Microprocessor

And Computer Architecture

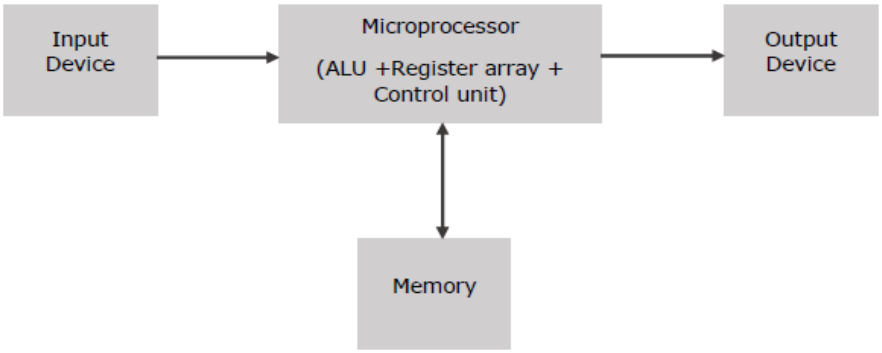
Unit I Fundamentals of Microprocessor

Introduction to Microprocessor

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.

Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator (register in which intermediate arithmetic and logic results are stored.). The control unit controls the flow of data and instructions within the computer.

Block Diagram of a Basic Microcomputer



How does a Microprocessor Work?

The microprocessor follows a sequence: Fetch, Decode, and then Execute.

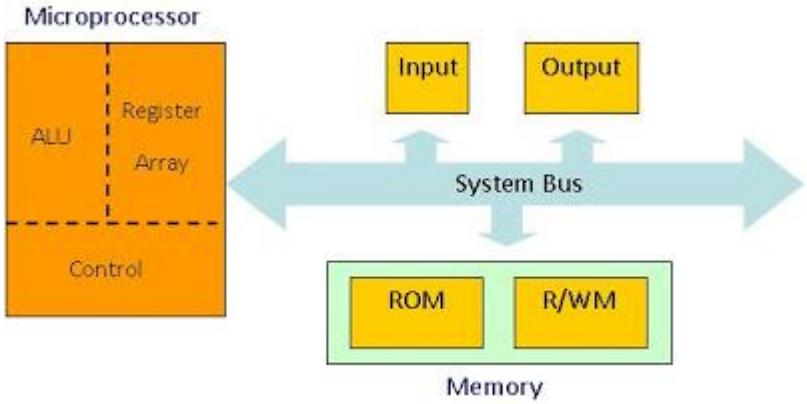
Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

Features of a Microprocessor

Here is a list of some of the most prominent features of any microprocessor −

a) **Cost-effective** − The microprocessor chips are available at low prices and results its low cost.
b) **Size** − The microprocessor is of small size chip, hence is portable.
c) **Low Power Consumption** − Microprocessors are manufactured by using metaloxide semiconductor technology, which has low power consumption.
d) **Versatility** − The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
e) **Reliability** − The failure rate of microprocessors is very low, hence it is reliable.

Microprocessor Architecture & Operation



ALU (Arithmetic/Logic Unit) – It performs such arithmetic operations as addition and subtraction, and such logic operations as AND, OR, and XOR. Results are stored either in registers or in memory.

Register Array – It consists of various registers identified by letter such as B, C, D, E, H, L, IX, and IY. These registers are used to store data and addresses temporarily during the execution of a program.

Control Unit – The control unit provides the necessary timing and control signals to all the operations in the microcomputer. It controls the flow of data between the microprocessor and memory and peripherals.

Input – The input section transfers data and instructions in binary from the outside world to the microprocessor. It includes such devices as a keyboard, switches, a scanner, and an analog-to-digital converter.

Output – The output section transfers data from the microprocessor to such output devices as LED, CRT, printer, magnetic tape, or another computer.

Memory – It stores such binary information as instructions and data, and provides that information to the microprocessor. To execute programs, the microprocessor reads instructions and data from memory and performs the computing operations in its ALU section. Results are either transferred to the output section for display or stored in memory for later use.

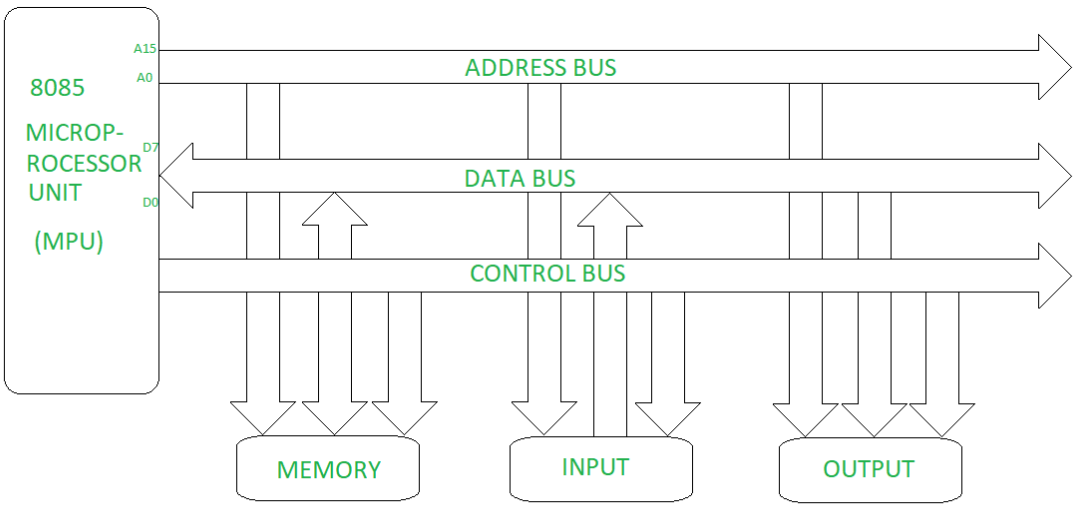
System bus – It is a communication path between the microprocessor and peripherals. The microprocessor communicates with only one peripheral at a time. The timing is provided by the control unit of the microprocessor.

Microprocessor Vs Microcontroller

Microprocessor	Microcontroller
CPU is stand alone, RAM,ROM, I/O & timer are separate.	CPU, RAM,ROM, I/O & timer all are on single chip.
Designer can decide amount of RAM,ROM, & I/O ports.	Fixed amount of on-chip RAM,ROM, & I/O ports.
High processing power	Low processing power
High power consumption	Low power consumption
Typically 32/64 bit	8/16 bit
General purpose	Single purpose(control oriented)
Less reliable	Highly reliable
Eg.- 8086,8085	8051

Bus organization of 8085 microprocessor

Bus is a group of conducting wires which carries information, all the peripherals are connected to microprocessor through Bus.



Bus organization system of 8085 Microprocessor

There are three types of buses.

a) **Address bus** – It is a group of conducting wires which carries address only. Address bus is unidirectional because data flow in one direction, from

microprocessor to memory or from microprocessor to Input/output devices (That is, Out of Microprocessor).

b) Data bus –

It is a group of conducting wires which carries Data only. Data bus is bidirectional because data flow in both directions, from microprocessor to memory or Input/output devices and from memory or Input/output devices to microprocessor.

c) Control bus –

It is a group of wires, which is used to generate timing and control signals to control all the associated peripherals, microprocessor uses control bus to process data, that is what to do with selected memory location. Some control signals are:

- Memory read
- Memory write
- I/O read
- I/O Write

8085 Microprocessor & its Operation

8085 is pronounced as "eighty-eighty-five" microprocessor. It is an 8-bit microprocessor designed by Intel in 1977 using NMOS technology.

It has the following configuration –

- 8-bit data bus
- 16-bit address bus, which can address up to 64KB
- A 16-bit program counter
- A 16-bit stack pointer
- Six 8-bit registers arranged in pairs: BC, DE, HL
- Requires +5V supply to operate at 3.2 MHZ single phase clock

It is used in washing machines, microwave ovens, mobile phones, etc.

8085 Microprocessor Architecture & Functional Units

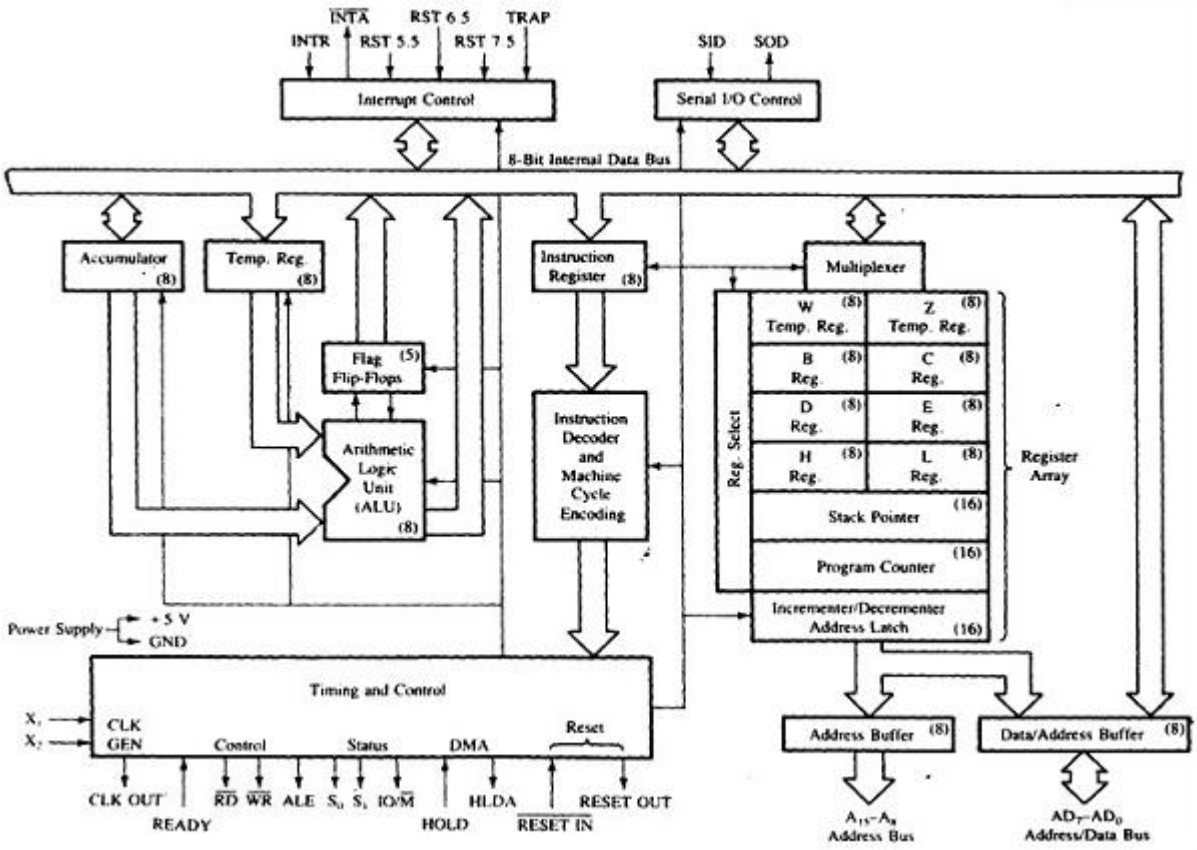


Fig: Block Diagram of 8085 Microprocessor

8085 consists of the following functional units −

1) **Accumulator**
   It is an 8-bit register used to perform arithmetic, logical, I/O & LOAD/STORE operations. It is connected to internal data bus & ALU.

2) **Arithmetic and logic unit**
   As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

3) **General purpose register**
   There are 6 general purpose registers in 8085 processor, i.e. B, C, D, E, H & L. Each register can hold 8-bit data. These registers can work in pair to hold 16-bit data and their pairing combination is like B-C, D-E & H-L.

4) **Program counter**
   It is a 16-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an

instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

5) Stack pointer

It is also a 16-bit register works like stack, which is always incremented/decremented by 2 during push & pop operations.

6) Temporary register

It is an 8-bit register, which holds the temporary data of arithmetic and logical operations.

7) Flag register

It is an 8-bit register having five 1-bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.

These are the set of 5 flip-flops –

- Sign (S)
- Zero (Z)
- Auxiliary Carry (AC)
- Parity (P)
- Carry (C)

Its bit position is shown in the following table –

D7	D6	D5	D4	D3	D2	D1	D0
S	Z		AC		P		CY

a) **Sign Flag (S) –** After any operation if result is negative sign flag becomes set, i.e. If result is positive sign flag becomes reset i.e. 0.

- **Example:**

MVI A 30 (load 30H in register A)
MVI B 40 (load 40H in register B)
SUB B (A = A – B)
These set of instructions will set the sign flag to 1 as 30 – 40 is a negative number.

MVI A 40 (load 40H in register A)
MVI B 30 (load 30H in register B)
SUB B (A = A – B)
These set of instructions will reset the sign flag to 0 as 40 – 30 is a positive number.

b) **Zero Flag (Z) –** After any arithmetical or logical operation if the result is 0 (00)H, the zero flag becomes set i.e. 1, otherwise it becomes reset i.e. 0.

- **Example:**

MVI A 10 (load 10H in register A)
SUB A (A = A – A)
These set of instructions will set the zero flag to 1 as 10H – 10H is 00H

c) **Auxiliary Carry Flag (AC) –** If intermediate carry is generated this flag is set to 1, otherwise it is reset to 0.

- **Example:**

  MOV A 2B (load 2BH in register A)
  MOV B 39 (load 39H in register B)
  ADD B (A = A + B)
  These set of instructions will set the auxiliary carry flag to 1, as on adding 2B and 39, addition of lower order nibbles B and 9 will generate a carry.

d) **Parity Flag (P) –** If after any arithmetic or logical operation the result has even parity, an even number of 1 bits, the parity register becomes set i.e. 1, otherwise it becomes reset.

1-accumulator has even number of 1 bits
0-accumulator has odd parity

e) **Carry Flag (CY) –** Carry is generated when performing n bit operations and the result is more than n bits, then this flag becomes set i.e. 1, otherwise it becomes reset i.e. 0.

During subtraction (A-B), if A>B it becomes reset and if (A<B) it becomes set.

Carry flag is also called borrow flag.

8) Instruction register and decoder

It is an 8-bit register. When an instruction is fetched from memory then it is stored in the Instruction register. Instruction decoder decodes the information present in the Instruction register.

9) Timing and control unit

It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals-

- Control Signals: READY, RD', WR', ALE
- Status Signals: S0, S1, IO/M'
- DMA Signals: HOLD, HLDA
- RESET Signals: RESET IN, RESET OUT

10) Interrupt control

As the name suggests it controls the interrupts during a process. When a microprocessor is executing a main program and whenever an interrupt occurs, the microprocessor shifts the control from the main program to process the incoming request. After the request is completed, the control goes back to the main program.

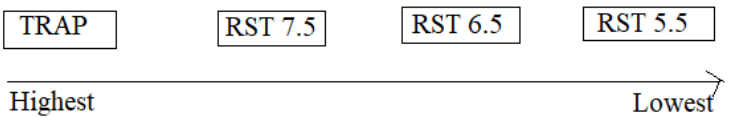
There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP. When microprocessor receives interrupt signals, it sends an acknowledgement (INTA) to the peripheral which is requesting for its service.

Maskable and Non-Maskable Interrupts

- ***Maskable*** *Interrupts* are those which can be disabled or ignored by the microprocessor. *INTR, RST 7.5, RST 6.5, RST 5.5* are maskable interrupts in 8085 microprocessor.
- **Non-Maskable** Interrupts are those which cannot be disabled or ignored by microprocessor. *TRAP* is a non-maskable interrupt.

Priority of Interrupts

When microprocessor receives multiple interrupt requests simultaneously, it will execute the interrupt service request (ISR) according to the priority of the interrupts.



11) Serial Input/output control

It controls the serial data communication by using these two instructions: SID (Serial input data) and SOD (Serial output data).

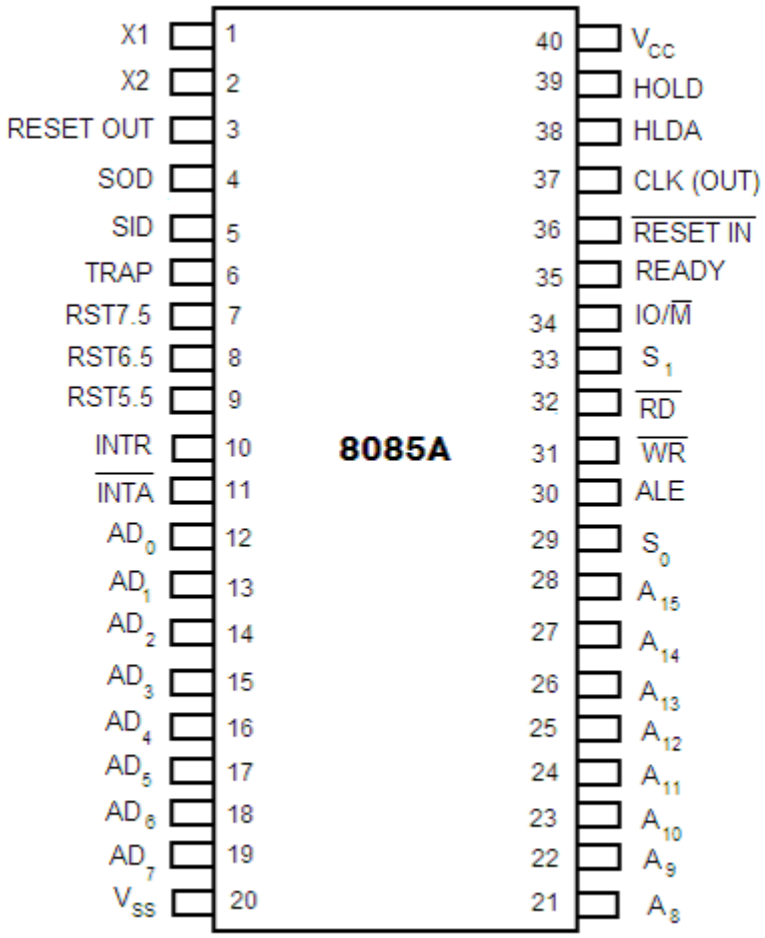
12) Address buffer and address-data buffer

The content stored in the stack pointer and program counter is loaded into the address buffer and address-data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

13) Address bus and data bus

Data bus carries the data to be stored. It is bidirectional, whereas address bus carries the location to where it should be stored and it is unidirectional. It is used to transfer the data & Address I/O devices.

8085 Pin Configuration



The pins of a 8085 microprocessor can be classified into seven groups −

1) **Address bus**
   A15-A8, it carries the most significant 8-bits of memory/IO address.

2) **Data bus**
   AD7-AD0, it carries the least significant 8-bit address and data bus.

3) **Control and status signals**
   These signals are used to identify the nature of operation. There are 3 control signal and 3 status signals.
   Three control signals are RD, WR & ALE.

a) **RD** − This signal indicates that the selected IO or memory device is to be read and is ready for accepting data available on the data bus.

b) **WR** − This signal indicates that the data on the data bus is to be written into a selected memory or IO location.

c) **ALE** − It is a positive going pulse generated when a new operation is started by the microprocessor. When the pulse goes high, it indicates address. When the pulse goes down it indicates data.

Three status signals are IO/M, S0 & S1.

IO/M
This signal is used to differentiate between IO and Memory operations, i.e. when it is high indicates IO operation and when it is low then it indicates memory operation.

S1 & S0
These signals are used to identify the type of current operation.

4) Power supply
There are 2 power supply signals − VCC & VSS. VCC indicates +5v power supply and VSS indicates ground signal.

5) Clock signals
There are 3 clock signals, i.e. X1, X2, CLK OUT.

6) Interrupts & externally initiated signals
Interrupts are the signals generated by external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.

