the hardware level is performed using **microinstructions**. A microinstruction is a low-level control instruction that specifies one or more micro-operations such as register transfer, ALU operation, or memory access.

Each machine-level instruction is executed as a sequence of microinstructions stored in control memory or generated by control logic. These microinstructions control:

- Data movement between registers
- ALU operation selection
- Memory read and write operations

For example, an ADD instruction may involve multiple microinstructions to fetch operands, perform addition, and store the result. This microinstruction-based execution provides precise control over hardware behavior.

6. Interrupt Cycle

An **interrupt** is a signal that temporarily suspends the normal execution of a program to service an urgent request. The **interrupt cycle** is the sequence of operations performed by the CPU to handle an interrupt.

When an interrupt occurs:

1. The CPU completes the execution of the current instruction.
2. The contents of the Program Counter and other necessary registers are saved.
3. Control is transferred to the interrupt service routine.
4. The interrupt service routine is executed.
5. The saved state is restored, and normal program execution resumes.

The interrupt cycle allows the processor to respond efficiently to external events such as I/O operations, hardware faults, or timer signals.

Significance of Instruction and Addressing Concepts

Understanding a simple computer organization and instruction set is essential because:

- It explains how software instructions control hardware
- It clarifies the role of registers, memory, and control logic
- It forms the basis for instruction set architecture design
- It helps in understanding advanced topics like pipelining and parallel execution

Instruction Execution in Terms of Microinstructions

At the hardware level, the execution of a machine-level instruction is carried out through a sequence of simpler operations known as **microinstructions**. A microinstruction is a low-level control instruction that specifies one or more micro-operations, such as data transfer between registers, arithmetic or logical operations, or memory access.

Each machine instruction is decomposed into a series of microinstructions that are executed sequentially. These microinstructions are either generated by a hardwired control unit or stored in a control memory in a microprogrammed control unit. Together, the sequence of microinstructions corresponding to a machine instruction is called a **microprogram**.

During instruction execution, microinstructions control the movement of data between registers, activate the Arithmetic and Logic Unit (ALU) for computation, and manage memory read or write operations. This approach provides precise control over hardware behavior and ensures that complex instructions can be executed systematically using simpler hardware actions.

7. Interrupt: Concept and Need

An **interrupt** is a signal that temporarily suspends the normal execution of a program to allow the processor to respond to an external or internal event. Interrupts are essential for efficient system operation because they enable the processor to react promptly to events such as input/output requests, hardware faults, or timer expirations.

Without interrupts, the processor would have to continuously poll devices to check their status, leading to inefficient use of CPU time. Interrupts allow the processor to perform useful work until an event requires attention.

Interrupts can be generated by:

- Input/output devices
- Internal hardware conditions
- Software instructions
- Timer units

Interrupt Cycle

The **interrupt cycle** is the sequence of operations performed by the CPU when an interrupt occurs. It is an extension of the normal instruction cycle and ensures that program execution can be resumed correctly after servicing the interrupt.

The interrupt cycle generally involves the following steps:

1. The processor completes the execution of the current instruction.
2. The contents of the Program Counter and other necessary registers are saved.
3. The processor transfers control to a predefined memory location known as the interrupt service routine.
4. The interrupt service routine is executed to handle the interrupt.
5. The saved processor state is restored.
6. Normal program execution resumes from the point of interruption.

The interrupt cycle allows the processor to handle asynchronous events without losing program continuity.

8. Simple Input/Output Organization

A **simple input/output (I/O) organization** provides basic communication between the CPU and peripheral devices such as keyboards, displays, printers, and storage devices. In this organization, I/O devices are connected to the CPU through input and output registers.

The CPU communicates with I/O devices using special I/O instructions. Data transfer occurs through I/O registers rather than directly accessing the devices. The control unit coordinates these operations by generating appropriate control signals.

In a simple I/O organization, the processor may use:

- Program-controlled I/O
- Interrupt-driven I/O

This approach is suitable for systems with limited I/O requirements and simple hardware design.

9. Synchronous Data Transfer

Synchronous data transfer is a method in which data transfer between two devices is controlled by a common clock signal. Both the sender and the receiver operate in synchronization with the same clock, ensuring that data is transmitted and received at predetermined time intervals.

In synchronous data transfer:

- Timing is fixed and predictable
- No additional handshaking signals are required
- Data transfer is fast and efficient

However, synchronous transfer requires both devices to operate at compatible speeds, which limits its flexibility.

10. Asynchronous Data Transfer

Asynchronous data transfer is a method in which data transfer occurs without a shared clock signal. Instead, control signals are used to coordinate the transfer between the sender and the receiver.

In asynchronous data transfer:

- The sender and receiver operate independently
- Handshaking signals are used to indicate readiness
- Data transfer occurs only when both devices are ready

This method is more flexible than synchronous transfer and is widely used in input/output operations where devices operate at different speeds.

Comparison Between Synchronous and Asynchronous Data Transfer

Synchronous data transfer offers higher speed and simpler control but requires strict timing coordination. Asynchronous data transfer provides greater flexibility and compatibility between devices but involves additional control overhead and slightly lower performance.

11. Data Transfer Modes

Data transfer modes define the methods by which data is transferred between the Central Processing Unit (CPU), main memory, and input/output (I/O) devices. Since I/O devices generally operate at speeds much slower than the CPU, efficient data transfer mechanisms are required to avoid unnecessary processor idle time. The choice of data transfer mode significantly affects system performance and processor utilization.

