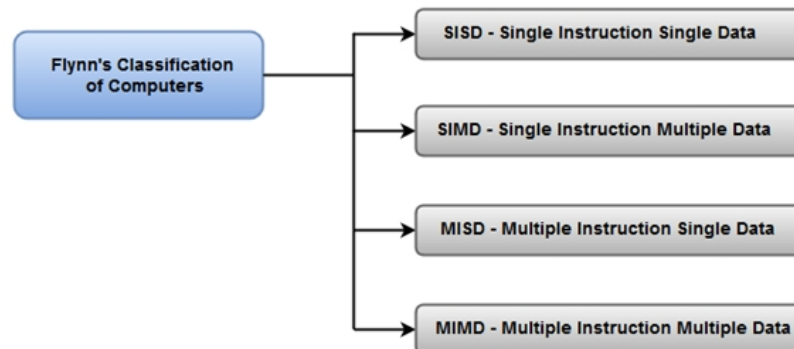
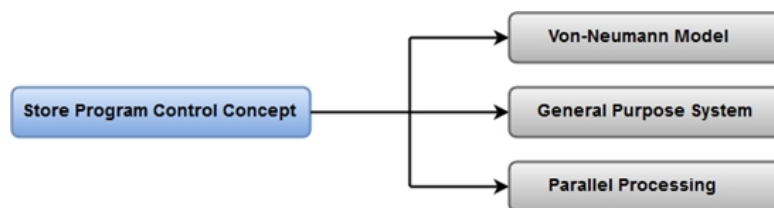


## General System Architecture

In Computer Architecture, the General System Architecture is divided into two major classification units.

1. Stored Program Control Concepts
2. Flynn's Classification of Computers



### **1. Store Program Control Concept**

The term Stored Program Control Concept refers to the storage of instructions in computer memory to enable it to perform a variety of tasks in sequence or not in sequence.

Stored program concept means that data and instructions are both logically the same and can both be stored in memory. Before introduction of this idea, instructions and data were considered two totally different entities and were thus stored separately. Von-Neumann architecture is built around this principle. Instructions can be stored in memory and executed in sequence referencing the data values on which it needs to operate.

The idea behind stored-program concept was to design a computer that includes an instruction set architecture and can store in memory, then a set of instructions (a program) that details the computation. A stored-program design also allows programs to modify themselves while running.

ENIAC (Electronic Numerical Integrator and Computer) was the first computing system designed in the early 1940s. It was based on Stored Program Concept in which machine use memory for processing data.

Stored Program Concept can be further classified in three basic ways:

1. Von-Neumann Model
2. General Purpose System
3. Parallel Processing

### ➤ **Von-Neumann Model**

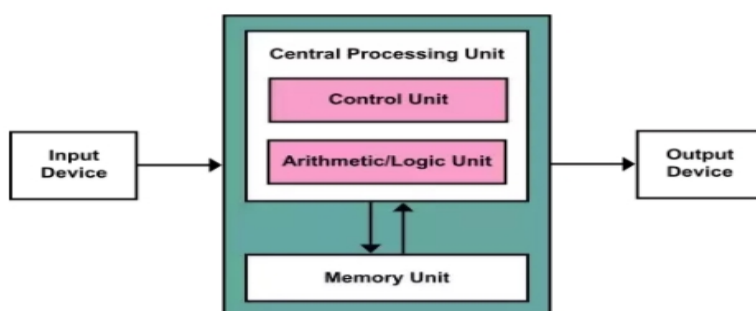
The Von-Neumann architecture, is a computer architecture that was described in 1945 by the mathematician and physicist John Von-Neumann. This describes a design architecture for an electronic digital computer with different parts.

There are essentially three different entities, a processing unit, an i/o unit and a storage unit. The units are connected over buses.

The processing unit can be broken down into a couple of subunits, the ALU, the processing control unit and the program counter. The ALU compute the arithmetic logic needed to run programs (adding and subtracting to registers etc.) To be simple, the control unit just controls the flow of data through the processor. The program essentially points to when you are in instruction memory, it keeps track of what instruction you are running at the moment and increments when done.

The i/o unit essentially encompasses all i/o that the computer could possibly do (printing to a monitor, to paper, inputs from a mouse or keyboard, etc.)

The storage unit stores anything the computer would need to store and retrieve. This includes local hard drive storage, cache storage, and ram. It is also called stored program computer. Here, both the instructions and the data (that the instructions operate on) are stored in the computer memory itself. Thus instructions like data can be read from the memory and written to the memory by the processor.



# Computer System

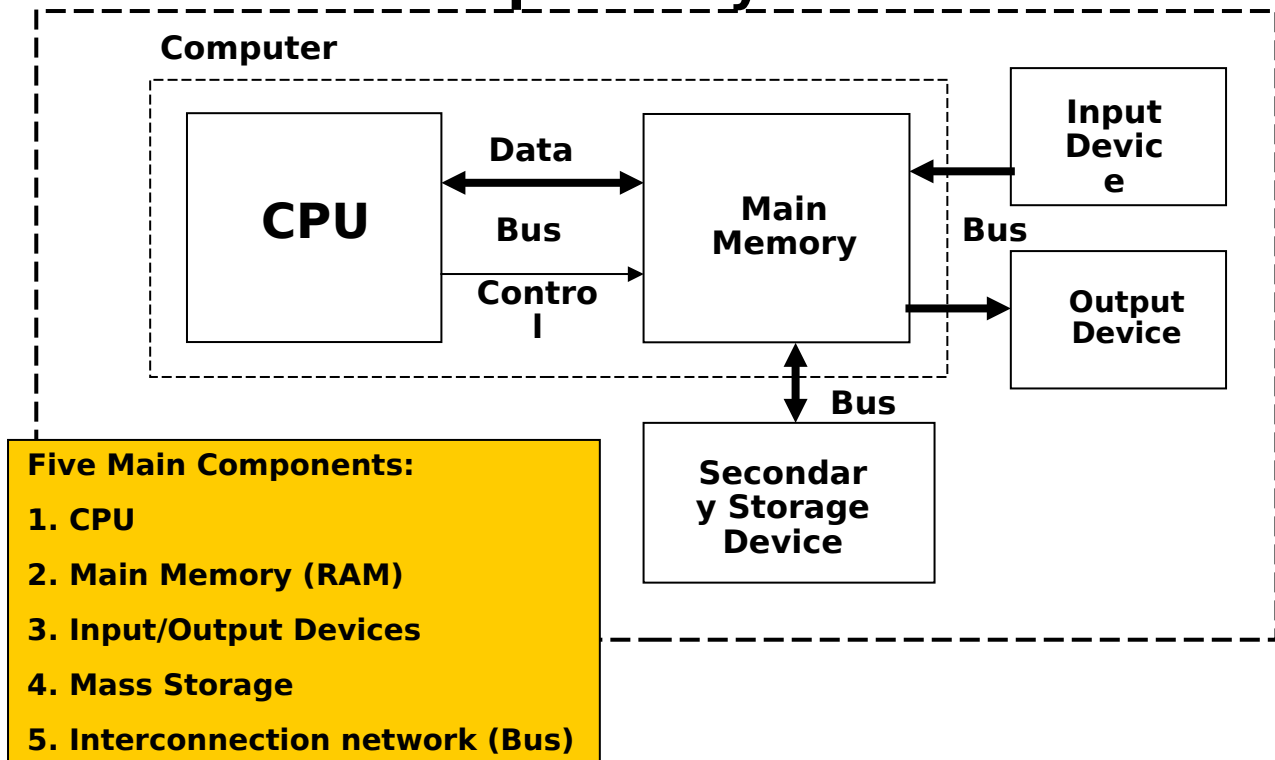


Fig: 5 component design of the *stored program digital computer*

The CPU does not distinguish in the way it deals with data or program. The read/write operations that are valid for data are valid for programming instructions as well.

Simply put the CPU addresses the memory, fetches the corresponding instruction from the memory, executes them and as per requirement addresses and then reads the data from the memory as well.

The Von Neumann Computer was the first computer to be based on the stored program concept. The architecture on which computers now are based is thus called the Von Neumann Architecture.

Von Neumann had proposed few registers in his model. They are discussed briefly below:

## Registers

Registers are high speed storage areas in the CPU. All data must be stored in a register before it can be processed.

<b>MAR</b>	Memory Address Register	Holds the memory location of data that needs to be accessed
<b>MDR</b>	Memory Data Register	Holds data that is being transferred to or from memory
<b>AC</b>	Accumulator	Where intermediate arithmetic and logic results are stored
<b>PC</b>	Program Counter	Contains the address of the next instruction to be executed
<b>CIR</b>	Current Instruction Register	Contains the current instruction during processing

## **Buses**

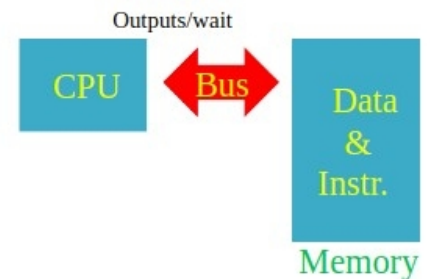
Buses are the means by which data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory.