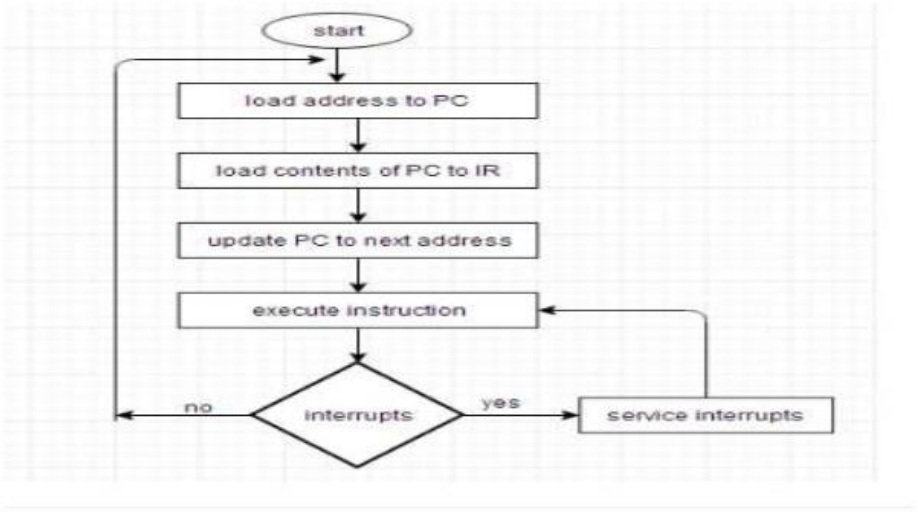
7) Serial I/O signals
There are 2 serial signals, i.e. SID (Serial output data line) and SOD (Serial input data line) and these signals are used for serial communication.

Instruction cycle in 8085 microprocessor

Time required to execute and fetch an entire instruction is called ***instruction cycle***. It consists:

- **Fetch cycle –** The next instruction is fetched by the address stored in program counter (PC) and then stored in the instruction register.
- **Decode instruction –** Decoder interprets the encoded instruction from instruction register.
- **Reading effective address –** The address given in instruction is read from main memory and required data is fetched. The effective address depends on direct addressing mode or indirect addressing mode.
- **Execution cycle –** consists memory read (MR), memory write (MW), input output read (IOR) and input output write (IOW)

General Flowchart for Instruction Cycle



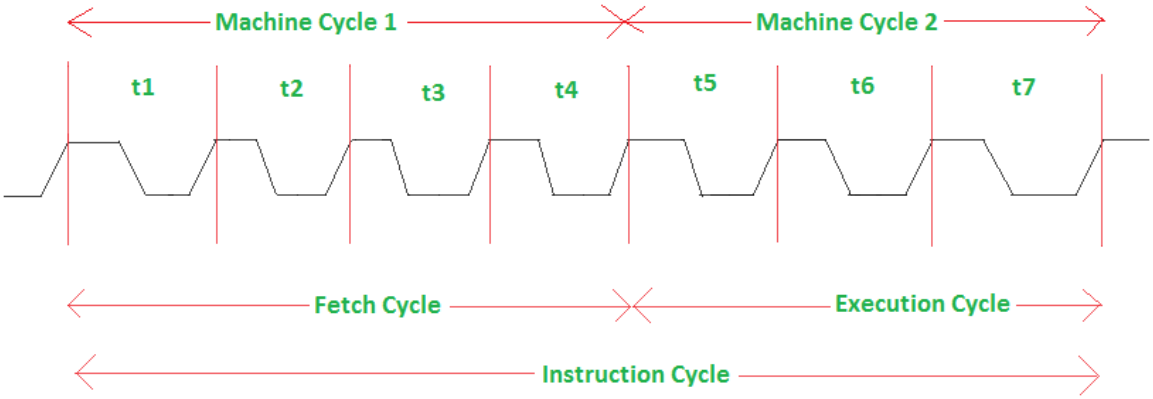
Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

The time required by the microprocessor to complete an operation of accessing memory or input/output devices is called **machine cycle**. Following are the different types of machine cycle:

- **Opcode fetch**, which takes 4 t-states
- **Memory Read**, which takes 3 t-states
- **Memory Write**, which takes 3 t-states
- **I/O Read**, which takes 3 t-states
- **I/O Write**, which takes 3 t-states

One-time period of frequency of microprocessor is called **t-state**. A t-state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse.

Fetch cycle takes four t-states and execution cycle takes three t-states. It is shown below:



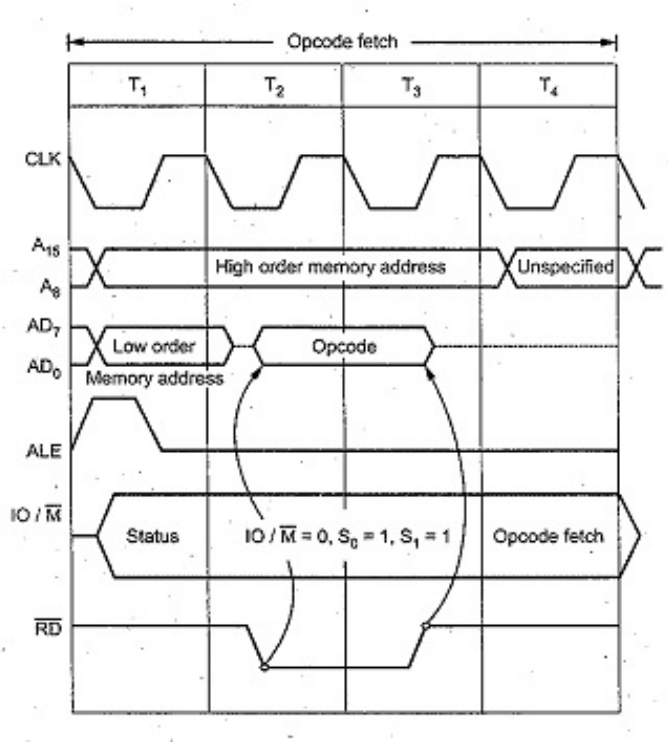
Instruction cycle in 8085 microprocessor

Different types of Machine Cycles:

Opcode Fetch Cycle

The first machine cycle of every instruction is **opcode fetch cycle** in which the 8085 finds the nature of the instruction to be executed. In this machine cycle, processor places the contents of the Program Counter on the address lines, and through the read process, reads the opcode of the instruction. The length of this cycle is not fixed. It varies from 4T states to 6T states as per the instruction.

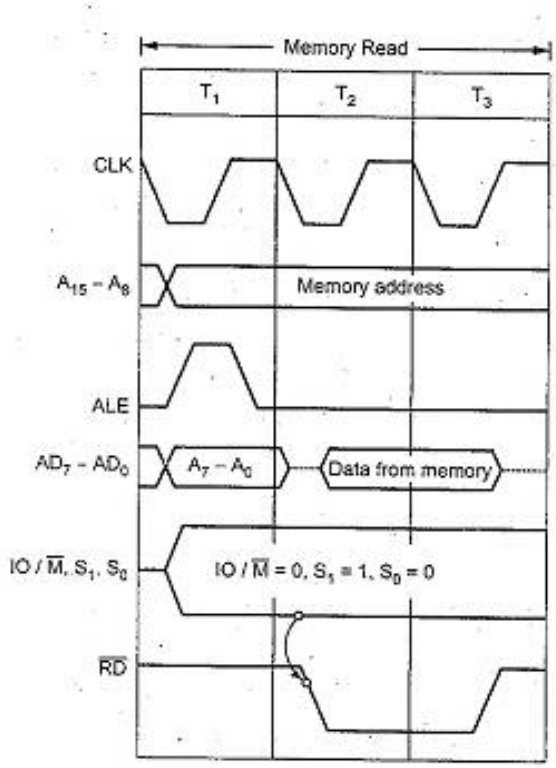
Below figure shows timing diagram of opcode fetch:



Memory Read Cycle

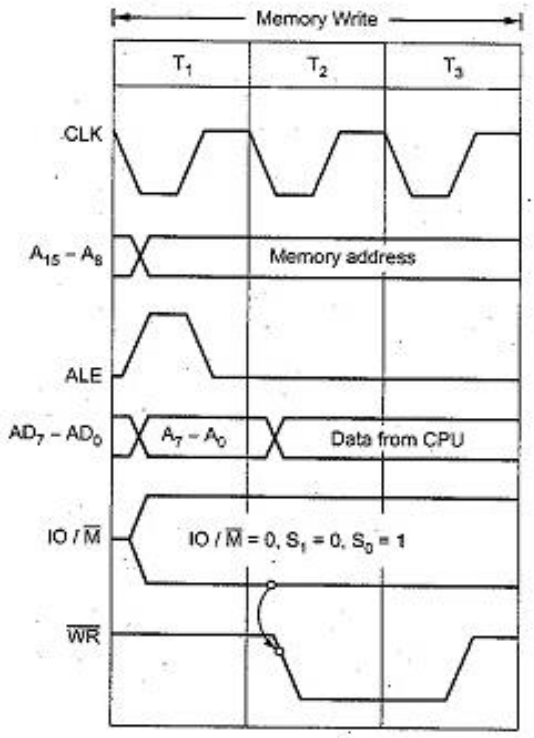
The 8085 executes the memory read cycle to read the contents of R/W memory or ROM. The length of this machine cycle is 3-T states (T1 – T3). In this machine cycle, processor places the address on the address lines from the stack pointer, general purpose register

pair or program counter, and through the read process, reads the data from the addressed memory location.
Timing diagram is shown below:



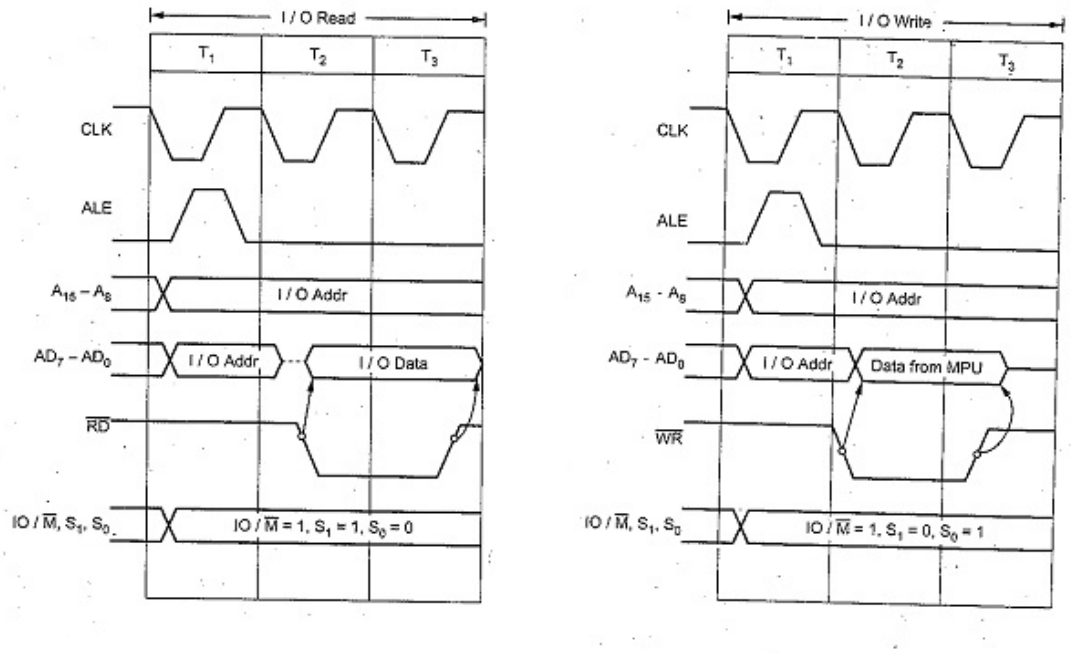
Memory Write Cycle

The 8085 executes the memory write cycle to store the data into data memory or stack memory. The length of this machine cycle is 3T states. (T1 — T3). In this machine cycle, processor places the address on the address lines from the stack pointer or general purpose register pair and through the write process, stores the data into the addressed memory location.



I/O Read and I/O Write cycles

The I/O read and I/O write machine cycles are similar to the memory read and memory write machine cycles, respectively, except that the I0/M signal is high for I/O read and I/O write machine cycles. High IO/M signal indicates that it is an I/O operation.



Timing Diagrams

Draw timing diagrams for following instructions:

a) MOV
b) MVI
c) IN
d) OUT
e) LDA
f) STA

Addressing Modes in 8085

To perform any operation, we have to give the corresponding instructions to the microprocessor. In each instruction, programmer has to specify 3 things:

1. Operation to be performed.
2. Address of source of data.
3. Address of destination of result.

The method by which the address of source of data or the address of destination of result is given in the instruction is called Addressing Modes. The term addressing mode refers to the way in which the operand of the instruction is specified.

Intel 8085 uses the following addressing modes:

1) Direct Addressing Mode
2) Register Addressing Mode
3) Register Indirect Addressing Mode
4) Immediate Addressing Mode
5) Implicit Addressing Mode

Direct Addressing Mode

In direct addressing mode, the data to be operated is available inside a memory location and that memory location is directly specified as an operand. The operand is directly available in the instruction itself.

Example:

LDA 2050 (load the contents of memory location into accumulator A)

Register Addressing Mode

In register addressing mode, the data to be operated is available inside the register(s) and register(s) is(are) operands. Therefore, the operation is performed within various registers of the microprocessor.

Examples:

MOV A, B (move the contents of register B to register A)
ADD B (add contents of registers A and B and store the result in register A)
INR A (increment the contents of register A by one)

Register Indirect Addressing Mode

In register indirect addressing mode, the data to be operated is available inside a memory location and that memory location is indirectly specified by a register pair.

Example:

MOV A, M (move the contents of the memory location pointed by the H-L pair to the accumulator)

Immediate Addressing Mode

In immediate addressing mode the source operand is always data. If the data is 8-bit, then the instruction will be of 2 bytes, if the data is of 16-bit then the instruction will be of 3 bytes.

Example:

MVI B 45 (move the data 45H immediately to register B)

Implied/Implicit Addressing Mode

In implied/implicit addressing mode the operand is hidden and the data to be operated is available in the instruction itself.

Examples:

RRC (rotate accumulator A right by one bit)

RLC (rotate accumulator A left by one bit)

Introduction to 8086

8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976. It is a 16-bit Microprocessor having 20 address lines and16 data lines that provides up to 1MB storage. It consists of powerful instruction set, which provides operations like multiplication and division easily.

It supports two modes of operation, i.e. Maximum mode and Minimum mode. Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

a) It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.
b) It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.
c) It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance. **Fetch stage** can prefetched up to 6 bytes of instructions and stores them in the queue. **Execute stage** executes these instructions.
d) It consists of 29,000 transistors.

Comparison between 8085 & 8086 Microprocessor

8085 Microprocessor	8086 Microprocessor
It is an 8 bit microprocessor.	It is a 16 bit microprocessor.
It has 16 ?bit address line.	It has 20 bit address line.
It has 8- bit data bus.	It has 16- bit data bus.
The memory capacity is 64 KB.	The memory capacity is 1 MB.
Clock speed of this microprocessor is 3 MHz.	Clock speed of this microprocessor varies between 5, 8 and 10 MHz for different versions.
It has 5 flags.	It has 9 flags.

8085 microprocessor does not support memory segmentation.	8086 microprocessor supports memory segmentation.
It does not support pipelining.	It supports pipelining.
It is accumulator based processor.	It is general purpose register based processor.
It has no minimum or maximum mode.	It has minimum and maximum modes.
In 8085, only one processor is used.	In 8086, more than one processor is used. Additional external processor can also be employed.
It contains less number of transistors compare to 8086 microprocessor. It contains about 6500 transistor.	It contains more number of transistors compare to 8085 microprocessor. It contains about 29000 in size.
The cost of 8085 is low.	The cost of 8086 is high.

Internal Architecture of 8086

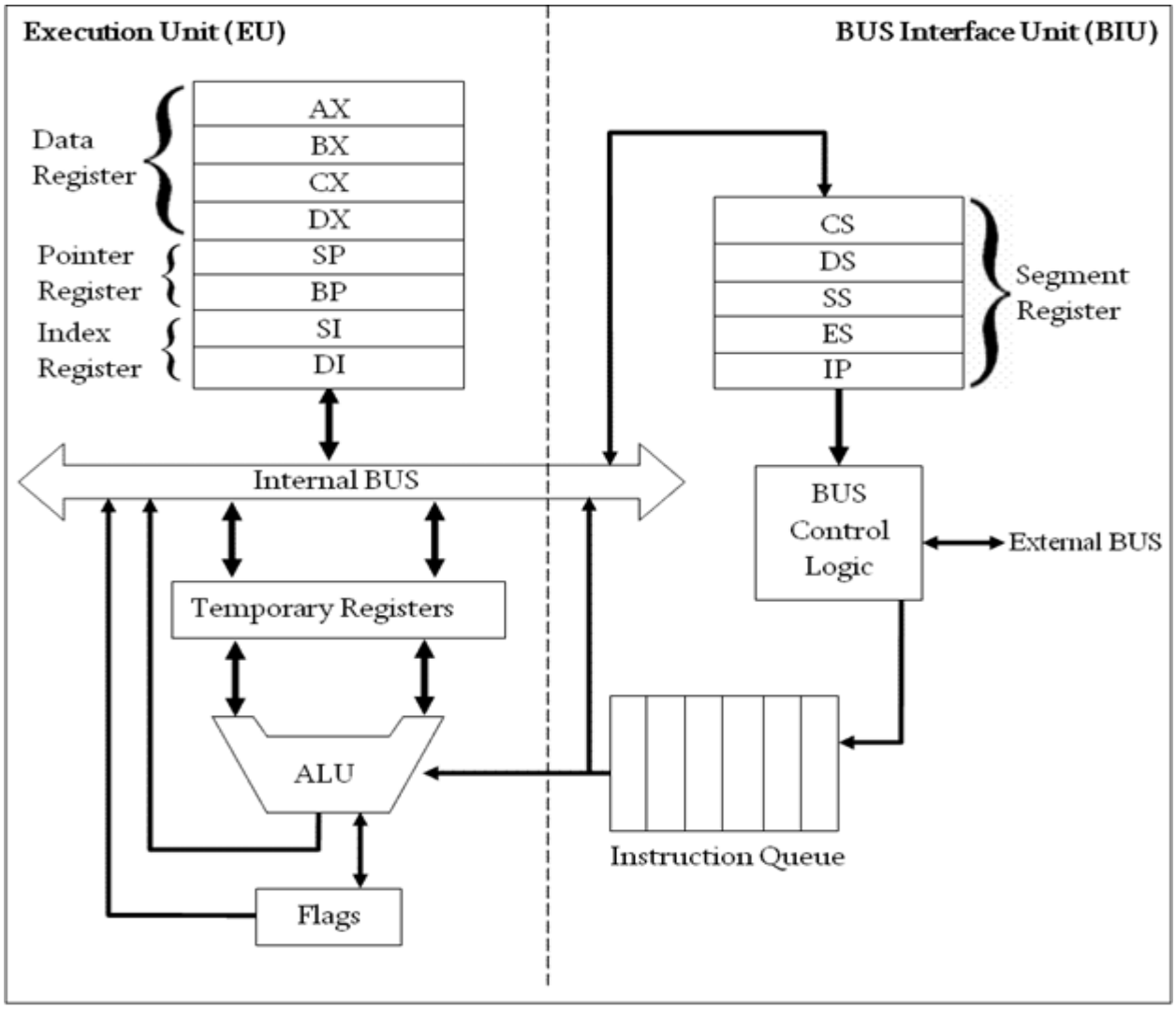


Fig: Functional Block Diagram of 8086

Functional Units

8086 contains two independent functional units: **a Bus Interface Unit (BIU)** and an **Execution Unit (EU)**.

Bus Interface Unit (BIU)

The segment registers, instruction pointer and 6-byte instruction queue are associated with the bus interface unit (BIU).

The BIU:

- Handles transfer of data and addresses,
- Fetches instruction codes, stores fetched instruction codes in first-in-first-out register set called a **queue**,
- Reads data from memory and I/O devices,
- Writes data to memory and I/O devices

It has the following functional parts:

- **Instruction Queue:** When EU executes instructions, the BIU gets 6-bytes of the next instruction and stores them in the instruction queue and this process is known as instruction pre fetch. This process increases the speed of the processor.
- **Segment Registers:** A segment register contains the addresses of instructions and data in memory which are used by the processor to access memory locations.
- There are 4 segment registers in 8086 as given below:
    - **Code Segment Register (CS):** Code segment of the memory holds instruction codes of a program.
    - **Data Segment Register (DS):** The data, variables and constants given in the program are held in the data segment of the memory.
    - **Stack Segment Register (SS):** Stack segment holds addresses and data of subroutines. It also holds the contents of registers and memory locations given in PUSH instruction.
    - **Extra Segment Register (ES):** Extra segment holds the destination addresses of some data of certain string instructions.
    - **Instruction Pointer (IP):** The instruction pointer in the 8086 microprocessor acts as a program counter. It indicates to the address of the next instruction to be executed.

