B.SC. CS/ BCA - SEMESTER EXAM NOTES

Computer Architecture & Organization

Complete unit-wise notes covering Basic Computer Design, RISC/CISC, Parallel Processing, Computer Arithmetic, Memory Organization, and 8085–Pentium Processors — with definitions, diagrams, and exam-focused explanations.

Unit I - Basic Org & Design

Unit II - RISC/CISC & Pipelining

Unit III - Computer Arithmetic

Unit IV - Memory Organization

Unit V - 8085 to Pentium

Table of Contents

UNIT I

Basic Computer Organization & Design — Instruction & Instruction Codes

1.1 Basic Computer Organization

DEFINITION

Computer Organization refers to the operational units and their interconnections that realize the architectural specifications of a computer. It includes the internal registers, timing and control structure, and the set of instructions the computer executes.

A basic computer consists of the following major components:

- **Memory Unit (RAM)** – Stores both data and instructions. The basic computer has 4096 (2^{12}) words, each 16 bits wide.
- **Arithmetic Logic Unit (ALU)** – Performs arithmetic (+, –, ×, ÷) and logical (AND, OR, NOT) operations.
- **Control Unit (CU)** – Fetches, decodes, and executes instructions; coordinates data flow between components.
- **Input / Output (I/O) Devices** – Keyboard, monitor, printer, etc. enable communication with the external environment.
- **Common Bus System** – A 16-bit shared data pathway connecting memory, registers, and ALU.

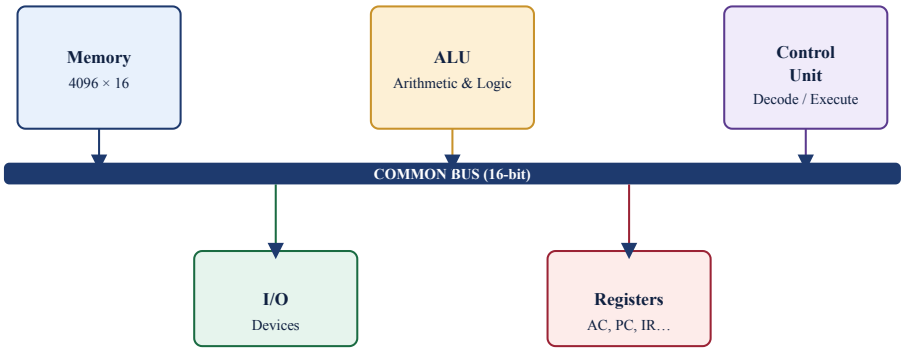

Fig 1.1 – Basic Computer Block Diagram

1.2 Instruction Codes

> **DEFINITION**
>
> **Instruction Code** is a group of bits that instruct the computer to perform a specific operation. It is divided into two parts: the *operation code (opcode)* and the *address field*.

16-bit Instruction Format (Basic Computer)

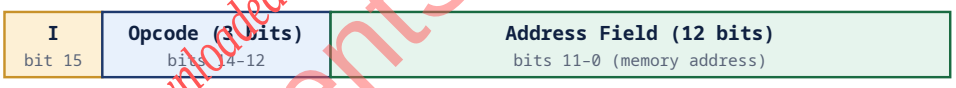

Fig 1.2 – 16-bit Instruction Format of Basic Computer

- **I (bit 15)** – Addressing mode bit: 0 = Direct, 1 = Indirect.
- **Opcode (bits 14–12)** – 3 bits → 8 possible opcodes (D0–D7).
- **Address field (bits 11–0)** – 12 bits → addresses 4096 memory locations.

Types of Instructions

Type	Identified By	Examples
Memory-Reference	Opcode ≠ 111 (D7=0)	AND, ADD, LDA, STA, BUN, BSA, ISZ
Register-Reference	Opcode = 111, I = 0 (D7=1, I=0)	CLA, CLE, CMA, CME, CIR, CIL, INC, SPA, SNA, SZA, SZE, HLT
I/O Instruction	Opcode = 111, I = 1 (D7=1, I=1)	INP, OUT, SKI, SKO, ION, IOF

📌 KEY POINT

When D7 = 1 and I = 0 → Register-reference instruction. When D7 = 1 and I = 1 → I/O instruction. For D7 = 0 → Memory-reference instruction.

1.3 Computer Registers

DEFINITION

Registers are small, high-speed storage locations within the CPU used to hold data temporarily during program execution. They are faster than main memory and directly accessible by the processor.