The three primary data transfer modes are **Program Controlled I/O**, **Interrupt-Driven I/O**, and **Direct Memory Access (DMA)**.

12. Program Controlled Data Transfer

Program controlled data transfer, also known as **programmed I/O**, is the simplest method of data transfer between the CPU and I/O devices. In this mode, the CPU directly controls and manages all data transfer operations through a sequence of instructions.

In program controlled I/O, the processor repeatedly checks the status of the I/O device to determine whether it is ready for data transfer. This process is known as **polling**. Once the device is ready, the CPU executes instructions to transfer data between the I/O device and the CPU registers.

Although this method is easy to implement, it is inefficient because the CPU remains busy waiting for the I/O device, even when no data transfer is occurring. As a result, processor time is wasted, leading to poor system performance.

13. Interrupt-Driven Data Transfer

Interrupt-driven data transfer improves efficiency by allowing the CPU to perform other tasks while waiting for an I/O device. In this mode, the I/O device sends an **interrupt signal** to the processor when it is ready for data transfer.

When an interrupt occurs, the processor temporarily suspends the execution of the current program, saves its state, and transfers control to an **interrupt service routine (ISR)**. The ISR performs the required data transfer between the CPU and the I/O device. After completing the operation, the processor restores its previous state and resumes normal program execution.

Interrupt-driven I/O significantly reduces CPU idle time compared to program controlled I/O. However, the CPU is still involved in every data transfer, which can become a limitation for high-speed or large-volume data transfers.

14. Direct Memory Access (DMA)

Direct Memory Access (DMA) is a data transfer technique that allows data to be transferred directly between an I/O device and main memory without continuous involvement of the CPU. In this mode, a special hardware component called the **DMA controller** manages the data transfer process.

When a DMA transfer is required, the CPU initializes the DMA controller by providing the memory address, the direction of transfer, and the number of data units to be transferred. The DMA controller then takes control of the system bus and performs the data transfer directly between the I/O device and memory.

Once the transfer is complete, the DMA controller generates an interrupt to notify the CPU. DMA greatly improves system performance by freeing the CPU from routine data transfer tasks and is commonly used in high-speed I/O devices such as disks and network interfaces.

Comparison of Data Transfer Modes

Program controlled data transfer is simple but inefficient due to continuous CPU involvement. Interrupt-driven data transfer improves efficiency by reducing busy waiting, but still requires CPU participation for each transfer. DMA provides the highest efficiency by offloading data transfer tasks to dedicated hardware, allowing the CPU to focus on computation.

15. Implementation of a Processor Using Building Blocks

The implementation of a processor using building blocks refers to the construction of a CPU by interconnecting fundamental digital components such as registers, multiplexers, arithmetic circuits, control logic, and memory interfaces. These building blocks form the **datapath** and **control path** of the processor.

Datapath Components

The datapath is responsible for data movement and processing within the processor. It consists of:

- Registers for temporary storage of data and instructions
- Arithmetic and Logic Unit (ALU) for computation
- Buses and multiplexers for data routing

The datapath executes micro-operations such as register transfer, arithmetic computation, and logical operations under the control of control signals.

Control Unit Implementation

The control unit coordinates the operation of the datapath by generating appropriate control signals. It interprets instruction opcodes, monitors status flags, and sequences micro-operations.

The control unit can be implemented using:

- Hardwired control logic
- Microprogrammed control logic

In both cases, the control unit ensures that instructions are executed correctly and in proper sequence.

Integration of Building Blocks

The processor is formed by integrating the datapath and control unit with memory and I/O interfaces. Clock signals synchronize all operations, ensuring reliable data transfer and processing.

This modular building-block approach simplifies processor design, improves scalability, and allows systematic analysis of processor behavior.

Importance of Building Block Implementation

Implementing a processor using building blocks:

- Provides a clear understanding of internal CPU operation
- Simplifies debugging and design verification
- Forms the basis for advanced processor architectures

UNIT 4– Memory System Design

Memory System Design: Memory Origination, Memory Hierarchy, Main Memory (RAM/ROM chips), Auxiliary memory, Associative memory, Cache Memory, Virtual Memory. Assembly Language Programs, Assembler Directives, Pseudo Instructions, Macroinstructions, Linking and Loading.

Memory System Design

The memory system is a vital component of a computer system that stores instructions and data required for program execution. Memory system design focuses on organizing different types of memory to achieve high performance, low cost, and efficient data access. This unit introduces the concepts of memory hierarchy, memory organization, and performance improvement techniques used in modern computer systems.

1. Memory Organization

Memory organization refers to the way memory units are structured, addressed, interconnected, and accessed in a computer system. It defines how instructions and data are stored in memory and how efficiently they can be retrieved during program execution.

Memory organization plays a critical role in determining system performance, as it affects access time, storage capacity, and cost.

Objectives of Memory Organization

The primary objectives of memory organization are:

- To provide fast access to frequently used data
- To efficiently utilize available memory resources
- To minimize memory access delay
- To support scalability and system expansion

Memory Organization Parameters

Memory organization is characterized by several important parameters:

- **Word Length**
  The number of bits that the processor can handle at one time.
- **Addressability**
  Defines whether memory is byte-addressable or word-addressable.
- **Access Time**
  The time required to read or write data from memory.
- **Memory Capacity**
  The total amount of data that can be stored.

Example

In a 32-bit system with byte-addressable memory, each memory location stores 8 bits, but the CPU processes 32 bits at a time.