Execution Unit(EU)

The **EU** receives opcode of an instruction from the queue, decodes it and then executes it. While Execution, unit decodes or executes an instruction, then the BIU fetches instruction codes from the memory and stores them in the queue.

- **General Purpose Registers:** There are four 16-bit general purpose registers: AX (Accumulator Register), BX (Base Register), CX (Counter) and DX.
- **Index Register:** The following four registers are in the group of pointer and index registers:
    - Stack Pointer (SP)
    - Base Pointer (BP)
    - Source Index (SI)
    - Destination Index (DI)
- **ALU:** It handles all arithmetic and logical operations. Such as addition, subtraction, multiplication, division, AND, OR, NOT operations.

- **Flag Register:** It is a 16 bit register which exactly behaves like a flip-flop, means it changes states according to the result stored in the accumulator. It has 9 flags and they are divided into 2 groups i.e. conditional and control flags.
    - **Conditional Flags:** This flag represents the result of the last arithmetic or logical instruction executed. Conditional flags are:
        - Carry Flag
        - Auxiliary Flag
        - Parity Flag
        - Zero Flag
        - Sign Flag
        - Overflow Flag
    - **Control Flags:** It controls the operations of the execution unit. Control flags are:
        - Trap Flag
        - Interrupt Flag
        - Direction Flag

- **Interrupts:** The Intel 8086 has two hardware interrupt pins:
  NMI (Non-Maskbale Interrupt)
  INTR (Interrupt Request) Maskable Interrupt.

Pin Configuration of 8086/Signal Diagram

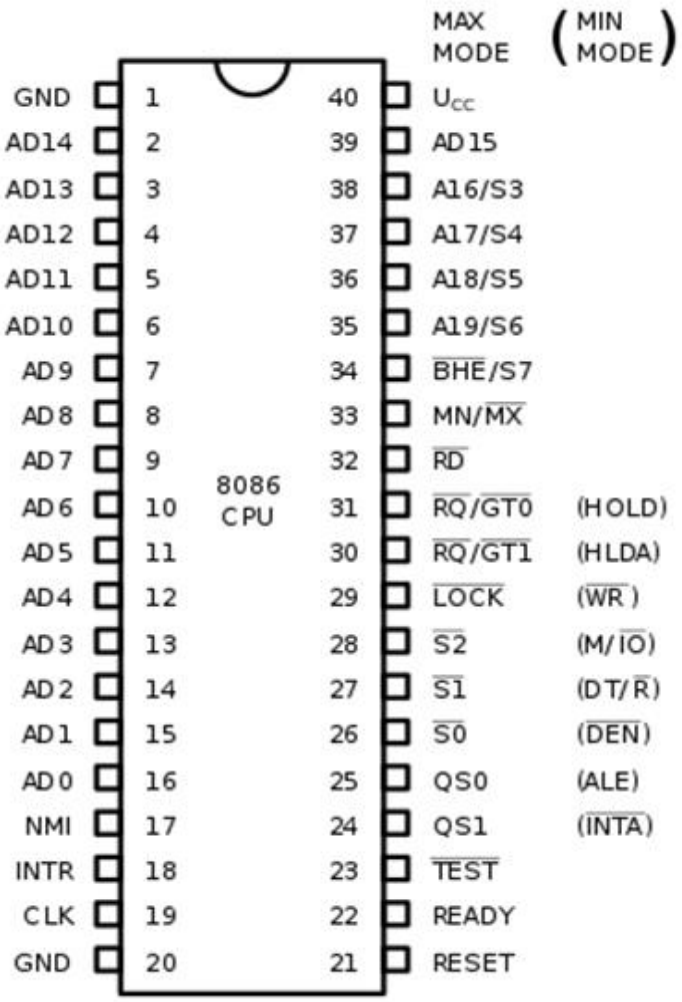


Fig: Pin Diagram of 8086

- **AD0-AD15 (Address Data Bus):** Bidirectional address/data lines. These are low order address bus. When these lines are used to transmit memory address the symbol A is used instead of AD for example A0- A15.
- **A16 - A19 (Output):** High order address lines. These are multiplexed with status signals.
- **A16/S3, A17/S4:** A16 and A17 are multiplexed with segment identifier signals S3 and S4.
- **A18/S5:** A18 is multiplexed with interrupt status S5.
- **A19/S6:** A19 is multiplexed with status signal S6.

- **BHE/S7 (Output):** Bus High Enable/Status. During T1, it is low. It enables the data onto the most significant half of data bus, D8-D15. 8-bit device connected to upper half of the data bus use BHE signal. It is multiplexed with status signal S7. S7 signal is available during T3 and T4.

- **RD (Read):** For read operation. It is an output signal. It is active when LOW.

- **Ready (Input):** The addressed memory or I/O sends acknowledgement through this pin. When HIGH it denotes that the peripheral is ready to transfer data.

- **RESET (Input):** System reset.

- **CLK (input):** Clock 5, 8 or 10 MHz.

- **INTR:** Interrupt Request.

- **NMI (Input):** Non-maskable interrupt request.

- **TEST (Input):** Wait for test control. When LOW the microprocessor continues execution otherwise waits.

- **VCC:** Power supply +5V dc.

- **GND:** Ground.

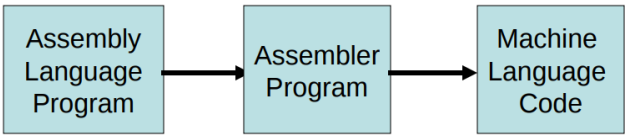
Unit II Introduction to Assembly Language Programming

Assembly Language Programming Basics

An assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations that are implemented directly on the physical CPU.

Assembly languages generally lack high-level conveniences such as variables and functions, and they are not portable between various families of processors. They have the same structures and set of commands as machine language, but allow a programmer to use names instead of numbers. This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.
Assembly language is specific to a given processor. For e.g. assembly language of 8085 is different than that of Motorola 6800 microprocessor.

Microprocessor cannot understand a program written in Assembly language. A program known as Assembler is used to convert Assembly language program to machine language.



Assembly language program to add two numbers

```
MVI A, 2H  ;Copy value 2H in register A
MVI B, 4H  ;Copy value 4H in register B
ADD B              ;A = A + B
```

Advantages of Assembly Language

a) The symbolic programming of Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
b) It is easier to correct errors and modify program instructions.
c) Assembly Language has the same efficiency of execution as the machine level language.

Disadvantages of Assembly Language

a) One of the major disadvantages is that assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration.
b) If you are programming in assembly language, you must have detailed knowledge of the particular microcomputer you are using.
c) Assembly language programs are not portable.

Instruction Set of 8085

An **instruction** is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions that a microprocessor supports is called **Instruction Set**.

8085 has **246** instructions. Each instruction is represented by an 8-bit binary value. These 8-bits of binary value is called **Op-Code** or **Instruction Byte**.

Following are the classification of instructions:

a) Data Transfer Instruction
b) Arithmetic Instructions
c) Logical Instructions
d) Branching Instructions
e) Control Instructions

a) Data Transfer Instruction

These instructions move data between registers, or between memory and registers. These instructions copy data from source to destination. While copying, the contents of source are not modified.

Example: MOV, MVI

b) Arithmetic Instructions

These instructions perform the operations like addition, subtraction, increment and decrement.

Example: ADD, SUB, INR, DCR

c) Logical Instructions

These instructions perform logical operations on data stored in registers and memory. The logical operations are: AND, OR, XOR, Rotate, Compare and Complement.

Example: ANA, ORA, RAR, RAL, CMP, CMA

d) Branching Instructions

Branching instructions refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction. The three types of branching instructions are: Jump, Call and Return.

e) Control Instructions

The control instructions control the operation of microprocessor. Examples: HLT, NOP, EI (Enable Interrupt), DI (Disable Interrupt).

Data Transfer Instructions

Instruction	Opcode	Addressing Mode	Bytes	Description
LDA address	3A	Direct	3	Transfers contents stored in memory location to accumulator.
STA address	32	Direct	3	Transfers contents stored in accumulator to memory address.
MOV A, A	7F	Register	1	Transfers the contents from one register to another
MOV A, B	78	Register	1	
MOV A, C	79	Register	1	
MOV A, D	7A	Register	1	
MOV A, E	7B	Register	1	
MOV A, H	7C	Register	1	
MOV A, L	7D	Register	1	
MOV B, A	47	Register	1	
MOV B, B	40	Register	1	
MOV B, C	41	Register	1	
MOV B, D	42	Register	1	
MOV B, E	43	Register	1	
MOV B, H	44	Register	1	
MOV B, L	45	Register	1	
MOV C, A	4F	Register	1	
MOV C, B	48	Register	1	
MOV C, C	49	Register	1	
MOV C, D	4A	Register	1	

MOV C, E	4B	Register	1	
MOV C, H	4C	Register	1	
MOV C, L	4D	Register	1	
MOV D, A	57	Register	1	
MOV D, B	50	Register	1	
MOV D, C	51	Register	1	
MOV D, D	52	Register	1	
MOV D, E	53	Register	1	
MOV D, H	54	Register	1	
MOV D, L	55	Register	1	
MOV E, A	5F	Register	1	
MOV E, B	58	Register	1	
MOV E, C	59	Register	1	
MOV E, D	5A	Register	1	
MOV E, E	5B	Register	1	
MOV E, H	5C	Register	1	
MOV E, L	5D	Register	1	
MOV H, A	67	Register	1	
MOV H, B	60	Register	1	
MOV H, C	61	Register	1	
MOV H, D	62	Register	1	
MOV H, E	63	Register	1	
MOV H, H	64	Register	1	
MOV H, L	65	Register	1	

MOV L, A	6F	Register	1	
MOV L, B	68	Register	1	
MOV L, C	69	Register	1	
MOV L, D	6A	Register	1	
MOV L, E	6B	Register	1	
MOV L, H	6C	Register	1	
MOV L, L	6D	Register	1	
MOV A, M	7E	Register Indirect	1	It moves / copies the data stored in memory location whose address is given in H-L register pair, to the given register.
MOV B, M	46	Register Indirect	1	
MOV C, M	4E	Register Indirect	1	
MOV D, M	56	Register Indirect	1	
MOV E, M	5E	Register Indirect	1	
MOV H, M	66	Register Indirect	1	
MOV L, M	6E	Register Indirect	1	
MOV M, A	77	Register Indirect	1	This instruction moves / copies the data in the given register to the memory location addressed by H-L register pair.
MOV M, B	70	Register Indirect	1	
MOV M, C	71	Register Indirect	1	
MOV M, D	72	Register Indirect	1	
MOV M, E	73	Register Indirect	1	
MOV M, H	74	Register Indirect	1	
MOV M, L	75	Register Indirect	1	

MVI A, data	3E	Immediate	2	This instruction transfers the given data immediately to the register.
MVI B, data	06	Immediate	2	
MVI C, data	0E	Immediate	2	
MVI D, data	16	Immediate	2	
MVI E, data	1E	Immediate	2	
MVI H, data	26	Immediate	2	
MVI L, data	2E	Immediate	2	
IN port-address (8-bit)	DB	Immediate	2	Load data from input port to accumulator.
OUT port-address (8-bit)	D3	Immediate	2	Transfer data to output port from accumulator.
LXI H, 16-bit data	21	Immediate	3	Transfer 16-bit data to H-L pair.
LXI D, 16-bit data	11	Immediate		Transfer 16-bit data to D-E pair.
LXI B, 16-bit data	01	Immediate	3	Transfer 16-bit data to B-C pair.
XCHG	EB	Implied/Implicit	1	Exchange contents of H-L pair and D-E pair.
LHLD address	2A	Direct	3	Load 16-bit contents of memory address to H-L pair.
SHLD address	22	Direct	3	Load 16-bit contents of H-L pair to memory address.
LDAX B	0A	Register Indirect	1	This instruction copies the contents of that memory location into the accumulator. The contents of either the register pair or the memory location are not altered.
LDAX D	1A	Register Indirect	1	

STAX B	02	Register Indirect	1	The contents of the accumulator are copied into the memory location specified by the contents of the operand (register pair). The contents of the accumulator are not altered.
STAX D	12	Register Indirect	1	
				Total = 84

Arithmetic Instructions

Instruction	Opcode	Addressing Mode	Bytes	Description
ADD A	87	Register	1	It adds the content stored in given register with the accumulator. The result of this addition is stored in accumulator.
ADD B	80	Register	1	
ADD C	81	Register	1	
ADD D	82	Register	1	
ADD E	83	Register	1	
ADD H	84	Register	1	
ADD L	85	Register	1	
ADD M	86	Register Indirect	1	adds the content of memory location whose address is given in H-L register pair with the accumulator and the answer is stored in accumulator.
ADI data	C6	Immediate	2	It immediately adds the given data with the accumulator and the answer will be stored in Accumulator.
ADC A	C6	Register	1	A=A+A+CY (Add with carry)
ADC B	8F	Register	1	A=A+B+CY

ADC C	88	Register	1	
ADC D	89	Register	1	
ADC E	8B	Register	1	
ADC H	8C	Register	1	
ADC L	8D	Register	1	
ADC M	8E	Register Indirect	1	
ACI data	CE	Immediate	2	
DAD B	09	Register	1	HL=HL+BC
DAD D	19	Register	1	HL=HL+DE
DAD H	29	Register	1	HL=HL+HL
SUB A	97	Register	1	A=A-A
SUB B	90	Register	1	A=A-B
SUB C	91	Register	1	
SUB D	92	Register	1	
SUB E	93	Register	1	
SUB H	94	Register	1	
SUB L	95	Register	1	
SUB M	96	Register Indirect	1	
SUI data	D6	Immediate	2	
SBB A	9F	Register	1	A=A-A-CY
SBB B	98	Register	1	A=A-B-CY
SBB C	99	Register	1	
SBB D	9A	Register	1	
SBB E	9B	Register	1	

SBB H	9C	Register	1	
SBB L	9D	Register	1	
SBB M	9E	Register Indirect	1	
SBI data	DE	Immediate	2	
INR A	3C	Register	1	A=A+1
INR B	04	Register	1	B=B+1
INR C	0C	Register	1	
INR D	14	Register	1	
INR E	1C	Register	1	
INR H	24	Register	1	
INR L	2C	Register	1	
INR M	34	Register Indirect	1	M=M+1 (pointed by HL)
INX B	03	Register	1	BC=BC+1
INX D	13	Register	1	DE=DE+1
INX H	23	Register	1	HL=HL+1
DCR A	3D	Register	1	A=A-1
DCR B	05	Register	1	
DCR C	0D	Register	1	
DCR D	15	Register	1	
DCR E	1D	Register	1	
DCR H	25	Register	1	
DCR L	2D	Register	1	
DCR M	35	Register Indirect	1	M=M-1
DCX B	0B	Register	1	BC=BC-1
DCX D	1B	Register	1	DE=DE-1

DCX H	2B	Register	1	HL=HL-1
RAL	17	Implied/Implicit	1	Rotate accumulator left
RAR	1F	Implied/Implicit	1	Rotate accumulator right
RLC	07	Implied/Implicit	1	Rotate accumulator left through carry
RRC	0F	Implied/Implicit	1	Rotate accumulator right through carry
				Total=65

Logical Instructions

Instruction	Opcode	Addressing Mode	Bytes	Description
ANA A	A7	Register	1	A=A AND A
ANA B	A0	Register	1	A=A AND B
ANA C	A1	Register	1	
ANA D	A2	Register	1	
ANA E	A3	Register	1	
ANA H	A4	Register	1	
ANA L	A5	Register	1	
ANA M	A6	Register Indirect	1	
ANI data	E6	Immediate	2	A=A AND data
ORA A	B7	Register	1	A=A OR A
ORA B	B0	Register	1	A=A OR B
ORA C	B1	Register	1	
ORA D	B2	Register	1	
ORA E	B3	Register	1	
ORA H	B4	Register	1	

ORA L	B5	Register	1	
ORA M	B6	Register Indirect	1	
ORI data	F6	Immediate	2	
XRA A	AF	Register	1	A=A XOR A
XRA B	A8	Register	1	A=A XOR B
XRA C	A9	Register	1	
XRA D	AA	Register	1	
XRA E	AB	Register	1	
XRA H	AC	Register	1	
XRA L	AD	Register	1	
XRA M	AE	Register Indirect	1	
XRI data	EE	Immediate	2	
CMP A	BF	Register	1	A=A-A (Accumulator remain unchanged)
CMP B	B8	Register	1	A=A-B
CMP C	B9	Register	1	
CMP D	BA	Register	1	
CMP E	BB	Register	1	
CMP H	BC	Register	1	
CMP L	BD	Register	1	
CMP M	BE	Register Indirect	1	
CPI data	FE	Immediate	2	A=A-data (Acc. Remain unchanged)
CMA	2F	Implied/implicit	1	Complement Accumulator contents
CMC	3F	Implied/implicit	1	Complement Carry flag
STC	37	Implied/implicit	1	CY=1 (It sets carry flag)
				Total=39

Branching Instructions

Instruction	Opcode	Addressing Mode	Bytes	Description
JMP address	C3	Immediate	3	Unconditional Jump
JC address	DA	Immediate	3	Jump if CY=1
JNC address	D2	Immediate	3	Jump if CY=0
JZ address	CA	Immediate	3	Jump if Z=1
JNZ address	C2	Immediate	3	Jump if Z=0
JM address	FA	Immediate	3	Jump if S=1
JP address	F2	Immediate	3	Jump if S=0
JPE address	EA	Immediate	3	Jump if P=1
JPO address	E2	Immediate	3	Jump if P=0
CALL address	CD	Immediate	3	Unconditional Call
CC address	DC	Immediate	3	Call Subroutine if CY=1
CNC address	D4	Immediate	3	Call Subroutine if CY=0
CZ address	CC	Immediate	3	Call Subroutine if Z=1
CNZ address	C4	Immediate	3	Call Subroutine if Z=0
CM address	FC	Immediate	3	Call Subroutine if S=1
CP address	F4	Immediate	3	Call Subroutine if S=0
CPE address	EC	Immediate	3	Call Subroutine if P=1
CPO address	FE	Immediate	3	Call Subroutine if P=1
RNZ	C0	Register Indirect	1	Return to main program if Z=0
RZ	C8	Register Indirect	1	Return to main program if Z=1
RNC	D0	Register Indirect	1	Return to main program if C=0
RC	D8	Register Indirect	1	Return to main program if C=1

RM	F8	Register Indirect	1	Return to main program if S=1
RP	F0	Register Indirect	1	Return to main program if S=0
RPO	E0	Register Indirect	1	Return to main program if P=0
RPE	E8	Register Indirect	1	Return to main program if P=1
				Total=26

Program Control & Stack Instructions

Instruction	Opcode	Addressing Mode	Bytes	Description
PUSH B	C5	Register Indirect	1	PUSH data from data from stack on the basis of address pointed by BC pair.
PUSH D	D5	Register Indirect	1	
PUSH H	E5	Register Indirect	1	
POP B	C1	Register Indirect	1	POP data from data from stack on the basis of address pointed by BC pair.
POP D	D1	Register Indirect	1	
POP H	E1	Register Indirect	1	
NOP	00	Implied/Implicit	1	No operation is performed
HLT	76	Implied/Implicit	1	Terminate program
DI	F3	Implied/Implicit	1	
EI	7B	Implied/Implicit	1	
RIM	20	Implied/Implicit	1	Read interrupt mask (read status of interrupt)
SIM	30	Implied/Implicit	1	Set interrupt mask (used to implement interrupt)
				Total=12 Grand Total=226

8085 Programs

1. **Program to add two 8-bit numbers.**
   Statement: *Add numbers 05H & 13H and display result in output port 03H.*

```
MVI A,05H    //Move data 05H to accumulator
MVI B,13H    //Move data 13H to B register
ADD B        //Add contents of accumulator and B register
OUT 03H      //Transfer result to output port 03H
HLT          //Terminate the program.
```