Types of Registers in Basic Computer

Register	Size	Name	Function
DR	16 bits	Data Register	Holds memory operand during ALU operations
AR	12 bits	Address Register	Holds memory address for read/write

Register	Size	Name	Function
AC	16 bits	Accumulator	Main working register; stores ALU results
IR	16 bits	Instruction Register	Holds the current instruction being executed
PC	12 bits	Program Counter	Points to the address of the next instruction
TR	16 bits	Temporary Register	Holds temporary data during micro-operations
OUTR	8 bits	Output Register	Holds output character to send to output device
INPR	8 bits	Input Register	Receives character from input device
SC	4 bits	Sequence Counter	Counts timing pulses T0–T15

Special Purpose vs General Purpose Registers

General Purpose Registers

- Can hold any data or address value
- Used by programmer for intermediate calculations
- Examples: AX, BX, CX, DX (in x86), R0–R15 (in ARM)
- Increase processor flexibility

Special Purpose Registers

- Dedicated to a specific function

- Examples: PC (Program Counter), SP (Stack Pointer), IR (Instruction Register), MAR, MDR
- Cannot be used freely by the programmer

Index Register

DEFINITION

Index Register is a special CPU register used to modify operand addresses during program execution. It is commonly used for array traversal and loop operations by adding an offset to a base address.

Example: If base address = 1000 and index register = 5, the effective address = 1000 + 5 = 1005.

1.4 Timing and Control Unit

DEFINITION

The **Control Unit** generates timing and control signals to coordinate the operations of all components of the computer. It translates the decoded instruction into a sequence of control signals.

Two types of control organization:

- **Hardwired Control** – Uses combinational logic circuits (gates, flip-flops). Very fast but inflexible. Difficult to modify.
- **Microprogrammed Control** – Uses a control memory (ROM) storing microinstructions. More flexible and easier to modify. Slightly slower.

Timing Signals

The sequence counter (SC) generates timing pulses **T0, T1, T2, T3, T4 …** in sequence, one per clock cycle. These pulses activate the correct registers and paths at each step of execution.

1.5 Instruction Cycle

DEFINITION

The **Instruction Cycle** (also called the Fetch-Decode-Execute cycle) is the sequence of steps that the CPU performs to fetch, decode, and execute each instruction in a program.

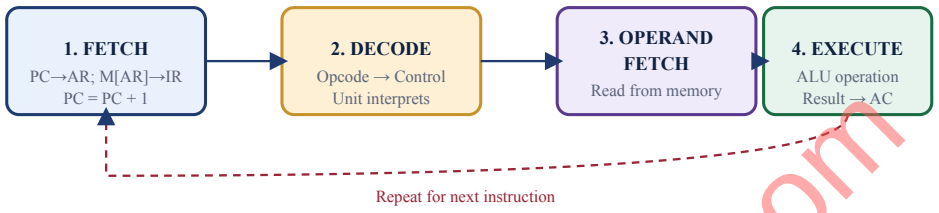

Fig 1.3 – Instruction Cycle (Fetch–Decode–Execute)

Micro-operations in Fetch Cycle

```
T0: AR ← PC              (Transfer PC to AR)
T1: IR ← M[AR], PC ← PC+1  (Load instruction, increment PC)
T2: AR ← IR(0-11), D ← Decode IR(12-14), I ← IR(15)
```

1.6 Register Transfer and Micro-Operations

DEFINITION

Register Transfer is the movement of data between registers within the CPU as part of executing instructions. It involves the controlled transfer of binary data from one register to another, often through arithmetic or logic operations.

Register Transfer Language (RTL) is a symbolic notation used to describe micro-operations:

```
R2 ← R1          (Transfer content of R1 to R2)
R1 ← R1 + R2     (Add R1 and R2, store in R1)
M[AR] ← DR       (Write DR content to memory at address AR)
DR ← M[AR]       (Read memory at AR into DR)
```

Types of Micro-Operations

- **Transfer** – Move data between registers (e.g., R1 ← R2)
- **Arithmetic** – Add, subtract, increment, decrement (e.g., AC ← AC + DR)
- **Logic** – AND, OR, XOR, complement (e.g., AC ← AC ∧ DR)
- **Shift** – Shift left or right (e.g., AC ← shl AC)

1.7 Memory Reference Instructions

Mnemonic	Operation	Hex (Direct)
AND	AC ← AC ∧ M[EA]	0xxx
ADD	AC ← AC + M[EA]	1xxx
LDA	AC ← M[EA]	2xxx
STA	M[EA] ← AC	3xxx
BUN	PC ← EA (unconditional branch)	4xxx
BSA	M[EA] ← PC; PC ← EA+1 (branch and save)	5xxx
ISZ	M[EA] ← M[EA]+1; if zero → skip next	6xxx

1.8 I/O and Interrupts

DEFINITION

An **Interrupt** is a signal sent to the CPU from an I/O device or software to request immediate attention, causing the CPU to temporarily suspend the current program and handle the interrupt service routine (ISR).

