

# Computer System Architecture – CE-310

- ❖ **Logic Gates and Boolean Theorems**
- ❖ **Combinational circuits**
  - 1. Adders
  - 2. Multiplexers
  - 3. Decoders
- ❖ **Sequential Circuits**
  - 1. **Flip Flop**
    - 1. S-R flip Flop
    - 2. D flip Flop
    - 3. J-K flip Flop
  - 2. **Registers** (Specific and General purpose registers)
  - 3. **Counters**
    - 4. Asynchronous Counters (Ripple counter)
    - 5. Synchronous Counter
  - 4. **Memory**
- ❖ **System Buses**
  - 1. Data Bus
  - 2. Address Bus
  - 3. Control Bus
- ❖ **Data Representation and Computer Arithmetic**
  - Number System
  - Binary arithmetic
  - Fixed and Floating point representation
  - Character representation
- ❖ **Instruction Set.**
- ❖ **Instruction Set Design Considerations.**
- ❖ **General Register Organization.**
- ❖ **Micro Operations.**
- ❖ **Stack Organization.**
- ❖ **Instruction Cycle.**
- ❖ **Instruction Format.**
- ❖ **Addressing Modes.**
- ❖ **Instruction Code.**
- ❖ **What is pipeline?**
- ❖ **Pipeline Hazards.**
- ❖ **Cache Coherence.**
- ❖ **Memory Management System.**
- ❖ **Memory Hierarchy.**
- ❖ **Page Replacement Algorithms.**
- ❖ **Cache Memory.**

Equipped by  
Dr. Sanjay Soni  
IAR-Gandhinagar

# Logic Gates and Boolean Theorems:

## AND Gate

A circuit which performs an AND operation is shown in figure. It has  $n$  input ( $n \geq 2$ ) and one output.

$$\begin{aligned} Y &= A \text{ AND } B \text{ AND } C \dots\dots N \\ Y &= A.B.C \dots\dots N \\ Y &= ABC \dots\dots N \end{aligned}$$

### Logic diagram



### Truth Table

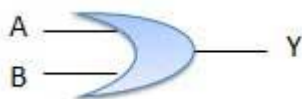
Inputs		Output
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

## OR Gate

A circuit which performs an OR operation is shown in figure. It has  $n$  input ( $n \geq 2$ ) and one output.

$$\begin{aligned} Y &= A \text{ OR } B \text{ OR } C \dots\dots N \\ Y &= A + B + C \dots\dots N \end{aligned}$$

### Logic diagram



### Truth Table

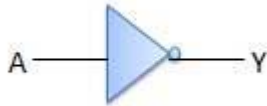
Inputs		Output
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

## NOT Gate

NOT gate is also known as **Inverter**. It has one input A and one output Y.

$$\begin{array}{l} Y \\ Y \end{array} = \begin{array}{l} \text{NOT } A \\ \overline{A} \end{array}$$

### Logic diagram



### Truth Table

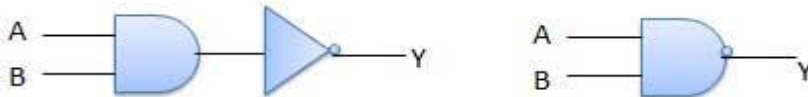
Inputs		Output
A	B	
0	1	
1	0	

## NAND Gate

A NOT-AND operation is known as NAND operation. It has n input ( $n \geq 2$ ) and one output.

$$\begin{array}{l} Y \\ Y \end{array} = \begin{array}{l} A \text{ NOT AND } B \text{ NOT AND } C \dots\dots N \\ A \text{ NAND } B \text{ NAND } C \dots\dots N \end{array}$$

### Logic diagram



### Truth Table

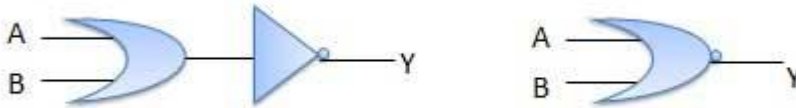
Inputs		Output
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

## NOR Gate

A NOT-OR operation is known as NOR operation. It has n input ( $n \geq 2$ ) and one output.

$$\begin{array}{l} Y \\ Y \end{array} = \begin{array}{l} A \text{ NOT OR } B \text{ NOT OR } C \dots\dots N \\ A \text{ NOR } B \text{ NOR } C \dots\dots N \end{array}$$

### Logic diagram



### Truth Table

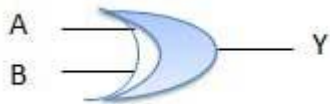
Inputs		Output
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

## XOR Gate

XOR or Ex-OR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-OR gate is abbreviated as EX-OR gate or sometime as X-OR gate. It has n input ( $n \geq 2$ ) and one output.

$$\begin{aligned} Y &= A \text{ XOR } B \text{ XOR } C \dots\dots N \\ Y &= A \oplus B \oplus C \dots\dots N \\ Y &= \overline{AB} + \overline{AB} \end{aligned}$$

### Logic diagram



### Truth Table

Inputs		Output
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

## XNOR Gate

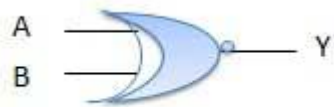
XNOR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-NOR gate is abbreviated as EX-NOR gate or sometime as X-NOR gate. It has n input ( $n \geq 2$ ) and one output.

$$Y = A \text{ XOR } B \text{ XOR } C \dots\dots N$$

$$Y = A \oplus B \oplus C \dots\dots N$$

$$Y = \overline{AB} + AB$$

## Logic diagram



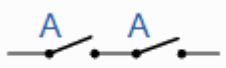
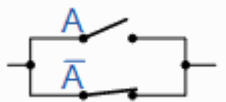
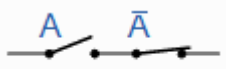
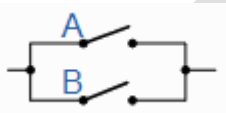
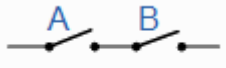
## Truth Table

Inputs		Output
A	B	$A \oplus B$
0	0	1
0	1	0
1	0	0
1	1	1

# Boolean Theorems/Laws:

## 1. Boolean algebra Rules

Boolean Expression	Description	Equivalent Switching Circuit	Boolean Algebra Law or Rule
$A + 1 = 1$	A in parallel with closed = "CLOSED"		Annulment
$A + 0 = A$	A in parallel with open = "A"		Identity
$A \cdot 1 = A$	A in series with closed = "A"		Identity
$A \cdot 0 = 0$	A in series with open = "OPEN"		Annulment
$A + A = A$	A in parallel with A = "A"		Idempotent

$A \cdot A = A$	A in series with A = "A"		Idempotent
$\text{NOT } A = A$	NOT NOT A (double negative) = "A"		Double Negation
$A + A = 1$	A in parallel with NOT A = "CLOSED"		Complement
$A \cdot A = 0$	A in series with NOT A = "OPEN"		Complement
$A+B = B+A$	A in parallel with B = B in parallel with A		Commutative
$A \cdot B = B \cdot A$	A in series with B = B in series with A		Commutative
$\text{Not}(A+B) = \text{not}(A) \cdot \text{Not}(B)$	invert and replace OR with AND		de Morgan's Theorem
$\text{Not}(A \cdot B) = \text{Not}(A) + \text{Not}(B)$	invert and replace AND with OR		de Morgan's Theorem
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $A + (B \cdot C) = (A + B) \cdot (A + C)$	Distribution of the operands		Distributive Law
$A + A \cdot B \text{ OR } A \cdot A + B = A$	Absorption Law		Absorption Law

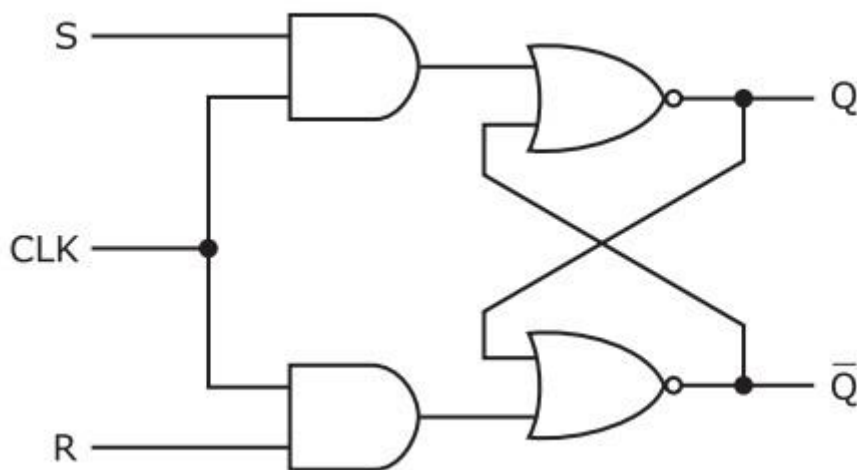
## S-R Flip Flop:

The SR flip flop is a 1-bit memory bistable (two stable state 0 and 1) device having two inputs, i.e., SET and RESET. The SET input 'S' set the device or produce the output 1, and the RESET input 'R' reset the device or produce the output 0. The SET and RESET inputs are labelled as **S** and **R**, respectively.

The SR flip flop stands for "Set-Reset" flip flop. The reset input is used to get back the flip flop to its original state from the current state with an output 'Q'. This output depends on the set and reset conditions, which is either at the logic level "0" or "1".

### Logic Circuit of S-R Flip Flop:

The logic circuit for SR Flip Flop constructed using NOR latch shown below



### Truth Table S-R Flip Flop:

INPUTS			OUTPUTS	REMARKS
S	R	$Q_n$ (Present State)	$Q_{n+1}$ (Next State)	States and Conditions
0	0	X	$Q_n$	Hold State condition $S = R = 0$
0	1	X	0	Reset state condition $S = 0, R = 1$
1	0	X	1	Set state condition $S = 1, R = 0$
1	1	X	Indeterminate	Indeterminate state condition $S = R = 1$

### What is excitation table?

For a given combination of present state  $Q_n$  and next state  $Q_{n+1}$ , excitation table tell the inputs required.

### Excitation Table S-R Flip Flop:

$Q_n$	$Q_{n+1}$	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR flip flop can be summarised as,

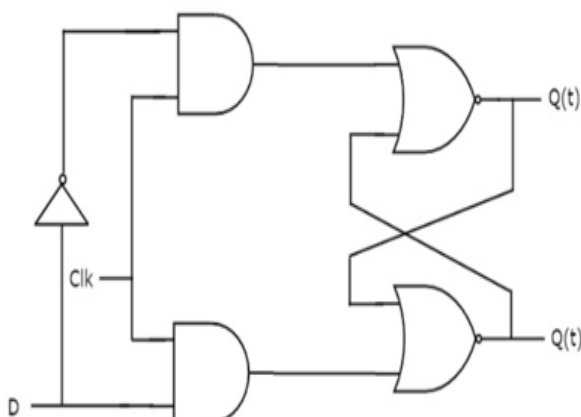
1. If no clock signal  $C=0$ , then output will no change.
2. IF clock signal  $C=1$ , and  $S=1$ ,  $R=0$  then, output  $Q=1$  and Not  $Q=0$  (Called Set).
3. IF  $R=1$ ,  $S=0$  and  $C=1$ , then output  $Q=0$ , and Not  $Q=1$ . (Called Reset).
4. IF  $C=1$  and  $S$ ,  $R=1$  then, output not defined, and it is called undefined (invalid) condition.

### D Flip Flop:

We need an **inverter** to prevent this from happening. We connect the inverter between the Set and Reset inputs for producing another type of flip flop circuit called D flip flop.

The D (Data) Flip Flop is modification of SR Flip Flop. The problem of undefined output in SR Flip Flop, when S & R become 1, get avoid in D Flip Flop, using single input.

#### Logic Circuit of D Flip Flop:



#### Truth Table for the D-type Flip Flop:

Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1



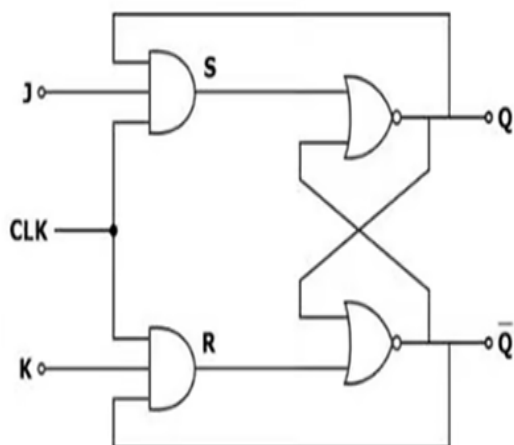
In D flip flop, the single input "D" is referred to as the "Data" input. When the data input is set to 1, the flip flop would be set, and when it is set to 0, the flip flop would change and become reset. However, this would be pointless since the output of the flip flop would always change on every pulse applied to this data input.