   Input: A=05H B=13H **Output**: (port 03H) = 18H

2. **Program to add two 8-bit numbers.**
   Statement: *Add numbers from memory location 2050H & 2051H and store result in memory location 2055H.*

```
LDA 2051H    //Load contents of memory location 2051 to accumulator
MOV B,A      //Move contents of accumulator to B register
LDA 2050H    //Load contents of memory location 2050 to accumulator
ADD B        // Add contents of accumulator and B register
STA 2055H    //Store contents of accumulator in memory location 2055H
HLT          //Terminate the program.
```

   Input:

Memory Location	Data
2050H	45H
2051H	53H

   Output:

Memory Location	Data
2055H	98H

3. **Program to subtract two 8-bit numbers.**
   Statement: *Subtract numbers 25H & 12H and display result in output port 01H.*

```
MVI A,25H    //Move data 05H to accumulator
MVI B,12H    //Move data 13H to B register
SUB B        //Add contents of accumulator and B register
OUT 01H      //Transfer result to output port 01H
HLT          //Terminate the program.
```

   Input: A=25H B=12H **Output**: (port 03H) = 13H

4. **Program to subtract two 8-bit numbers.**
   Statement: *Subtract numbers from memory location 2050H & 2051H and store result in memory location 2055H.*

```
LDA 2051H    //Load contents of memory location 2051 to accumulator
MOV B,A      //Move contents of accumulator to B register
```

```
LDA 2050H    //Load contents of memory location 2050 to accumulator
SUB B        // Add contents of accumulator and B register
STA 2055H    //Store contents of accumulator in memory location 2055H
HLT          //Terminate the program.
```

Input:

Memory Location	Data
2050H	65H
2051H	53H

Output:

Memory Location	Data
2055H	12H

5. **Program to find 1's complement of a number.**
   Statement: *Input number from memory location 2013H and store result in memory location 2052H.*

```
LDA 2013H    //Load contents from memory location 2013H to accumulator
CMA          //Complement contents of accumulator
STA 2052H    //Store result in memory location 2052H
HLT          //Terminate the program.
```

Input:

Memory Location	Data
2013H	12H

Output:

Memory Location	Data
2052H	EDH

6. **Program to find 2's complement of a number.**
   Statement: *Input number from memory location 2013H and store result in memory location 2052H.*

```
LDA 2013H    //Load contents from memory location 2013H to accumulator
CMA          //Complement contents of accumulator
ADI 01H      //Add 01H to the contents of accumulator
STA 2052H    //Store result in memory location 2052H
HLT          //Terminate the program.
```

Input:

Memory Location	Data
2013H	12H

Output:

Memory Location	Data
2052H	EEH

7. **Program to right shift 8-bit numbers.**
   Statement*: Shift an eight-bit data four bits right. Assume data is in memory location 2051H. Store result in memory location 2055H.*

```
LDA 2051H     //Load data from memory location 2051H to accumulator
RAR           //Rotate accumulator 1-bit right
RAR
RAR
RAR
STA 2055H     //Store result in memory location 2055H
HLT           //Terminate the program.
```

8. **Program to left shift 8-bit numbers.**

***Statement**: Shift an eight-bit data four bits left. Assume data is in memory location 2051H. Store result in memory location 2055H.*

```
LDA 2051H     //Load data from memory location 2051H to accumulator
RAL           //Rotate accumulator 1-bit left
RAR
RAR
RAR
STA 2055H     //Store result in memory location 2055H
HLT           //Terminate the program.
```

9. **Program to add two 16-bit numbers.**

***Statement:** Add numbers 1124H & 2253H and store result in memory location 2055H & 2056H.*

```
LXI H,1124H   //Load 16-bit data 1124H to HL pair
LXI D,2253H   //Load 16-bit data 2253H to DE pair
MOV A,L       //Move contents of register L to Accumulator
ADD E         //Add contents of Accumulator and E register
MOV L,A       //Move contents of Accumulator to L register
MOV A,H       //Move contents of register H to Accumulator
ADC D         //Add contents of Accumulator and D register with carry
MOV H,A       //Move contents of Accumulator to register H
SHLD 2055H    //Store contents of HL pair in memory address 2055H & 2056H
HLT           //Terminate the program.
```

Input:

Register Pair	Data
HL	1124H
DE	2253H

Output:

Memory Location	Data
2055H	77H
2056H	33H

10. **Program to add two 16-bit numbers.**

***Statement:** Input first number from memory location 2050H & 2051H and second number from memory location 2052H & 2053H and store result in memory location 2055H & 2056H.*

```
LHLD 2052H    //Load 16-bit number from memory location 2052H & 2053H to HL pair
XCHG          //Exchange contents of HL pair and DE pair
LHLD 2050H    //Load 16-bit number from memory location 2050H & 2051H to HL pair
MOV A,L       //Move contents of register L to Accumulator
ADD E         //Add contents of Accumulator and E register
MOV L,A       //Move contents of Accumulator to L register
MOV A,H       //Move contents of register H to Accumulator
ADC D         //Add contents of Accumulator and D register with carry
MOV H,A       //Move contents of Accumulator to register H
SHLD 2055H    //Store contents of HL pair in memory address 2055H & 2056H
HLT           //Terminate the program.
```

Input:

Memory Location	Data
2050H	33H
2051H	45H
2052H	24H
2053H	34H

Output:

Memory Location	Data
2055H	57H
2056H	79H

11. Program to subtract two 16-bit numbers.

Statement: *Subtract number 1234H from 4897H and store result in memory location 2055H & 2056H.*

```
LXI H,4567H   //Load 16-bit data 4897H to HL pair
LXI D,1234H   //Load 16-bit data 1234H to DE pair
MOV A,L       //Move contents of register L to Accumulator
SUB E         //Subtract contents of Accumulator and E register
MOV L,A       //Move contents of Accumulator to L register
MOV A,H       //Move contents of register H to Accumulator
SBB D         //Subtract contents of Accumulator and D register with borrow
MOV H,A       //Move contents of Accumulator to register H
SHLD 2055H    //Store contents of HL pair in memory address 2055H & 2056H
HLT           //Terminate the program.
```

Input:

Register Pair	Data
HL	4897H
DE	1234H

Output:

Memory Location	Data
2055H	63H
2056H	36H

12. Program to subtract two 16-bit numbers.

Statement: *Input first number from memory location 2050H & 2051H and second number from memory location 2052H & 2053H and store result in memory location 2055H & 2056H.*

```
LHLD 2052H    //Load 16-bit number from memory location 2052H & 2053H to HL pair
XCHG          //Exchange contents of HL pair and DE pair
LHLD 2050H    //Load 16-bit number from memory location 2050H & 2051H to HL pair
MOV A,L       //Move contents of register L to Accumulator
SUB E         //Subtract contents of Accumulator and E register
MOV L,A       //Move contents of Accumulator to L register
MOV A,H       //Move contents of register H to Accumulator
SBB D         //Subtract contents of Accumulator and D register with carry
MOV H,A       //Move contents of Accumulator to register H
SHLD 2055H    //Store contents of HL pair in memory address 2055H & 2056H
HLT           //Terminate the program.
```

Input:

Memory Location	**Data**
2050H	78H
2051H	45H
2052H	24H
2053H	34H

Output:

Memory Location	**Data**
2055H	54H
2056H	11H

13. Program to multiply two 8-bit numbers.

Statement: Multiply 06 and 03 and store result in memory location 2055H.

```
MVI A,00H
MVI B,06H
MIV C,03H
X: ADD B
DCR C
JNZ X
STA 2055H
HLT
```

14. Program to divide to 8-bit numbers.

Statement: Divide 08H and 03H and store quotient in memory location 2055H and remainder in memory location 2056H.

```
MVI A,08H
MVI B,03H
MVI C,00H
X: CMP B
    JC Y
    SUB B
    INR C
    JMP X
Y: STA 2056H
MOV A,C
STA 2055H
HLT
```

15. Program to find greatest among two 8-bit numbers.

Statement: Input numbers from memory location 2050H & 2051H and store greatest number in memory location 2055H.

```
LDA 2051H
MOV B,A
LDA 2050H
CMP B
JNC X
MOV A,B
X: STA 2055H
HLT
```

16. Program to find smallest among two 8-bit numbers.

Statement: Input numbers from memory location 2050H & 2051H and store smallest number in memory location 2055H.

```
LDA 2051H
MOV B,A
LDA 2050H
CMP B
JC X
MOV A,B
X: STA 2055H
HLT
```

17. Program to find whether a number is odd or even.

Statement: Input number from memory location 2050H and store result in 2055H.

```
LDA 2050H
ANI 01H
JZ X
MVI A,0DH
```

```
JMP Y
X: MVI A,0EH
Y: STA 2055H
HLT
```

18. Program to count no. of 1's in given number.

Statement: Input number from memory location 2050H and store result in 2055H.

```
LDA 2050H
MVI C,08H
MVI B,00H
X: RAR
    JNC Y
    INR B
Y: DCR C
    JNZ X
MOV A,B
STA 2055H
HLT
```

19. Display number from 1 to 10.

```
LXI H,2050H
MVI B,01H
MVI C,0AH
X: MOV M,B
   INX H
   INR B
   DCR C
   JNZ X
HLT
```

20. Find sum of numbers from 1 to 10.

```
LXI H,2050H
MVI B,01H
MVI C,0AH
MVI A,00H
X: ADD B
   INX H
   INR B
   DCR C
   JNZ X
STA 2055H
HLT
```

21. Display all odd numbers from 1 to 10.

```
LXI H,2050H
MVI B,01H
MVI C,0AH
X: MOV M,B
   INX H
   INR B
```

```
    INR B
    DCR C
    DCR C
    JNZ X
HLT
```

22. Display all even numbers from 1 to 20.

```
LXI H,2050H
MVI B,02H
MVI C,14H
X: MOV M,B
    INX H
    INR B
    INR B
    DCR C
    DCR C
    JNZ X
HLT
```

23. Display all even numbers from 10 to 50.

```
LXI H,2050H
MVI B,0AH
MVI C,32H
X: MOV M,B
    INX H
    INR B
    INR B
    DCR C
    DCR C
    JNZ X
HLT
```

24. Find sum of 10 numbers in array.

```
LXI H,2050H
MVI C,OAH
MVI A,00H
X: MOV B,M
    ADD B
    INX H
    DCR C
    JNZ X
STA 2060H
HLT
```

25. Find the largest element in a block of data. The length of the block is in the memory location 2200H and block itself starts from memory location 2201H. Store the maximum number in memory location 2300H.

```
LDA 2200H
MOV C,A
LXI H,2201
MVI A,00H
X: CMP M
   JNC Y
   MOV A,M
Y: INX H
   DCR C
   JNZ X
 STA 2300H
HLT
```

26. Find smallest number in array.

```
LDA 2200H
MOV C,A
LXI H,2201H
MVI A,00H
X: CMP M
   JC Y
   MOV A,M
Y: INX H
   DCR C
   JNZ X
 STA 2300H
HLT
```

27. Generate Fibonacci series upto 10^{th} term.

```
LXI H,2050H
MVI C,08H
MVI B,00H
MVI D,01H
MOV M,B
INX H
MOV M,D
X: MOV A,B
   ADD D
    MOV B,D
    MOV D,A
    INX H
    MOV M,A
    DCR C
    JNZ X
HLT
```

28. Sort 10 numbers in ascending order in array.

```
MVI C,0AH
DCR C
X: MOV D,C
   LXI H,2050H

Y: MOV A,M
   INX H
   CMP M
   JC Z

MOV B,M
MOV M,A
DCX H
MOV M,B
INX H

Z: DCR D
   JNZ Y
   DCR C
   JNZ X
HLT
```

29. Sort numbers in descending order in array. Length of array is in memory location 2050H.

```
LDA 2050H
MVI C,A
DCR C
X: MOV D,C
   LXI H,2051H

Y: MOV A,M
   INX H
   CMP M
   JNC Z

MOV B,M
MOV M,A
DCX H
MOV M,B
INX H

Z: DCR D
   JNZ Y
   DCR C
   JNZ X
HLT
```

30. Multiply two 8 bit numbers 43H & 07H. Result is stored at address 3050 and 3051.

```
LXI H,0000H
MVI D,00H
MVI E,43H
MVI C,07H
X: DAD D
   DCR C
  JNZ X
SHLD 2050H
HLT
```

31. Multiply two 8 bit numbers stored at address 2050 and 2051. Result is stored at address 3050 and 3051.

```
LDA 2050H
MOV E,A
LDA 2051H
MOV C,A
MVI D,00H
LXI H,0000H
X: DAD D
   DCR C
  JNZ X
SHLD 3050H
HLT
```

Input Data ⇨	07	43
Memory Address ⇨	2051	2050

Output Data ⇨	01	D5
Memory Address ⇨	3051	3050

Unit III Basic Computer Architecture

Introduction to Computer Architecture

Computer architecture is a specification detailing how a set of software and hardware technology standards interact to form a computer system or platform. In short, computer architecture refers to how a computer system is designed and what technologies it is compatible with.

As with other contexts and meanings of the word architecture, computer architecture is likened to the art of determining the needs of the user/system/technology, and creating a logical design and standards based on those requirements.

A very good example of computer architecture is von Neumann architecture, which is still used by most types of computers today. This was proposed by the mathematician John von Neumann in 1945. It describes the design of an electronic computer with its CPU, which includes the arithmetic logic unit, control unit, registers, memory for data and instructions, an input/output interface and external storage functions.

There are three categories of computer architecture:

a) **System Design:** This includes all hardware components in the system, including data processors aside from the CPU, such as the graphics processing unit and direct memory access. It also includes memory controllers, data paths and miscellaneous things like multiprocessing and virtualization.

b) **Instruction Set Architecture (ISA):** This is the embedded programming language of the central processing unit. It defines the CPU's functions and capabilities based on what programming it can perform or process. This includes the word size, processor register types, memory addressing modes, data formats and the instruction set that programmers use.

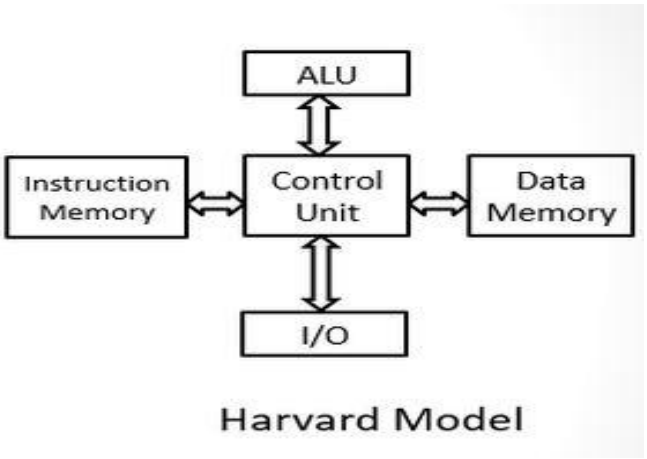
c) **Microarchitecture:** Otherwise known as computer organization, this type of architecture defines the data paths, data processing and storage elements, as well as how they should be implemented in the ISA.

History of Computer Architecture

The first document of Computer Architecture was a correspondence between Charles Babbage and Ada Lovelace, that describes the analytical engine. Here is the example of other early important machines: John Von Neumann and Alan Turing.
Computer architecture is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible with economic status and as well as the technological constraints. In ancient period, computer architectures were designed and prepared on the paper and then, directly built into the final hardware form. Later, in today`s computer architecture, prototypes were physically built in the form of transistor logic (TTL) computer such as the prototypes of the 6800 and the PA-RISC tested, and tweaked before committing to the final hardware form.

Harvard Architecture

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for instructions and data. The term originated from the Harvard Mark I, which stored instructions on punched tape and data in electro-mechanical counters. These early machines had data storage entirely contained within the central processing unit, and provided no access to the instruction storage as data. It required two memories for their instruction and data. Harvard architecture requires separate bus for instruction and data.



Harvard Model

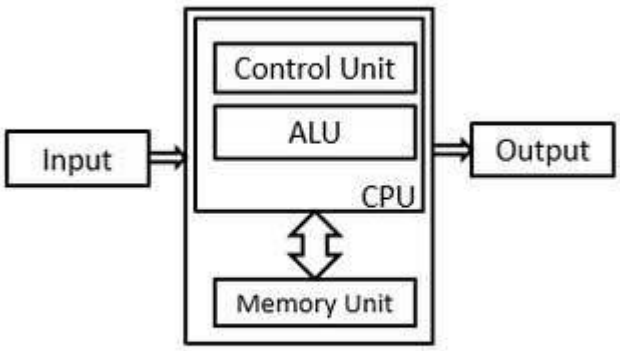
Von Neumann architecture

Von Neumann architecture was first published by John von Neumann in 1945.

His computer architecture design consists of a Control Unit, Arithmetic and Logic Unit (ALU), Memory Unit, Registers and Inputs/Outputs.

Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory. This design is still used in most computers produced today.

The modern computers are based on a stored-program concept introduced by John Von Neumann. In this stored-program concept, programs and data are stored in a separate storage unit called memories and are treated the same. Von Neumann architecture requires only one bus for instruction and data.



Von Neumann Model

Overview of Computer Organization

Computer organization refers to the operational units and their interconnection that realize the architecture specification. Computer organization deals with physical aspects of computer design, memory and their types and microprocessors design. Computer organization is concerned with the way the hardware components operate and the way they are connected together to form a computer system.

It describes how the computer performs. Ex, circuit design, control signals, memory types and etc.

Computer Organization vs Architecture

1) **Computer Architecture** refers to those attributes of a system that have a direct impact on the logical execution of a program. Examples:
   - the instruction set
   - the number of bits used to represent various data types
   - I/O mechanisms
   - memory addressing techniques

   Computer Organization refers to the operational units and their interconnections that realize the architectural specifications. Examples are things that are transparent to the programmer:
   - control signals
   - interfaces between computer and peripherals
   - the memory technology being used.

2) So, for example, the fact that a multiply instruction is available is a computer architecture issue. How that multiply is implemented is a computer organization issue.

3) **Architecture** is those attributes visible to the programmer
   - Instruction set, number of bits used for data representation, I/O mechanisms, addressing techniques. e.g. Is there a multiply instruction?

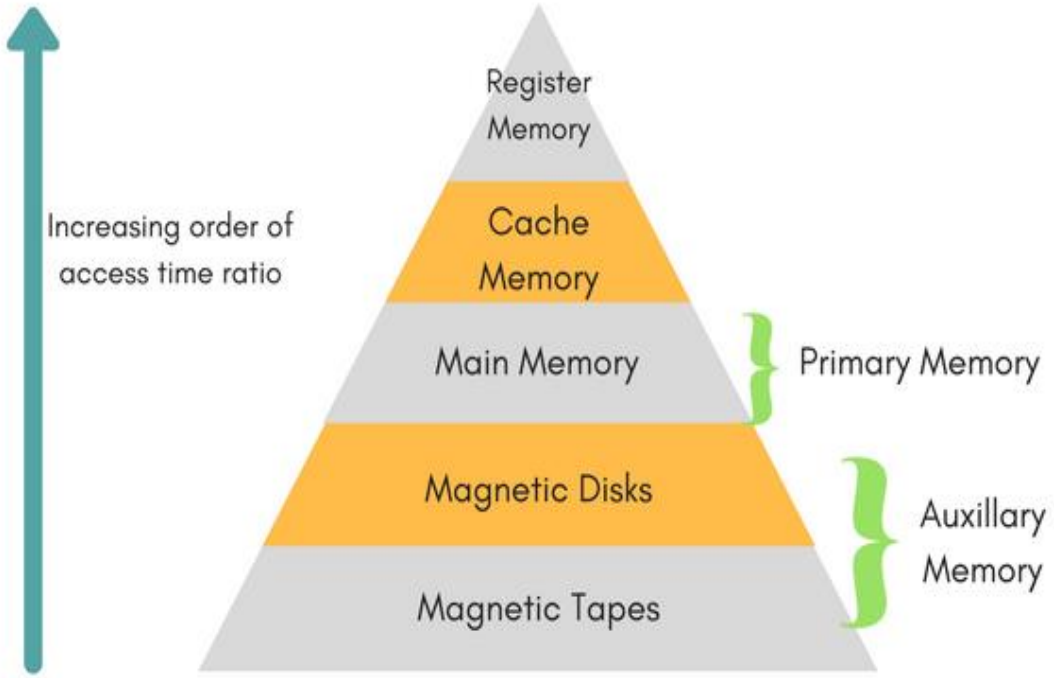
   Organization is how features are implemented
   - Control signals, interfaces, memory technology. e.g. Is there a hardware multiply unit or is it done by repeated addition?