2. Memory Hierarchy

Memory hierarchy is an arrangement of different types of memory in a computer system based on speed, capacity, cost, and proximity to the CPU. Since no single memory type can satisfy all requirements, multiple memory levels are organized hierarchically.

Levels of Memory Hierarchy

The memory hierarchy typically consists of the following levels:

1. **Registers**
   Fastest memory, located inside the CPU, very small in size.

2. **Cache Memory**
   High-speed memory placed between CPU and main memory.

3. **Main Memory**
   Stores active programs and data.

4. **Auxiliary Memory**
   Provides permanent storage for large volumes of data.

Principle of Locality of Reference

The effectiveness of memory hierarchy relies on the principle of locality:

- **Temporal Locality**
  Recently accessed data is likely to be accessed again soon.

- **Spatial Locality**
  Data located near recently accessed data is likely to be accessed.

Example

Loop variables are repeatedly accessed and therefore remain in cache memory.

3. Main Memory

Main memory is the primary storage directly accessed by the CPU during instruction execution. It stores currently running programs, data, and intermediate results.

Main memory is typically implemented using semiconductor memory chips and is volatile in nature.

3.1 Random Access Memory (RAM)

Random Access Memory (RAM) is a volatile memory that allows data to be read or written at any location with equal access time.

Types of RAM

Static RAM (SRAM)

- Stores data using flip-flops
- Does not require refreshing
- Very fast access time
- Expensive and low density
- Used in cache memory

Dynamic RAM (DRAM)

- Stores data as electric charge in capacitors
- Requires periodic refreshing
- Slower than SRAM
- High storage density
- Used as main memory

Example

System RAM in personal computers is implemented using DRAM chips.

3.2 Read Only Memory (ROM)

Read Only Memory (ROM) is a non-volatile memory that stores permanent data and programs required for system initialization.

Types of ROM

- **PROM (Programmable ROM)** – Can be programmed once
- **EPROM (Erasable PROM)** – Erased using ultraviolet light
- **EEPROM (Electrically Erasable PROM)** – Electrically erasable
- **Flash Memory** – Faster and widely used EEPROM variant

Example

BIOS firmware is stored in ROM.

4. Auxiliary Memory

Auxiliary memory, also known as secondary storage, provides permanent storage for large amounts of data and programs.

Characteristics of Auxiliary Memory

- Non-volatile
- Large storage capacity
- Low cost per bit
- Slower than main memory

Types of Auxiliary Memory

- Hard disk drives
- Solid state drives
- Magnetic tapes
- Optical disks

Example

Operating systems and user files are stored on hard disks or SSDs.

<u>5. Associative Memory (Content Addressable Memory)</u>

Associative memory is a special type of memory that retrieves stored data based on content rather than memory address. It is also known as Content Addressable Memory (CAM).

Working Principle

All memory entries are searched simultaneously in parallel. If the input data matches stored content, the corresponding memory location is accessed.

Advantages

- Extremely fast searching
- Parallel comparison of entries

Limitations

- High cost
- Complex hardware design
- Limited storage capacity

Applications

- Cache tag comparison
- Translation Lookaside Buffer (TLB) in virtual memory

6. Cache Memory

Cache memory is a small, high-speed memory placed between the CPU and main memory to reduce average memory access time.

Working of Cache Memory

When the CPU requests data:

- **Cache Hit** occurs if data is found in cache
- **Cache Miss** occurs if data is not found and must be fetched from main memory

Types of Cache Memory

- **Level 1 (L1)** – Located inside CPU, fastest
- **Level 2 (L2)** – Larger and slightly slower
- **Level 3 (L3)** – Shared among multiple cores

Cache Mapping Techniques

- Direct mapping
- Associative mapping
- Set-associative mapping

Example

Instruction cache stores frequently executed instructions.

7. Virtual Memory

Virtual memory is a memory management technique that allows programs larger than physical main memory to be executed by using auxiliary memory as an extension of main memory.

Concept of Virtual Memory

- Programs are divided into fixed-size blocks called pages
- Only required pages are loaded into main memory
- Remaining pages stay on secondary storage

Key Terms

- **Page** – Fixed-size memory block

- **Page Table** – Maps virtual addresses to physical addresses
- **Page Fault** – Occurs when required page is not in main memory

Advantages

- Efficient use of memory
- Supports multitasking
- Allows execution of large programs

Disadvantages

- Slower access during page faults
- Requires complex hardware and OS support

Example

Modern operating systems use paging-based virtual memory.

8. Assembly Language Programs

Assembly language is a low-level programming language that provides a symbolic representation of machine-level instructions. An **assembly language program** consists of a sequence of instructions written using mnemonics, symbolic addresses, and labels that are easier for humans to understand compared to binary machine code.

Each assembly language instruction corresponds directly to a machine instruction, making assembly language **machine-dependent**. Assembly language programs provide greater control over hardware resources and are commonly used in system programming, embedded systems, and performance-critical applications.

Structure of an Assembly Language Program

An assembly language program typically consists of the following components:

- **Label Field** – Used to define symbolic names for memory locations
- **Opcode Field** – Specifies the operation to be performed
- **Operand Field** – Specifies registers, memory locations, or constants
- **Comment Field** – Provides explanatory remarks for the programmer

Example

```
START:  MOV R1, R2   ; Move data from R2 to R1
        ADD R1, R3   ; Add contents of R3 to R1
        END
```

In this example, symbolic instructions are used instead of machine code, making the program easier to read and maintain.