## J-K Flip Flop:

The JK flip flop work in the same way as the SR flip flop work. The JK flip flop has 'J' and 'K' flip flop instead of 'S' and 'R'. The only difference between JK flip flop and SR flip flop is that when both inputs of SR flip flop is set to 1, the circuit produces the invalid states as outputs, but in case of J-K flip flop, there are no invalid states even if both 'J' and 'K' flip flops are set to 1 as toggle.

The J-K flip-flop has four possible input combinations, i.e., **1, 0, "no change" and "toggle"**. The symbol of JK flip flop is the same as **S-R Bistable Latch** except for the addition of a clock input.

Logic Diagram of the J K Flip Flop



Truth Table J-K Flip Flop:

INPUTS			OUTPUTS	REMARKS
S	R	$Q_n$ (Present State)	$Q_{n+1}$ (Next State)	States and Conditions
0	0	X	$Q_n$	Hold State condition $S = R = 0$
0	1	X	0	Reset state condition $S = 0, R = 1$
1	0	X	1	Set state condition $S = 1, R = 0$
1	1	X	<b>Toggle</b>	Toggle state condition $S = R = 1$

When both of the inputs of J-K flip flop are set to 1 and clock input is also pulse "High" then from the SET state to a RESET state, the circuit will be toggled. The J-K flip flop work as a T-type toggle flip flop when both of its inputs are set to 1.

Remember it, The J-K flip-flop has a drawback of timing problem known as "**RACE**". The condition of *RACE* arises if the output Q changes its state before the timing pulse of the clock input has time to go in OFF state.

# Adders:

An adder is the combinational circuit that perform arithmetic and logic functions. There are two types of adder circuits named **half-adder** and **full-adder**. Both the half adder and the full adder circuits are used to perform addition and also widely used for performing various arithmetic functions in the digital circuits.

## What is a Half Adder?

A combinational logic circuit which is designed to add two binary digits is known as **half adder**. The half adder provides the output along with a carry value (if any). The half adder circuit is designed by connecting an EX-OR gate and one AND gate. It has two input terminals and two output terminals for sum and carry.

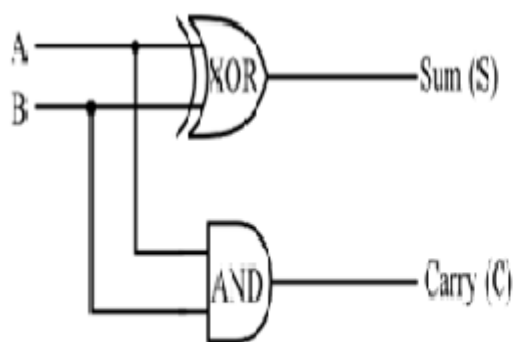


Figure - Half Adder Circuit

Truth Table of Half Adder			
Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$\text{Sum} = A \text{ XOR } B$$

$$\text{Carry} = A \text{ AND } B$$

## Full Adder:

Full adder is the combinational circuit which is designed to add three binary digits and produce two outputs is known as full adder. The **full adder** circuit adds three binary digits, where two are the inputs and one is the carry forwarded from the previous addition.

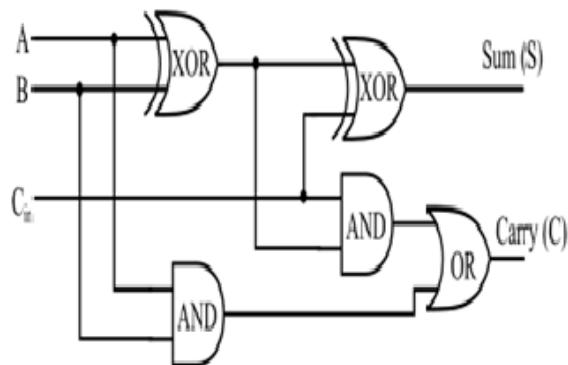


Figure - Full Adder Circuit

Truth Table of Full Adder				
Input			Output	
A	B	C <sub>in</sub>	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The circuit of the full adder consists of two EX-OR gates, two AND gates and one OR gate, which are connected together as shown in the full adder circuit. The output equations of the full adder are,

$$\text{Sum, } S = A \oplus B \oplus C_{in}$$

$$\text{Carry, } C = AB + BC_{in} + AC_{in}$$

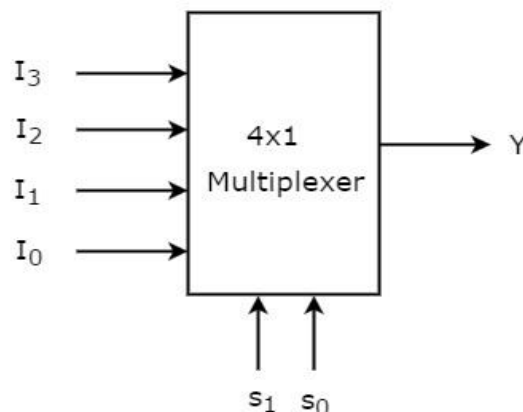
## Multiplexers:

**Multiplexer** is a combinational circuit that has maximum of  $2^n$  data inputs, 'n' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are 'n' selection lines, there will be  $2^n$  possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

### 4x1 Multiplexer:

4x1 Multiplexer has four data inputs  $I_3$ ,  $I_2$ ,  $I_1$  &  $I_0$ , two selection lines  $s_1$  &  $s_0$  and one output Y. The **block diagram** of 4x1 Multiplexer is shown in the following figure.



One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

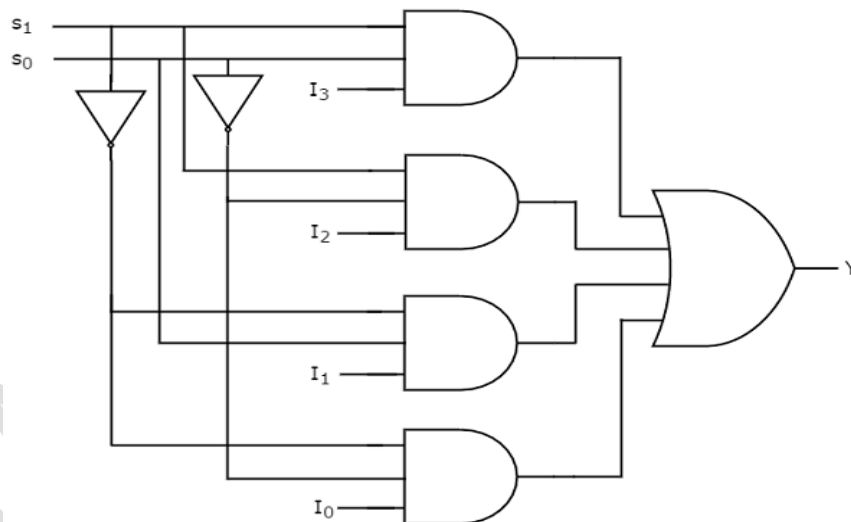
Selection Lines		Output
S <sub>1</sub>	S <sub>0</sub>	Y
0	0	I <sub>0</sub>
0	1	I <sub>1</sub>
1	0	I <sub>2</sub>
1	1	I <sub>3</sub>

From Truth table, we can directly write the **Boolean function** for output, Y as

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.

**Logic/Circuit Diagram of the 4x1 Multiplexer**



We can easily understand the operation of the above circuit. Similarly, you can implement 8 x 1 Multiplexer and 16 x 1 multiplexer by following the same procedure.

## 8 x 1 Multiplexer:

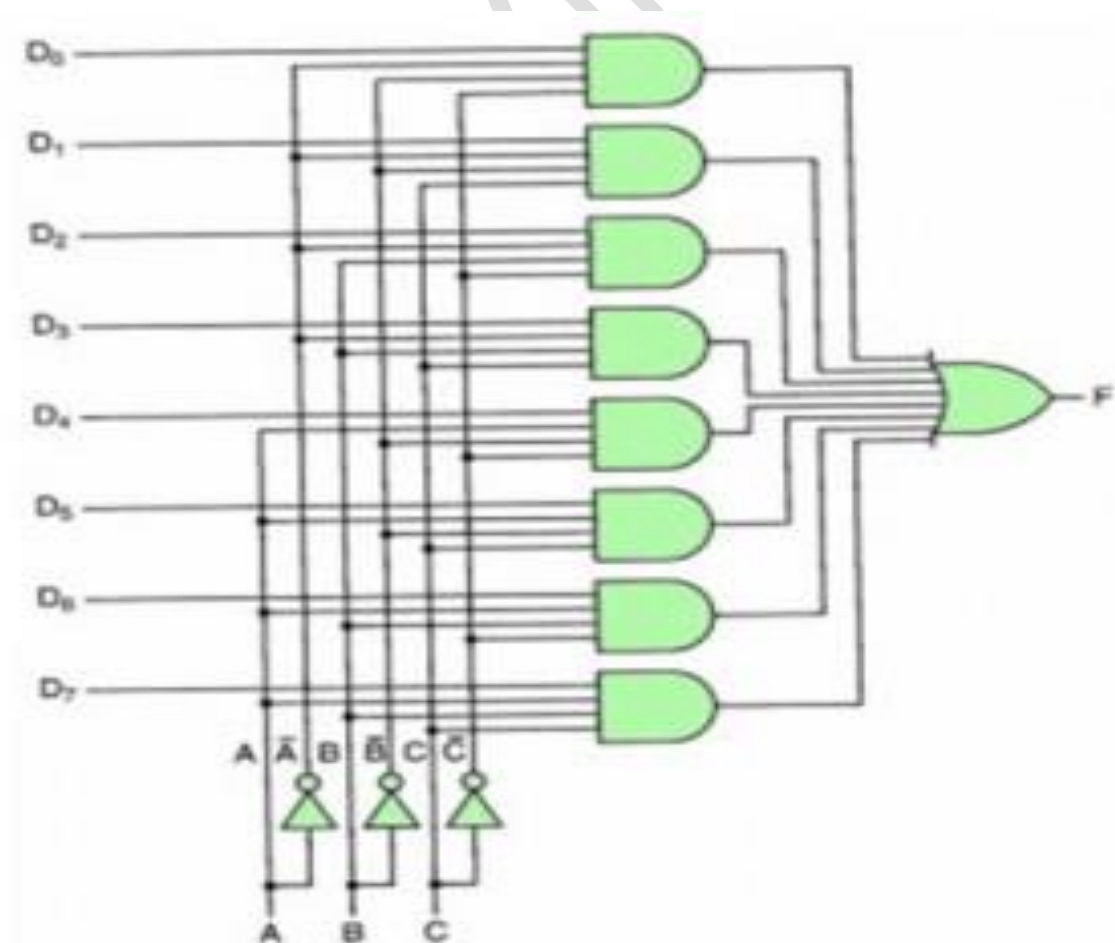
Let the 8x1 Multiplexer has eight data inputs I<sub>7</sub> to I<sub>0</sub>, three selection lines s<sub>2</sub>, s<sub>1</sub> & s<sub>0</sub> and one output Y.

The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs	Output
------------------	--------

$S_2$	$S_1$	$S_0$	$Y$
0	0	0	$I_0$
0	0	1	$I_1$
0	1	0	$I_2$
0	1	1	$I_3$
1	0	0	$I_4$
1	0	1	$I_5$
1	1	0	$I_6$
1	1	1	$I_7$

**Logical Diagram/circuit of the 8x1 Multiplexer**



The logical expression of the term Y is as follows:

$$Y = S_0' \cdot S_1' \cdot S_2' \cdot A_0 + S_0 \cdot S_1' \cdot S_2' \cdot A_1 + S_0' \cdot S_1 \cdot S_2' \cdot A_2 + S_0 \cdot S_1 \cdot S_2' \cdot A_3 + S_0' \cdot S_1' \cdot S_2 \cdot A_4 + S_0 \cdot S_1' \cdot S_2 \cdot A_5 + S_0' \cdot S_1 \cdot S_2 \cdot A_6 + S_0 \cdot S_1 \cdot S_2 \cdot A_7$$

**Multiplexers are used in various applications wherein multiple-data need to be transmitted by using a single line.**

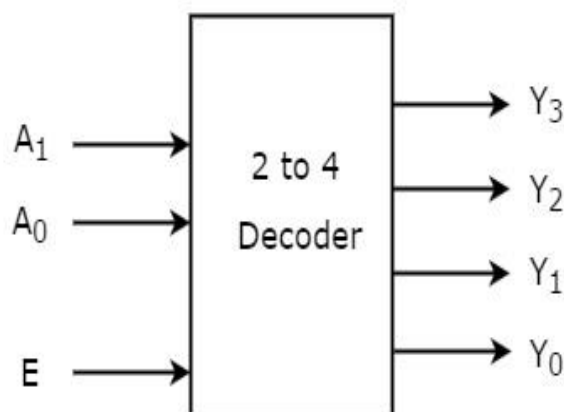
1. Data Communication System.
2. Computer Memory.
3. Telephone Network.
4. Transmission from the Computer System of a Satellite.
5. Arithmetic Logic Unit.
6. Serial to Parallel Converter etc.