A standard CPU system bus is comprised of a control bus, data bus and address bus.

<b>Address Bus</b>	Carries the addresses of data (but not the data) between the processor and memory
<b>Data Bus</b>	Carries data between the processor, the memory unit and the input/output devices
<b>Control Bus</b>	Carries control signals/commands from the CPU (and status signals from other devices) in order to control and coordinate all the activities within the computer

The main drawback:

CPU is unable to access program memory and data memory simultaneously. This case is called the "bottleneck" that affects system performance.



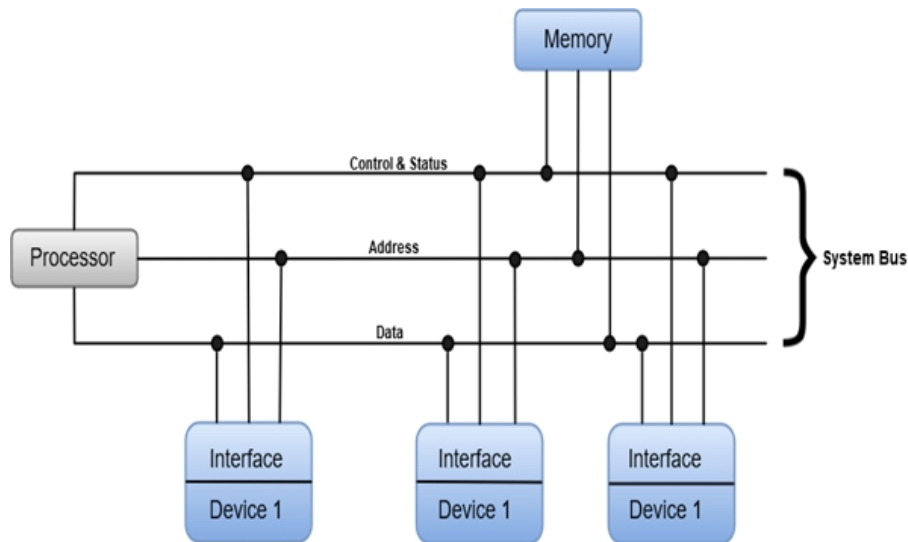
### **➤ General Purpose System**

The General Purpose Computer System is the modified version of the Von-Neumann Architecture. In simple words, we can say that a general purpose computer system is a modern day architectural representation of Computer System.

The CPU (Central Processing Unit) consists of the ALU (Arithmetic and Logic Unit), Control Unit and various processor registers.

The CPU, Memory Unit and I/O subsystems are interconnected by the system bus which includes data, address, and control-status lines.

The following image shows how CPU, Memory Unit and I/O subsystems are connected through common single bus architecture.



However, practical systems may differ from the single bus architecture in the sense that it may be configured around multiple buses.

Multiple Bus Architecture favors high throughput as compared to Single Bus Architecture.

### ➤ **Parallel processing**

Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

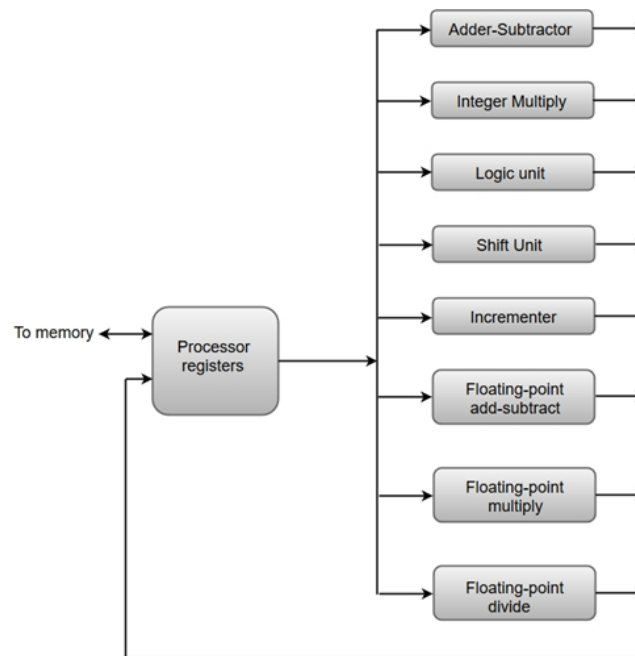
A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

The primary purpose of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.

A parallel processing system can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. The data can be distributed among various multiple functional units.

The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel.

The operation performed in each functional unit is indicated in each block if the diagram:



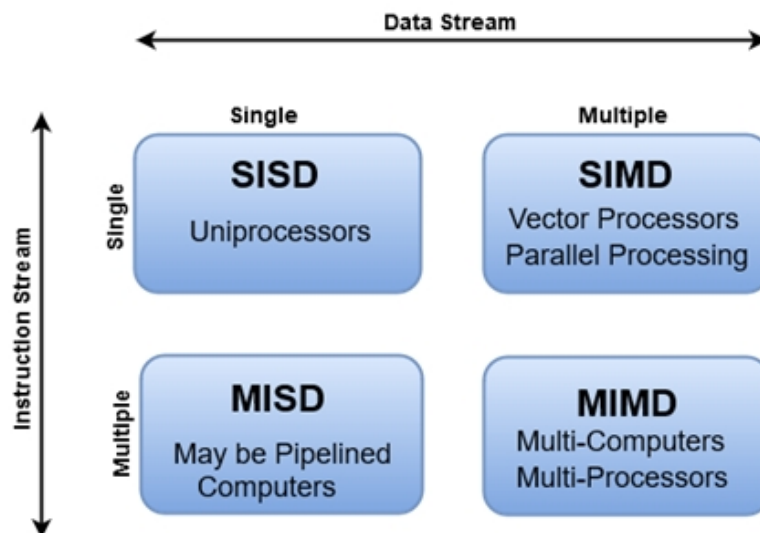
- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.

## 2. Flynn's Classification of Computers

- Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- The sequence of instructions read from memory forms an **instruction stream**.
- The operations performed on the data in the processor forms a **data stream**.
- Parallel processing may occur in the instruction stream, in the data stream, or both.
- Flynn's classification divides computers into four major groups that are:
  - Single Instruction stream, Single Data stream (SISD)
  - Single Instruction stream, Multiple Data stream (SIMD)

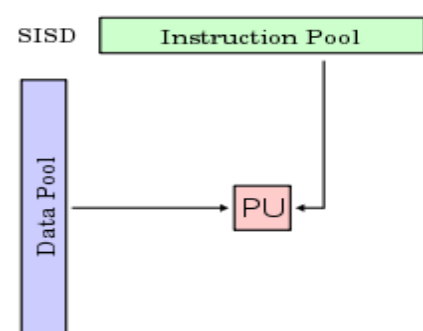
- Multiple Instruction stream, Single Data stream (MISD)
- Multiple Instruction stream, Multiple Data stream (MIMD)

### Flynn's Classification of Computers



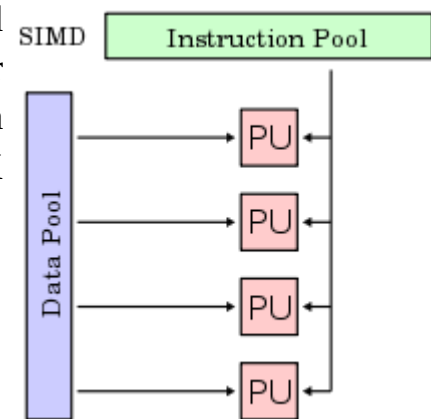
### ➤ SISD (Single Instruction, Single Data stream)

- SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- SISD refers to an Instruction Set Architecture in which a single processor (one CPU) executes exactly one instruction stream at a time and also fetches or stores one item of data at a time to operate on data stored in a single memory unit.
- Most conventional computers have SISD architecture like the traditional Von-Neumann computers.
- The SISD model is a typical non-pipelined architecture with the general-purpose registers, as well as dedicated special registers such as the Program Counter (PC), the Instruction Register (IR), Memory Address Registers (MAR) and Memory Data Registers (MDR).
- Data Stream flows between the processors and memory bi-directionally.
- Example: Older generation computers, minicomputers, and workstations.



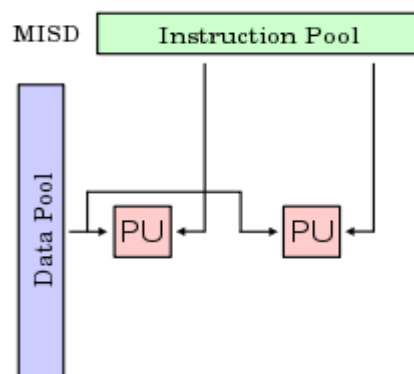
## ➤ SIMD (Single Instruction, Multiple Data streams)

- SIMD represents an organization that includes many processing units under the supervision of a common control unit.
- SIMD is an Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU) that operates like a Von Neumann machine by executing a single instruction stream over PUs, handled through the CU.
- The CU generates the control signals for all of the PUs and by which executes the same operation on different data streams. The SIMD architecture, in effect, is capable of achieving data level parallelism just like with vector processor.
- Some of the examples of the SIMD based systems include IBM's AltiVec and SPE for PowerPC, HP's PA-RISC Multimedia Acceleration eXtensions (MAX), Intel's MMX and iwMMXt Etc.