Advantages of Assembly Language Programs

- Efficient use of hardware
- Faster execution compared to high-level languages
- Precise control over memory and registers

Limitations

- Machine dependent
- Difficult to write and debug
- Poor portability

9. Assembler Directives

Assembler directives are special instructions provided to the assembler that **do not generate machine code**. Instead, they guide the assembler during the assembly process by defining data, reserving memory, or controlling program structure.

Assembler directives are also known as **assembler commands**.

Common Assembler Directives

- **START** – Specifies the starting address of the program
- **END** – Indicates the end of the assembly program
- **ORG** – Sets the origin (starting address) for subsequent instructions
- **EQU** – Assigns a constant value to a symbol
- **DB / DW** – Define byte or define word

Example

```
NUM  EQU  10
DATA DB   25
```
Here, EQU assigns a constant value, and DB allocates memory for data.

Purpose of Assembler Directives

- Control memory allocation
- Define constants and data structures
- Organize program layout
- Assist in symbol resolution

10. Pseudo Instructions

Pseudo instructions are instructions that **appear like machine instructions** but are **not directly translated into machine code**. They are used to simplify programming and improve readability.

Pseudo instructions are handled by the assembler and may expand into one or more actual machine instructions.

Characteristics of Pseudo Instructions

- Do not represent real CPU instructions
- Converted internally by the assembler
- Improve program clarity

Example

MOVE R1, R2
The assembler may translate this into multiple machine instructions depending on the architecture.

Difference Between Machine Instructions and Pseudo Instructions

Machine instructions are executed directly by hardware, whereas pseudo instructions are translated by the assembler before execution.

11. Macroinstructions

A macroinstruction (or macro) is a **user-defined sequence of assembly instructions** that can be invoked using a single macro name. Macros help reduce repetitive code and improve program maintainability.

A macro is expanded by the assembler during assembly time.

Structure of a Macro

- **Macro definition** – Specifies the macro name and body
- **Macro invocation** – Uses the macro name within the program

Example

MACRO INCR
ADD R1, #1
ENDM
Using INCR in the program will expand into ADD R1, #1.

Advantages of Macros

- Reduces code duplication
- Improves program readability
- Simplifies modification

Disadvantages

- Increases program size due to expansion
- Difficult to debug expanded code

12. Linking

Linking is the process of combining multiple object programs into a single executable program. A **linker** resolves symbolic references between different modules and assigns final memory addresses.

Functions of a Linker

- Combines object files
- Resolves external references
- Relocates code and data
- Produces executable file

Types of Linking

- **Static Linking** – All library routines are included at compile time
- **Dynamic Linking** – Libraries are linked at runtime

Example

If one module defines a function and another module uses it, the linker resolves this dependency.

13. Loading

Loading is the process of placing an executable program into main memory for execution. A **loader** allocates memory and initializes program execution.

Functions of a Loader

- Allocate memory
- Load program into memory
- Perform relocation

- Start program execution

Types of Loaders

- **Absolute Loader** – Loads program at fixed address
- **Relocating Loader** – Adjusts addresses during loading
- **Dynamic Loader** – Loads routines as needed

Relationship Between Assembler, Linker, and Loader

- **Assembler** converts assembly code into object code
- **Linker** combines object files into executable code
- **Loader** loads the executable into memory for execution

Together, these system programs enable translation and execution of assembly language programs.

UNIT 5– Vector and Array Processing & Microprocessor Concepts

Vector and Array Processing: Shared-Memory, Multiprocessing, Distributed Mufti Computing.

Microprocessor Concepts: Pin Diagram of 8085, Architecture of 8085, Addressing Mode of 8085, functional block diagram of 8085 assembly language, instruction set of 8085.

Introduction to Vector and Array Processing

Modern computing applications such as scientific simulations, image processing, weather forecasting, machine learning, and big data analytics require the execution of a large number of similar operations on large sets of data. To meet these performance demands, computer systems use **vector and array processing techniques**, which allow multiple data elements to be processed simultaneously.

Vector and array processing exploit **parallelism** at the data level, where the same operation is applied to multiple data elements at the same time. This unit discusses the principles of vector and array processing and examines different parallel computing models such as shared-memory systems, multiprocessing systems, and distributed multi-computing systems.

1. Vector Processing

Vector processing is a computing technique in which a single instruction operates on a vector, which is a sequence of data elements. Instead of processing one data element at a time, vector processors handle multiple elements in parallel, significantly improving execution speed.

Characteristics of Vector Processing

- Operates on one-dimensional arrays called vectors
- Uses vector instructions instead of scalar instructions
- Reduces instruction fetch and decode overhead
- Achieves high performance for repetitive computations

Vector Instructions

Vector instructions specify:

- The operation to be performed (addition, multiplication, etc.)
- The source vectors
- The destination vector

For example, a single vector addition instruction can add two vectors element by element.

Advantages of Vector Processing

- High computational throughput
- Efficient use of hardware resources
- Suitable for scientific and numerical applications

Limitations

- Less effective for irregular data structures
- Requires large memory bandwidth
- Not suitable for control-intensive programs

2. Array Processing

Array processing is a parallel computing technique in which a large number of processing elements operate simultaneously on different elements of an array. Each processing element performs the same operation on different data, following the **Single Instruction Multiple Data (SIMD)** model.