## Decoders:

**Decoder** is a combinational circuit that they are used in code conversions like **binary to decimal** and has 'n' input lines and maximum of  $2^n$  output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the min terms of 'n'

### 2 to 4 Decoder

Let 2 to 4 Decoder has two inputs  $A_1$  &  $A_0$  and four outputs  $Y_3$ ,  $Y_2$ ,  $Y_1$  &  $Y_0$ . The **block diagram** of 2 to 4 decoder is shown in the following figure.



One of these four outputs will be '1' for each combination of inputs when enable, E is '1'. The **Truth table** of 2 to 4 decoder is shown below.

Enable	Inputs		Outputs			
E	A <sub>1</sub>	A <sub>0</sub>	Y <sub>3</sub>	Y <sub>2</sub>	Y <sub>1</sub>	Y <sub>0</sub>
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

From Truth table, we can write the **Boolean functions** for each output as

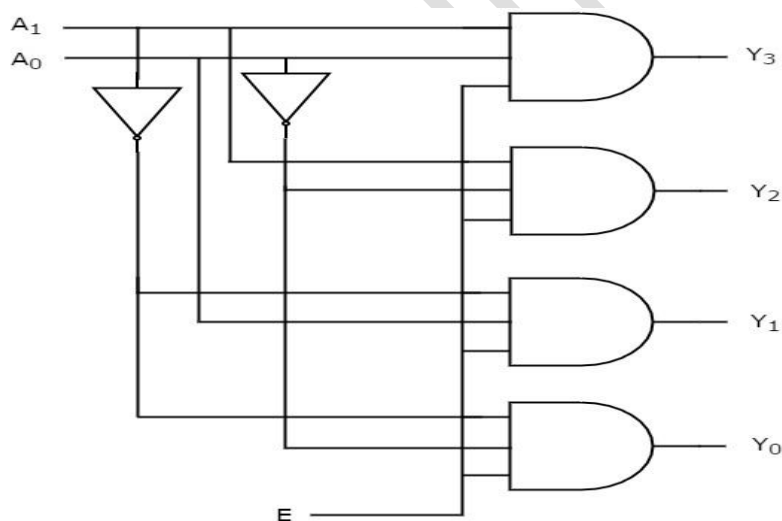
$$Y_3 = E \cdot A_1 \cdot A_0$$

$$Y_2 = E \cdot A_1 \cdot A_0'$$

$$Y_1 = E \cdot A_1' \cdot A_0$$

$$Y_0 = E \cdot A_1' \cdot A_0'$$

The Logic/Circuit diagram of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A<sub>1</sub> & A<sub>0</sub>, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces **eight min terms** of three input variables A<sub>2</sub>, A<sub>1</sub> & A<sub>0</sub> and 4 to 16 decoder produces sixteen min terms of four input variables A<sub>3</sub>, A<sub>2</sub>, A<sub>1</sub> & A<sub>0</sub>.

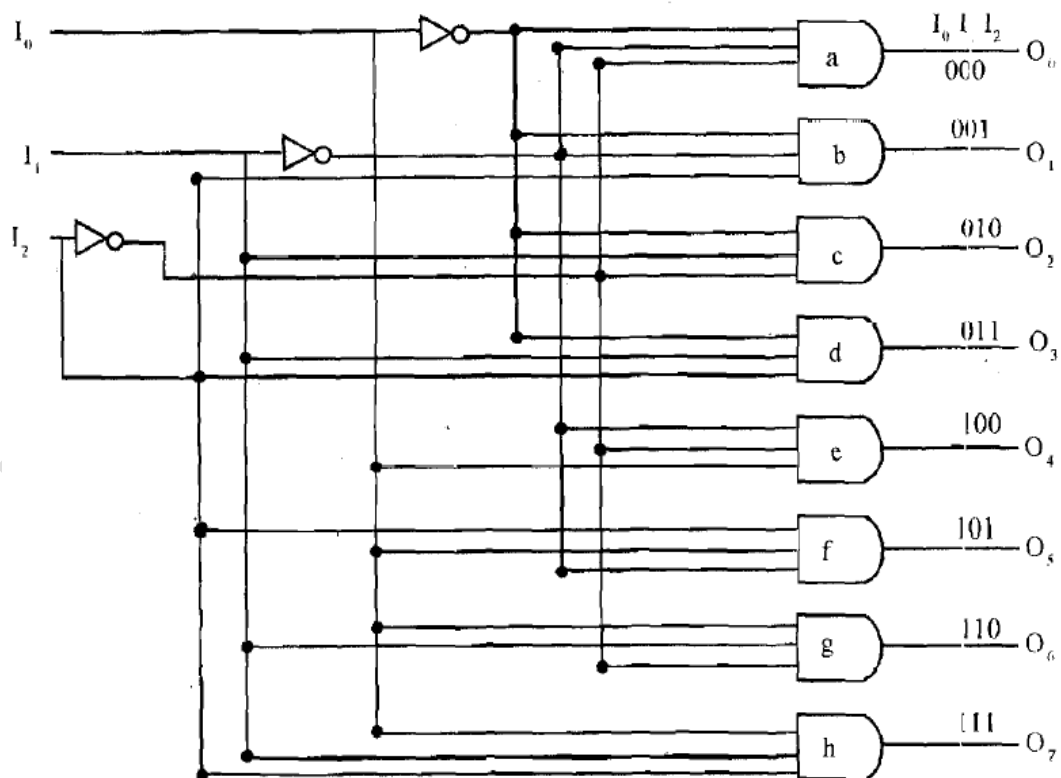
### 3 to 8 Decoder:

In this section, let us implement 3 to 8 decoder using 2 to 4 decoders. We know that 2 to 4 Decoder has two inputs, A<sub>1</sub> & A<sub>0</sub> and four outputs, Y<sub>3</sub> to Y<sub>0</sub>. Whereas, 3 to 8 Decoder has three inputs A<sub>2</sub>, A<sub>1</sub> & A<sub>0</sub> and eight outputs, Y<sub>7</sub> to Y<sub>0</sub>.

### 3 to 8 Line Decoder and Truth Table

<i>I</i> 0	<i>I</i> 1	<i>I</i> 2	<i>D</i> 0	<i>D</i> 1	<i>D</i> 2	<i>D</i> 3	<i>D</i> 4	<i>D</i> 5	<i>D</i> 6	<i>D</i> 7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

### 3 to 8 Line Decoder and Logic Diagram/Circuit





# Registers:

A Register is a group of flip-flops with each flip-flop capable of storing one bit of information. An  $n$ -bit register has a group of  $n$  flip-flops and is capable of storing binary information of  $n$ -bits. Here flip-flop transfer bit by bit into register, using set and Reset mechanism.

A register consists of a group of flip-flops and gates. The flip-flops hold the binary information and gates control when and how new information is transferred into a register. The below types of registers are used in the CPU.

**Accumulator:** This is the most common register, used to store data taken out from the memory.

**General Purpose Registers:** This is used to store data intermediate results during program execution. It can be accessed via assembly programming. General Purpose, Index, Status & Control, and Segment. The four general purpose registers are the **AX, BX, CX, and DX, etc.**

Also general purpose registers are extra registers that are present in the CPU and are utilized anytime data or a memory location is required. **These registers are used for storing operands and pointers.** These are mainly used for holding operands for logical and arithmetic operations.

**Special Purpose Registers:** Users do not access these registers. These registers are for Computer system, **SP (Stack pointer), BP (Base Pointer), DI (Destination Index register) etc.** are the example of the Special Purpose Register

**MAR:** Memory Address Register are those registers that holds the address for memory unit.

**MBR:** Memory Buffer Register stores instruction and data received from the memory and sent from the memory.

**DR:** Data Register contains 16 bits which hold the operand value, read from the memory location.

**PC:** Program Counter points to the next instruction to be executed.

**IR:** Instruction Register holds the instruction to be executed.

# Counters:

A counter is a register, which goes through a predetermined sequence of states when clock pulse is applied. In principle, the value of counter is incremented by 1 module the capacity of register i.e. when the value stored in a counter reaches its maximum value, the next increment value becomes zero. The counters are mainly used in circuits of digital systems where sequence and control operations are performed, for example, in CPU we have program counter (PC). The gates in the counter are connected in such a way as to produce the prescribed sequence of binary states.

The counting sequence is often depicted by a graph called a state diagram. A modulus-m counter (i.e., a counter with m states) has the following state diagram:

Each node  $S_i$  denotes the states of the counter and the arrows in the graph denote the order in which the states occur. Counters are available in two categories: ripple counters and synchronous counters.

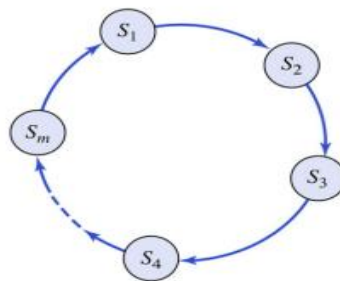
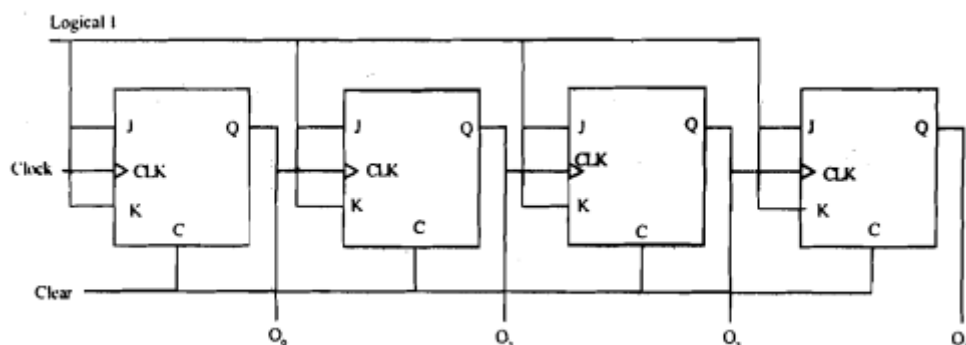


Figure: State Diagram of the Counter

## Asynchronous Counters

Counters can be classified into two categories, based on the way they operate: P Asynchronous and synchronous counters. In Asynchronous counter, the change in state of one flip flop triggers the other flip-flops. Synchronous counters are relatively faster because the state of all flip-flops can be changed at the same time,

This is more often referred to as ripple counter, as the change, which occurs in order to increment the counter ripples through it from one end to the other. Below figure shows an implementation of 4-bit ripple counter using J-K flip flops. This counter is incremental on the occurrence of each clock pulse and counts from 0000 to 1111 (i.e. 0 to 15).



### Figure: 4 bit Ripple Counter

The input line to J & K of all flip flops is kept high i.e. logic. Each time a clock pulse occurs the value of flip flop is complemented. Please note that the clock pulse is given only to first flip flop and second flip flop onwards, the output of previous flip flop is fed as clock signal. This implies that these flip flops will be complemented if the previous flip flop has a value 1. Thus, the effect of complement will ripple through these flip flops.

### Synchronous Counters:

The major disadvantage of ripple counter is the delay in changing the value. How? To understand this, take an instance when the state of ripple counter is 0111. Now the next state will be 1000, which means change in the state of all flip-flops. But will it occur simultaneously in ripple counter? No, first  $Q_0$  will change then  $Q_1$ ,  $Q_2$  & lastly  $Q_3$ . The delay is proportional to the length of the counter. Therefore, to avoid this disadvantage of ripple counters, synchronous counters are used in which all flip-flops change their states at same time. Below figure shows 3-bit synchronous counter.