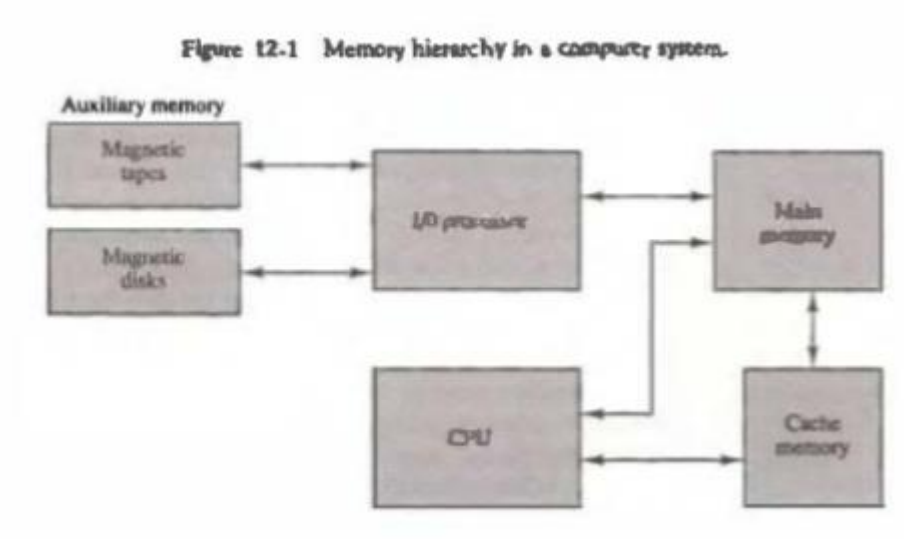
4) **Computer architecture** is concerned with the structure and behavior of computer system as seen by the user.
   Computer organization is concerned with the way the hardware components operate and the way they are connected together to form a computer system.

Memory Hierarchy

The memory unit Is an essential component in any digital computer since It Is needed for storing programs and data. A very small computer with an unlimited application may be able to fulfill its intended task without the use of additional storage capacity. Most general purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU.

The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information Is stored in auxiliary memory and transferred to main memory when needed.





The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. Above Figure illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

Cache Memory

A special very high speed memory called a Cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time Is dose to processor logic dock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a cache memory. It is placed between the CPU and main memory as illustrated in Fig. below. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

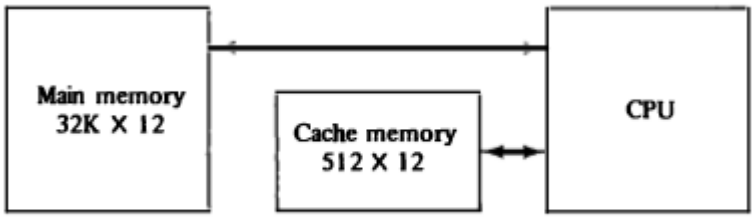


Figure 12-10 Example of cache memory.

The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

Organization of Hard Disk

A hard disk drive (HDD) is a non-volatile computer storage device containing magnetic disks or platters rotating at high speeds. It is a secondary storage device used to store data permanently. Non-volatile means data is retained when the computer is turned off. A hard disk drive is also known as a hard drive.

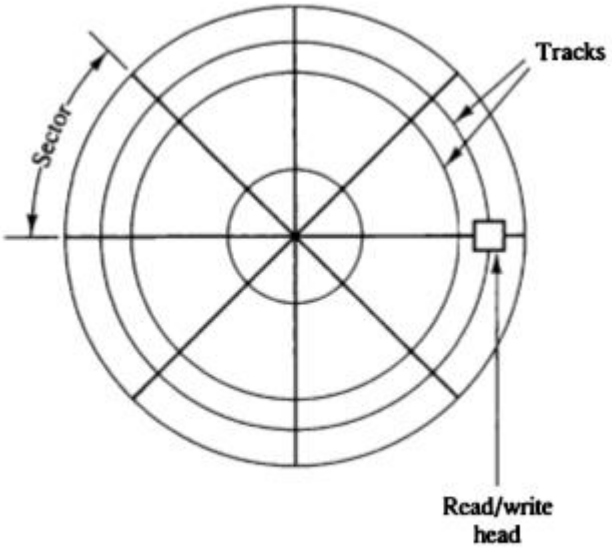


Figure Magnetic disk.

A hard drive consists of the following:

Magnetic platters - Platters are the round plates in the image above. Each platter holds a certain amount of information, so a drive with a lot of storage will have more platters than one with less storage. When information is stored and retrieved from the platters it is done so in concentric circles, called tracks, which are further broken down into segments called sectors.

Arm - The arm is the piece sticking out over the platters. The arms will contain read and write heads which are used to read and store the magnetic information onto the platters. Each platter will have its own arm which is used to read and write data off of it.

Motor - The motor is used to spin the platters from 4,500 to 15,000 rotations per minute (RPM). The faster the RPM of a drive, the better performance you will achieve from it.

When a computer wants to retrieve data off of the hard drive, the motor will spin up the platters and the arm will move itself to the appropriate position above the platter where the data is stored. The heads on the arm will detect the magnetic bits on the platters and convert them into the appropriate data that can be used by the computer. Conversely, when data is sent to the drive, the heads will this time, send magnetic pulses at the platters changing the magnetic properties of the platter, and thus storing your information.

Instruction Code

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.

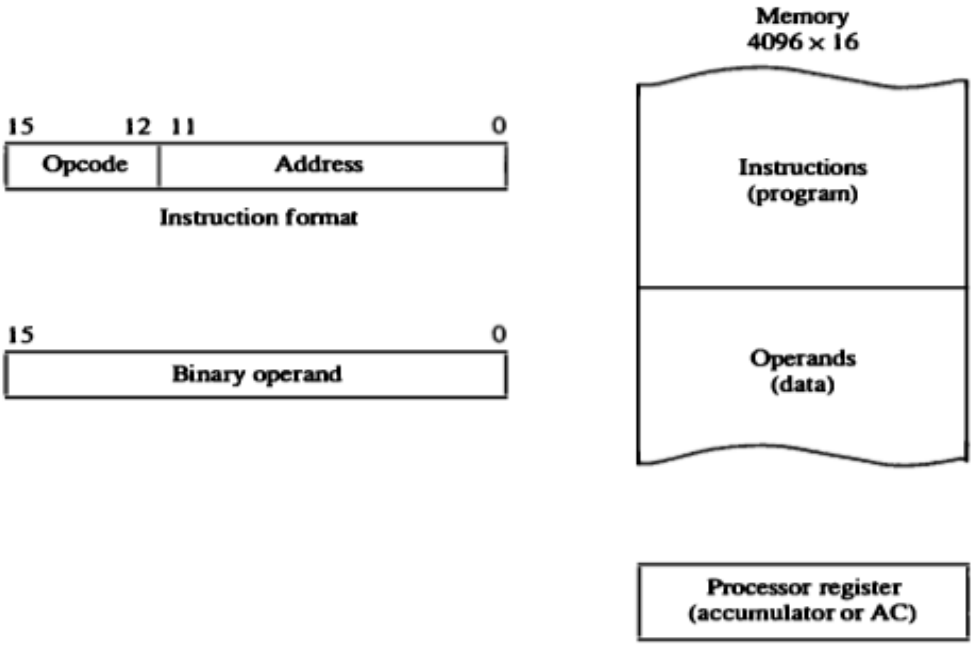
Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of micro operations. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.
The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored.

Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Below figure depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since 2^{12} = 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 5-1 Stored program organization.



Indirect Address

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.

In register indirect addressing mode, the data to be operated is available inside a memory location and that memory location is indirectly specified by a register pair.

Example:
MOV A, M (move the contents of the memory location pointed by the H-L pair to the accumulator)

Computer Registers

Registers are the fastest and smallest type of memory elements available to a processor. Registers are normally measured by the number of bits they can hold, for example, an "8-bit register", "32-bit register" or a "64-bit register" (or even with more bits). A processor often contains several kinds of registers, which can be classified according to their content or instructions that operate on them.

TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Common Bus System

The basic computer has eight registers, a memory unit, and a control unit. Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus.

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to O's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

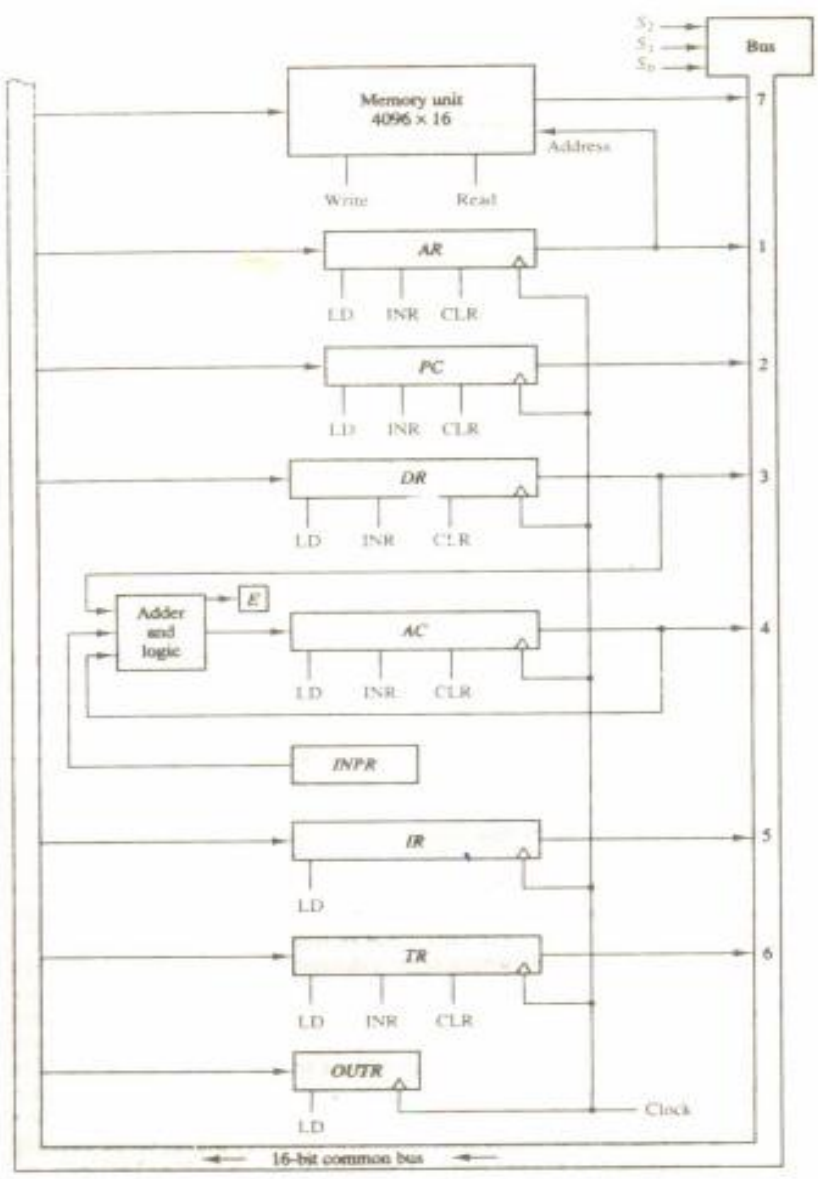


Figure 1.4 Basic registers which are connected to a common bus

The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers. The 16lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear).

The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be

specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC.

Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and micro operations for the accumulator.

Instruction Cycle

Time required to execute and fetch an entire instruction is called instruction cycle. A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

UNIT IV – Microprogrammed Control

Microprogrammed vs Hardwired Control

A hardwired control differs from microprogrammed control in the following ways:

HARDWIRED CONTROL UNIT	MICROPROGRAMMED CONTROL UNIT
The control unit whose control signals are generated by the hardware through a sequence of instructions is called a hardwired control unit.	The control unit whose control signals are generated by the data stored in control memory and constitute a program on the small scale is called a microprogrammed control unit
The control logic of a hardwired control is implemented with gates, flip flops, decoders etc.	The control logic of a micro-programmed control is the instructions that are stored in control memory to initiate the required sequence of microoperations.
Wiring changes are made in the hardwired control unit if there are any changes required in the design.	Changes in a microprogrammed control unit are done by updating the microprogram in control memory.
Hardwired control unit are faster and known to have complex structure.	Microprogrammed control unit is comparatively slow compared but are simple in structure.

Terminologies

Hardwired Control Unit:
When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Micro programmed control unit:
A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Control Memory:
Control Memory is the storage in the microprogrammed control unit to store the microprogram.

Control Word:
The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

Microoperations:
Micro-operations perform basic operations on data stored in one or more registers, including transferring data between registers or between registers and external buses of the central processing unit (CPU), and performing arithmetic or logical operations on registers.

Microcode:
A very low-level instruction set which is stored permanently in a computer or peripheral controller and controls the operation of the device.

Microinstruction
A single instruction in microcode. It is the most elementary instruction in the computer, such as moving the contents of a register to the arithmetic logic unit (ALU).

Microprogram
A set or sequence of microinstructions.

Design of Basic Computer

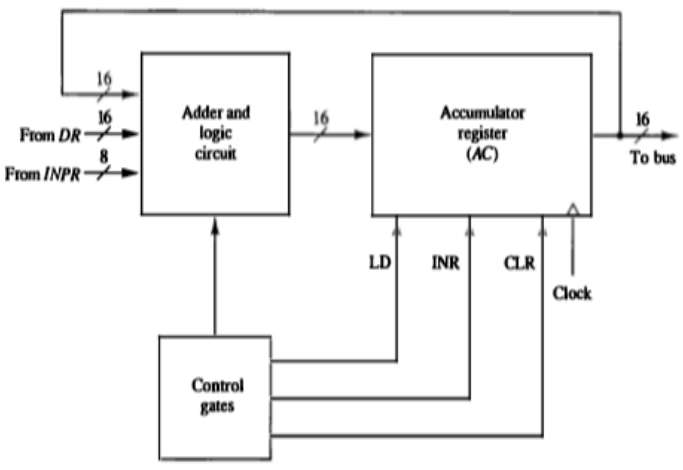
The basic computer consists of the following hardware components:

- A memory unit with 4096 words of 16 bits each.
- Nine registers: AR(Address Reg.), PC (Program Counter), DR(Data Reg.), AC (Accumulator), IR (Instruction Reg.), TR (Temp. Reg.), OUTR (Output Reg.), INPR (Input Reg.), and SC (Sequence Counter).
- Flip-flops: IEN (Interrupt Enable), FGI (Input Flag), and FGO (Output Flag).
- Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder
- A 16-bit common bus.
- Control logic gates.
- Adder and logic circuit connected to the input of AC.

Design of Accumulator Logic

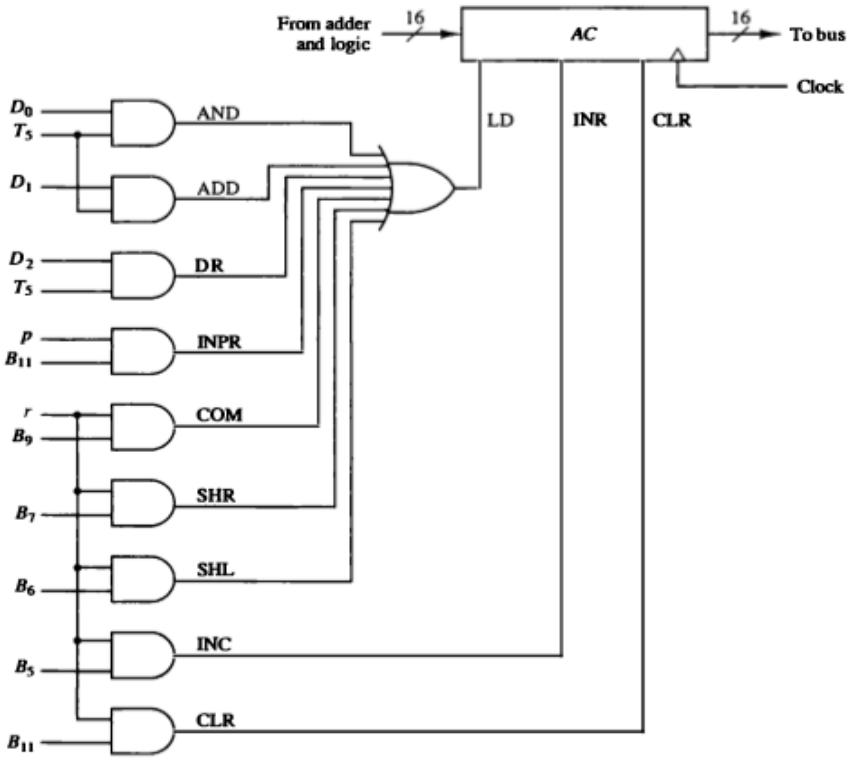
- The circuits associated with the AC register are shown in Fig. The adder and logic circuit has three sets of inputs.
- One set of 16 inputs comes from the outputs of AC.
- Another set of 16 inputs comes from the data register DR.
- A third set of eight inputs comes from the input register INPR.
- The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit.

Figure 5-19 Circuits associated with AC.



Control of AC Register – Gate Structure

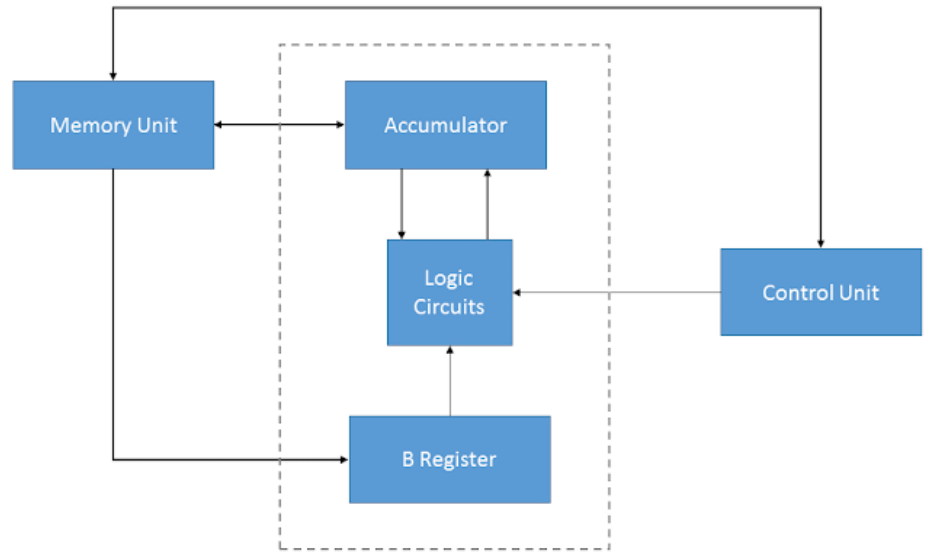
Figure 5-20 Gate structure for controlling the LD, INR, and CLR of AC.



- The gate structure that controls the LD, INR, and CLR inputs of AC is shown in Fig.
- The output of the AND gate that generates this control function is connected to the CLR input of the register.
- Similarly, the output of the gate that implements the increment micro operation is connected to the INR input of the register.
- The other seven micro operations are generated in the adder and logic circuit and are loaded into AC at the proper time.
- The outputs of the gates for each control function is marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

ALU Organization

- Various circuits are required to process data or perform arithmetical operations which are connected to microprocessor's ALU.
- Accumulator and Data Buffer stores data temporarily. These data are processed as per control instructions to solve problems. Such problems are addition, multiplication etc.



Organization of Arithmetic and Logic Unit

Functions of ALU:

Functions of ALU or Arithmetic & Logic Unit can be categorized into following 3 categories:

1. Arithmetic Operations:

Additions, multiplications etc. are example of arithmetic operations. Finding greater than or smaller than or equality between two numbers by using subtraction is also a form of arithmetic operations.

2. Logical Operations:

Operations like AND, OR, NOR, NOT etc. using logical circuitry are examples of logical operations.

3. Data Manipulations:

Operations such as flushing a register is an example of data manipulation. Shifting binary numbers are also example of data manipulation.