## ➤ MISD (Multiple Instruction, Single Data stream)

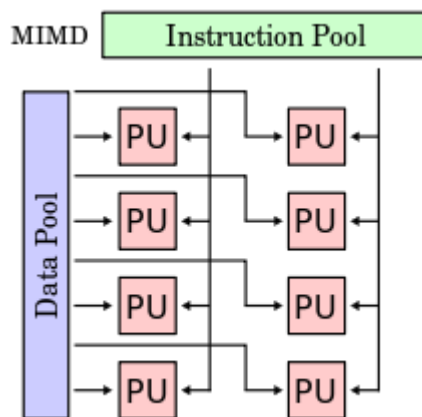
- MISD is an Instruction Set Architecture for parallel computing where many functional units perform different operations by executing different instructions on the same data set.
- This type of architecture is common mainly in the fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors.





## ➤ MIMD (Multiple Instruction, Multiple Data streams)

- Multiple Instruction stream, Multiple Data stream (MIMD) is an Instruction Set Architecture for parallel computing that is typical of the computers with multiprocessors. Using the MIMD, each processor in a multiprocessor system can execute different set of the instructions independently on the different set of data units.
- In MIMD, each processor has a separate program and an instruction stream is generated from each program.
- MIMD based computer systems can use the shared memory in a memory pool or work using distributed memory across heterogeneous network computers in a distributed environment.
- The MIMD architectures is primarily used in a number of application areas such as computer- aided design/ computer-aided manufacturing, simulation, modeling, communication switches etc.



Examples:

Cray T90, Cray T3E, IBM-SP2

## Digital Design


### Logic Gates

Logic gates are the basic building blocks of any digital system. It is an electronic circuit having one or more than one input and only one output. The relationship between the input and the output is based on a **certain logic**. Based on this, logic gates are named as AND gate, OR gate, NOT gate etc.

### AND Gate:

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

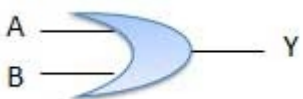
$$\begin{aligned} Y &= A \text{ AND } B \text{ AND } C \dots\dots N \\ Y &= A.B.C \dots\dots N \\ Y &= ABC \dots\dots N \end{aligned}$$

Logic Diagram	Truth Table																		
	<table><tr><th colspan="2">Inputs</th><th>Output</th></tr><tr><th>A</th><th>B</th><th>AB</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	Inputs		Output	A	B	AB	0	0	0	0	1	0	1	0	0	1	1	1
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.

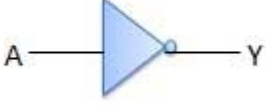
$$\begin{aligned} Y &= A \text{ OR } B \text{ OR } C \dots\dots N \\ Y &= A + B + C \dots\dots N \end{aligned}$$

Logic Diagram	Truth Table																		
	<table><tr><th colspan="2">Inputs</th><th>Output</th></tr><tr><th>A</th><th>B</th><th>A + B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	Inputs		Output	A	B	A + B	0	0	0	0	1	1	1	0	1	1	1	1
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.



$$\begin{aligned} Y &= \text{NOT } A \\ Y &= \overline{A} \end{aligned}$$

Logic Diagram	Truth Table								
	<table><tr><th>Inputs</th><th>Output</th></tr><tr><td>A</td><td>B</td></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	Inputs	Output	A	B	0	1	1	0
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.

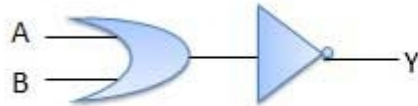
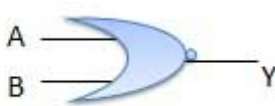
$$\begin{aligned} Y &= A \text{ NOT AND } B \text{ NOT AND } C \dots\dots N \\ Y &= A \text{ NAND } B \text{ NAND } C \dots\dots N \end{aligned}$$

Logic Diagram	Truth Table																		
 	<table><tr><th colspan="2">Inputs</th><th>Output</th></tr><tr><th>A</th><th>B</th><th><math>\overline{AB}</math></th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	Inputs		Output	A	B	$\overline{AB}$	0	0	1	0	1	1	1	0	1	1	1	0
Inputs		Output																	
A	B	$\overline{AB}$																	
0	0	1																	
0	1	1																	
1	0	1																	
1	1	0																	

## NOR Gate:

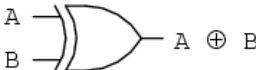
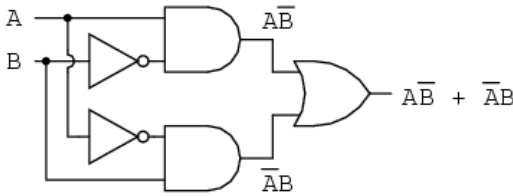
A NOT-OR operation is known as NOR operation. It has n input ( $n \geq 2$ ) and one output.

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

Logic Diagram	Truth Table																		
 	<table><tr><th colspan="2">Inputs</th><th>Output</th></tr><tr><th>A</th><th>B</th><th><math>\overline{A+B}</math></th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	Inputs		Output	A	B	$\overline{A+B}$	0	0	1	0	1	0	1	0	0	1	1	0
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.

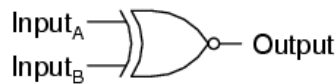
Logic Diagram	Truth Table																		
<div><p><math>A \oplus B</math></p><p>... is equivalent to ...</p><p><math>A \oplus B = \overline{A}B + A\overline{B}</math></p></div>	<table><tr><th colspan="2">Inputs</th><th>Output</th></tr><tr><th>A</th><th>B</th><th><math>A \oplus B</math></th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	Inputs		Output	A	B	$A \oplus B$	0	0	0	0	1	1	1	0	1	1	1	0
Inputs		Output																	
A	B	$A \oplus B$																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	0																	

### XNOR Gate:

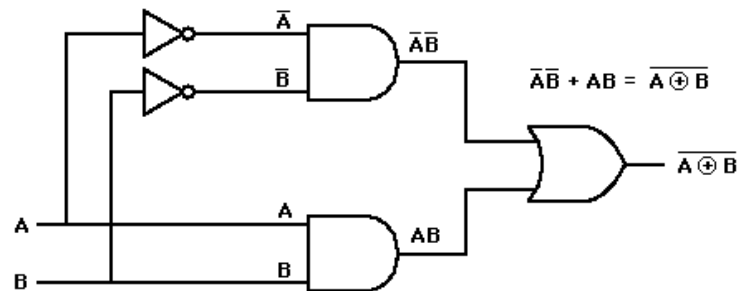
XNOR gate is a special type of gate. It can be used in the half adder, full adder and subtractor. The exclusive-NOR gate is abbreviated as EX-NOR gate or sometime as X-NOR gate. It has n input ( $n \geq 2$ ) and one output.

## Logic Diagram and Truth table

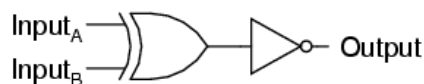
*Exclusive-NOR gate*



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



*Equivalent gate circuit*



## Combinational circuits

Combinational circuit is a circuit in which we combine the different gates in the circuit. Some of the characteristics of combinational circuits are following –

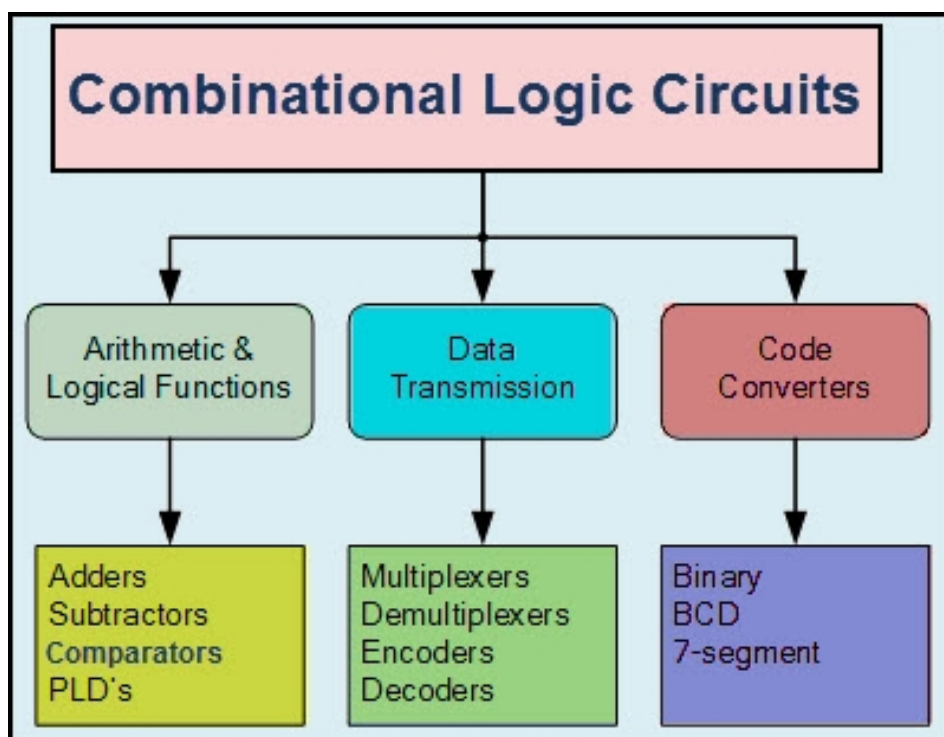
- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.
- **Examples** of common **combinational** logic **circuits** include: half adders, full adders, multiplexers, demultiplexers, encoders and decoders.