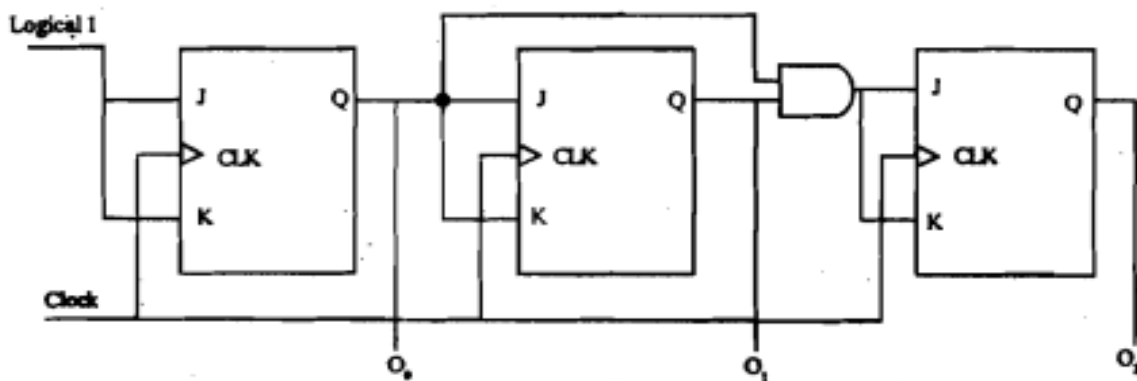


Figure: Logic diagram of 3-bit synchronous counter

You can understand the working of this counter by analysing the sequence of states ( $Q_2$ ,  $Q_1$ ,  $Q_0$ ) given in below table.

$Q_2$	$Q_1$	$Q_0$
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0

Figure: Truth table for 3 bit synchronous counter

***The operation can be summarized as:***

- i. The first flip-flop is complemented in every clock cycle.
- ii. The second flip-flop is complemented on occurrence of a clock cycle if the current state of first flip-flop is 1.
- iii. The third flip-flop is fed by an AND gate which is connected with output of first and second flip-flops. It will be complemented only when first & second flip flops are in Set State.

## **Memory:**

Memory is the smallest and fastest buffer (temporary storage) in a computer. Here Registers hold a small amount of data around 32 bits to 64 bits. But registers are not a part of the main memory and is located in the CPU in the form of registers, which are the smallest data holding elements. A register temporarily holds frequently used data, instructions, and memory address that are to be used by CPU. They hold instructions that are currently processed by the CPU. All data is required to pass through registers before it can be processed. So, they are used by CPU to process the data entered by the users. The speed of a CPU depends on the number and size (no. of bits) of registers that are built into the CPU. The widely used Registers include Accumulator or AC, Data Register or DR, the Address Register or AR, Program Counter (PC), I/O Address Register, and more are hold the memory.

## **Bus System:**

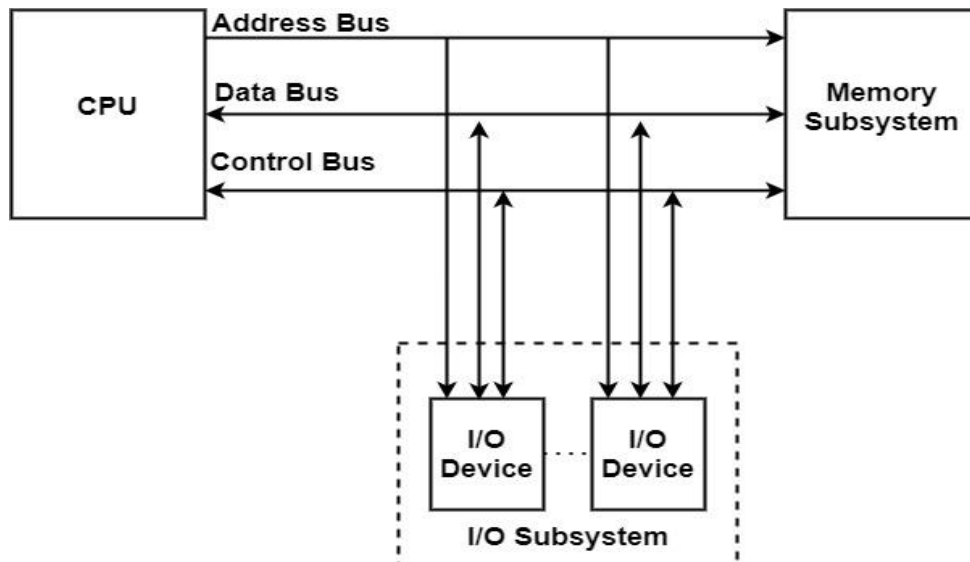
A bus is a set of wires. The elements and devises of the computer are linked to the buses. Buses can transfer data from one element/Device to another, the source element outputs data onto the bus. The destination element then inputs this information from the bus. Basic there are three types of main buses are as,

1. Address Bus
2. Data Bus and
3. Control Bus

The system has three buses as shown in the following figure. The uppermost bus is the address bus. When the CPU reads data or instructions from or writes data to memory, it should determine the address of the memory location it needs to access.

It outputs this address to the memory bus, memory inputs this address from the address bus and uses it to access the suitable memory location. Each I/O device including a keyboard, monitor, or disk drive, has a specific address as well.

When accessing an I/O device, the CPU locates the address of the device on the address bus. Each device can read the address off of the bus and specify whether it is the device being accessed by the CPU.



**Figure: System Bus Diagram**

Data is shared via the data bus. When the CPU fetches information from memory, it first outputs the memory address on its address bus. Therefore memory outputs the data onto the data bus, the CPU can read the information from the data bus. When writing data to memory, the CPU first outputs the address onto the address bus, therefore outputs the data onto the data bus.

The control bus is different from the other two buses. The address bus includes  $n$  lines, which associate to transit one  $n$ -bit address value. The lines of the data bus work simultaneously to send a single, multi-bit value. In contrast, the control bus is a collection of individual control signals. These signals indicate whether data is to be read into or written out of the CPU, whether the CPU is accessing memory or an I/O device, and whether the I/O device or memory is read to transfer data.

# Instruction Set:

## What is an Instruction Set?

Instruction set is the collection of machine language instructions that a particular processor understands and executes. In other words, a set of assembly language mnemonics represents the machine code of a particular computer. Therefore, if we define all the instructions of a computer, we can say we have defined the instruction set. It should be noted here that the instructions available in a computer are machine dependent, that is, a different processors have different instruction sets. However, a newer processor that may belong to some family may have a compatible but extended instruction set of an old processor of that family. Instructions can take different formats. The instruction format involves:

- The instruction length;
- The type;
- Length and position of operation codes in an instruction; and
- The number and length of operand addresses etc.

## What are the elements of an instruction?

As the purpose of instruction is to communicate to CPU what to do, it requires a minimum set of communication as:

- What operation to perform?
- On what operands?

Thus, each instruction consists of several fields. The most common fields found in instruction formats are:

### Opcode: (What operation to perform?)

- An operation code field termed as opcode that specifies the operation to be performed.

### Operands: (Where are the operands?)

- An address field of operand on which data processing is to be performed.
- An operand can reside in the memory or a processor register or can be incorporated within the operand field of instruction as an immediate constant. Therefore a mode field is needed that specifies the way the operand or its address is to be determined.

A sample instruction format is given in figure 1.

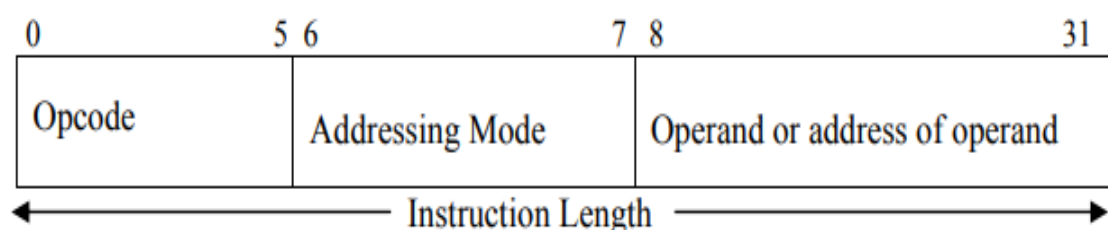


Figure 1: A Hypothetical Instruction Format of 32 bits

Please note the following points in Figure 1:

- The opcode size is 6 bits. So, in general it will have  $2^6 = 64$  operations.
- There is only one operand address machine
- There are two bits for addressing modes. Therefore, there are  $2^2 = 4$  different addressing modes possible for this machine.
- The last field (8 – 31 bits = 24 bits) here is the operand or the address of operand field.

In case of immediate operand the maximum size of the unsigned operand would be  $2^{24}$ .

In case it is an address of operand in memory, then the maximum physical memory size supported by this machine is  $2^{24} = 16$  MB.

For this machine there may be two more possible addressing modes in addition to the immediate and direct. However, let us not discuss addressing modes right now. They will be discussed in general, details in section 1.4 of this unit.

The opcode field of an instruction is a group of bits that define various processor operations such as LOAD, STORE, ADD, and SHIFT to be performed on some data stored in registers or memory.

The operand address field can be data, or can refer to data – that is address of data, or can be labels, which may be the address of an instruction you want to execute next, such labels are commonly used in Subroutine call instructions. An operand address can be:

- The memory address
- CPU register address
- I/O device address

**Please note that if the operands are placed in processor registers then an instruction executes faster than that of operands placed in memory, as the registers are very high-speed memory used by the CPU. However, to put the value of a memory operand to a register you will require a register LOAD instruction.**

How is an instruction represented? Instruction is represented as a sequence of bits. A layout of an instruction is termed as instruction format. Instruction formats are primarily machine dependent. A CPU instruction set can use many instruction formats at a time. Even the length of opcode varies in the same processor.

## How many instructions in a Computer?

A computer can have a large number of instructions and addressing modes. The older computers with the growth of Integrated circuit technology have a very large and complex set of instructions. These are called “complex instruction set computers” (CISC). Examples of CISC architectures are the Digital Equipment Corporation VAX computer and the IBM 370 computer.

However, later it was found in the studies of program style that many complex instructions found CISC are not used by the program. This led to the idea of making a simple but faster computer, which could execute simple instructions much faster. These computers have simple instructions, registers addressing and move registers. These are called Reduced Instruction Set Computers (RISC).

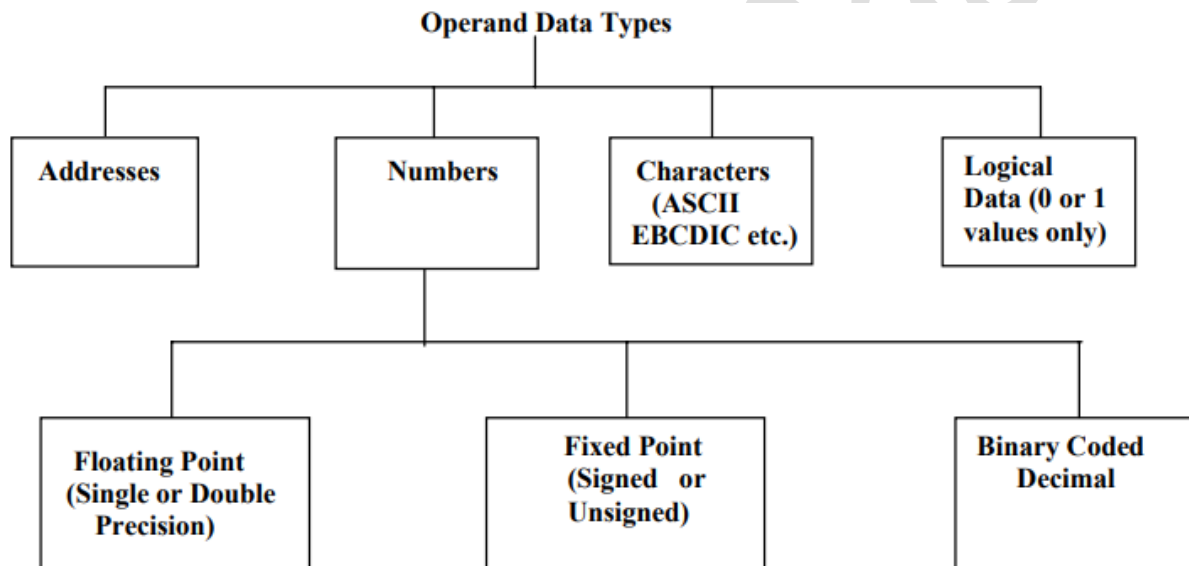
## 1.3 INSTRUCTION SET DESIGN CONSIDERATIONS

Some of the basic considerations for instruction set design include selection of:

- A set of data types (e.g. integers, long integers, doubles, character strings etc.).
- A set of operations on those data types.
- A set of instruction formats. Includes issues like number of addresses, instruction length etc.
- A set of techniques for addressing data in memory or in registers.
- The number of registers which can be referenced by an instruction and how they are used.