I/O Instructions

- **INP** – Transfer 8-bit character from INPR to AC(0–7)
- **OUT** – Transfer 8-bit character from AC(0–7) to OUTR
- **SKI** – Skip next if input flag (FGI) is set
- **SKO** – Skip next if output flag (FGO) is set
- **ION** – Enable interrupt (IEN ← 1)
- **IOF** – Disable interrupt (IEN ← 0)

Interrupt Cycle

When IEN=1 and (FGI=1 or FGO=1), after instruction execution:

```
T0: M[0] ← PC, PC ← 1  (Save return address)
T1: IEN ← 0, R ← 0, SC ← 0  (Disable interrupts, clear R fl
```

🔑 KEY POINT

Programmed I/O (polling) wastes CPU cycles; Interrupt-driven I/O is more efficient because the CPU only responds when the device is ready.

UNIT II

RISC/CISC, Parallel Processing, Pipelining & Vector Processing

2.1 CISC – Complex Instruction Set Computer

DEFINITION

CISC is a CPU design philosophy where a single instruction can execute several low-level operations such as loading from memory, performing arithmetic, and storing to memory — all in one instruction.

Characteristics of CISC

- Large number of instructions (300–500+)
- Variable-length instructions
- Many addressing modes
- Instructions take multiple clock cycles
- Pipelining is difficult to implement
- Hardware-intensive; complex decoder circuits
- Examples: Intel x86, VAX, IBM System/360

REMEMBER

CISC was designed to minimize the number of instructions per program, especially when memory was expensive. The idea: "do more with fewer instructions."

2.2 RISC – Reduced Instruction Set Computer

DEFINITION

RISC is a CPU design based on a small, highly optimized set of instructions that each executes in exactly one clock cycle, enabling efficient pipelining and simpler hardware design.

Characteristics of RISC

- Small, uniform instruction set (≈50–100 instructions)
- Fixed-length instructions (easy to pipeline)
- Single-cycle execution (CPI ≈ 1)
- Large number of general-purpose registers
- Only LOAD and STORE instructions access memory
- Simple addressing modes
- Low power consumption, smaller chip size
- Examples: ARM, MIPS, SPARC, PowerPC

Berkeley RISC

The Berkeley RISC project (early 1980s, UC Berkeley) pioneered RISC concepts with the **RISC-I** and **RISC-II** processors. Key innovations:

- **Register Windows** – Multiple overlapping sets of registers to reduce procedure call overhead
- **Overlapping Execution** – Instruction phases overlap (pipelining)
- RISC-II had 138 registers organized in 8 windows of 32 registers each

RISC vs CISC Comparison

Feature	RISC	CISC
Instruction Size	Fixed (32-bit)	Variable (1–15 bytes)
Instructions per Task	More instructions	Fewer instructions
Execution Time (CPI)	≈1 cycle	2–15+ cycles
Addressing Modes	Few (2–3)	Many (10–20+)
Pipelining	Easy	Complex/Difficult
Registers	Many (32+)	Few (8–16)
Control Unit	Hardwired	Microprogrammed

Feature	RISC	CISC
Power	Low	High
Examples	ARM, MIPS	Intel x86, VAX

2.3 Parallel Processing

DEFINITION

Parallel Processing is the simultaneous use of multiple processing units to execute multiple instructions or tasks at the same time, improving computational throughput and speed.

Flynn's Classification (1966)

Class	Full Name	Description	Example
SISD	Single Instruction Single Data	One instruction, one data stream — classical uniprocessor	Traditional Von Neumann
SIMD	Single Instruction Multiple Data	One instruction operates on multiple data — array processors	GPU, Vector processor
MISD	Multiple Instruction Single Data	Multiple instructions on same data — rare, theoretical	Fault-tolerant systems
MIMD	Multiple Instruction Multiple Data	Multiple processors, each executing different instructions on different data	Multi-core CPUs, Clusters

2.4 Pipelining

> **DEFINITION**
>
> **Pipelining** is a technique in which multiple instructions are overlapped in execution, similar to an assembly line. Different stages of different instructions are executed simultaneously, increasing instruction throughput.