Control Memory

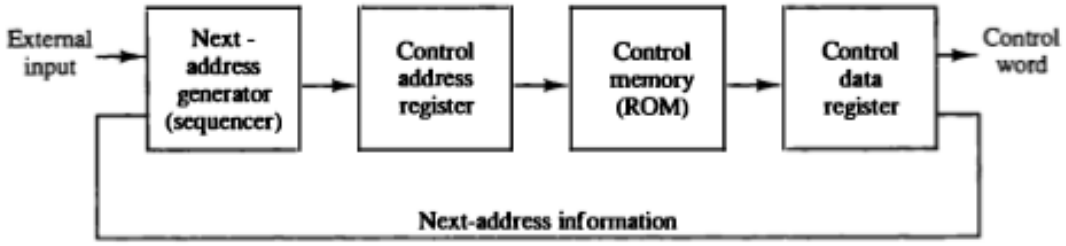
- A computer that employs a **microprogrammed control unit** will have two separate memories: a **main memory** and a **control memory**.
- The **main memory** is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data.
- In contrast, the **control memory** holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations.
- Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

- The control unit initiates a series of sequential steps of rnicrooperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a **control word**.
- As such, control words can be programmed to perform various operations on the components of the system.
- The **microinstruction** specifies one or more **microoperations** for the system. A sequence of microinstructions constitutes a **microprogram**.

Microprogrammed Control Organization

- The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig. The control memory is assumed to be a ROM, within which all control information is permanently stored.

Figure 7-1 Microprogrammed control organization.



- The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a **routine**.
- Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction.
- To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.
- An **initial address** is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. At the end of the fetch routine, the instruction is in the instruction register of the computer.
- The control memory next must go through the routine that determines the effective address of the operand. When the effective address computation routine is completed, the address of the operand is available in the memory address register.
- The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction.

- When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.
- In summary, the address sequencing capabilities required in a control memory are:
- Incrementing of the control address register.
- Unconditional branch or conditional branch, depending on status bit conditions.
- A mapping process from the bits of the instruction to an address for control memory.
- A facility for subroutine call and return.

Conditional Branching

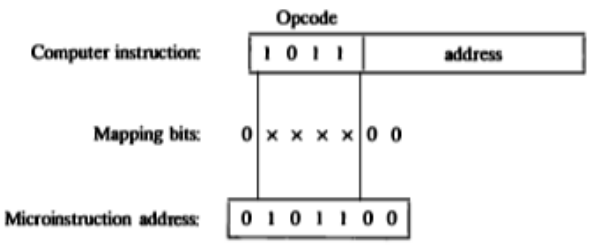
- The branch logic provides decision-making capabilities in the control unit.
- The **status conditions** are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.

- Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- The branch logic hardware may be implemented in a variety of ways. The simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented.

Mapping of Instruction

- Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a **mapping process**.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.
- For example, a computer with a simple instruction format as shown in Fig. 7-3 has an operation code of four bits. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction.

Figure 7-3 Mapping from instruction code to microinstruction address.



- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111.
- If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

Subroutines

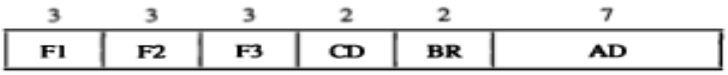
- **Subroutines** are programs that are used by other routines to accomplish a particular task.
- A subroutine can be called from any point within the main body of the microprogram.
- Frequently, many microprograms contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode.
- For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.
- Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented output from the control address register into a **subroutine register** and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine.

Microprogram

- **Microprogram** is a sequence of microinstructions that controls the operation of an arithmetic and logic unit so that machine code instructions are executed.
- It is a microinstruction program that controls the functions of a central processing unit or peripheral controller of a computer.

Microinstruction Format

The microinstruction format for the control memory is shown in Fig. The **20 bits of the microinstruction** are divided into **four functional parts**. The three fields **F1, F2, and F3** specify microoperations for the computer. The **CD** field selects status bit conditions. The **BR** field specifies the type of branch to be used. The **AD** field contains a branch address. The address field is seven bits wide, since the control memory has $128 = 2^7$ words.



F1, F2, F3: Microoperation fields

CD: Condition for branching

BR: Branch field

AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

Symbols & Binary Code for Microoperations

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow$ shl AC	SHL
100	$AC \leftarrow$ shr AC	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	$DR(15)$	I	Indirect address bit
10	$AC(15)$	S	Sign bit of AC
11	$AC = 0$	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{–}5) \leftarrow DR(11\text{–}14)$, $CAR(0,1,6) \leftarrow 0$

- Each microoperation in Table is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type microoperations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer AC <-DR (F1 = 100) has the symbol DRTAC, which stands for a transfer from DR to AC.
- The **CD (condition)** field consists of two bits which are encoded to specify four status bit conditions as listed in Table. The first condition is always a 1, so that a reference to CD = 00 (or the symbol U) will always find the condition to be true. We will use the symbols U, I, S, and Z for the four status bits when we write microprograms in symbolic form.

- The **BR (branch)** field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in Table, when BR = 00, the control performs a jump GMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from SBR to CAR. The mapping from the operation code bits of the instruction to an address for CAR is accomplished when the BR field is equal to 11.

Symbolic Vs Binary Microprogram

Symbolic Microprogram	**Binary Microprogram**
1. It is a set of microinstructions written in a symbolic form.	1. It is a set of microinstructions written in a binary form.
2. It is a convenient form for writing microprograms in a way that people can read and understand.	2. It is written in binary form so can be difficult for people to read and understand.
3. To be stored in memory symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough.	3. Translation is not required for storing in memory because it is already written in binary form.
4. The symbolic representation is useful for writing microprograms in an assembly language format.	4. The binary representation is the actual internal content that must be stored in control memory.

5. Example: Symbolic microprogram for fetch routine:
 	
`ORG 64`
`FETCH: PCTAR U JMP NEXT`
`READ, INCPC U JMP NEXT`
`DRTAR U MAP`
 	
Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. Here ORG 64 refers to the origin address or staring address.
	5. Example: Translation of symbolic microprogram to binary microprogram.
	
	Binary Address | F1 | F2 | F3 | CD | BR | AD
	1000000 | 110 | 000 | 000 | 00 | 00 | 1000001
	1000001 | 000 | 100 | 101 | 00 | 00 | 1000010
	1000010 | 101 | 000 | 000 | 00 | 11 | 0000000

Design of Control Unit/Structure of CU

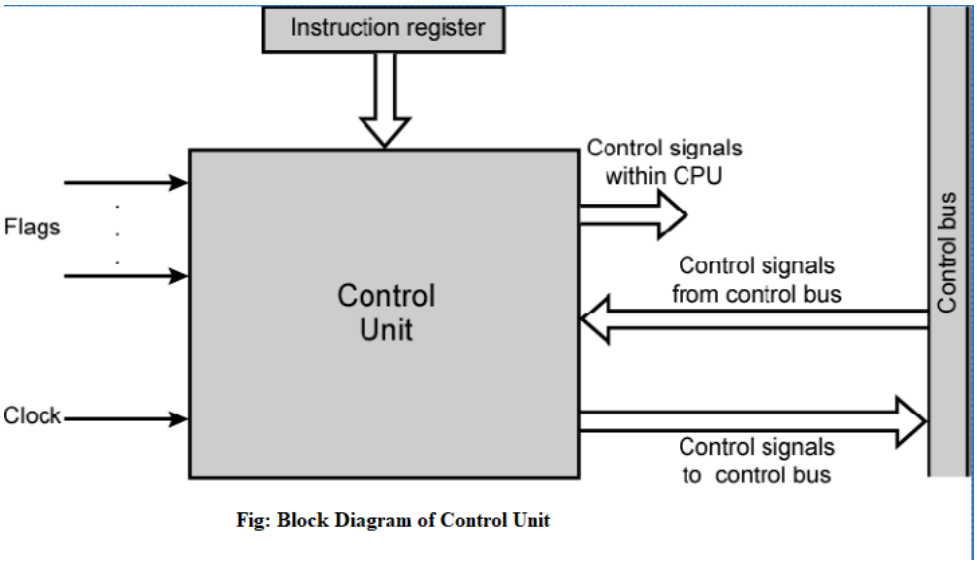


Fig: Block Diagram of Control Unit

- Control unit generates timing and control signals for the operations of the computer. The control unit communicates with ALU and main memory. It also controls the transmission between processor, memory and the various peripherals. It also instructs the ALU which operation has to be performed on data.
- Control unit can be designed by two methods:
  1. Hardwired control Unit
  2. Micro programmed Control Unit

Basic Requirement of Control Unit

The functional requirements of control unit are those functions that the control unit must perform and these are the basis for the design and implementation of the control unit. A three step process that lead to the characterization of the Control Unit:

1. Define the basis elements of the processor
2. Describe the micro-operations that the processor performs
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

1. Basic Elements of Processor

The following are the basic functional elements of a CPU:

- **ALU**: is the functional essence of the computer.
- **Registers**: are used to store data internal to the CPU.

2. Types of Micro-operation

These operations consist of a sequence of micro operations. All micro instructions fall into one of the following categories:

- Transfer data between registers
- Transfer data from register to external
- Transfer data from external to register
- Perform arithmetic or logical operations

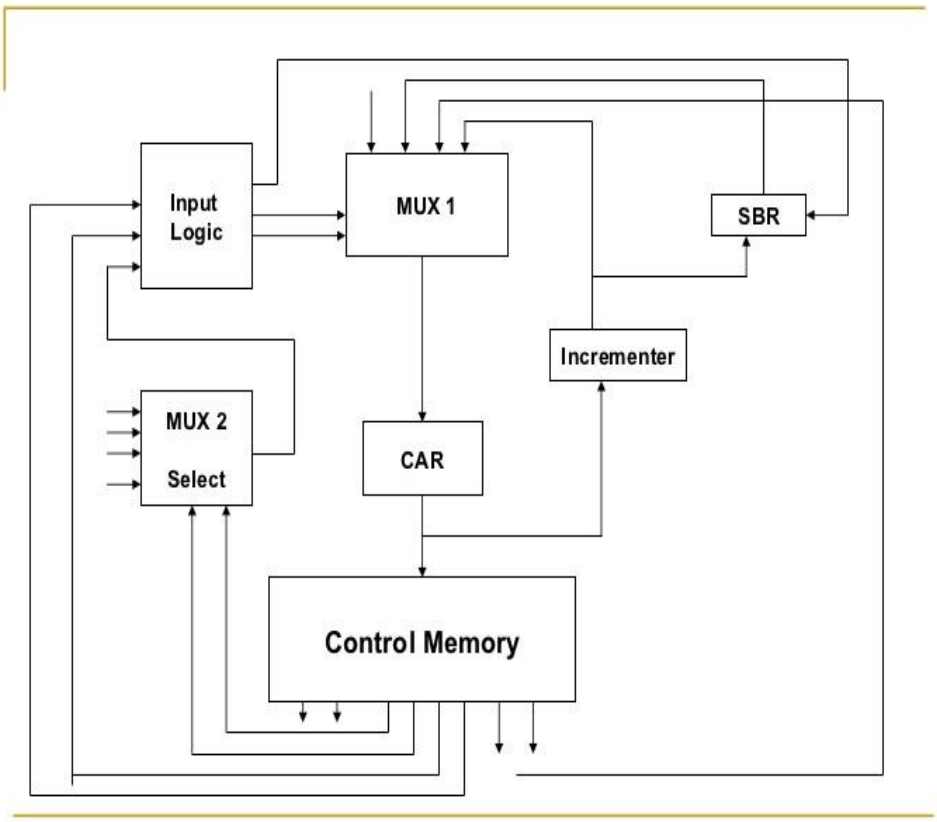
3. Functions of Control Unit

The control unit perform two tasks:

- **Sequencing**: The control unit causes the CPU to step through a series of micro-operations in proper sequence based on the program being executed.
- **Execution**: The control unit causes each micro-operation to be performed.

Microprogram Sequencer

- The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a **microprogram sequencer**.
- The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed.
- The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction.
- Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.



Microprogram sequencer for a control Mmemory

- The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it.
- There are two multiplexers in the circuit.
- The first multiplexer selects an address from one of the four sources and routes it into the **CAR(Control Address Register)**.
- The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit.
- The output from **CAR** provides the address for the control memory.
- The contents of **CAR** is incremented and applied to one of the multiplexer inputs and to the **SBR**.
- The other three input come from the address field of the present microinstruction, from the output of **SBR(Subroutine Register)** and from an external source that maps the instruction.

Unit V – Central Processing Unit

Introduction to CPU

The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
The CPU is made up of three major parts, as shown in Fig.

- The register set stores intermediate data used during the execution of the instructions.
- The arithmetic logic unit (ALU) performs the required micro operations for executing the instructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

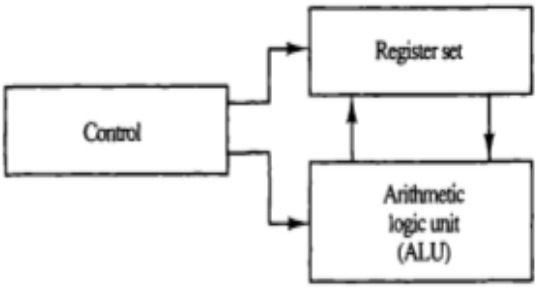


Figure 8-1 Major components of CPU.

General Register Organization

When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various micro operations.

Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift micro operations in the processor. Generally, CPU has seven general registers. Register organization show how registers are selected and how data flow between register and ALU.

- A decoder is used to select a particular register. The output of each register is connected to two multiplexers to form the two buses A and B. The selection lines in each multiplexer select the input data for the particular bus.
- The A and B buses form the two inputs of an ALU. The operation select lines decide the micro operation to be performed by ALU.
- The result of the micro operation is available at the output bus. The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.

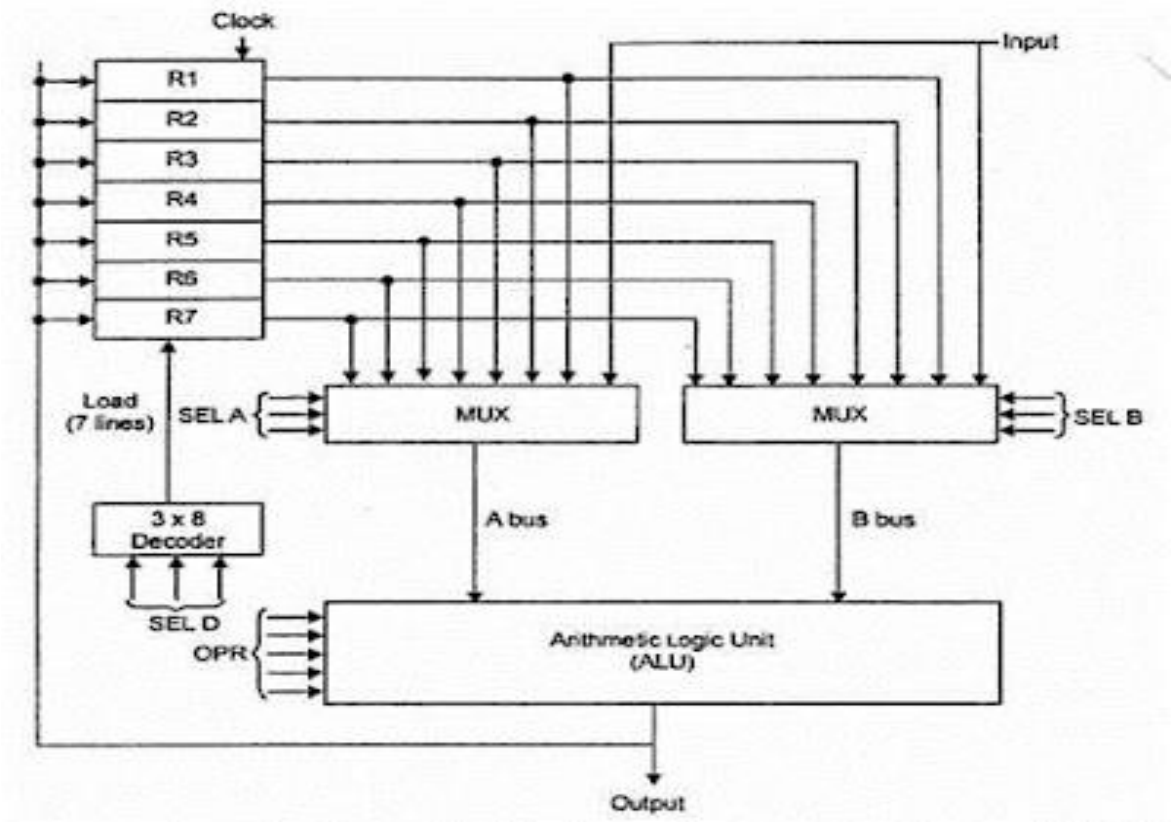


Fig: General Register Organization

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

R1<--R2 + R3

the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A+ B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R 1.

Control Word

There are 14 binary selection inputs in the unit, and their combined value specifies a control word.

SEL A	SELB	SELREG OR SELD	SELOPR

FORMATE OF CONTROL WORD

1. The three bit of SELA select a source registers of the **a** input of the ALU.
2. The three bits of SELB select a source registers of the **b** input of the ALU.
3. The three bits of SELED or SELREG select a destination register using the decoder.
4. The four bits of SELOPR select the operation to be performed by ALU.

TABLE 8-1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	*R1*	*R1*	*R1*
010	*R2*	*R2*	*R2*
011	*R3*	*R3*	*R3*
100	*R4*	*R4*	*R4*
101	*R5*	*R5*	*R5*
110	*R6*	*R6*	*R6*
111	*R7*	*R7*	*R7*

TABLE 8-2 Encoding of ALU Operations

OPR Select	Operation	Symbol
00000	Transfer *A*	TSFA
00001	Increment *A*	INCA
00010	Add *A* + *B*	ADD
00101	Subtract *A* − *B*	SUB
00110	Decrement *A*	DECA
01000	AND *A* and *B*	AND
01010	OR *A* and *B*	OR
01100	XOR *A* and *B*	XOR
01110	Complement *A*	COMA
10000	Shift right *A*	SHRA
11000	Shift left *A*	SHLA

A control word of 14 bits is needed to specify a micro operation in the CPU. The control word for a given micro operation can be derived from the selection variables.

For example, the subtract micro operation given by the statement
R1<-R2 - R3
specifies R2 for the A input of the ALU, R3 for the B input of the ALU, R1 for the destination register, and an ALU operation to subtract A - B. Thus the control word is specified by the four fields and the corresponding binary value for each field is obtained from the encoding.

Stack Organization

- A useful feature that is included in the CPU of most computers is a stack or last-in, first-out (LIFO) list.
- A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved.
- The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.
- The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in the stack.
- The two operations of a stack are the insertion and deletion of items. The operation of insertion is called push (or push-down) because it can be thought of as the result of pushing a new item on top. The operation of deletion is called pop (or pop-up) because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

Register Stack

A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers. Figure 8-3 shows the organization of a 64-word register stack.

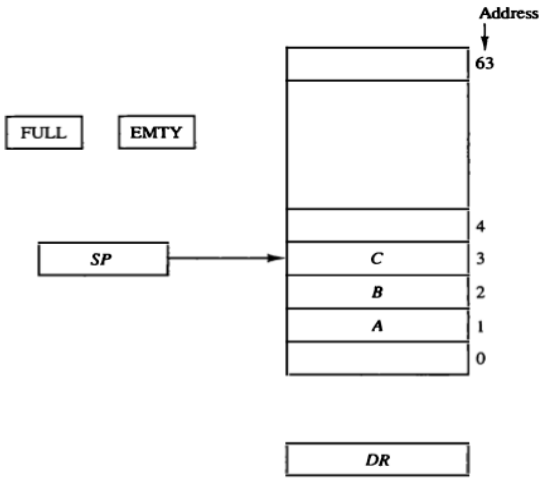


Figure 8-3 Block diagram of a 64-word stack.

The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3.

To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2.