To design a combinational logic circuit use the following procedures:

1. The problem is stated (Verbal description).
2. Specify the number of inputs and required numbers of outputs.
3. The input and output variables are assigned letter symbols.
4. Construct the truth table to define relationship between inputs and outputs.
5. The simplified Boolean function for each output is obtained (using K-Map, Tabulation method and Boolean Algebra rules).
6. The logic diagram is drawn.

### Classification of Combinational Logic:



### Adder:

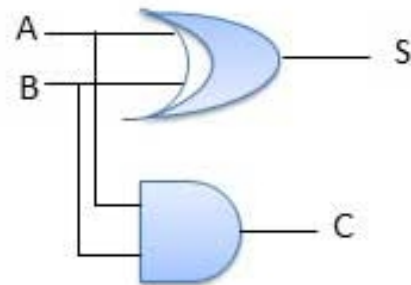
An **adder** is a digital circuit that performs addition of numbers. In many computers and other kinds of processors adders are used in the arithmetic and logic units (**ALU**).

## Half Adder

Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two **single** bit numbers. This circuit has two outputs **carry** and **sum**. The common representation uses a XOR logic gate and an AND logic gate.



Block Diagram

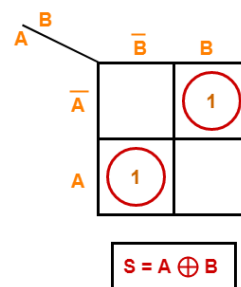


Circuit Diagram

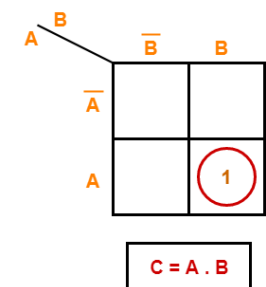
Inputs		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth table

For S:



For C:



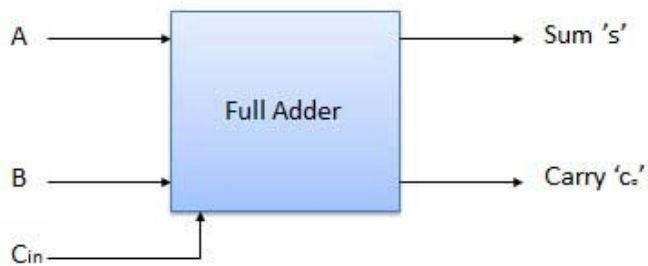
K Maps

$$S = \bar{A}B + A\bar{B} = A \oplus B$$
$$C = AB$$

## Full Adder:

Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry C. The full adder is a three input and two output combinational circuit.

## Block diagram



For S:

	$BC_{in}$	$\overline{B}\overline{C}_{in}$	$\overline{B}C_{in}$	$BC_{in}$	$B\overline{C}_{in}$
A					
$\overline{A}$		1			1
A	1		1		

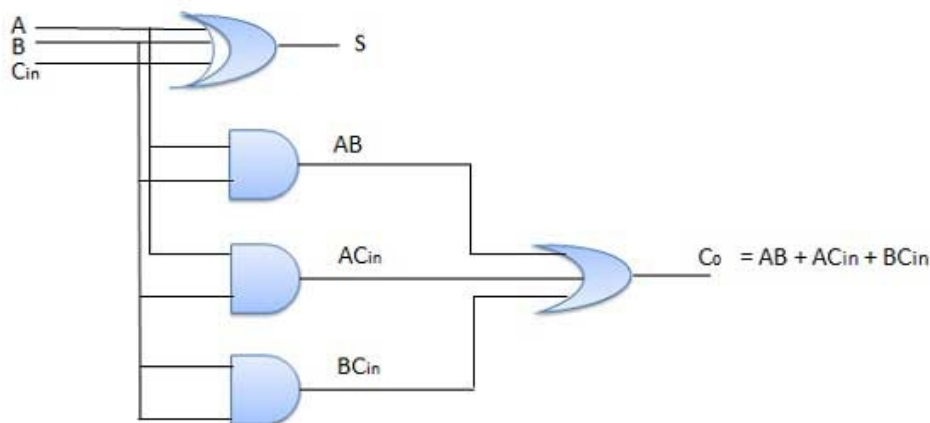
$$S = A \oplus B \oplus C_{in}$$

Inputs			Output	
A	B	C <sub>in</sub>	S	Co
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

For C<sub>in</sub>:

	$BC_{in}$	$\overline{B}\overline{C}_{in}$	$\overline{B}C_{in}$	$BC_{in}$	$B\overline{C}_{in}$
A					
$\overline{A}$			1		
A		1		1	1

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



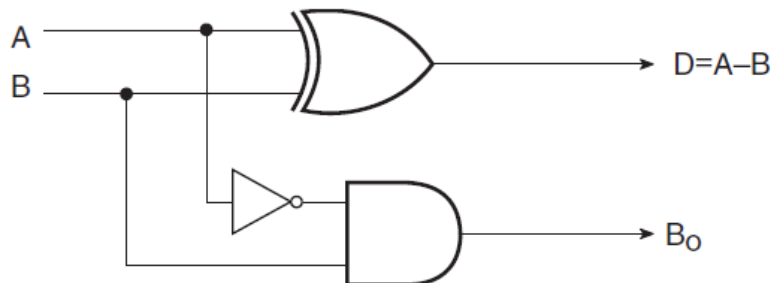
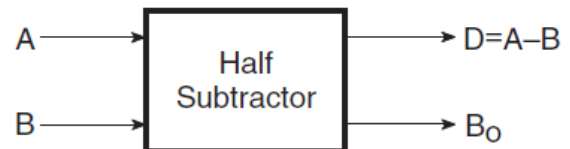
## Half Subtractor

Half subtractor is a combination circuit with two inputs and two outputs (difference and borrow). It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called as Minuend bit and B is called as Subtrahend bit.



## Truth Table, Block Diagram and Circuit Diagram

A	B	D	B <sub>0</sub>
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



## K-map

### For Difference

A \ B	0	1
0	0	1
1	1	0

$$\begin{aligned}\text{Difference} &= A\bar{B} + \bar{A}B \\ &= A \oplus B\end{aligned}$$

### For Borrow

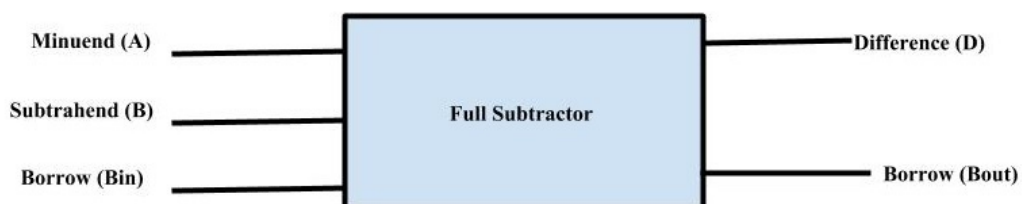
A \ B	0	1
0	0	1
1	0	0

$$\text{Borrow} = \bar{A}B$$

## Full Subtractor

The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A, B, B<sub>in</sub> and two outputs D and B<sub>out</sub>.