4-Stage Instruction Pipeline

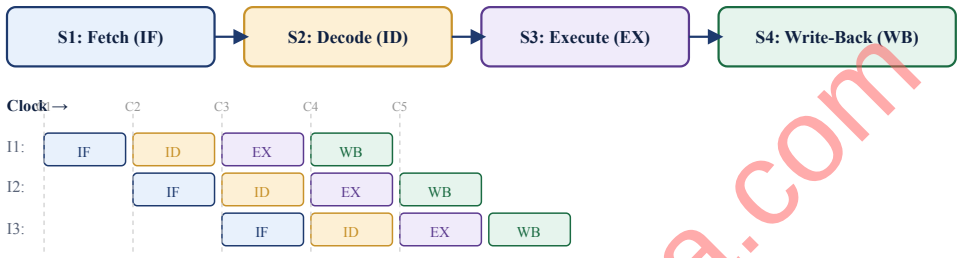

Fig 2.1 – Instruction Pipeline Timing Diagram

Types of Pipelines

- **Arithmetic Pipeline** – Used for floating-point operations; breaks computation into sub-operations (add exponents, multiply mantissa, normalize).
- **Instruction Pipeline** – Overlaps stages of instruction execution (IF, ID, EX, WB).
- **RISC Pipeline** – RISC processors use simple, uniform instructions designed for easy pipelining with CPI ≈ 1.

Pipeline Hazards

- **Data Hazard** – An instruction needs data not yet computed by a previous instruction.
- **Control Hazard** – Branch instructions disturb the sequential flow.
- **Structural Hazard** – Two instructions need the same hardware resource simultaneously.

⚠ IMPORTANT FORMULA

Speedup of k-stage pipeline = k × (execution time without pipeline) / (execution time with pipeline)

For n instructions and k stages: Time = (k + n – 1) × clock cycle

2.5 Vector Processing & Array Processor

DEFINITION

Vector Processing refers to executing a single instruction on an entire array (vector) of data elements simultaneously, rather than operating on individual scalar values.

- Operates on **vectors** (ordered sequences of data elements)
- Single instruction: e.g., `VADD V1, V2, V3` adds two 64-element vectors at once
- Used extensively in scientific computing (matrix operations, weather simulations)
- Uses **vector registers** to hold arrays

Array Processor

DEFINITION

An **Array Processor** is a parallel computing unit consisting of multiple Processing Elements (PEs) that all execute the same instruction simultaneously on different data elements (SIMD architecture).

- Classic example: ILLIAC IV (first large-scale parallel processor)
- All PEs share a single control unit but have their own local memory
- Highly efficient for regular, data-parallel computations

- Modern equivalent: GPU (Graphics Processing Unit)

UNIT III

Computer Arithmetic

3.1 Addition and Subtraction

DEFINITION

Binary Addition and Subtraction are the fundamental arithmetic operations performed in the ALU using combinational circuits called adders, operating on 2's complement binary numbers.

Binary Addition Rules

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10  (sum=0, carry=1)
```

2's Complement Subtraction

Subtraction (A – B) is performed by adding A and the 2's complement of B:

```
A - B  =  A + (2's complement of B)
       =  A + (~B + 1)