To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full. If the stack is not full (if FULL = 0), a new item is inserted with a push operation. The push operation is implemented with the following sequence of micro operations;

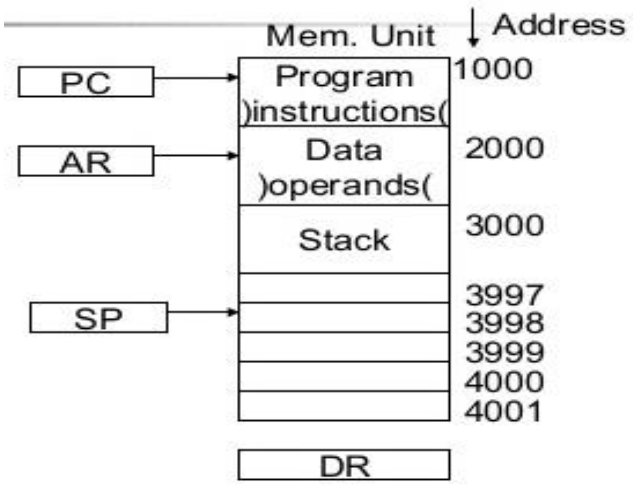
SP <- SP + 1	Increment stack pointer
M[SP]<-DR	Write item on top of the stack

A new item is deleted from the stack if the stack is not empty (if EMTY = 0). The pop operation consists of the following sequence of micro operations:

DR <--M[SP]	Read item from the top of stack
SP<--SP - 1	Decrement stack pointer

Memory Stack

A stack can be implemented in a random-access memory attached to a CPU.
The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.
The portion of computer memory is partitioned into three segments: program, data, and stack.
The program counter PC points at the address of the next instruction in the program. The address register AR points at an array of data. The stack pointer SP points at the top of the stack.
PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or &om the stack.



Computer memory with
program, data & stack segments

As shown in Fig., the initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000 the second item is stored at address 3999 and the last address that can be used for the stack Is 3000.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

SP<-SP - 1
M[SP]<-DR
The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from DR into the top of the stack.
A new item is deleted with a pop operation as follows:
DR <-M[SP]
SP<-SP + 1
The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

Program Interrupt

Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request.
Control returns to the original program after the service program is executed.
The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations:

(1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt);
(2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and

an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External Interrupts

External interrupts come from input/output (I/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.
Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure.
Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation.
Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal Interrupts

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called **traps**.
Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken.

Internal vs External Interrupts

The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event.

If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

Software Interrupts

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A software interrupt is initiated by executing an instruction.

Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.

The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt.

Instruction Format

Computer perform task on the basis of instruction provided. **An instruction format** defines layout of bits of an instruction.

An instruction in computer comprises of groups called fields. These field contains different information as for computers every thing is in 0 and 1 so each field has different significance on the basis of which a CPU decide what so perform.

The most common fields are:

- Operation field which specifies the operation to be performed like addition.
- Address field which contain the location of operand, i.e., register or memory location.
- Mode field which specifies how operand is to be founded.

An instruction is of various length depending upon the number of addresses it contain. Generally, CPU organization are of three types on the basis of number of address fields:

- Single Accumulator organization
- General register organization
- Stack organization

In **first** organization operation is done involving a special register called accumulator. In **second** on multiple registers are used for the computation purpose. In **third** organization the work on stack basis operation due to which it does not contain any address field.

1. Single Accumulator organization

All operations are performed with an accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as:

ADD X

where X is the address of the operand. The ADD instruction in this case results in the operation AC <--AC + M[X]. AC is the accumulator register and M[X] symbolizes the memory word located at address X.

2. General register organization

The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as:

ADD R1, R2, R3

to denote the operation R 1 <---R2 + R3.

The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

ADD R1, R2

would denote the operation R 1 <---R 1 + R2.

3. Stack organization

Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction

PUSH X

will push the word at address X to the top of the stack. The stack pointer is updated automatically.

Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction

ADD

in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack.

Instruction Format

To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

X = (A + B) * (C + D)

using zero, one, two, or three address instructions. We will use the symbols **ADD**, **SUB**, **MUL**, and **DIV** for the four arithmetic operations; **MOV** for the transfer-type operation; and **LOAD** and **STORE** for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

1. Three Address Instructions

Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand.

The program in assembly language that evaluates **X = (A + B) * (C + D)** is shown below, together with comments that explain the register transfer operation of each instruction.

```
ADD    R1, A, B      R1←M[A] + M[B]
ADD    R2, C, D      R2←M[C] + M[D]
MUL    X, R1, R2     M[X]←R1*R2
```

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

2. Two-Address Instructions

Two-address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate **X = (A + B) * (C + D)** is as follows:

```
MOV    R1, A      R1←M[A]
ADD    R1, B      R1←R1 + M[B]
MOV    R2, C      R2←M[C]
ADD    R2, D      R2←R2 + M[D]
MUL    R1,R2      R1←R1*R2
MOV    X, R1      M[X]←R1
```

The **MOV** instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

3. One-Address Instructions

One-address instructions use an accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second register and assume that the AC contains the result of all operations. The program to evaluate **X = (A + B) * (C + D)** is

```
LOAD     A     AC←M[A]
ADD      B     AC←AC + M[B]
STORE    T     M[T]←AC
LOAD     C     AC←M[C]
ADD      D     AC←AC + M[D]
MUL      T     AC←AC*M[T]
STORE    X     M[X]←AC
```

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

4. Zero-Address Instructions

A stack-organized computer does not use an address field for the instructions **ADD** and **MUL**. The **PUSH** and **POP** instructions, however, need an address field to specify the operand that communicates with the stack.

The following program shows how **X = (A + B) * (C + D)** will be written for a stack organized computer. (**TOS** stands for top of stack.)

```
PUSH    A    TOS←A
PUSH    B    TOS←B
ADD          TOS←(A + B)
PUSH    C    TOS←C
PUSH    D    TOS←D
ADD          TOS←(C + D)
MUL          TOS←(C + D)*(A + B)
POP     X    M[X]←TOS
```

5. RISC (Reduced Instruction Set Computer) Instructions

The instruction set of a typical RISC processor is restricted to the use of load and store instructions when communicating between memory and CPU. All other instructions are executed within the registers of the CPU without referring to memory.

The following is a program to evaluate **X = (A + B) * (C + D).**

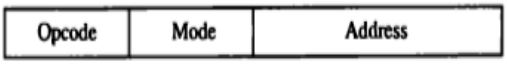
```
LOAD    R1, A          R1←M[A]
LOAD    R2, B          R2←M[B]
LOAD    R3, C          R3←M[C]
LOAD    R4, D          R4←M[D]
ADD     R1, R1, R2     R1←R1 + R2
ADD     R3, R3, R2     R3←R3 + R4
MUL     R1, R1, R3     R1←R1*R3
STORE   X, R1          M[X]←R1
```

The load instructions transfer the operands from memory to CPU registers. The add and multiply operations are executed with data in the registers without accessing memory. The result of the computations is then stored in memory with a store instruction.

Addressing Modes

An example of an instruction format with a distinct addressing mode field is shown in Fig. 8-6.

Figure 8-6 Instruction format with mode field.



The **operation code** specifies the operation to be performed. The **mode field** is used to locate the operands needed for the operation.

There may or may not be an **address field** in the instruction. If there is an **address field**, it may designate a memory address or a processor register. Moreover, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

1. Implied Mode

Implied Addressing Mode also known as "Implicit" or "Inherent" addressing mode is the addressing mode in which, no operand (register or memory location or data) is specified in the instruction. As in this mode the operand is specified implicit in the definition of instruction.

As an example: The instruction: “Complement Accumulator” is an Implied Mode instruction because the operand in the accumulator register is implied in the definition of instruction. In assembly language it is written as:

CMA: Take complement of content of AC

Similarly, the instruction,

RLC: Rotate the content of Accumulator is an implied mode instruction.

All register reference instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

2. Immediate Mode

In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field.

The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

The address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

3. Register Mode

In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction.

4. Register Indirect Mode

In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself.

The instruction specifies the name of the register in which the address of the data is available. Here the data will be in memory and the address will be in the register pair.

Example: **MOV A, M** - The memory data addressed by H L pair is moved to A register.

5. Auto increment or Auto Decrement Mode:

This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

After accessing the operand, the contents of this register are automatically incremented to the next value.

Before accessing the operand, the contents of this register are automatically decremented and then the value is accessed.

6. Direct Address Mode:

Here, the operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

7. Indirect Address Mode:

In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The **effective address** in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

8. Relative Address Mode:

In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

The address part of the instruction is usually a signed number (in 2' s complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction.

To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to 826. The effective address computation for the relative address mode is 826 + 24 = 850.

9. Indexed Addressing Mode:

In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory.

Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.

Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value.

10. Base Register Addressing Mode:

In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.

This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

The difference between the two modes is in the way they are used rather than in the way that they are computed.
An **index register** is assumed to hold an index number that is relative to the address part of the instruction. A **base register** is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address.

Data Transfer & Manipulation

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks.
The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields.
The actual operations available in the instruction set are not very different from one computer to another. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction.
Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instructions cause transfer of data from one location to another without changing the binary information content.
Data manipulation instructions are those that perform arithmetic, logic, and shift operations.
Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.
The instruction set of a particular computer determines the register transfer operations and control decisions that are available to the user.

Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content.
The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
Table 8-5 gives a list of eight data transfer instructions used in many computers.

TABLE 8-5 Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

The ***load*** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
The **store** instruction designates a transfer from a processor register into memory.
The ***move*** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words.
The ***exchange*** instruction swaps information between two registers or a register and a memory word.
The ***input*** and ***output*** instructions transfer data among processor registers and input or output terminals.
The ***push*** and ***pop*** instructions transfer data between processor registers and a memory stack.

Data Manipulation Instructions

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

1. Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions.

TABLE 8-7 Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

The **increment** instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1' s when incremented produces a result of all 0' s.
The **decrement** instruction subtracts 1 from a value stored in a register or memory word. A number with all 0's, when decremented, produces a number with all 1's.

2. Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.

TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

The **clear** instruction causes the specified operand to be replaced by 0's.

The **complement** instruction produces the 1's complement by inverting all the bits of the operand.

The **AND**, **OR**, and **XOR** instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations.

A few other bit manipulation instructions are included in Table 8-8. Individual bits such as a carry can be **cleared**, **set**, or **complemented** with appropriate instructions. Another example is a flip-flop that controls the interrupt facility and is either **enabled** or **disabled** by means of bit manipulation instructions.

3. Shift Instructions

Shifts are operations in which the bits of a word are moved to the left or right.

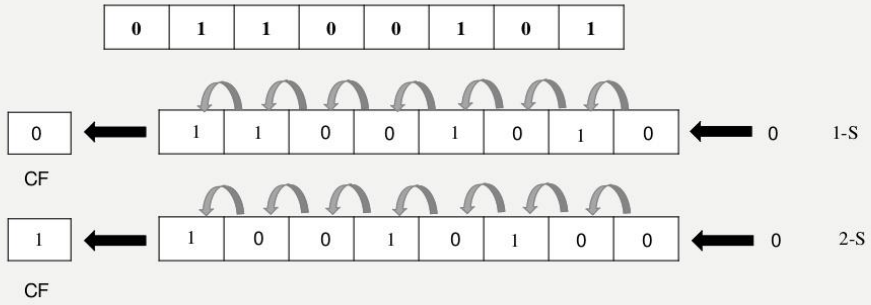
The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either **logical shifts, arithmetic shifts, or rotate-type** operations. In either case the shift may be to the right or to the left.

TABLE 8-9 Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

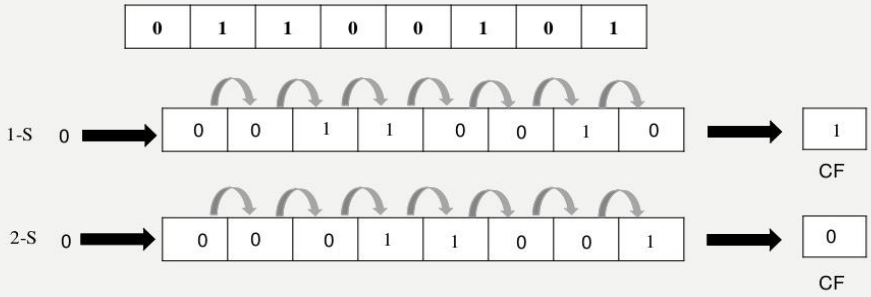
SHIFT INSTRUCTION: SHL AND SAL

- The SHL (shift left) instruction shifts the bits in the destination to the left.
- A 0 is shifted into the right most bit position and the msb is shifted into CF.



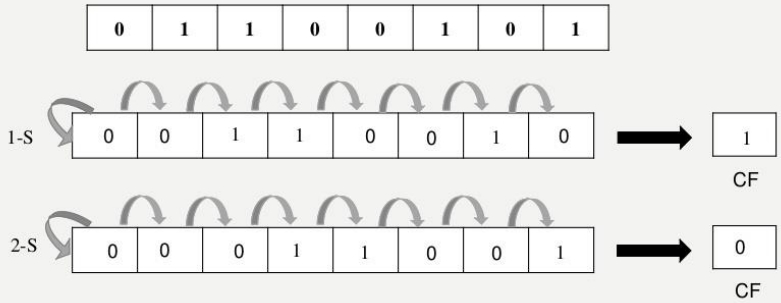
SHIFT INSTRUCTION: SHR

- The instruction SHR(shift right) performs right shift on the destination operand.
- A 0 is shifted into the msb position ,and the rightmost bit is shifted into CF.



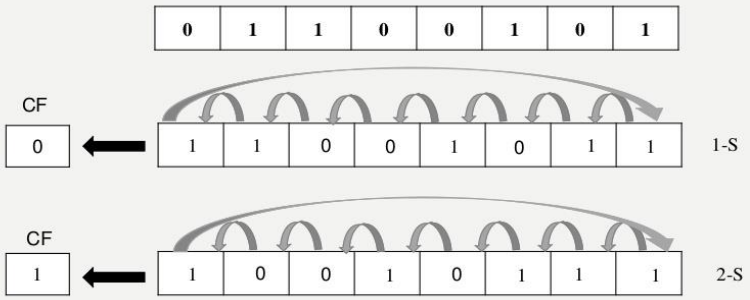
SHIFT INSTRUCTION: SAR

- The SAR instruction (shift arithmetic right) operates like SHR, with one difference: the msb retains its original value.



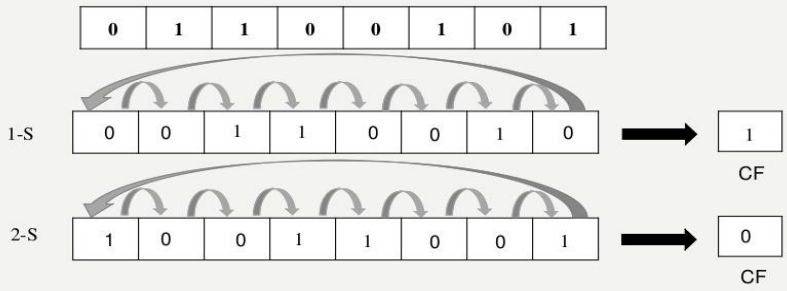
ROTATE INSTRUCTION: ROL

- The instruction ROL (rotate left)shifts bits to the left .The msb is shifted into the rightmost bit . The CF also gets the bit shifted out of the msb .



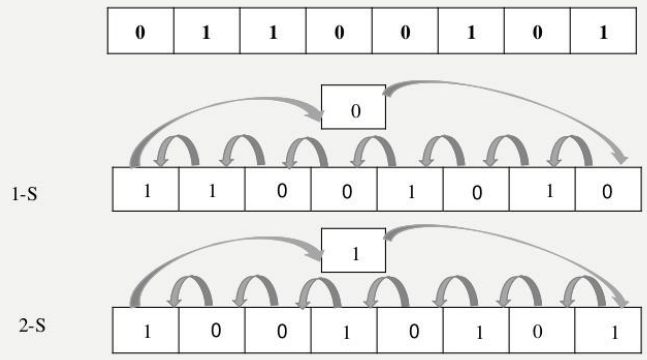
ROTATE INSTRUCTION: ROR

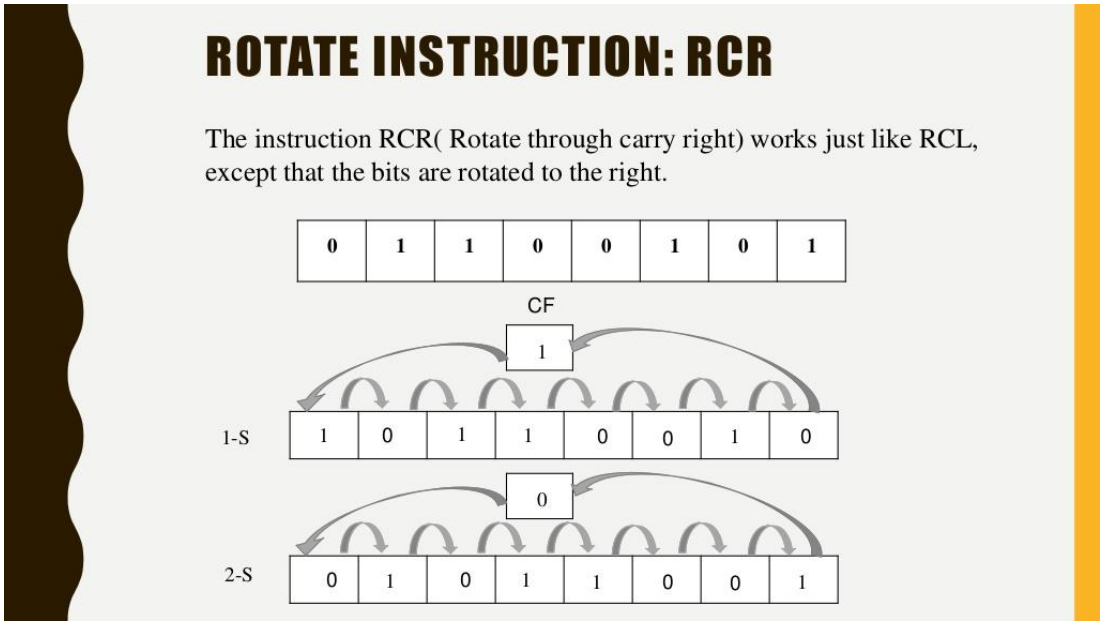
- The instruction ROR (rotate right) works just like ROL , except that the bits are rotate to the right .The rightmost bit is shifted into the msb , and also into The CF.



ROTATE INSTRUCTION: RCL

- The instruction **RCL** (Rotate through carry left) shifts the bits of the destination to the left. The msb is shifted into the CF, and the previous value of CF is shifted into the rightmost bit.





Program Control Instructions

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed.
Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.
After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

Program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
This is an important feature in **digital computers**, as it provides control over the flow of program execution and a capability for branching to different program segments.

TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

The **branch** and **jump** instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes.
The **branch** is usually a one-address instruction. It is written in assembly language as **BR ADR**, where **ADR** is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of **ADR** into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location **ADR.**

Branch and jump instructions may be **conditional** or **unconditional**. An **unconditional branch** instruction causes a branch to the specified address without any conditions.
The **conditional branch** instruction specifies a condition such as branch if positive or branch if zero.
If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.
The **skip** instruction does not need an address field and is therefore a zero-address instruction. A **conditional skip** instruction will skip the next instruction if the condition is met. The **call and return instructions** are used in conjunction with subroutines.

Status Bit Conditions

Status bits are also called **condition-code bits** or **flag bits**. Figure below shows the block diagram of an 8-bit ALU with a 4-bit status register. The four status bits are symbolized by C, S, Z, and V.