## Block Diagram

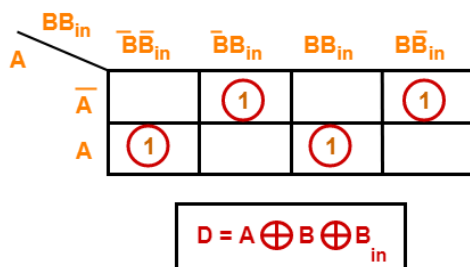


**Truth Table**

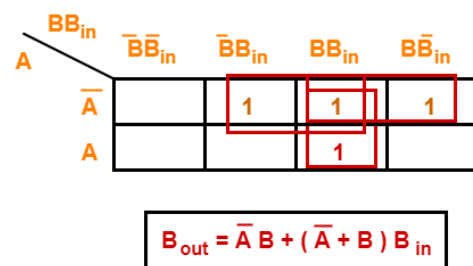
INPUT			OUTPUT	
A	B	B <sub>in</sub>	D	B <sub>out</sub>
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

## K-Map

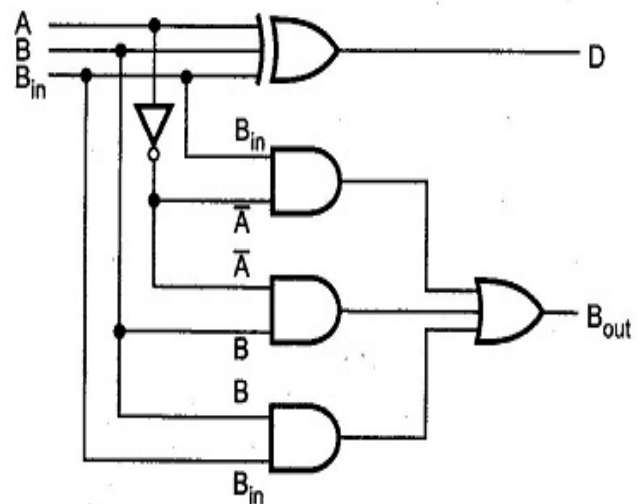
For D:



For B<sub>in</sub>:



## Circuit Diagram



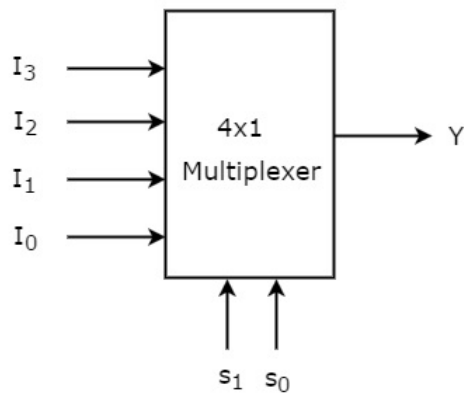
## Multiplexer

**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<sub>3</sub>, I<sub>2</sub>, I<sub>1</sub> & I<sub>0</sub>, two selection lines s<sub>1</sub> & s<sub>0</sub> 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.

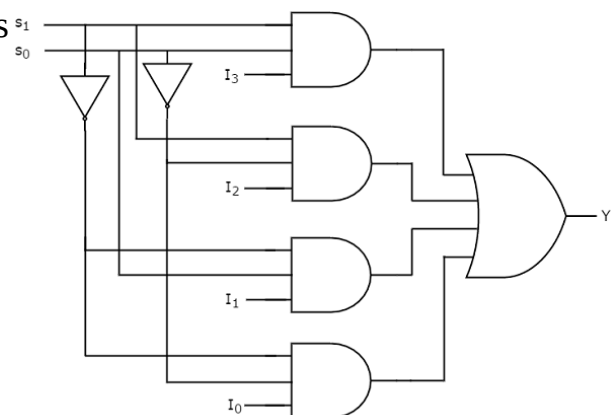
Selection Lines		Output
S <sub>1</sub>	S <sub>0</sub>	Y
0	0	I <sub>0</sub>
0	1	I <sub>1</sub>
1	0	I <sub>2</sub>
1	1	I <sub>3</sub>

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.



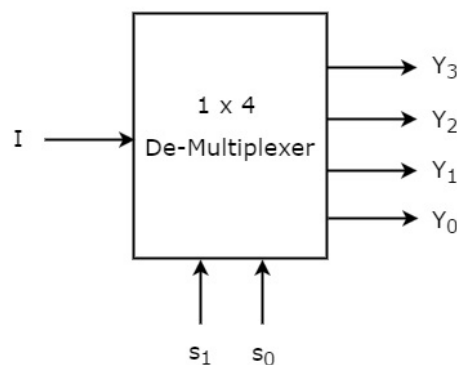
## De-multiplexer

**De-Multiplexer** is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of  $2^n$  outputs. The input will be connected to one of these outputs 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 can select only one output. De-Multiplexer is also called as **De-Mux or Data distributor**.

### 1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s1 & s0 and four outputs Y3, Y2, Y1 & Y0. The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



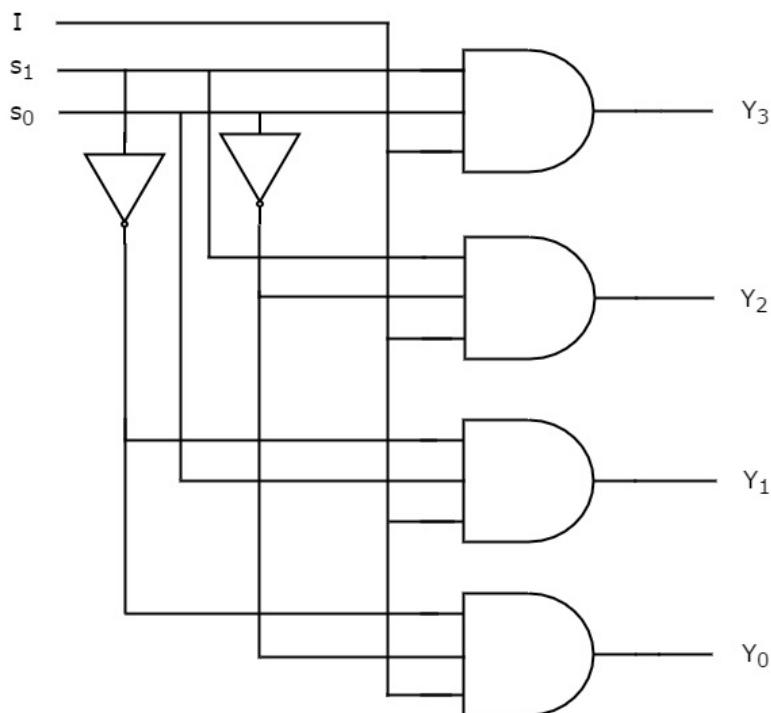
The single input 'I' will be connected to one of the four outputs, Y3 to Y0 based on the values of selection lines s1 & s0. The **Truth table** of 1x4 De-Multiplexer is shown below.

Selection Inputs		Outputs			
S1	S0	Y3	Y2	Y1	Y0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

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

$$Y3 = s_1 s_0 I \quad Y2 = s_1 s_0' I \quad Y1 = s_1' s_0 I \quad Y0 = s_1' s_0' I$$

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure:



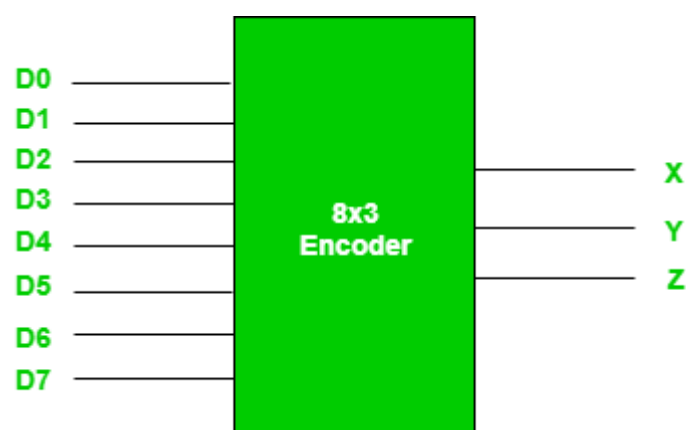
From above diagram, we can understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

### Encoder and Decoder in digital logic:

Binary code of  $N$  digits can be used to store  $2^N$  distinct elements of coded information. This is what encoders and decoders are used for. **Encoders** convert  $2^N$  lines of input into a code of  $N$  bits and **Decoders** decode the  $N$  bits into  $2^N$  lines.

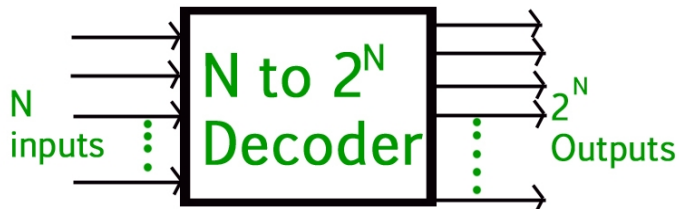
#### Encoder:

An encoder is a combinational circuit that converts binary information in the form of a  $2^N$  input lines into  $N$  output lines, which represent  $N$  bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.



## Decoder:

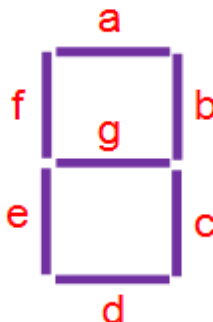
In Digital Electronics, discrete quantities of information are represented by binary codes. A binary code of  **$n$  bits** is capable of representing up to  **$2^n$  distinct elements** of coded information. The name “**Decoder**” means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output. A **decoder** is a **combinational circuit** that converts binary information from  **$n$  input lines** to a maximum of  **$2^n$  unique output lines**.