Example:  9 - 5 in 4-bit:
  9 = 1001
  5 = 0101  →  2's complement = 1011
  1001 + 1011 = 10100  →  ignore carry → 0100 = 4 ✓
```

Overflow Detection

Overflow occurs when the result of an arithmetic operation exceeds the representable range.

- Overflow if: carry into the sign bit ≠ carry out of the sign bit
- Alternatively: V = C_n XOR C_{n-1}

3.2 Multiplication Algorithm

> **DEFINITION**
>
> Binary multiplication involves forming partial products by multiplying the multiplicand by each bit of the multiplier, then shifting and adding all partial products to form the final result.

Booth's Multiplication Algorithm

> **DEFINITION**
>
> **Booth's Algorithm** is an efficient signed binary multiplication algorithm that encodes the multiplier to reduce the number of additions, handling both positive and negative numbers in 2's complement form.

Booth's Algorithm Steps

1. Initialize: AC = 0, Q_{-1} = 0
2. Examine the last bit of Q (Q_0) and Q_{-1}
3. If Q_0Q_{-1} = 10: AC ← AC – M (subtract multiplicand)
4. If Q_0Q_{-1} = 01: AC ← AC + M (add multiplicand)
5. If Q_0Q_{-1} = 00 or 11: No operation
6. Arithmetic right shift (AC, Q, Q_{-1})
7. Repeat for n cycles (n = number of bits)

Array Multiplier

Uses an array of AND gates to form all partial products simultaneously, then adds them using a cascade of adders. Fast but requires more hardware ($O(n^2)$ gates for n-bit multiplier).

3.3 Division Algorithm

> **DEFINITION**
>
> Binary division produces a quotient and a remainder. The two main hardware division algorithms are **Restoring Division** and **Non-Restoring Division**.

Restoring Division

1. Left-shift (AC, Q) by 1
2. AC ← AC – M (subtract divisor)
3. If AC ≥ 0: set $Q_0 = 1$ (quotient bit)
4. If AC < 0: set $Q_0 = 0$, restore AC ← AC + M
5. Repeat for n steps

Non-Restoring Division

More efficient — does not restore the partial remainder after each step:

- If AC ≥ 0: shift left, AC ← AC – M
- If AC < 0: shift left, AC ← AC + M
- Final step: if AC < 0, add M to restore the remainder

> **KEY DIFFERENCE**
>
> Non-restoring division is faster because it eliminates the restore step, but requires a final correction step if the final remainder is negative.

3.4 Floating-Point Arithmetic Operations

DEFINITION

A **Floating-Point Number** represents very large or very small values using a mantissa (significand) and an exponent: $N = M \times B^E$, where M = mantissa, B = base (usually 2), E = exponent.

IEEE 754 Single Precision (32-bit) Format

Fig 3.1 – IEEE 754 Single Precision Floating-Point Format

Floating-Point Addition / Subtraction Algorithm

1. **Check for zeros** – If either operand is zero, result is the other operand
2. **Align exponents** – Shift the mantissa of the smaller exponent number right until exponents are equal
3. **Add / Subtract mantissas** – Perform addition or subtraction
4. **Normalize** – Shift result to ensure leading 1 in mantissa; adjust exponent
5. **Check for overflow / underflow** in exponent

Floating-Point Multiplication

```
Product = (M1 × M2) × 2^(E1 + E2 - bias)
Steps:
  1. Add exponents: E = E1 + E2 - 127 (adjust bias)
  2. Multiply mantissas: M = M1 × M2
  3. Normalize the result
  4. Set sign: S = S1 XOR S2
```

Floating-Point Division

```
Quotient = (M1 / M2) × 2^(E1 - E2 + bias)
Steps:
  1. Subtract exponents: E = E1 - E2 + 127
```

```
2. Divide mantissas: M = M1 / M2
3. Normalize the result
4. Set sign: S = S1 XOR S2
```

3.5 Decimal Arithmetic Unit & Operations

DEFINITION

A **Decimal Arithmetic Unit** performs arithmetic operations directly on BCD (Binary-Coded Decimal) numbers, where each decimal digit (0–9) is represented by a 4-bit binary code.

BCD Addition

BCD addition requires a correction step: if the sum of two BCD digits exceeds 9 (or a carry is generated), add 6 (0110) to the result to correct the invalid BCD code.