### 1.3.1 Operand Data Types

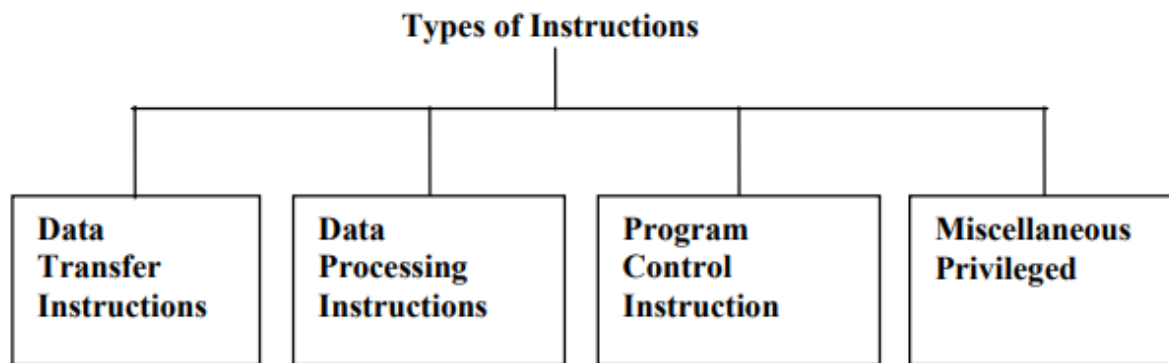
Operand is that part of an instruction that specifies the address of the source or result, or the data itself on which the processor is to operate. Operand types usually give operand size implicitly. In general, operand data types can be divided in the following categories.



### 1.3.2 Types of Instructions

Computer instructions are the translation of high level language code to machine level language programs. Thus, from this point of view the machine instructions can be classified under the following categories.





**Figure 3: Types of Instructions**

**Data Transfer Instructions** These instructions transfer data from one location in the computer to another location without changing the data content. The most common transfers are between:

- Processor registers and memory,
- Processor registers and I/O, and
- Processor registers themselves.

Operation Name	Mnemonic	Description
Load	LD	Loads the contents from memory to register.
Store	ST	Store information from register to memory location.
Move	MOV	Data Transfer from one register to another or between CPU registers and memory.

Exchange	XCH	Swaps information between two registers or a register and a memory word.
Clear	CLEAR	Causes the specified operand to be replaced by 0's.
Set	SET	Causes the specified operand to be replaced by 1's.
Push	PUSH	Transfers data from a processor register to top of memory stack.
Pop	POP	Transfers data from top of stack to processor register.

## Data Processing Instructions:

These instructions perform arithmetic and logical operations on data. Data Manipulation Instructions can be divided into three basic types:

**Arithmetic:** The four basic operations are ADD, SUB, MUL and DIV. An arithmetic instruction may operate on fixed-point data, binary or decimal data etc. The other possible operations include a variety of single-operand instructions, for example ABSOLUTE, NEGATE, INCREMENT, DECREMENT.

The execution of arithmetic instructions requires bringing in the operands in the operational registers so that the data can be processed by ALU. Such functionality is implemented generally within instruction execution steps.

Logical: AND, OR, NOT, XOR operate on binary data stored in registers. For example, if two registers contain the data:

**R1 = 1011 0111**

**R2 = 1111 0000**

Then,

R1 AND R2 = 1011 0000. Thus, the AND operation can be used as a mask that selects certain bits in a word and zeros out the remaining bits. With one register is set to all 1's, the XOR operation inverts those bits in R1 register where R2 contains 1.

**R1 XOR R2 = 0100 0111**

Shift: Shift operation is used for transfer of bits either to the left or to the right. It can be used to realize simple arithmetic operation or data communication/recognition etc. Shift operation is of three types:

1. Logical shifts LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT insert zeros to the end bit position and the other bits of a word are shifted left or right respectively. The end bit position is the leftmost bit for shift right and the rightmost bit position for the shift left. The bit shifted out is lost.
2. Arithmetic shifts ARITHMETIC SHIFT LEFT and ARITHMETIC SHIFT RIGHT are the same as LOGICAL SHIFT LEFT and LOGICAL SHIFT RIGHT the Central Processing Unit except that the sign bit it remains unchanged. On an arithmetic shift right, the sign bit is replicated into the bit position to its right. On an arithmetic shift left, a logical shift left is performed on all bits but the sign bit, which is retained.
3. Circular shifts ROTATE LEFT and ROTATE RIGHT. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end.

**Character and String Processing Instructions:** String manipulation typically is done in memory. Possible instructions include COMPARE STRING, COMPARE CHARACTER, MOVE STRING and MOVE CHARACTER. While compare character usually is a byte-comparison operation, compare string always involves memory address.

**Stack and register manipulation:** If we build stacks, stack instructions prove to be useful. LOAD IMMEDIATE is a good example of register manipulation (the value being loaded is part of the instruction). Each CPU has multiple registers, when instruction set is designed; one has to specify which register the instruction is referring to.

$$A = B * C + D * E.$$

Evaluation of Stack Machine		Accumulator Machine	
Program	Comments	Programs	Comments
PUSH B	Push the value B	LOAD B	Load B in AC
PUSH C	Push C	MULT C	Multiply AC with C in AC
MULT	Multiply (B×C) and store result on stack top	STORE T	Store B×C into Temporary T
PUSH D	Push D	LOAD D	Load D in AC
PUSH E	Push E	MULT E	Multiply E in AC

MULT	Multiply D×E and store result on stack top	ADD T	B×C + D×E
ADD	Add the top two values on the stack	STORE A	Store Result in A
POP A	Store the value in A		

## General Register Organization:

**General Register Organization** is the processor architecture that stores and manipulates data for computations. The main components of a register organization include registers, memory, and instructions.

A bus organization for seven CPU registers is shown in Fig. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic micro operation that is to be performed. The result of the micro operation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

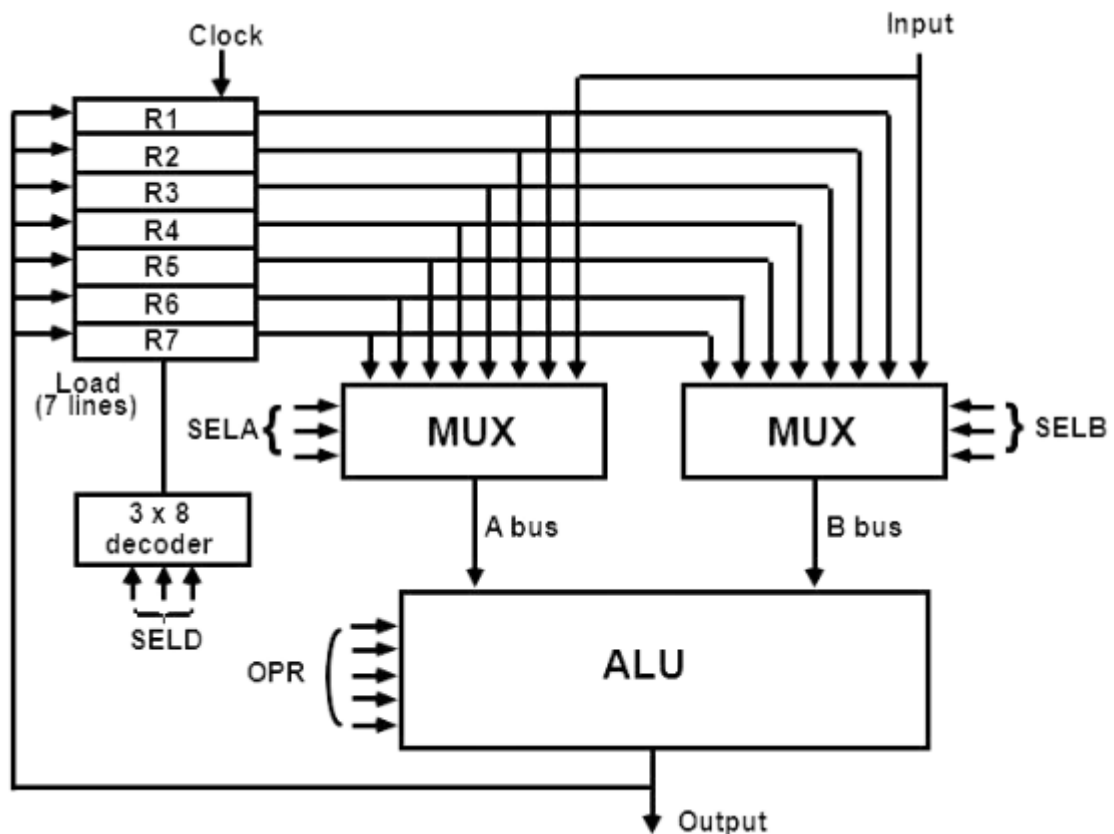
The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

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.
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 R1.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary information from the output bus is transferred into R1. To achieve a fast response time, the ALU is constructed with high-speed circuits.



**Figure:** Register set with common ALU

## Micro Operations:

1. Arithmetic Micro operation
2. Logic Micro operation
3. Shift Micro operation

### Arithmetic micro-operations:

Micro-operations are operations done on data contained in registers. A micro-operation is a simple operation performed on data stored in one or more registers.

Addition, subtraction, increment and decrement are examples of micro-operations. Let us understand arithmetic micro-operations with examples.

**TABLE** Arithmetic Microoperations

Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

**Logic micro-operations:**

Logic micro-operations are used on the bits of data stored in registers. These micro-operations treat each bit independently and create binary variables from them. There are a total of 16 micro-operations available.

**TABLE** Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer $A$
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer $B$
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement $B$
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement $A$
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

## Sift Micro operation:

The operation that changes the adjacent bit position of the binary values stored in the register is known as shift micro-operation. Shift micro operations are involved in shifting of bits of a register. Shift micro operations are used for serial transfer of data. Shift Micro-operations are categorized in detail as follows:

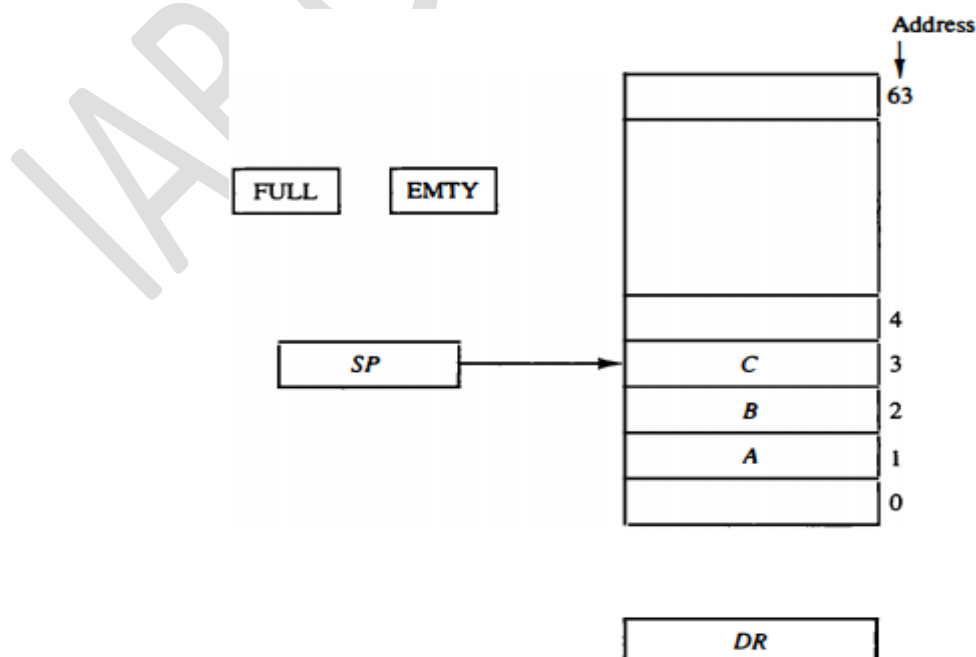
**TABLE**      Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register $R$
$R \leftarrow \text{shr } R$	Shift-right register $R$
$R \leftarrow \text{cil } R$	Circular shift-left register $R$
$R \leftarrow \text{cir } R$	Circular shift-right register $R$
$R \leftarrow \text{ashl } R$	Arithmetic shift-left $R$
$R \leftarrow \text{ashr } R$	Arithmetic shift-right $R$

## Stack Organization:

### 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 3 shows the organization of a 64-word register 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.