## Binary to BCD converter:

Binary Code	Decimal Number	BCD Code
A B C D		B <sub>5</sub> B <sub>4</sub> B <sub>3</sub> B <sub>2</sub> B <sub>1</sub>
0 0 0 0	0	0 0 0 0 0
0 0 0 1	1	0 0 0 0 1
0 0 1 0	2	0 0 0 1 0
0 0 1 1	3	0 0 0 1 1
0 1 0 0	4	0 0 1 0 0
0 1 0 1	5	0 0 1 0 1
0 1 1 0	6	0 0 1 1 0
0 1 1 1	7	0 0 1 1 1
1 0 0 0	8	0 1 0 0 0
1 0 0 1	9	0 1 0 0 1
1 0 1 0	10	1 0 0 0 0
1 0 1 1	11	1 0 0 0 1
1 1 0 0	12	1 0 0 1 0
1 1 0 1	13	1 0 0 1 1
1 1 1 0	14	1 0 1 0 0
1 1 1 1	15	1 0 1 0 1

## BCD To 7-Segment Decoder



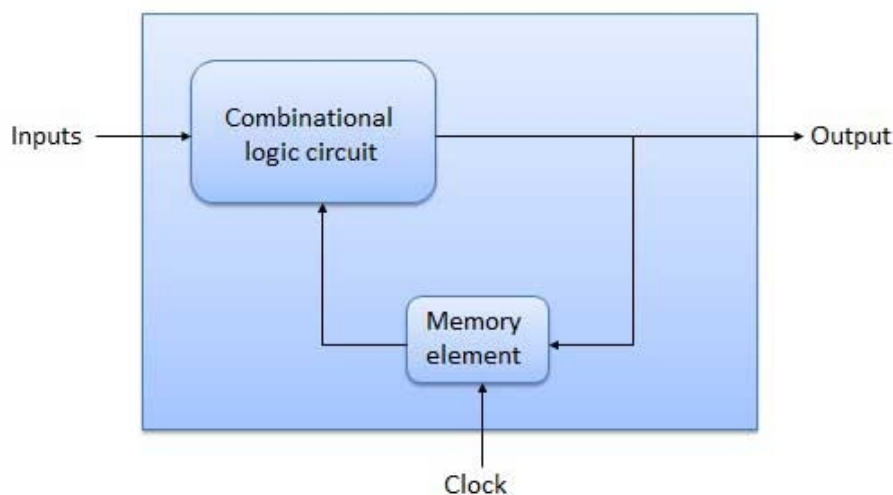
## Truth Table for 7 Segment Decoder

Decimal Digit	Input lines				Output lines							Display pattern
	A	B	C	D	a	b	c	d	e	f	g	
0	0	0	0	0	1	1	1	1	1	1	0	0
1	0	0	0	1	0	1	1	0	0	0	0	1
2	0	0	1	0	1	1	0	1	1	0	1	2
3	0	0	1	1	1	1	1	1	0	0	1	3
4	0	1	0	0	0	1	1	0	0	1	1	4
5	0	1	0	1	1	0	1	1	0	1	1	5
6	0	1	1	0	1	0	1	1	1	1	1	6
7	0	1	1	1	1	1	1	0	0	0	0	7
8	1	0	0	0	1	1	1	1	1	1	1	8
9	1	0	0	1	1	1	1	1	0	1	1	9

### Sequential Circuits

Sequential Circuit is a system whose output depends on present input and also on past output which is stored in a memory element. The combinational circuit does not use any memory. Hence the previous state of input does not have any effect on the present state of the circuit. But sequential circuit has memory so output can vary based on input. This type of circuits uses previous input, output, clock and a memory element. Examples: cross coupled inverters, the shift registers and the counters are the common examples.

### **Block diagram**



## Types of sequential circuits

**Asynchronous sequential circuit** – These circuit **do not use a clock signal** but uses the pulses of the inputs. These circuits are **faster** than synchronous sequential circuits because there is in change state immediately when there is a change in the input signal. We use asynchronous sequential circuits when speed of operation is important and **independent** of internal clock pulse. But these circuits are more **difficult** to design and their output is **uncertain**.



Figure: Asynchronous Sequential Circuit

**Synchronous sequential circuit** – These circuit **uses clock signal** and input levels. The output pulse is the same duration as the clock pulse for the clocked sequential circuits. Since they wait for the next clock pulse to arrive to perform the next operation, so these circuits are bit **slower** compared to asynchronous. Level output changes state at the start of an input pulse and remains in that until the next input or clock pulse. We use synchronous sequential circuit in synchronous counters, flip flops, and in the design of MOORE-MEALY state management machines.

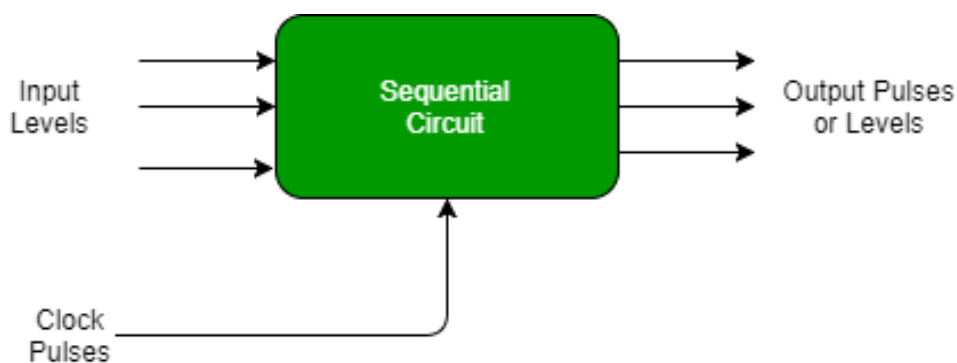


Figure: Synchronous Sequential Circuit

## Latch and Flip flops:

- Latch and flip flops are the building blocks of the sequential circuits. Latches and flip-flops are the basic elements for storing information. One latch or flip-flop can store one bit of information. The main difference between latches and flip-flops is that for latches, their outputs are constantly affected by their inputs as long as the enable signal is asserted. In other words, when they are enabled, their content changes immediately when their inputs change. Flip-flops, on the



other hand, have their content change only either at the rising or falling edge of the enable signal. This enable signal is usually the controlling clock signal.

- After the rising or falling edge of the clock, the flip-flop content remains constant even if the input changes.
- There are basically four main types of latches and flip-flops: SR, D, JK, and T. The major differences in these flip-flop types are the number of inputs they have and how they change state. For each type, there are also different variations that enhance their operations.

Latches	Flip Flops
Latches are building blocks of sequential circuits and these can be built from logic gates	Flip flops are also building blocks of sequential circuits. But, these can be built from the latches.
Latch continuously checks its inputs and changes its output correspondingly.	Flip flop continuously checks its inputs and changes its output correspondingly only at times determined by clocking signal
The latch is sensitive to the duration of the pulse and can send or receive the data when the switch is on	Flipflop is sensitive to a signal change. They can transfer data only at the single instant and data cannot be changed until next signal change. Flip flops are used as a register.
It is based on the enable function input	It works on the basis of clock pulses
It is a level triggered, it means that the output of the present state and input of the next state depends on the level that is binary input 1 or 0.	It is an edge triggered, it means that the output and the next state input changes when there is a change in clock pulse whether it may a +ve or -ve clock pulse.

## **K-Map**

- K-Map stands for Karnaugh Map. Maurice Karnaugh introduced it in 1953.
- It is the graphical representation of given expression(Used for simplification of Boolean Algebra).We can construct K-Map using 2 variable,3 variable, 4 variable and so on.
- The k-map is a two dimensional representation of a truth table it provides a simpler method for minimizing logic expressions. A karnaugh map is a diagram consisting of squares. Each square of the map represent either min term or max term. Any logic expression can be written as either sum of product

term or product of sum term which is also called sum of min term and product of max term respectively. Therefore a logic expression can be easily represented on a karnaugh map.

- A karnaugh map for n variables made up of  $2^n$  squares.