```
Example:  5 + 8 = 13
  0101 + 1000 = 1101 (D in hex = 13 in decimal)
  Since result > 9: add 0110
  1101 + 0110 = 1 0011 = BCD 13 ✓ (carry=1, digit=3)
```

BCD Subtraction

Performed using 10's complement: subtract using 9's complement + 1 method, or add the 10's complement of the subtrahend.

🔑 KEY POINT

The BCD correction factor (+6) is added when the 4-bit sum exceeds 1001 (9) or a carry out of the BCD group is generated. BCD is widely used in financial and commercial applications where decimal accuracy is critical.

UNIT IV

Memory Organization & Cache Memory

4.1 Memory Hierarchy

DEFINITION

Memory Hierarchy is the organization of different types of storage in a computer system, arranged by speed, cost, and capacity — from fast and expensive (registers) to slow and cheap (secondary storage).

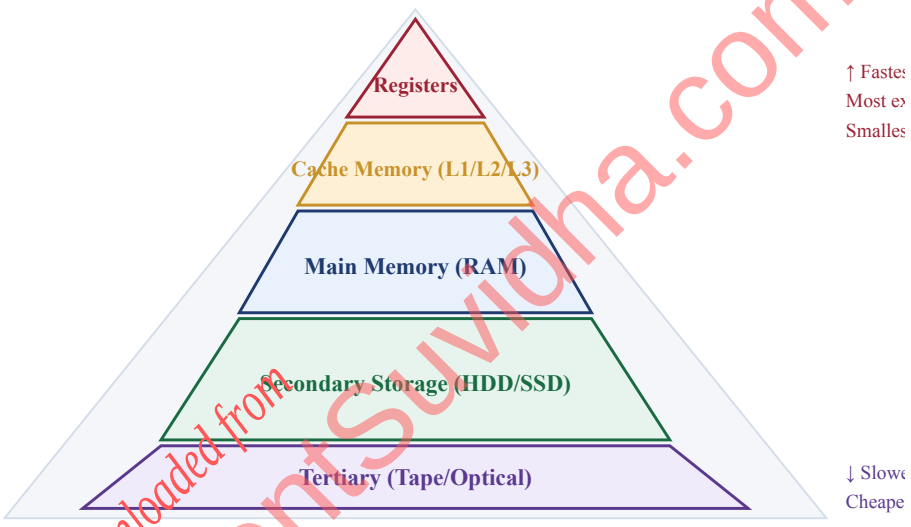

Fig 4.1 – Memory Hierarchy Pyramid

4.2 Cache Memory

DEFINITION

Cache Memory is a small, ultra-fast memory placed between the CPU and main memory to reduce the average time to access data. It stores frequently used data and instructions temporarily.

Cache Mapping Techniques

- **Direct Mapping** – Each main memory block maps to exactly one cache line. Simple but can cause conflicts. Formula: Cache line = (Block number) mod (Number of cache lines)
- **Fully Associative Mapping** – Any block can be placed in any cache line. Most flexible, but expensive hardware (all cache lines searched in parallel).
- **Set-Associative Mapping** – Cache divided into sets; each block maps to a specific set, but can occupy any line within the set. Best compromise (n-way set associative).

Cache Replacement Policies

- **LRU (Least Recently Used)** – Replace the line not used for the longest time. Most common and effective.
- **FIFO (First In First Out)** – Replace the oldest loaded line.
- **LFU (Least Frequently Used)** – Replace the line used least frequently.
- **Random** – Replace a randomly selected line. Simple but unpredictable.

Hit Ratio & Effective Access Time

> ⚠ IMPORTANT FORMULAS
>
> **Hit Ratio (h)** = Number of cache hits / Total memory accesses
>
> **Effective Access Time (T_{eff})** = $h \times T_{cache} + (1–h) \times T_{main}$

4.3 Virtual Memory

> DEFINITION
>
> **Virtual Memory** is a memory management technique that gives programs the illusion of having more memory than physically available, by temporarily storing inactive pages/segments on secondary storage (disk).

Paging

- Physical memory divided into fixed-size **frames**
- Logical memory divided into fixed-size **pages** (same size as frames)
- A **page table** maps logical pages to physical frames
- Page fault: required page not in memory → OS loads it from disk

Segmentation

Divides memory into logical variable-size **segments** (code, stack, data). Each segment has a base address and limit. More flexible than paging but suffers from external fragmentation.

Page Replacement Algorithms

- **FIFO** – Replace oldest page. Simple but may replace frequently used pages.
- **LRU** – Replace the page not used for the longest time. Performs well but hardware-intensive.
- **Optimal (OPT)** – Replace page not needed for the longest time in future. Theoretical benchmark only.

4.4 Associative Memory (Content-Addressable Memory)

> DEFINITION
>
> **Associative Memory (CAM)** is a type of memory that is searched by content (the data value) rather than by address. It compares the input word with all stored words simultaneously and returns the address of the matching word.

- Very fast but expensive (parallel comparators for every word)
- Used in TLBs (Translation Lookaside Buffers) for virtual memory
- Key operation: Match + Read output

UNIT V

Overview of 8085 to Intel Pentium Processors & Assembly Language

5.1 Evolution of Intel Processors: 8085 → Pentium

Processor	Year	Bits	Transistors	Key Feature
Intel 8085	1977	8-bit	~6,500	5-interrupt system, 16-bit address bus