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

- **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. Note that item C has been read out but not physically removed.
- **This does not matter because when the stack is pushed**, a new item is written in its place. In a 64-word stack, the stack pointer contains 6 bits because  $2^6 = 64$ .
- **Since SP has only six bits**, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since  $111111 + 1 = 1000000$  in binary, but SP can accommodate only the six least significant bits.
- **Similarly, when 000000 is decremented by 1**, the result is 111111. The one-bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of 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;
  - $SP \leftarrow SP + 1$  Increment stack pointer
  - $M[SP] \leftarrow DR$  Write item on top of the stack
  - If (SP = 0) then (FULL  $\leftarrow$  1) Check if stack is full
  - $EMTY \leftarrow 0$  Mark the stack not empty
- **The stack pointer** is incremented so that it points to the address of the next-higher word. A memory write operation inserts the word from DR into the top of the stack. Note that SP holds the address of the top of the stack and that  $M[SP]$  denotes the memory word specified by the address presently available in SP.
- **The first item stored** in the stack is at address L. The last item is stored at address 0.
- **If SP reaches 0**, the stack is full of items, so FULL is set to 1. This condition is reached if the top item prior to the last push was in location 63 and, after incrementing SP, the last item is stored in location 0.
- **Once an item is stored** in location 0, there are no more empty registers in the stack. If an item is written in the stack, obviously the stack cannot be empty, so EMTY is cleared to 0.
- **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:
  - $DR \leftarrow M[SP]$  Read item from the top of stack
  - $SP \leftarrow SP - 1$  Decrement stack pointer
  - If (SP = 0) then (EMTY  $\leftarrow$  1) Check if stack is empty
  - $FULL \leftarrow 0$  Mark the stack not full

## Memory Stack:

A **stack can exist** as a stand-alone unit as in Fig. 3 or 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.

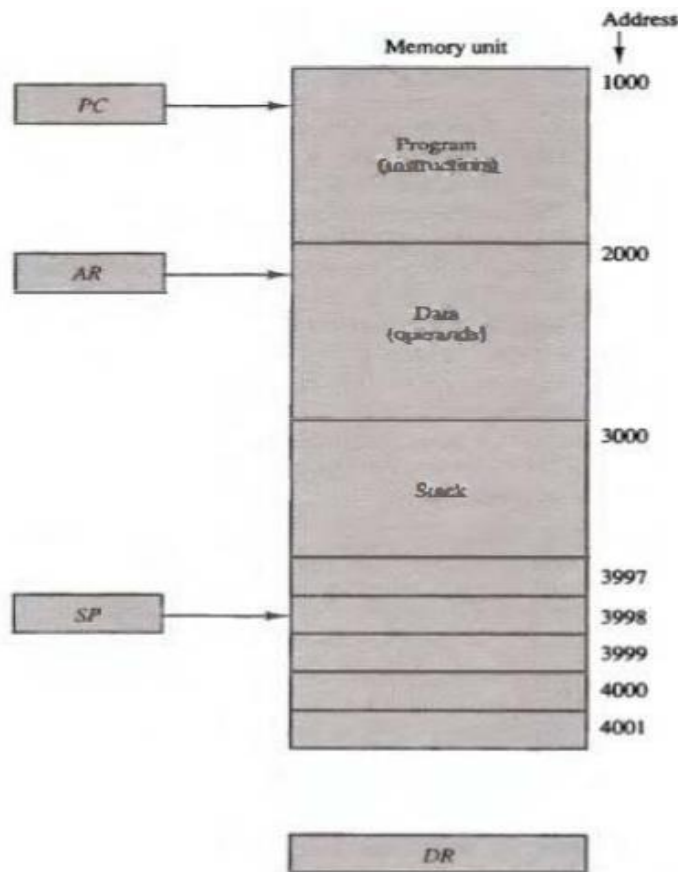


Figure 4 Computer memory with program, data, and stack segments.

- **The stack pointer SP** points at the top of the stack. The three registers are connected to a common address bus, and either one can provide an address for memory.
- **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 the stack. As shown in Fig. 4, 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.
- **No provisions** are available for stack limit checks.
- **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:
  - $SP \leftarrow SP - 1$
  - $M[SP] \leftarrow DR$
- **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:



- $DR \leftarrow M[SP]$
- $SP \leftarrow SP + 1$
- **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.
- **Most computers** do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).
- **The stack limits** can be checked by using two processor registers: one to hold the upper limit (3000 in this case), and the other to hold the lower limit (4001 in this case).
- **After a push operation**, SP is compared with the upper-limit register and after a pop operation, SP is compared with the lower-limit register.
- **The two micro operations** needed for either the push or pop are (1) an access to memory through SP, and (2) updating SP. Which of the two micro operations is done first and whether SP is updated by incrementing or decrementing depends on the organization of the stack.
- **In Fig. 4 the stack grows** by decreasing the memory address. The stack may be constructed to grow by increasing the memory address as in above Fig. 3.
- **In such a case, SP is incremented** for the push operation and decremented for the pop operation. A stack may be constructed so that SP points at the next empty location above the top of the stack.
- **In this case the sequence** of micro operations must be interchanged. A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. Henceforth, SP is automatically decremented or incremented with every push or pop operation.
- **The advantage of a memory stack** is that the CPU can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

## 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 subcycles 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.

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.

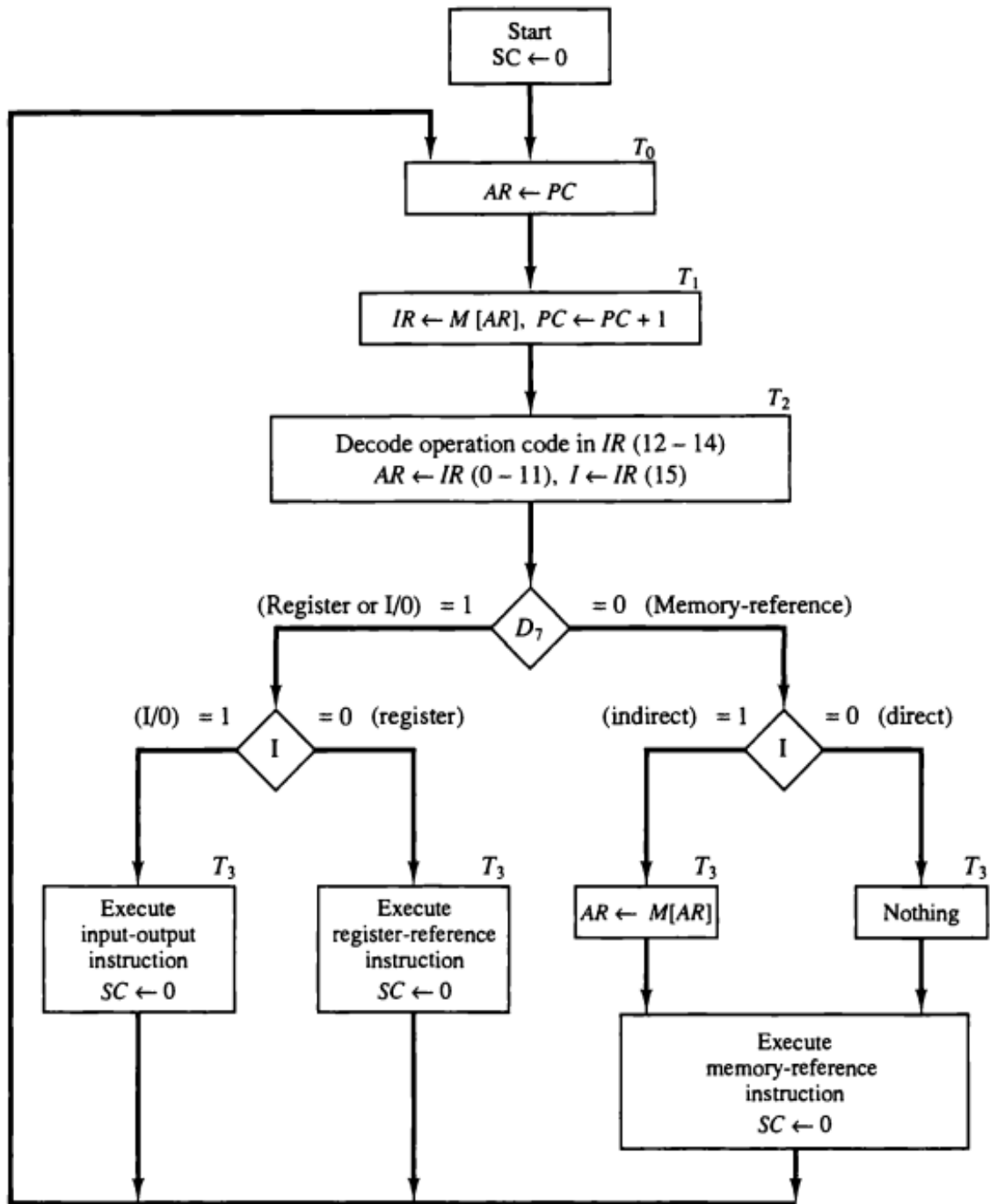


Figure 5-9 Flowchart for instruction cycle (initial configuration).

Following are the different types of registers involved in each instruction cycle:

1. **Memory address registers (MAR):** It is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
2. **Memory Buffer Register (MBR):** It is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from the memory.
3. **Program Counter (PC):** Holds the address of the next instruction to be fetched.
4. **Instruction Register (IR):** Holds the last instruction fetched.

### 1. Fetch Cycle:

The fetching of instruction is the first phase. The fetch instruction is common for each instruction executed in a central processing unit. In this phase, the central processing unit sends the PC to MAR and then sends the READ command into a control bus. After sending a read command on the data bus, the memory returns the instruction, which is stored at that particular address in the memory. Then, the CPU copies data from the data bus into MBR and then copies the data from MBR to registers. After all this, the pointer is incremented to the next memory location so that the next instruction can be fetched from memory.

### 2. Decode Cycle:

The decoding of instruction is the second phase. In this phase, the CPU determines which instruction is fetched from the instruction and what action needs to be performed on the instruction. The opcode for the instruction is also fetched from memory and decodes the related operation which needs to be performed for the related instruction.

### 3. Read Cycle:

The reading of an effective address is the third phase. This phase deals with the decision of the operation. The operation can be of any type of memory type non-memory type operation.

**Memory instruction can be categorized into two categories: direct memory instruction and indirect memory instruction.**

### 4. Execute Cycle:

The execution of the instruction is the last phase. In this stage, the instruction is finally executed. The instruction is executed, and the result of the instruction is stored in the register. After the execution of an instruction, the CPU prepares itself for the execution of the next instruction.

## Instruction Format:

A computer performs a task based on the instruction provided. Instruction in computers comprises groups called fields. These fields contain different information as for computers everything is in 0 and 1 so each field has different significance based on which a CPU decides what to perform. The most common fields are:

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

Instruction is of variable length depending upon the number of addresses it contains. Generally, CPU organization is of three types based on the number of address fields:

1. Single Accumulator organization
2. General register organization
3. Stack organization

Note that we will use  $X = (A+B)*(C+D)$  expression to showcase the procedure.

### 1. Zero Address Instructions:

A stack-based computer does not use the address field in the instruction. To evaluate an expression first it is converted to reverse Polish Notation i.e. Postfix Notation.

Expression:  $X = (A+B)*(C+D)$

Postfixed:  $X = AB+CD+*$

TOP means top of stack

M[X] is any memory location

PUSH	A	TOP = A
PUSH	B	TOP = B
ADD		TOP = A+B
PUSH	C	TOP = C
PUSH	D	TOP = D
ADD		TOP = C+D
MUL		TOP = (C+D)*(A+B)
POP	X	M[X] = TOP

## 2. One Address Instructions

This uses an implied ACCUMULATOR register for data manipulation. One operand is in the accumulator and the other is in the register or memory location. Implied means that the CPU already knows that one operand is in the accumulator so there is no need to specify it.

Expression:  $X = (A+B)*(C+D)$

AC is accumulator

M[] is any memory location

M[T] is temporary location

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

## 2. Two Address Instructions

This is common in commercial computers. Here two addresses can be specified in the instruction. Unlike earlier in one address instruction, the result was stored in the accumulator, here the result can be stored at different locations rather than just accumulators, but require more number of bit to represent address.

Expression:  $X = (A+B)*(C+D)$

R1, R2 are registers

M [] is any memory location

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

### 3. Three Address Instructions

This has three address field to specify a register or a memory location. Program created are much short in size but number of bits per instruction increase. These instructions make creation of program much easier but it does not mean that program will run much faster because now instruction only contain more information but each micro operation (changing content of register, loading address in address bus etc.) will be performed in one cycle only.