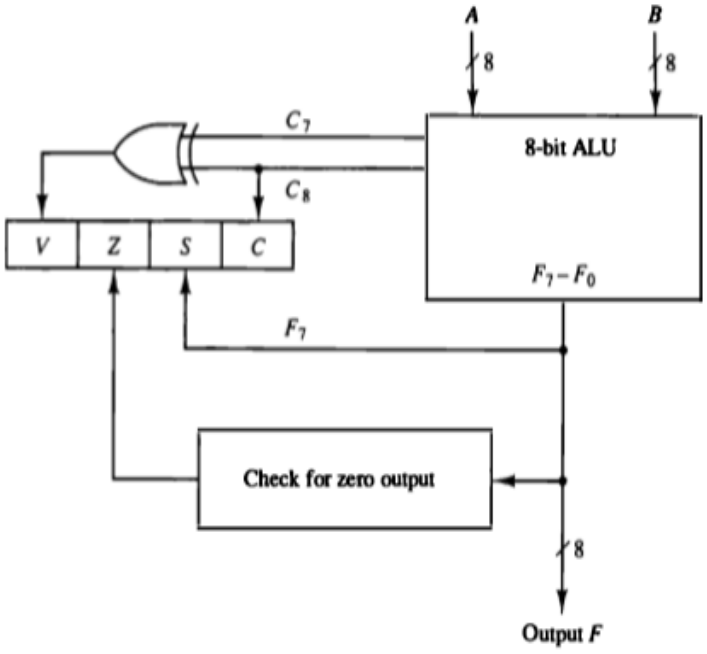


Figure 8-8 Status register bits.

The bits are set or cleared as a result of an operation performed in the ALU.

1. **Bit C (carry)** is set to 1 if the end carry C8 is 1. It is cleared to 0 if the carry is 0.
2. **Bit S (sign)** is set to 1 if the highest-order bit F, is 1. It is set to 0 if the bit is 0.
3. **Bit Z (zero)** is set to 1 if the output of the ALU contains all O's. !tis cleared to 0 otherwise. In other words, Z = 1 if the output is zero and Z = 0 if the output is not zero.
4. **Bit V (overflow)** is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise.

Conditional Branch Instructions

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
	Unsigned compare conditions ($A - B$)	
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
	Signed compare conditions ($A - B$)	
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state.

Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the jump, skip, call, or return type of program control instructions.

Subroutine Call and Return

A **subroutine** is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.

Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are ***call subroutine, jump to subroutine, branch to subroutine, or branch and save address***.

A **call subroutine** instruction consists of an operation code together with an address that specifies the beginning of the subroutine. The instruction is executed by performing two operations:

(1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and
(2) control is transferred to the beginning of the subroutine.

The last instruction of every subroutine, commonly called **return** from subroutine, transfers the **return address** from the temporary location into the program counter.

This results in a transfer of program control to the instruction whose address was originally stored in the temporary location.

RISC & CISC

A computer with a large number of instructions is classified as a **complex instruction set computer**, abbreviated **CISC**.

In the early 1980s, a number of computer designers recommended that computers use fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often. This type of computer is classified as a **reduced instruction set computer or RISC**.

CISC Characteristics

1. A large number of instructions-typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes-typically from 5 to 20 different modes.
4. Variable-length instruction formats.
5. A relatively large number of registers in the processor unit.

RISC Characteristics

The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

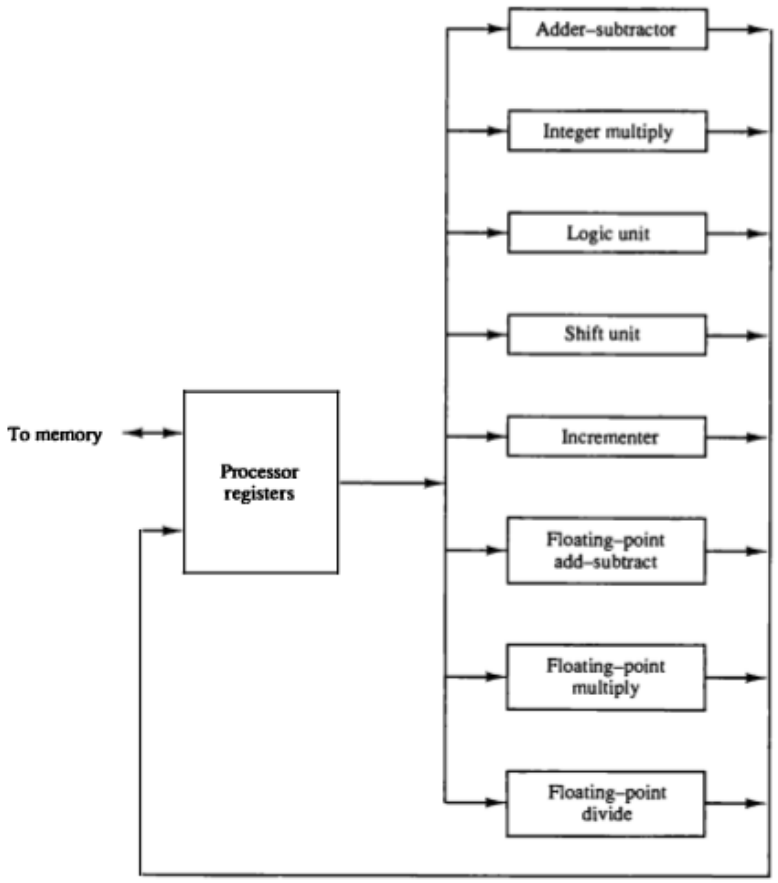
1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operations done within the registers of the CPU
5. Fixed-length, easily decoded instruction format

UNIT VI – Pipeline, Vector Processing & Microprocessors

Parallel Processing

- **Parallel processing** is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system.
- Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.
- For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time.
- Furthermore, the system may have two or more processors operating concurrently. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.
- The amount of hardware increases with parallel processing. and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Figure 9-1 Processor with multiple functional units.



- Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.
- The adder and integer multiplier perform the arithmetic operations with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data.
- All units are independent of each other, so one number can be shifted while another number is being incremented.
- A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

Flynn's Classification of Parallel Processing

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system.

One classification introduced by **M. J. Flynn** considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.

The normal operation of a computer is to fetch instructions from memory and execute them in the processor. The sequence of instructions read from memory constitutes an **instruction stream**. The operations performed on the data in the processor constitutes a **data stream**. Parallel processing may occur in the **instruction stream, in the data stream, or in both**.

Flynn's classification divides computers into **four major groups** as follows:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)

- **SISD** represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.
- **SIMD** represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.
- **MISD** structure is only of theoretical interest since no practical system has been constructed using this organization.

- **MIMD** organization refers to a computer system capable of processing several programs at the same time. Most multiprocessor and multicomputer systems can be classified in this category.

Pipelining

- **Pipelining** is a technique of decomposing a sequential process into sub-operations, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments.
- A pipeline can be visualized as a collection of processing segments through which binary information flows.
- Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
- It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.
- The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Pipelining Example

The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each sub-operation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2.

R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The sub-operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

Figure 9-2 Example of pipeline processing.

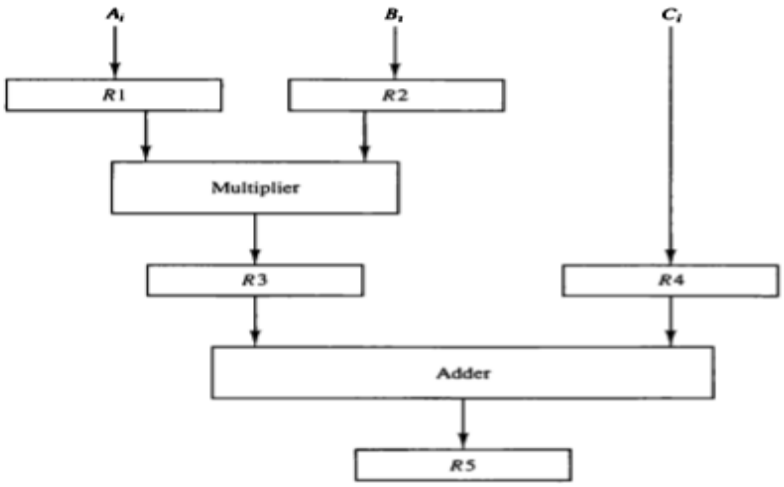


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. **An instruction pipeline** reads consecutive instructions from memory while previous instructions are being executed in other segments.

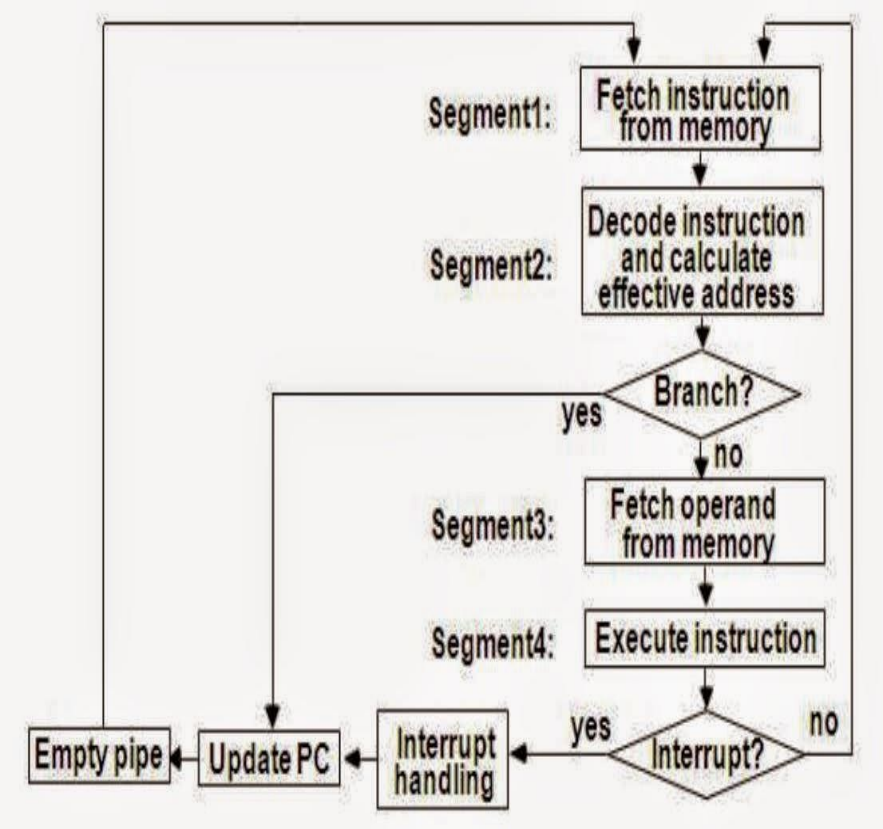
This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way itis executed.

Example: Four-Segment Instruction Pipeline



The above figure shows operation of 4-segment instruction pipeline. The four segments are represented as:

1. **FI**: segment 1 that fetches the instruction.
2. **DA**: segment 2 that decodes the instruction and calculates the effective address.
3. **FO**: segment 3 that fetches the operands.
4. **EX**: segment 4 that executes the instruction.

The space time diagram for the 4-segment instruction pipeline is given below:

Step	1	2	3	4	5	6	7	8	9
1	FI	DA	FO	EX					
2		FI	DA	FO	EX				
3			FI	DA	FO	EX			
4				FI	DA	FO	EX		
5					FI	DA	FO	EX	
6						FI	DA	FO	EX

Fig: timing diagram for 4-segment instruction pipeline

Pipeline Conflicts (Hazards)

A pipeline hazard occurs when the instruction pipeline deviates at some phases, some operational conditions that do not permit the continued execution. In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. **Resource conflicts** caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. **Data dependency** conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. **Branch difficulties** arise from branch and other instructions that change the value of PC.

Data Dependency

It arises when instructions depend on the result of previous instruction but the previous instruction is not available yet.

For example, an instruction in segment may need to fetch an operand that is being generated at same time by the previous instruction in the segment.

The most common techniques used to resolve data hazard are:

(a) **Hardware interlock** - a hardware interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. It then inserts enough number of clock cycles to delays the execution of such instructions.

(b) **Operand forwarding -** This method uses a special hardware to detect conflicts in instruction execution and then avoid it by routing the data through special path between pipeline segments. For example, instead of transferring an ALU result into a destination result, the hardware checks the destination operand, and if it is needed in next instruction, it passes the result directly into ALU input, bypassing the register.

(c) **Delayed load -** It is software solutions where the compiler is designed in such a way that it can detect the conflicts; re-order the instructions to delay the loading of conflicting data by inserting no operation instruction.

Handling of Branch Instructions

Branch hazard arises from branch and other instruction that change the value of program counter (PC). The conditional branch provides plenty of instruction branch line and it is difficult to determine which branches will be taken or not taken. A variety of approaches have been used to deal with branch hazard and they are described below.

(a) **Multiple streaming -** It is a brute-force approach which replicates the initial portions of the pipeline and allows the pipeline to fetch both instructions, making use of two streams (branches).

(b) **Prefetch branch target -** When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

(c) **Branch prediction -** uses additional logic to prediction the outcomes of a (conditional) branch before it is executed. The popular approaches are - predict never taken, predict always taken, predict by opcode, taken/not taken switch and using branch history table.

(d) **Loop buffer -** A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.

(e) **Delayed branch -** This technique is employed in most RISC processors. In this technique, compiler detects the branch instructions and re-arranges the instructions by inserting useful instructions to avoid pipeline hazards.

Vector Processing

- **Vector processing** is a procedure for speeding the processing of information by a computer, in which pipelined units perform arithmetic operations on uniform, linear arrays of data values, and a single instruction involves the execution of the same operation on every element of the array.
- There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems are characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete.
- In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.
- To achieve the required level of high performance it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

Application Areas of Vector Processing

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

- Long-range weather forecasting
- Petroleum explorations
- Seismic data analysis
- Medical diagnosis
- Aerodynamics and space flight simulations
- Artificial intelligence and expert systems
- Mapping the human genome
- Image processing

Vector Operations

- Many scientific problems require arithmetic operations on large arrays of numbers. These numbers are usually formulated as vectors and matrices of floating-point numbers.
- A **vector** is an ordered set of a one-dimensional array of data items. A vector V of length n is represented as a row vector by V = [V_1 , V_2 , V_3, · · · V_n].
- A conventional sequential computer is capable of processing operands one at a time. Consequently, operations on vectors must be broken down into single computations with subscripted variables. The element V_i of vector V is written as V(I) and the index I refers to a memory address or register where the number is stored.
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

```
      DO 20 I = 1, 100
20    C(I) = B(I) + A(I)
```

- This is a program for adding two vectors A and B of length 100 to produce a vector C.
- A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop. It allows operations to be specified with a single vector instruction of the form

  C(1 : 100) = A(1 : 100) + B(1: 100)
- The vector instruction includes the initial address of the operands, the length of the vectors, and the operation to be performed, all in one composite instruction.

Conventional computer

```
      Initialize I = 0
20    Read A(I)
      Read B(I)
      Store C(I) = A(I) + B(I)
      Increment I = I + 1
      If I ≤ 100 goto 20
```

Matrix Multiplication

Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors. An n x m matrix of numbers has n rows and m columns and may be considered as constituting a set of n row vectors or a set of m column vectors. Consider, for example, the multiplication of two 3 x 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

For example, the number in the first row and first column of matrix C is calculated by letting i = 1, j = 1, to obtain

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

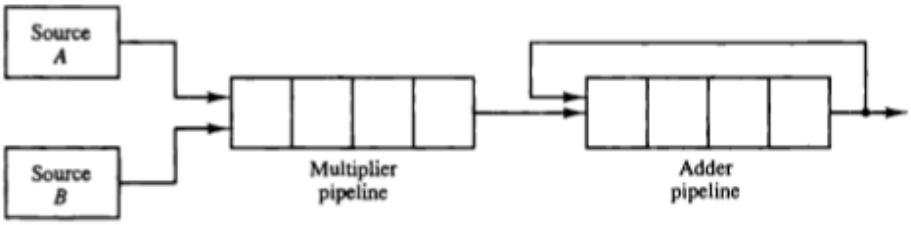
Inner Product

In general, the inner product consists of the sum of k product terms of the form

$$C = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4 + \cdots + A_k B_k$$

In a typical application k may be equal to 100 or even 1000. The inner product calculation on a pipeline vector processor is shown below:

$$\begin{aligned} C = {} & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \cdots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \cdots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \cdots \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \cdots \end{aligned}$$



Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.

Let's take an example of a pipeline unit for floating-point addition and subtraction. The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

$$X = A \times 2^a \quad \text{(Exponent)}$$
$$Y = B \times 2^b$$

Mantissa (fraction)

4-Segment Pipeline :

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

Floating-point Add/Subtraction Pipeline Example :

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain $3 - 2 = 1$. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$Z = 0.10324 \times 10^4$$

Multiprocessor System

- A **multiprocessor** is a computer system with two or more central processing units (CPUs), with each one sharing the common main memory as well as the peripherals. This helps in simultaneous processing of programs.
- The key objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching.
- A multiprocessor is regarded as a means to improve computing speeds, performance and cost-effectiveness, as well as to provide enhanced availability and reliability.

Characteristics:

- ❖ Consists of more than one CPU.
- ❖ Fast processing.
- ❖ Reliability
- ❖ Cost – Effective
- ❖ Simultaneous processing of programs.

Interconnection Structures for Multiprocessor System

The components that form a multiprocessor system are CPUs, IOPs(Input Output Processors) connected to input output devices, and a memory unit.

There are several physical forms available for establishing an interconnection network.

- Time-shared common bus
- Multiport memory
- Crossbar switch
- Multistage switching network

1. Time Shared Common Bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.

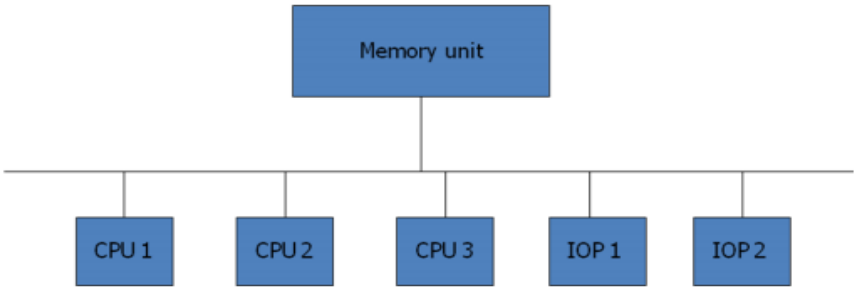


Fig: Time shared common bus organization

2. Multiport Memory

A multiport memory system employs separate buses between each memory module and each CPU.

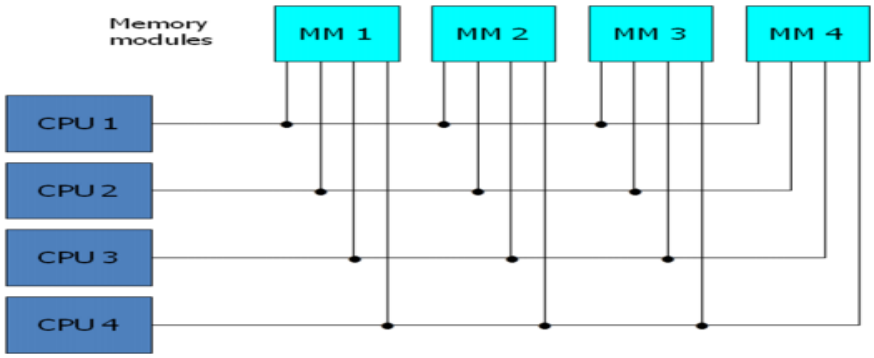


Fig: Multiport memory organization

3. Crossbar Switch

Consists of a number of cross points that are placed at intersections between processor buses and memory module paths.

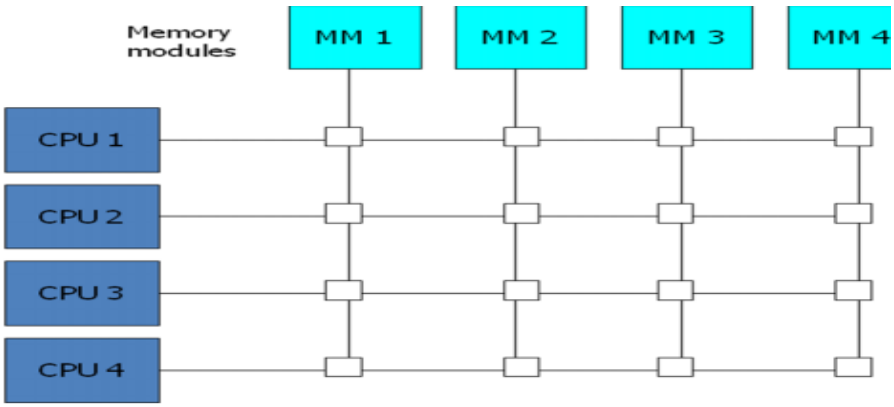


Fig: Crossbar switch