Intel 8086	1978	16-bit	~29,000	First x86, 1MB memory address space
Intel 80286	1982	16-bit	134,000	Protected mode, 16MB address
Intel 80386	1985	32-bit	275,000	32-bit registers, virtual memory
Intel 80486	1989	32-bit	1.2 million	Built-in FPU, on-chip 8KB cache
Pentium	1993	32-bit	3.1 million	Superscalar (2 pipelines), 64-bit bus
Pentium Pro	1995	32-bit	5.5 million	Out-of-order execution, branch prediction
Pentium MMX	1997	32-bit	4.5 million	MMX instructions (multimedia)
Pentium III/IV	1999–2000	32-bit	9.5–42M	SSE instructions, hyper-threading

5.2 Intel 8085 Architecture

> **DEFINITION**
>
> The **Intel 8085** is an 8-bit microprocessor with a 16-bit address bus, capable of addressing up to 64KB of memory. It operates at 3–6 MHz and uses a single 5V supply.

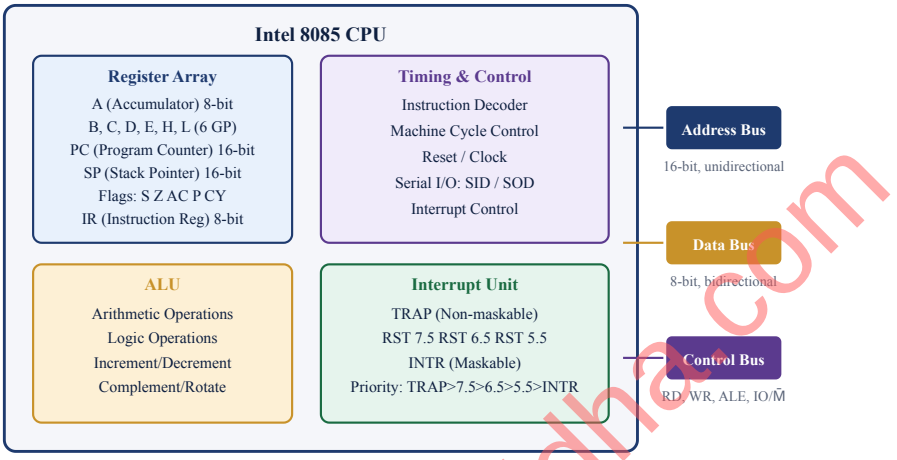

Fig 5.1 – Intel 8085 Architecture Block Diagram

Key Features of 8085

- 8-bit data bus, 16-bit address bus (can address 64 KB = 2^{16} bytes)
- 6 general-purpose registers: B, C, D, E, H, L (can be paired: BC, DE, HL)
- Accumulator A (8-bit) — main arithmetic register
- 5 flags: Sign (S), Zero (Z), Auxiliary Carry (AC), Parity (P), Carry (CY)
- 5 interrupt signals with priority: TRAP > RST7.5 > RST6.5 > RST5.5 > INTR
- Serial communication pins: SID (Serial Input Data), SOD (Serial Output Data)

5.3 Assembly Language & Assemblers

DEFINITION

Assembly Language is a low-level programming language that uses mnemonics (symbolic names) to represent machine-level instructions. It is specific to a particular processor architecture.

DEFINITION

An **Assembler** is a system software program that translates assembly language programs (source code) into machine language (binary/object code) that the microprocessor can directly execute.

Types of Assemblers

- **One-pass Assembler** – Reads source code once; only forward references resolved with backpatching (limited).
- **Two-pass Assembler** – Pass 1: Build symbol table; Pass 2: Generate machine code using the table. More common.
- **Cross Assembler** – Runs on a different machine (host) than the target machine.
- **Meta Assembler** – Can assemble for multiple processor architectures.

8085 Instruction Set Categories

Category	Examples	Function
Data Transfer	MOV, MVI, LDA, STA, LHLD, SHLD	Move data between registers or memory
Arithmetic	ADD, ADC, SUB, SBB, INR, DCR, DAD	Arithmetic operations on registers
Logical	ANA, ORA, XRA, CMA, CMP, RLC, RRC	Bitwise logical operations and rotations
Branch Control	JMP, JZ, JNZ, JC, JNC, CALL, RET	Change program flow based on conditions

Category	Examples	Function
Stack / I/O	PUSH, POP, IN, OUT, XTHL	Stack operations and I/O transfer
Machine Control	NOP, HLT, EI, DI, RIM, SIM	Control processor operation / interrupts

5.4 Level Instructions & Addressing Modes

DEFINITION

Addressing Mode specifies how the operand of an instruction is identified or located. It defines the method used to access data in memory or registers.

Addressing Mode	Description	Example (8085)
Immediate	Operand is part of the instruction itself	MVI A, 05H (A ← 05H)
Register	Operand is in a CPU register	MOV A, B (A ← B)
Direct	Instruction contains the memory address	LDA 2050H (A ← M[2050])
Indirect	Register holds the memory address (M[reg])	MOV A, M (A ← M[HL])
Implied/Implicit	Operand implied by the opcode	CMA (A ← complement of A)

5.5 Macros and Use of Macros in I/O Instructions

DEFINITION

A **Macro** is a named sequence of assembly language instructions that can be invoked (called) by name anywhere in a program. When the assembler encounters a macro call, it replaces the call with the actual macro body (macro expansion).

Macros vs Subroutines

✔ Macro Advantages

- No CALL/RET overhead
- Faster execution (inline expansion)
- Can accept parameters (arguments)
- Good for small, frequently used code

⚖ Subroutine Advantages

- One copy in memory; saves space
- Better for large, complex procedures
- Uses CALL and RET instructions

Macro Definition & Usage (8085 Style)