➤ **2 variable k-map:** – In the 2 variable k-map, four squares are constructed. Each square contains one term of expression with two variables. **A 2 variable k-map for SOP and POS form are as follows: –**

**A. SOP: -**

		B	
		$\bar{B}$ 0	B 1
A	$\bar{A}$ 0	$\bar{A}.\bar{B}$	$\bar{A}.B$
	A 1	$A.\bar{B}$	$A.B$

**B. POS: -**

		B	
		0	$\bar{B}$ 1
A	0	$A+B$	$A+\bar{B}$
	$\bar{A}$ 1	$\bar{A}+B$	$\bar{A}+\bar{B}$

➤ **3 variable k-map:** – In the three variable k-map, 8 square is required. The 3-variables k-map can content either horizontally or vertically. **An example of 3 variables k-map for SOP and POS form are as follows: –**

**A. SOP: -**

		BC			
		$\bar{B}\bar{C}$ 00	$\bar{B}C$ 01	$BC$ 11	$B\bar{C}$ 10
A	$\bar{A}$ 0	$\bar{A}\bar{B}\bar{C}$ 0	$\bar{A}\bar{B}C$ 1	$\bar{A}BC$ 3	$\bar{A}B\bar{C}$ 2
	A 1	$A\bar{B}\bar{C}$ 4	$A\bar{B}C$ 5	$ABC$ 7	$AB\bar{C}$ 6

		C	
		$\bar{C}$ 0	C 1
AB	$\bar{A}\bar{B}$ 00	$\bar{A}\bar{B}\bar{C}$ 0	$\bar{A}\bar{B}C$ 1
	$\bar{A}B$ 01	$\bar{A}B\bar{C}$ 2	$\bar{A}BC$ 3
	$AB$ 11	$AB\bar{C}$ 6	$ABC$ 7
	$A\bar{B}$ 10	$A\bar{B}\bar{C}$ 4	$A\bar{B}C$ 5

**B. POS: -**

		B+C			
		$B+C$ 00	$B+\bar{C}$ 01	$\bar{B}+C$ 11	$\bar{B}+\bar{C}$ 10
A	0	$A+B+C$ 0	$A+B+\bar{C}$ 1	$A+\bar{B}+C$ 3	$A+\bar{B}+\bar{C}$ 2
	$\bar{A}$ 1	$\bar{A}+B+C$ 4	$\bar{A}+B+\bar{C}$ 5	$\bar{A}+\bar{B}+C$ 7	$\bar{A}+\bar{B}+\bar{C}$ 6

		C	
		0	$\bar{C}$ 1
A+B	00	$A+B+C$ 0	$A+B+\bar{C}$ 1
	01	$A+\bar{B}+C$ 2	$A+\bar{B}+\bar{C}$ 3
	11	$\bar{A}+\bar{B}+C$ 6	$\bar{A}+\bar{B}+\bar{C}$ 7
	10	$\bar{A}+B+C$ 4	$\bar{A}+B+\bar{C}$ 5

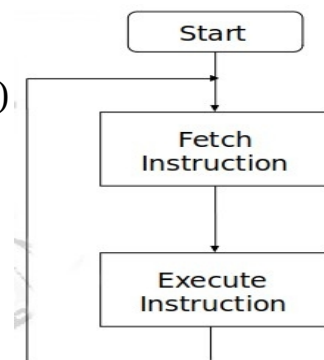
➤ **4 variable k-map:** – In the 4 variable k-map, 16 square are required. An example of 4 variable k-map for SOP and POS form are as follows: –

A. SOP: -					B. POS: -				
AB \ CD	$\overline{C}\overline{D}$ 00	$\overline{C}D$ 01	$CD$ 11	$C\overline{D}$ 10	A+B \ C+D	$C+D$ 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}+D$ 10
$\overline{A}\overline{B}$ 00	$\overline{A}\overline{B}\overline{C}\overline{D}$ 0	$\overline{A}\overline{B}\overline{C}D$ 1	$\overline{A}\overline{B}CD$ 3	$\overline{A}\overline{B}C\overline{D}$ 2	A+B 00	$A+B+C+D$ 0	$A+B+C+\overline{D}$ 1	$A+B+\overline{C}+\overline{D}$ 3	$A+B+\overline{C}+D$ 2
$\overline{A}B$ 01	$\overline{A}B\overline{C}\overline{D}$ 4	$\overline{A}B\overline{C}D$ 5	$\overline{A}BCD$ 7	$\overline{A}BC\overline{D}$ 6	A+B 01	$A+\overline{B}+C+D$ 4	$A+\overline{B}+C+\overline{D}$ 5	$A+\overline{B}+\overline{C}+\overline{D}$ 7	$A+\overline{B}+\overline{C}+D$ 6
$A\overline{B}$ 11	$AB\overline{C}\overline{D}$ 12	$AB\overline{C}D$ 13	$ABCD$ 15	$ABC\overline{D}$ 14	$\overline{A}+B$ 11	$\overline{A}+\overline{B}+C+D$ 12	$\overline{A}+\overline{B}+C+\overline{D}$ 13	$\overline{A}+\overline{B}+\overline{C}+\overline{D}$ 15	$\overline{A}+\overline{B}+\overline{C}+D$ 14
$AB$ 10	$A\overline{B}\overline{C}\overline{D}$ 8	$A\overline{B}\overline{C}D$ 9	$A\overline{B}CD$ 11	$A\overline{B}C\overline{D}$ 10	$\overline{A}+B$ 10	$\overline{A}+B+C+D$ 8	$\overline{A}+B+C+\overline{D}$ 9	$\overline{A}+B+\overline{C}+\overline{D}$ 11	$\overline{A}+B+\overline{C}+D$ 10

**Instruction cycle:** - Basic - Intermediate - Exceptions

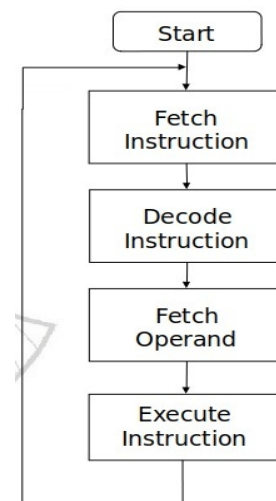
### The Instruction Cycle - Basic View

- Once the computer has been started (bootstrapped) it continually executes instructions (until the computer is stopped)
- Typically, different instructions take different amounts of time to execute
- All instructions and data are contained in main memory



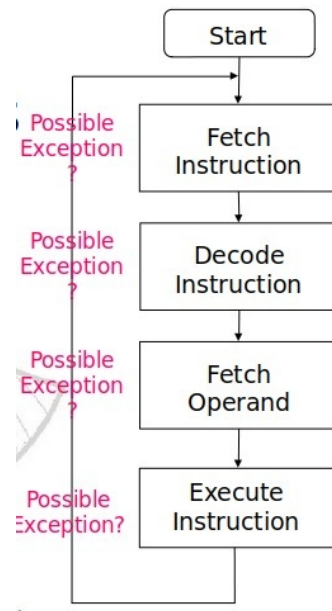
**The Instruction Cycle - Intermediate View:** A complete instruction consists of

- operation code
- addressing mode
- zero or more operands
  - immediately available data (embedded within the instruction)
  - address where the data can be found in main memory



## The Instruction Cycle – Exceptions

Exceptions, or errors, may occur at various points in the instruction cycle, for example:

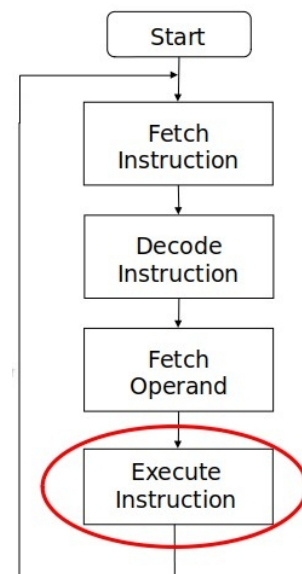
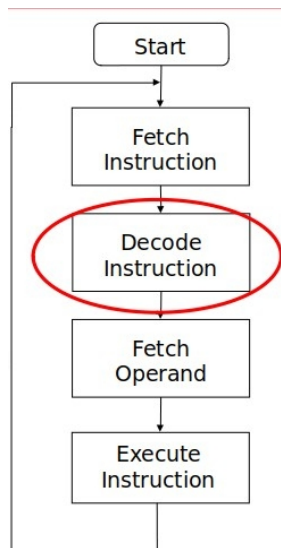
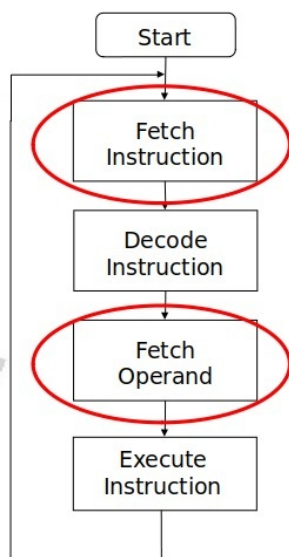


**Addressing-** the memory does not exist or is inaccessible

**Operation** - the operation code does not denote a valid operation

**Execution** - the instruction logic fails, typically due to the input data

- divide by zero
- integer addition/subtraction overflow
- floating point underflow/overflow



## Instruction Architecture

- Software Design
- Hardware circuits

### Instruction Architecture: Software Design

Each computer CPU must be designed to accommodate and understand instructions according to specific formats.

Examples:

- All instructions must have an operation code specified

NOP : no operation

TSTST : test and set

<b>OpCode</b>
---------------

- Most instructions will require one, or more, operands.