Characteristics of Array Processing

- Uses multiple processing units
- Executes the same instruction across all processors
- High degree of parallelism
- Often implemented using processor arrays

Applications of Array Processing

- Image and signal processing
- Matrix computations
- Pattern recognition
- Scientific simulations

3. Shared-Memory Systems

A shared-memory system is a parallel computing architecture in which multiple processors share a common memory address space. All processors can directly access and modify shared data.

Working Principle

In shared-memory systems:

- Processors communicate by reading and writing shared variables

- Synchronization mechanisms are required to avoid data conflicts
- Memory access time may vary depending on architecture

Types of Shared-Memory Architectures

- **Uniform Memory Access (UMA)**
  All processors have equal access time to memory.
- **Non-Uniform Memory Access (NUMA)**
  Memory access time depends on the memory location relative to the processor.

Advantages

- Easy communication between processors
- Simplified programming model
- Efficient data sharing

Disadvantages

- Memory contention
- Scalability limitations
- Synchronization overhead

4. Multiprocessing

Multiprocessing refers to the use of two or more processors within a single computer system to execute multiple processes simultaneously. Each processor operates independently but cooperates with others to improve overall system performance.

Types of Multiprocessing

- **Symmetric Multiprocessing (SMP)**
  All processors are equal and share the same memory and operating system.
- **Asymmetric Multiprocessing (AMP)**
  One processor controls the system, while others perform specific tasks.

Benefits of Multiprocessing

- Increased throughput
- Better resource utilization
- Improved system reliability

Challenges

- Process synchronization
- Load balancing
- Memory access conflicts

5. Distributed Multi Computing

Distributed multi computing is a computing model in which multiple independent computers, connected through a network, work together to solve a problem. Each computer has its own local memory and processing resources.

Characteristics of Distributed Computing

- No shared memory
- Communication through message passing
- High scalability
- Fault tolerance

Working Mechanism

In distributed systems:

- Tasks are divided among multiple nodes
- Nodes communicate using messages
- Each node executes its assigned task independently

Advantages of Distributed Computing

- Highly scalable
- Cost-effective using commodity hardware
- Suitable for large-scale problems

Limitations

- Communication latency
- Complex programming model
- Data consistency challenges

6. Comparison of Shared-Memory and Distributed Systems

Shared-memory systems provide faster communication through shared variables but face scalability issues. Distributed systems scale efficiently and provide fault tolerance but incur communication overhead due to message passing.

7. Importance of Vector and Parallel Processing

Vector and parallel processing techniques:

- Improve computational performance
- Enable handling of large data sets
- Support modern high-performance applications

MICROPROCESSOR CONCEPTS – INTEL 8085

1. Introduction to Microprocessor

A microprocessor is a programmable digital device that performs arithmetic, logical, and control operations on binary data. It acts as the **central processing unit (CPU)** of a computer system and executes instructions stored in memory. The **Intel 8085** is an 8-bit microprocessor widely used for educational purposes to explain fundamental microprocessor concepts due to its simple architecture and instruction set.

The 8085 operates on 8-bit data, has a 16-bit address bus, and can address up to 64 KB of memory.

2. Pin Diagram of 8085

Explanation

The Intel 8085 is packaged in a **40-pin Dual In-Line Package (DIP)**. Each pin has a specific function and can be grouped based on functionality.

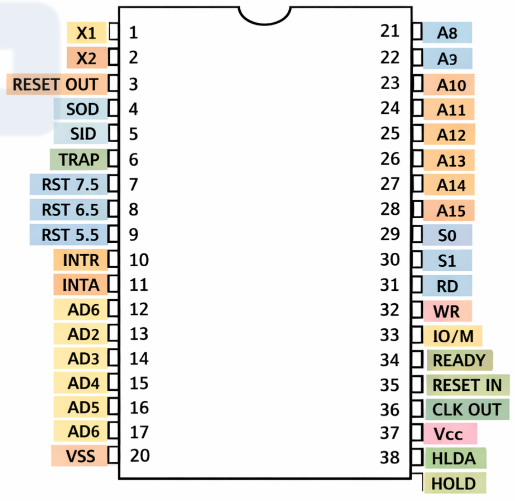


Figure 1: Pin Diagram of Intel 8085

Pin Groups and Their Functions

1. **Address/Data Bus (AD0–AD7)**
   These pins are multiplexed. They carry the lower 8 bits of the address during the first clock cycle and data during subsequent cycles.