```
; Define a macro to output a character
PRINT_CHAR  MACRO  CHAR
    MVI  A, CHAR      ; Load character
    OUT  01H          ; Output to port 01
ENDM

; Usage:
    PRINT_CHAR  'H'   ; Expands inline to: MVI A,'H' / OUT
    PRINT_CHAR  'I'
```

Use of Macros in I/O Instructions

Macros are commonly used to simplify repetitive I/O operations in assembly programs:

```
; Macro to read a byte from an input port
READ_PORT  MACRO  PORT_NO
    IN  PORT_NO       ; Read from I/O port
    MOV B, A          ; Store result in B
ENDM

; Macro to write to output port
WRITE_PORT  MACRO  PORT_NO, DATA
    MVI  A, DATA
    OUT  PORT_NO
ENDM
```

5.6 Program Loops

DEFINITION

A **Program Loop** is a sequence of instructions that is executed repeatedly until a specific condition is satisfied. Loops are fundamental for implementing repetitive tasks in assembly language.

Types of Loops

- **Counter-controlled Loop** – Uses a register as a counter; loop executes a fixed number of times (use DCR and JNZ)
- **Conditional Loop** – Continues until a flag condition is met (JZ, JC, JNC, etc.)
- **Infinite Loop** – Loops indefinitely (JMP to start of loop)

Example: Add 5 numbers in array at memory location 2050H

```
LXI  H, 2050H    ; HL points to array start
MVI  C, 05H      ; Counter = 5
MVI  A, 00H      ; Clear accumulator (sum = 0)
LOOP:
```

```
    ADD  M          ; AC = AC + M[HL]
    INX  H          ; Point to next element
    DCR  C          ; Decrement counter
    JNZ  LOOP       ; Repeat if counter ≠ 0
STA  3000H       ; Store sum at 3000H
HLT              ; Stop
```

5.7 Programming Arithmetic and Logic

Arithmetic Programs (8085)

16-bit Addition

```
LHLD  2050H   ; Load 16-bit number from 2050-51 into HL
XCHG          ; HL ↔ DE
LHLD  2052H   ; Load second number into HL
DAD   D       ; HL = HL + DE (16-bit add)
SHLD  2054H   ; Store result at 2054-55
HLT
```

Logical Programs

Masking – Clear lower nibble using AND

```
MVI  A, 0F5H    ; A = 1111 0101
ANI  F0H        ; AND with 1111 0000 → clears lower nibble
                ; A = 1111 0000 = F0H
STA  3000H
HLT
```

Setting bits using OR

```
MVI  A, 35H     ; A = 0011 0101
ORI  0FH        ; OR with 0000 1111 → sets lower nibble
                ; A = 0011 1111 = 3FH
HLT
```

> 🗝 KEY INSTRUCTIONS SUMMARY

> **ADD B** – A = A + B | **SUB B** – A = A – B | **ANA B** – A = A AND B | **ORA B** – A = A OR B | **XRA B** – A = A XOR B | **CMA** – A = complement of A | **RLC** – Rotate A left through carry | **RRC** – Rotate A right through carry

References

1. **Morris Mano** – *Computer System Architecture*, Prentice Hall of India (PHI)
2. **John L. Hennessy & David A. Patterson** – *Computer Organization and Design*, Morgan Kaufmann
3. **A.P. Mathur** – *Introduction to Microprocessors*, Tata McGraw-Hill
4. **William Stallings** – *Computer Organization and Architecture*, Pearson Education
5. **Ramesh Gaonkar** – *Microprocessor Architecture, Programming and Applications with 8085*, Penram International