These may be (immediate) data to be used directly

or, addresses of memory locations where data will be found (including the address of yet another location)

<b>OpCode</b>	<b>Operand (Address)</b>
---------------	--------------------------

- Sometimes the instruction format requires a code, called the Mode, that specifies a particular addressing format to be distinguished from other possible formats:

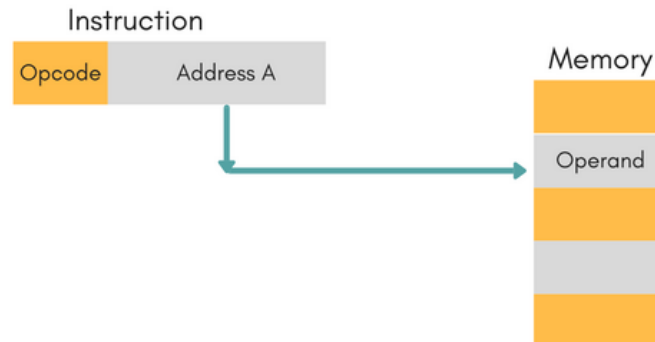
- direct addressing
- indirect addressing
- Register direct addressing
- Register indirect addressing
- index addressing etc...

### Addressing Modes

- When a microprocessor accesses memory, to either read or write data, it must specify the memory address it needs to access
- Several addressing modes to generate this address are known, a microprocessor instruction set architecture may contain some or all of those modes, depending on its design

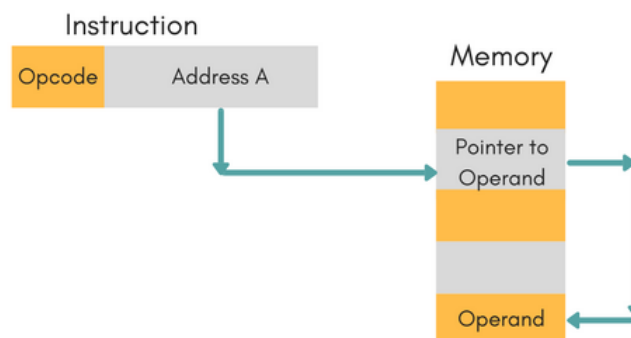
- In the following examples we will use the LDAC instruction (loads data from a memory location into the AC (accumulator) microprocessor register)

## Direct mode



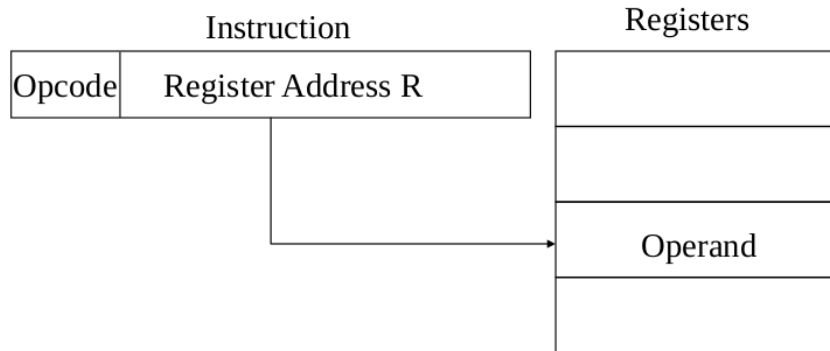
- Instruction includes the A memory address
- For eg: LDAC 5 – accesses memory location 5, reads the data (suppose 10) and stores the data in the microprocessor's accumulator
- This mode is usually used to load variables and operands into the CPU

## Indirect mode



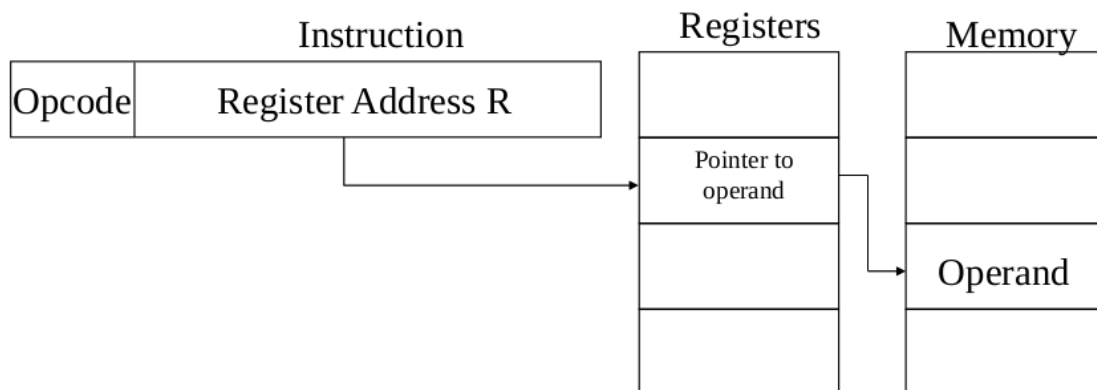
- Starts like the direct mode, but it makes an extra memory access. The address specified in the instruction is not the address of the operand, it is the address of a memory location that contains the address of the operand.
- LDAC @5 or LDAC (5), first retrieves the content of memory location 5, suppose 10, and then CPU goes to location 10, reads the content (20) of that location and loads the data into the CPU

## Register direct mode



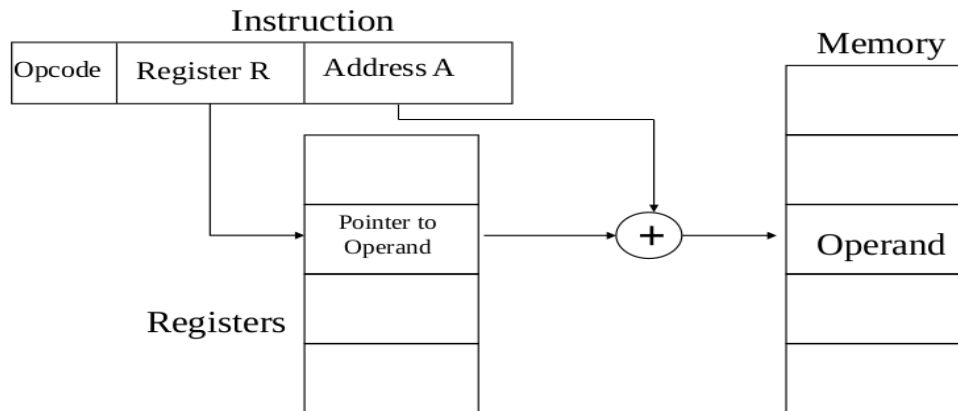
- It specifies a register instead a memory address
- LDAC R – if register R contains an value 5, then the value 5 is copied into the CPU's accumulator
- No memory access
- Very fast execution
- Very limited address space

## Register indirect mode



- LDAC @R or LDAC (R) – the register contains the address of the operand in the memory
- Register R (selected by the operand), contains value 5 which represents the address of the operand in the memory (10)

## Displacement Addressing mode



- Effective Address =  $A + (\text{content of } R)$
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

## Relative Addressing mode

It is a version of Displacement addressing mode.

In this the contents of PC(Program Counter) is added to address part of instruction to obtain the effective address.

$EA = A + (PC)$ , where EA is effective address and PC is program counter.

For eg: Consider that the relative instruction LDAC \$5 is located at memory address 10 and it takes two memory locations; the next instruction is at location 12, so the operand is actually located at  $(12 + 5)$  17; the instruction loads the operand at address 17 and stores it in the CPU's accumulator

## Index Addressing mode

Works like relative addressing mode; instead adding the A to the content of program counter (PC), the A is added to the content of an index register

For eg: If the index register contains value 10, then the instruction LDAC 5(X) reads data from memory at location  $(5+10)$  15 and stores it in the accumulator.

## Instruction Architecture - Hardware Circuits

- Everything that the computer can do is the result of designing and building devices to carry out each function.
- At the most elementary level the devices are called logic gates.
- There are many possible gate types, each perform a specific Boolean operation (e.g. AND, OR, NOT, NAND, NOR, XOR, XNOR)



- ALL circuits, hence all functions, are defined in terms of the basic gates.
- We apply Boolean Algebra and Boolean Calculus in order to design circuits and then optimize our designs.
- Data is represented by various types of “signals”, including electrical, magnetic, optical and so on. Data “moves” through the computer along wires that form the various bus networks (address, data, control) and which interconnect the gates.
- Combinations of gates are called integrated circuits (IC).
- All computer functions are defined and controlled by IC’s of varying complexity in design. The manufacture of these may be scaled according to size/complexity:

LSI                      large scale integration  
VLSI                    very large scale integration  
ULSI                    ultra large scale integration

### **Instruction Architecture – CU**

- The control unit must decode instructions, set up for communication with RAM addresses and manage the data stored in register and accumulator storages.
- Each such operation requires separate circuitry to perform the specialized tasks.
- It is also necessary for computer experts to have knowledge of the various data representations to be used on the machine in order to design components that have the desired behaviours.

### **Instruction Architecture – ALU**

- All instructions together are called the instruction set:

CISC                    complex instruction set  
RISC                    reduced instruction set

CISC	RISC
Variable length instructions	Fixed-length instructions
Abundant instructions and addressing modes	Fewer instructions and addressing modes
Longer decoding	Easier decoding
Mem-to-mem operations	Load/store architecture
Use on-core microcode	No microinstructions, directly executed by HW logic
Less pipelineability	Better pipelineability

- Each ALU instruction requires a separate circuit, although some instructions may incorporate the circuit logic of other instructions.