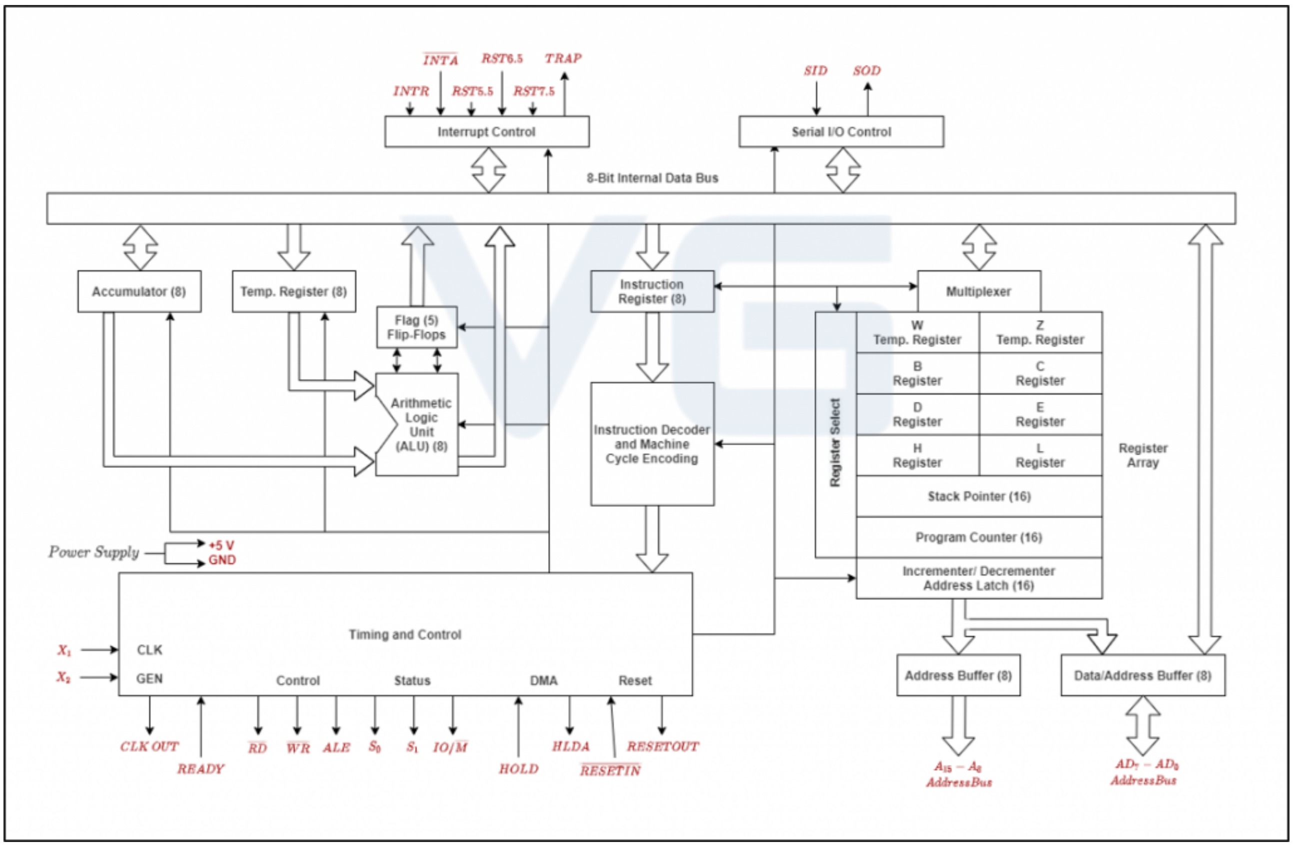
2. **Address Bus (A8–A15)**
   These pins carry the higher 8 bits of the 16-bit address and are unidirectional.

3. **Control and Status Signals**
   - **RD**: Read signal
   - **WR**: Write signal
   - **IO/M**: Selects memory or I/O operation

4. **Interrupt Pins**
   - **TRAP** (non-maskable, highest priority)
   - **RST 7.5, RST 6.5, RST 5.5** (maskable interrupts)
   - **INTR** (general interrupt)
5. **Power and Clock Pins**
   - **Vcc** and **GND** supply power
   - **X1, X2** provide clock input

The pin diagram defines how the microprocessor communicates with memory, I/O devices, and external hardware.

3. Architecture of 8085



Overview

The architecture of the 8085 describes its internal organization and interconnection of functional units. It follows a **Von Neumann architecture**, where both data and instructions share the same memory.

Main Components of 8085 Architecture

1. Arithmetic and Logic Unit (ALU)

The ALU performs arithmetic operations (addition, subtraction) and logical operations (AND, OR, XOR, comparison).

2. Accumulator (A)

An 8-bit register used to store operands and results of ALU operations.

3. Register Array

Includes six general-purpose registers: **B, C, D, E, H, L**, which can be paired as BC, DE, and HL.

4. Program Counter (PC)

A 16-bit register that stores the address of the next instruction to be executed.

5. Stack Pointer (SP)

A 16-bit register that points to the top of the stack in memory.

6. Instruction Register and Decoder

Stores the fetched instruction and decodes it to generate control signals.

7. Timing and Control Unit

Generates timing and control signals required for instruction execution.

4. Addressing Modes of 8085

Definition

An addressing mode specifies the method by which the operand of an instruction is accessed.

Types of Addressing Modes

1. Immediate Addressing Mode

The operand is specified directly in the instruction.
Example: MVI A, 05H

2. Register Addressing Mode

The operand is stored in a register.
Example: MOV A, B

3. Direct Addressing Mode

The memory address of the operand is given in the instruction.
Example: LDA 2050H