Expression:  $X = (A+B)*(C+D)$

R1, R2 are registers

M[] is any memory location

ADD	R1, A, B	$R1 = M[A] + M[B]$
ADD	R2, C, D	$R2 = M[C] + M[D]$
MUL	X, R1, R2	$M[X] = R1 * R2$

## Addressing Modes:

### 1. Implied Addressing 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 are specified implicit in the definition of 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

### 2. Immediate Addressing Mode:

In Immediate Addressing Mode operand is specified in the instruction itself. In other words, an immediate mode instruction has an operand field rather than an address field, which contain actual operand to be used in conjunction with the operand specified in the instruction. That is, in this mode, the format of instruction is:



**As an example: The Instruction:**

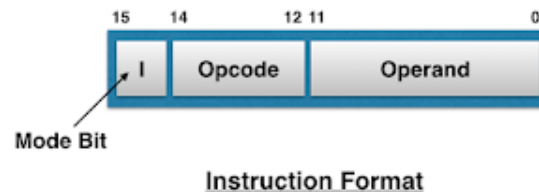
**MVI 06**    Move 06 to the accumulator

**ADD 05**    ADD 05 to the content of accumulator

In addition to this, this mode is very useful for initialising the register to a constant value.

### 3. Direct and Indirect Addressing Modes:

The instruction format for direct and indirect addressing mode is shown below:



It consists of 3-bit opcode, 12-bit address and a mode bit designated as (I). The mode bit (I) is zero for Direct Address and 1 for Indirect Address. Now we will discuss about each in detail one by one.

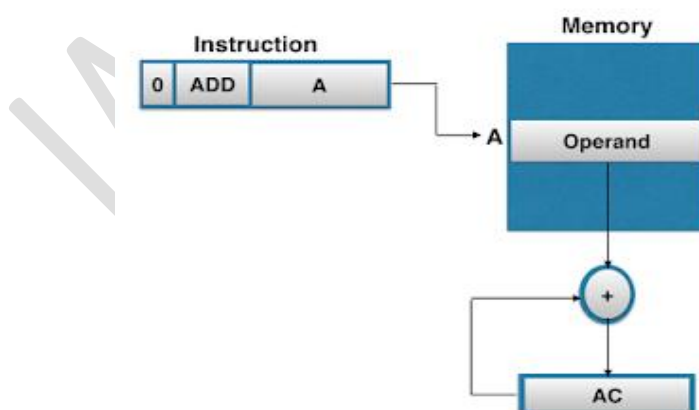
### 4. Direct Addressing Mode:

Direct Addressing Mode is also known as “Absolute Addressing Mode”. In this mode the address of data (operand) is specified in the instruction itself. That is, in this type of mode, the operand resides in memory and its address is given directly by the address field of the instruction.

Means, in other words, in this mode, the address field contain the Effective Address of operand i.e.,  $EA=A$

As an example: Consider the instruction:

**ADD A** Means add contents of cell A to accumulator. It would look like as shown below:



Here, we see that in it Memory Address=Operand.

### 5. Indirect Addressing Mode:

In this mode, the address field of instruction gives the memory address where on, the operand is stored in memory. That is, in this mode, the address field of the instruction gives the address

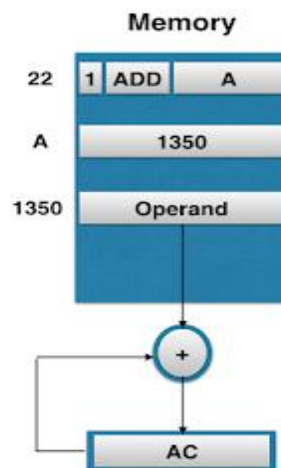
where the “Effective Address” is stored in memory.

i.e.,  $EA = (A)$

Means, here, Control fetches the instruction from memory and then uses its address part to access memory again to read Effective Address.

**As an example: Consider the instruction:**

**ADD (A) Means** adds the content of cell pointed to contents of A to Accumulator. It look like as shown in figure below:



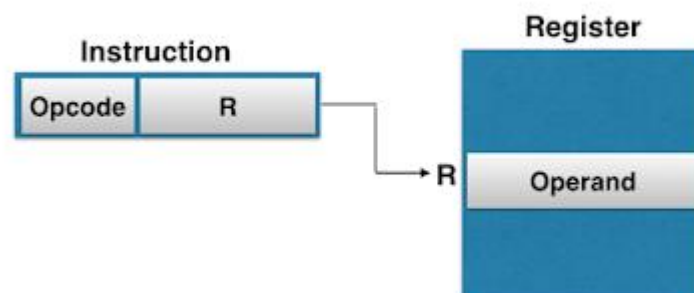
Thus in it,  $AC \leftarrow M[M[A]]$  [M=Memory]  
i.e.,  $(A) = 1350 = EA$

## 6. Register Addressing Mode:

In Register Addressing Mode, the operands are in registers that reside within the CPU. That is, in this mode, instruction specifies a register in CPU, which contain the operand. It is like Direct Addressing Mode, the only difference is that the address field refers to a register instead of memory location.

i.e.,  $EA = R$

It look like as:



Example of such instructions are:

**MOV AX, BX**      Move contents of Register BX to AX  
**ADD AX, BX**      Add the contents of register BX to AX

Here, AX, BX are used as register names which is of 16-bit register.

Thus, for a Register Addressing Mode, there is no need to compute the actual address as the operand is in a register and to get operand there is no memory access involved.

## 7. Register Indirect Addressing Mode:

In Register Indirect Addressing Mode, the instruction specifies a register in CPU whose contents give the operand in memory. In other words, the selected register contain the address of operand rather than the operand itself. That is,

i.e.,  $EA=(R)$

Means, control fetches instruction from memory and then uses its address to access Register and looks in Register(R) for effective address of operand in memory.

It look like as:



Here, the parentheses are to be interpreted as meaning contents of.

Example of such instructions are:

**MOV AL, [BX]**

Code example in Register:

```
MOV BX, 1000H
MOV 1000H, operand
```

From above example, it is clear that, the instruction(MOV AL, [BX]) specifies a register[BX], and in coding of register, we see that, when we move register [BX], the register contain the address of operand(1000H) rather than address itself.

## 8. Auto-increment addressing Mode:

Auto-increment Addressing Mode are similar to Register Indirect Addressing Mode except that the register is incremented after its value is loaded (or accessed) at another location like accumulator (AC).

That is, in this case also, the Effective Address is equal to

$EA=(R)$



## 9. Auto-decrement Addressing Mode:

Auto-decrement Addressing Mode is reverse of auto-increment, as in it the register is decrement before the execution of the instruction. That is, in this case, effective address is equal to

$$EA = (R) - 1$$

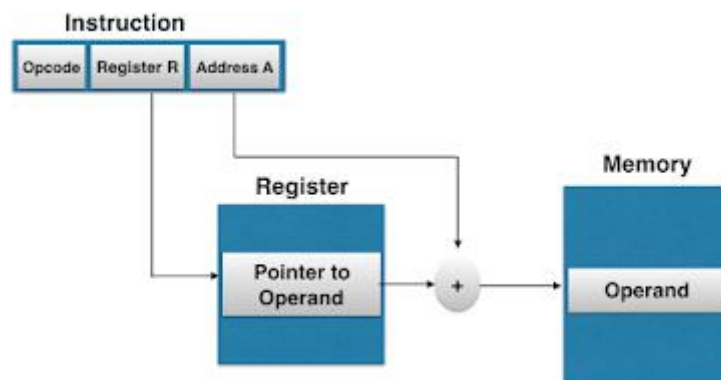
## 10. Displacement Based Addressing Modes:

Displacement Based Addressing Modes is a powerful addressing mode as it is a combination of direct addressing or register indirect addressing mode.

i.e.,  $EA = A + (R)$

Means, Displacement Addressing Modes requires that the instruction have two address fields, at least one of which is explicit means, one is address field indicate direct address and other indicate indirect address. That is, value contained in one addressing field is A, which is used directly and the value in other address field is R, which refers to a register whose contents are to be added to produce effective address.

It look like as shown in figure below:



There are three areas where Displacement Addressing modes are used. In other words, Displacement Based Addressing Modes are of three types. These are:

1. Relative Addressing Mode
2. Base Register Addressing Mode
3. Indexing Addressing Mode

## 11. Relative Addressing Mode:

*In Relative Addressing Mode, the contents of program counter is added to the address part of instruction to obtain the Effective Address.*

That is, in Relative Addressing Mode, the address field of the instruction is added to implicitly reference register Program Counter to obtain effective address.

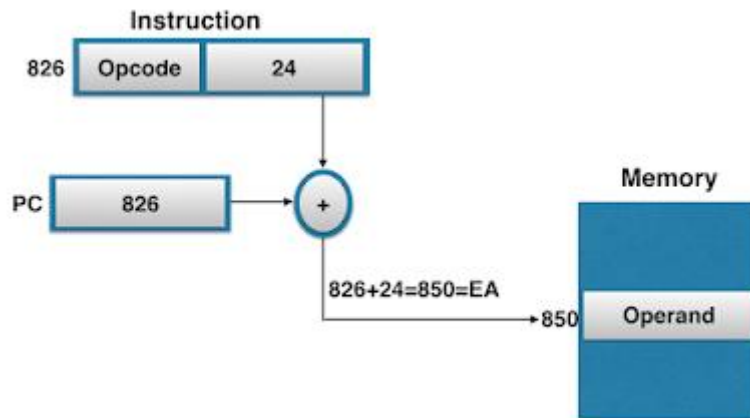
i.e.,  $EA = A + PC$

*It becomes clear with an example:*

Assume that PC contains the no.- 825 and the address part of instruction contain the no.- 24, then the instruction at location 825 is read from memory during fetch phase and the Program Counter is then incremented by one to 826.

The effective address computation for relative address mode is  $826+24=850$

It is depicted in fig. below:



Thus, Effective Address is displacement relative to the address of instruction. Relative Addressing is often used with branch type instruction.

## 12. Index Register Addressing Mode:

In indexed addressing mode, the content of Index Register is added to direct address part (or field) of instruction to obtain the effective address. Means, in it, the register indirect addressing field of instruction point to Index Register, which is a special CPU register that contain an Indexed value, and direct addressing field contain base address.

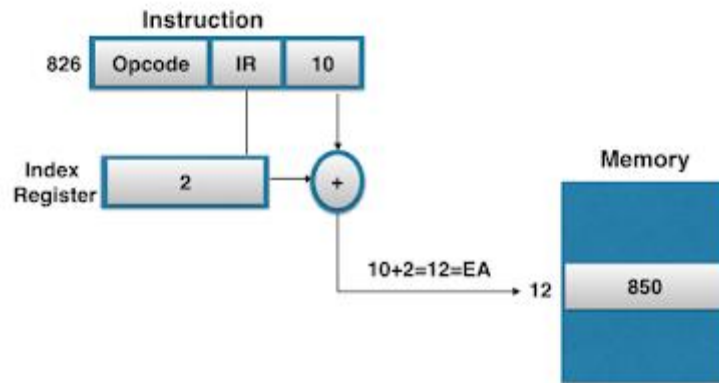
As, indexed type instruction make sense that data array is in memory and each operand in the array is stored in memory relative to base address. And the distance between the beginning address and the address of operand is the indexed value stored in indexed register.

Any operand in the array can be accessed with the same instruction, which provided that the index register contains the correct index value i.e., the index register can be incremented to facilitate access to consecutive operands.

Thus, in index addressing mode

$$EA = A + \text{Index}$$

It looks like as shown in fig. below:

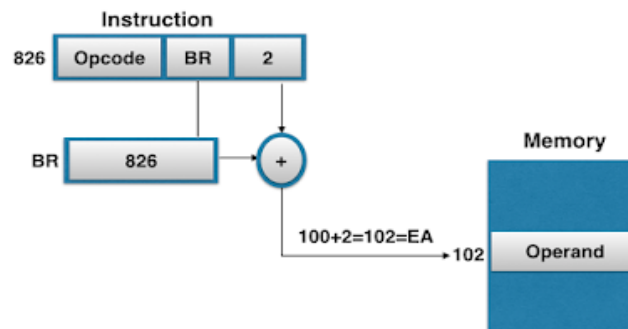


### 13. Base Register Addressing Mode:

In this mode, the content of the Base Register is added to the direct address part of the instruction to obtain the effective address. Means, in it the register indirect address field point to the Base Register and to obtain EA, the contents of Instruction Register, is added to direct address part of the instruction. This is similar to indexed addressing mode except that the register is now called as Base Register instead of Index Register.

That is, the  $EA = A + \text{Base}$

It looks like as shown in fig. below:



Thus, the difference between Base and Index mode is in the way they are used rather than the way they are computed. An Index Register is assumed to hold an index number that is relative to the address part of the instruction. And a Base Register is assumed to hold a base address and the direct address field of instruction gives a displacement relative to this base address. Thus, the Base register addressing mode is used in computer to facilitate the relocation of programs in memory.