4. Indirect Addressing Mode

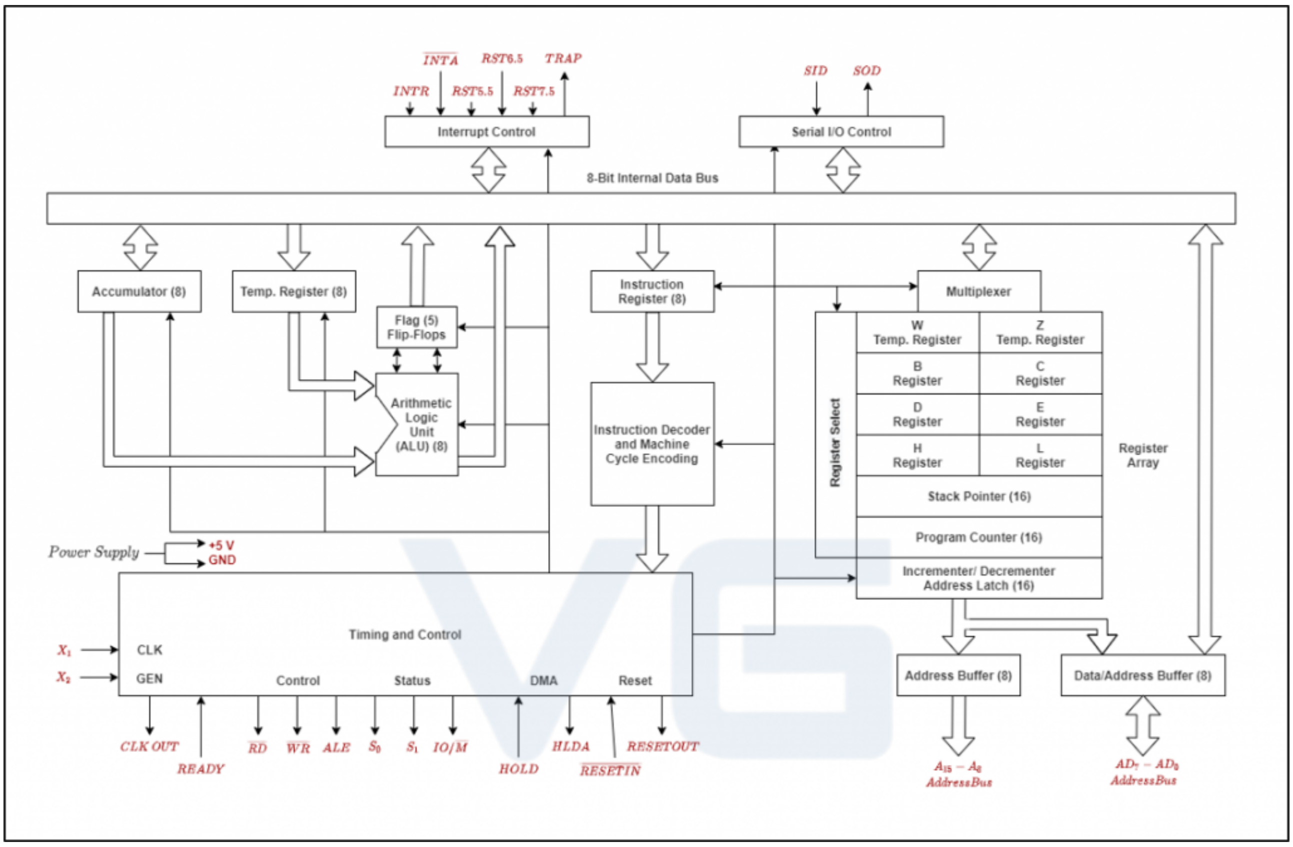
The address of the operand is stored in a register pair.
Example: MOV A, M

5. Implied Addressing Mode

The operand is implied by the instruction itself.
Example: CMA

Addressing modes provide flexibility in accessing data efficiently.

<u>5. Functional Block Diagram of 8085</u>



The functional block diagram of the Intel 8085 microprocessor represents the **logical organization of its internal units** and explains how these units cooperate to execute instructions. Instead of focusing only on physical layout, this diagram emphasizes the **functional interaction** among different blocks during instruction execution and data processing.

The internal structure of the 8085 can be broadly divided into three major functional groups: **Data Path Blocks**, **Control Path Blocks**, and **Interface Blocks**. Each group performs a specific role in the overall operation of the microprocessor.

<u>Data Path Blocks</u>

The data path blocks are responsible for the **movement, storage, and processing of data** inside the microprocessor. These blocks form the core computation mechanism of the 8085.

The data path includes the **register array**, the **accumulator**, the **Arithmetic and Logic Unit (ALU)**, and the **internal data buses**. The general-purpose registers store temporary data, operands, and intermediate results. The accumulator acts as a central register that holds operands and results of most arithmetic and logical operations.

The ALU performs arithmetic operations such as addition and subtraction, as well as logical operations such as AND, OR, XOR, and comparison. Internal buses provide communication paths between registers, the ALU, and other internal blocks, ensuring smooth data transfer within the processor.

Control Path Blocks

The control path blocks manage and coordinate the execution of instructions. These blocks generate the necessary control signals that regulate data movement and processing.

The control path includes the **Instruction Register**, **Instruction Decoder**, and the **Timing and Control Unit**. The instruction register temporarily holds the fetched instruction, while the instruction decoder interprets the opcode to determine the type of operation to be performed.

The timing and control unit generates precise timing signals and control signals required for instruction execution. It synchronizes all micro-operations with the system clock and ensures that each step of instruction execution occurs in the correct sequence.

Interface Blocks

The interface blocks enable communication between the microprocessor and the external world, including **memory and input/output devices**.

These blocks consist of the **address buffer**, **data buffer**, and **control signals** such as RD, WR, and IO/M. The address buffer outputs the memory or I/O address, while the data buffer handles the transfer of data between the microprocessor and external devices.

Through these interface blocks, the 8085 can read instructions and data from memory, write results back to memory, and exchange information with I/O devices.

Overall Significance of the Functional Block Diagram

The functional block diagram provides a clear understanding of **how instructions flow through the microprocessor**, starting from instruction fetch, decoding, execution, and finally storing the result. It also explains how data is processed internally and how control signals coordinate all activities.

By studying this diagram, one can understand the complete working mechanism of the 8085 microprocessor, including the interaction between computation units, control logic, and external interfaces. This makes the functional block diagram an essential tool for learning microprocessor architecture and operation.

6. Assembly Language of Intel 8085

Assembly language is a low-level programming language specifically designed to program a particular microprocessor. In the case of the Intel 8085, assembly language uses **mnemonics** to represent machine-level instructions in a symbolic and human-readable form. Each assembly language instruction corresponds **exactly to a single machine code instruction** of the 8085 microprocessor.

Assembly language provides a direct interface between software and hardware, allowing programmers to control registers, memory locations, and I/O devices explicitly.

Characteristics of 8085 Assembly Language

1. Machine Dependent

Assembly language programs written for the 8085 are **not portable**. They are designed specifically for the 8085 architecture and cannot be executed on other microprocessors without modification. This is because mnemonics, registers, and instruction formats are unique to the 8085.

2. One-to-One Correspondence with Machine Instructions

Each assembly instruction translates into a **single machine instruction**. This one-to-one correspondence ensures precise control over hardware operations and predictable execution behavior.

For example:

ADD B
is translated directly into the corresponding machine opcode for addition.

3. Use of Mnemonics

Mnemonics are abbreviated symbolic names used to represent machine instructions. They make programs easier to read, write, and debug compared to binary machine code.

Examples of mnemonics include:

- MOV for data transfer
- ADD for addition
- JMP for unconditional jump

4. Use of Symbolic Names and Labels

Assembly language allows the use of **labels** and **symbolic names** to represent memory addresses. This improves program readability and simplifies modification, as the programmer does not need to remember absolute memory addresses.

5. Efficient and Fast Execution

Since assembly language instructions correspond directly to machine instructions, programs written in assembly language execute faster and use memory efficiently. This makes assembly language suitable for real-time systems and performance-critical applications.

Structure of an Assembly Language Instruction

An assembly instruction for the 8085 follows a fixed format:

LABEL: OPCODE OPERAND ; COMMENT
Explanation of Fields

- **Label**
  A symbolic name representing a memory location. It is optional and used mainly in branching instructions.
- **Opcode**
  The mnemonic that specifies the operation to be performed by the microprocessor.

- **Operand**
  Specifies the data, register, or memory location involved in the operation.

- **Comment**
  A non-executable part of the instruction used to describe the purpose of the instruction.

Example

MOV A, B ; Copy contents of register B into accumulator A
This instruction transfers the contents of register B into the accumulator.

Advantages of Assembly Language

- Precise control over hardware
- Efficient use of memory and CPU resources
- Fast execution speed

Limitations of Assembly Language

- Difficult to write and debug
- Machine dependent
- Large program size for complex applications

7. Instruction Set of Intel 8085

The instruction set of the Intel 8085 is the complete collection of machine-level instructions that the microprocessor can execute. These instructions define the functional capabilities of the 8085 and determine how it performs data processing, logical operations, program control, and system management.

Classification of Instructions

The instruction set of the 8085 is classified into several groups based on their functionality.

1. Data Transfer Instructions

Data transfer instructions are used to move data between registers, between memory and registers, and between the microprocessor and I/O devices. These instructions **do not alter the content of the source**, except in specific operations.

Common Data Transfer Instructions

- **MOV** – Transfer data between registers or between register and memory
- **MVI** – Move immediate data to a register or memory
- **LDA** – Load accumulator directly from memory
- **STA** – Store accumulator directly into memory

Example

MVI A, 05H

Loads the hexadecimal value 05 into the accumulator.

2. Arithmetic Instructions

Arithmetic instructions perform mathematical operations on binary data. Most arithmetic operations are performed using the **accumulator** and affect status flags such as Zero, Carry, Sign, and Parity.

Common Arithmetic Instructions

- **ADD** – Add register or memory content to accumulator
- **SUB** – Subtract register or memory content from accumulator
- **INR** – Increment register or memory by one
- **DCR** – Decrement register or memory by one

Example

ADD B

Adds the contents of register B to the accumulator.

3. Logical Instructions

Logical instructions perform bitwise logical operations and comparisons. These instructions are used for masking, testing bits, and implementing decision-making logic.

Common Logical Instructions

- **ANA** – Logical AND operation
- **ORA** – Logical OR operation
- **XRA** – Logical Exclusive OR operation
- **CMA** – Complement accumulator

Example

ANA C

Performs a logical AND between accumulator and register C.

4. Branching Instructions

Branching instructions are used to alter the normal sequence of program execution. These instructions enable conditional and unconditional jumps, subroutine calls, and returns.

Common Branching Instructions

- **JMP** – Unconditional jump
- **JZ** – Jump if Zero flag is set
- **CALL** – Call a subroutine
- **RET** – Return from subroutine

Example

JZ LOOP

Transfers control to label LOOP if the Zero flag is set.

5. Machine Control Instructions

Machine control instructions control the overall operation of the microprocessor. These instructions manage processor states such as halt, interrupt enable, and disable.

Common Machine Control Instructions

- **HLT** – Halt program execution
- **NOP** – No operation
- **EI** – Enable interrupts
- **DI** – Disable interrupts

Example

HLT

Stops program execution until an interrupt or reset occurs.

Importance of the Instruction Set

The instruction set enables the 8085 microprocessor to:

- Perform data manipulation
- Control program flow
- Handle interrupts and system operations