### Machine Language and Assembly Language:

Machine Language	Assembly Language
Machine language is a low-level programming language made out of binary numbers or bits that can only be read by machines. It is also known as machine code or object code, in which instructions are executed directly by the CPU.	Assembly language is a human-only language that is not understood by computers. As a result, it acts as a link between high-level programming languages and machine languages, requiring the usage of an assembler to convert instructions into machine or object code.

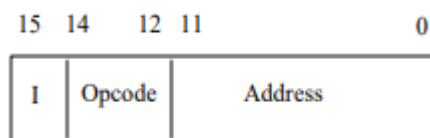
Machine language includes binary digits (0s and 1s), hexadecimal and octal decimal, which can be comprehended only by computers and cannot be deciphered by humans.	Mnemonics such as Mov, Add, Sub, End, and others make up the assembly language, which people can understand, utilise, and apply.
In machine language, error fixing and modifications cannot be done, and the features of machine languages are varied accordingly.	Assembly language has conventional instruction sets, as well as the ability to correct errors and modify programs.
Machine languages are platform-dependent and very difficult to understand by human beings.	The syntaxes of Assembly languages are similar to the English language; therefore, it is easy to understand by a human.
Machine language is not possible to learn as it is difficult to memorize and serves as a machine code only.	Assembly language is easy to memorize, and it is used for microprocessor-based applications/devices and real-time systems.
In machine language, all data is present in binary format that makes it fast in execution.	As compared to machine language, the execution speed of assembly language is slow.
The sequences of bits are used by Machine language to give commands. Zero represents the off or false state, while one represents the on or true state. It is dependent on the CPU to the conversion of high-level programming language to machine language.	Instead of using raw sequences of bits, assembly language uses "mnemonics" names and symbols; therefore, users do not need to remember op-codes with assembly language. In assembly languages, humans can map the code to machine code, and the codes are slightly more readable
The first-generation programming languages are Machine languages, which do not need a translator.	The second generation of programming languages is assembly languages, which use assembler as a translator to convert mnemonics into machine-understandable form.
Machine language is hardware-dependent and does not allow for modification.	Assembly language is not portable, and it is machine-dependent and can be modified easily.
In the syntax of machine language, there are more chances of errors.	As compared to machine language, there are fewer chances of syntax errors in assembly language.

# Instruction Code:

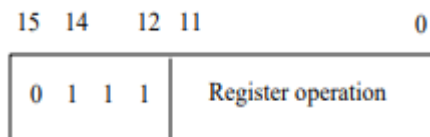
## Types of Instruction Code:

In general, there are three code formats for computer instructions. Of 16-bit code, the opcode usually contains 3 bits, and the rest of 13-bit is subjective, depending on the operation code encountered. Based on the instruction code format, the three types of instruction code are:

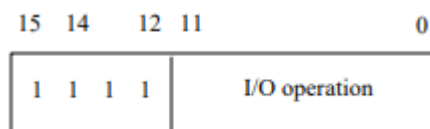
**1. Memory-reference instruction:** In this type of code, 12 bits are used to specify the memory address, 3 bits for the opcode (000 to 110), and 1 bit to specify the mode as indirect addressing mode (I).



**2. Register-reference instruction:** In this type of code, 12 bits indicate the register operation address, 3 bits for the opcode (111), and 1 bit is utilised for setting the mode as 0. The instructions are executed on the register.



**3. Input-output instruction:** This type of code contains a 12-bit input/output operation address, 3 bits for the opcode (111), and 1 bit is utilised for setting the mode as 1. These instructions are required to transfer to and from the AC register and output device.



## What is pipeline?

The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

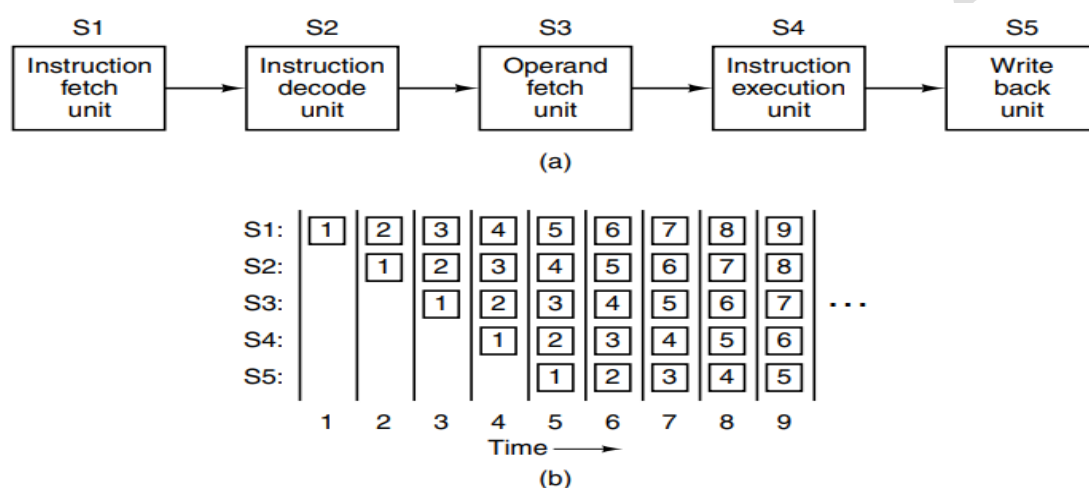
Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.

To fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a special set of registers called the prefetch buffer. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a pipeline carries this strategy much further. Instead of being divided into only two parts, instruction execution is often divided into many (often a dozen or more) parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Figure 2-4(a) illustrates a pipeline with five units, also called stages. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs.

Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path, finally, stage 5 writes the result back to the proper register.



**Figure 2-4.** (a) A five-stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2-4(b) we see how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction 2, and stage S1 fetches the third instruction. During cycle 4, stage S4 executes instruction 1, S3 fetches the operands for instruction 2, S2 decodes instruction 3, and S1 fetches instruction 4. Finally, in cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

## Pipeline Hazards:

**Pipeline Hazards** In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in a more systematic way. Chapter 14 revisits this issue, in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a pipeline bubble. There are three types of hazards: resource, data, and control.

### RESOURCE HAZARDS:

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than

parallel for a portion of the pipeline. A resource hazard is sometime referred to as a structural hazard.

Let us consider a simple example of a resource hazard. Assume a simplified five stage pipeline, in which each stage takes one clock cycle. Figure 12.15a shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 12.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			FI	DI	FO	EI	WO		
	I4				FI	DI	FO	EI	WO	

(a) Five-stage pipeline, ideal case

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			Idle	FI	DI	FO	EI	WO	
	I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

Figure 12.15 Example of Resource Hazard

## DATA HAZARDS:

A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce

a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining. As an example, consider the following x86 machine instruction sequence:

ADD EAX, EBX /\*  $EAX = EAX + EBX$

SUB ECX, EAX /\*  $ECX = ECX - EAX$

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. Figure 12.16 shows the pipeline behavior. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clock cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage. There are three types of data hazards,

- **Read after write (RAW)**, or true dependency: An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (RAW)**, or antidependency: An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (RAW)**, or output dependency: Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

### CONTROL HAZARDS:

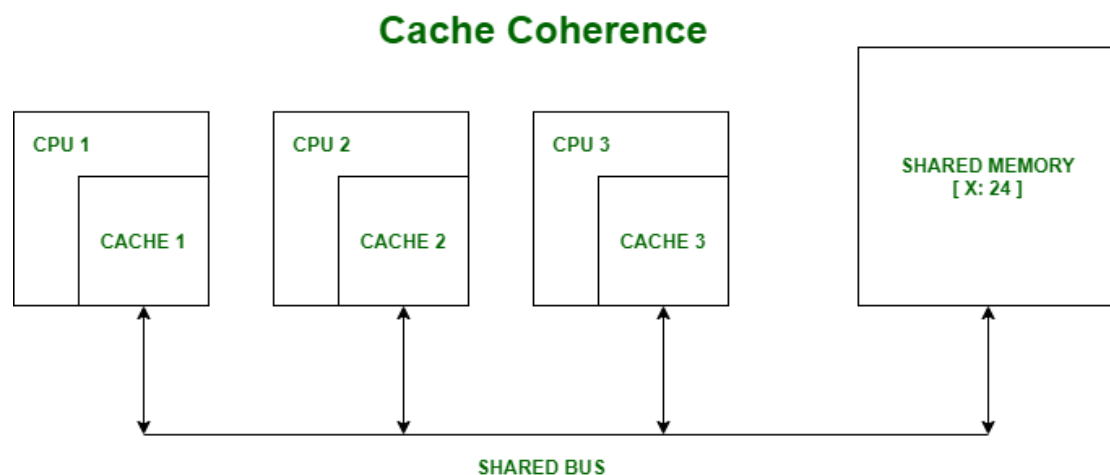
A control hazard, also known as a branch hazard, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.



## Cache Coherence:

In a multiprocessor system, data inconsistency may occur among adjacent levels or within the same level of the memory hierarchy. In a shared memory multiprocessor with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also.

**Example:** Cache and the main memory may have inconsistent copies of the same object.



Suppose there are three processors, each having cache. Suppose the following scenario:-

- **Processor 1 read X:** obtains 24 from the memory and caches it.
- **Processor 2 read X:** obtains 24 from memory and caches it.
- **Again, processor 1 writes as X: 64,** its locally cached copy is updated. Now, processor 3 reads X, what value should it get?
- Memory and processor 2 thinks it is 24 and processor 1 thinks it is 64.

As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates a cache coherence problem. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion. There are three distinct level of cache coherence.

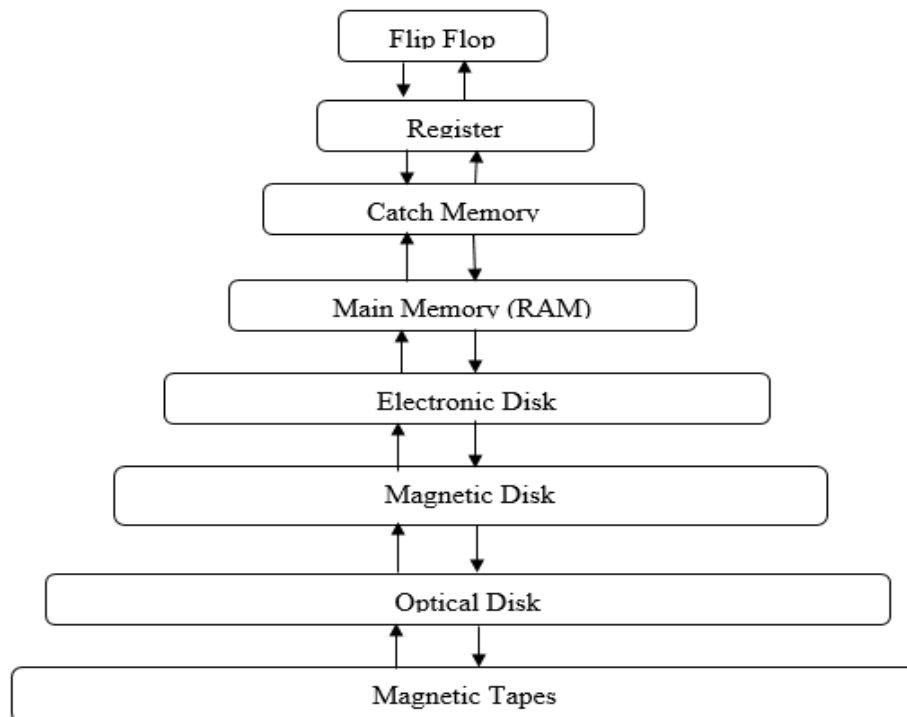
## Memory Management System:

Memory is the important part of the computer that is used to store the data. Its management is critical to the computer system because the amount of main memory available in a computer system is very limited. At any time, many processes are competing for it. Moreover, to increase performance, several processes are executed simultaneously. For this, we must keep several processes in the **main memory**, so it is even more important to manage them effectively. Here is the hierarchy of the computer memories.

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of

either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

**Memory Hierarchy:** Computer memory are arranged according to lowest to highest, as flip flop store (one bit only) lowest memory and optical disk and Magnetic tapes store highest memory in the system.



**Figure:** Memory Hierarchy

## ❖ Page Replacement Algorithms

1. **Optimal Page Replacement algorithm** → this algorithm replaces the page which will not be referred for so long in future. Although it cannot be practically implementable but it can be used as a benchmark. Other algorithms are compared to this in terms of optimality.
2. **Least recent used (LRU) page replacement algorithm** → this algorithm replaces the page which has not been referred for a long time. This algorithm is just opposite to the optimal page replacement algorithm. In this, we look at the past instead of staring at future.
3. **FIFO** → in this algorithm, a queue is maintained. The page which is assigned the frame first will be replaced first. In other words, the page which resides at the rare end of the queue will be replaced on the every page fault.