Unit-1: Binary Systems

Digital Computer and Digital System:

Digital System:

Digital systems are designed to store, process, and communicate information in digital form. They are found in a wide range of applications, including process control, communication systems, digital instruments, and consumer products. The digital computer, more commonly called the *computer*, is an example of a typical digital system.

A computer manipulates information in digital, or more precisely, binary form. A binary number has only two discrete values zero or one. Each of these discrete values is represented by the OFF and ON status of an electronic switch called a *transistor*. All computers, therefore, only understand binary numbers. Any decimal number (base 10, with ten digits from 0 to 9) can be represented by a binary number (base 2, with digits 0 and 1).

Some of the basic examples of digital systems are Personal computers, Desktops, Laptops, Smartphones, and Mobile

Digital Computer:

A Digital computer can be considered as a digital system that performs various computational tasks. The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

Digital Computer is a machine or a device that helps to process any kind of information. By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

.

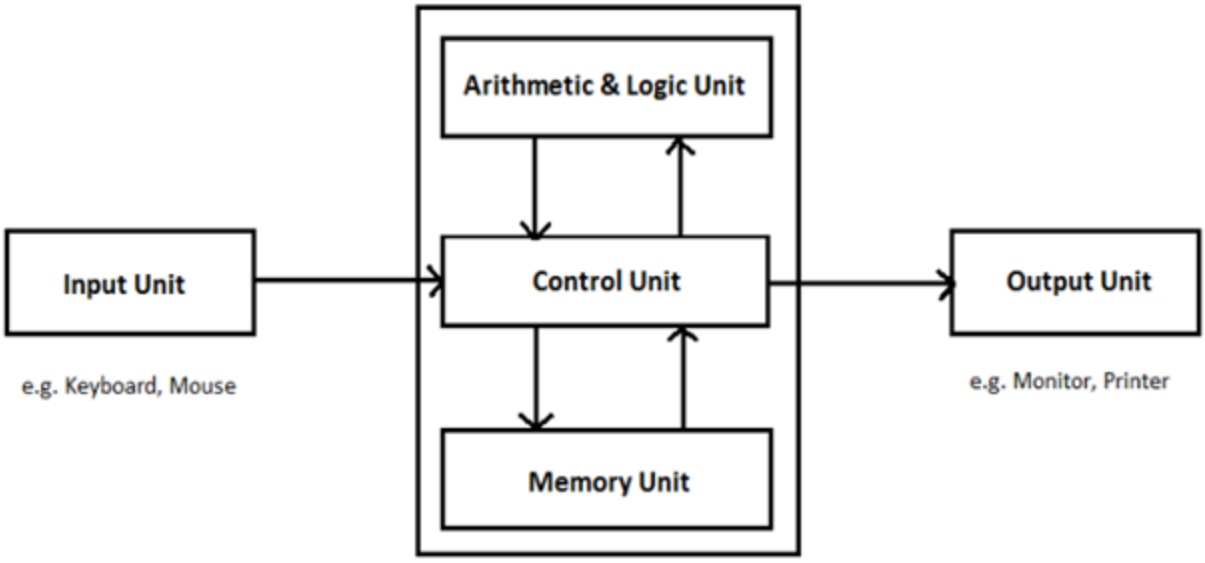


Fig: Block Diagram of Digital Computer

Components of digital computers are:

Input Devices:

The user normally provides the data to the device that is known as input. Eg: Keyboard, Mouse, Scanner, Camera, Joysticks, and Microphones etc.

CPU:

The input that is provided by the user is processed internally using some defined sequence. This action is performed by CPU. CPU has three parts CU, ALU and Memory Unit.

CU (Control Unit):

It is the unit which controls all the operations of the different units but does not carry out any actual data processing operation. **Control unit** transfers data or instruction among different units of a computer system. It receives the instructions from the memory, interprets them and sends the operation to various units as instructed.

ALU (Arithmetic and Logic Unit):

ALU can also be subdivided into 2 sections namely, arithmetic unit and logic unit. It is a complex digital circuit which consists of registers and which performs arithmetic and logical operations.

Arithmetic sections perform arithmetic operations like addition, subtraction, multiplication, division etc. All other Complex operations can also be performed by repetition of these above basic operations.

The logic unit is responsible for performing logical operations such as comparing, selecting, matching and merging of different data or information.

Memory Unit:

Memory can classified in two parts

- Primary Memory
- Secondary Memory

Primary Memory:

Primary memory is also known as **main memory** or may also refer to "*Internal memory*" and primary storage. All those types of computer memories that are directly accessed by the processor using data bus are called primary memory. That allows a processor to access stores running programs and currently processed data that stored in a memory location. This memory also called the main memory. Primary memory is volatile in nature.

For example: RAM, ROM etc.

Secondary or Auxiliary Memory:

Auxiliary memory (also referred to as *secondary storage*) is the non-volatile memory lowest-cost, highest-capacity, and slowest-access storage in a computer system. It is where programs and data kept for long-term storage or when not in immediate use.

The most common examples of auxiliary memories are Hard Disk, Pendrive, CD, DVD etc.

Output Devices:

Once the processing is completed, based on the input, the output is displayed to the user through output devices. Eg: Screen, printer, speakers etc.

Different types of Digital Computer

Digital computers are a device that needs to be programmed in order to receive the desired output. It uses electronic technology to generate, store and process different types of data.

Based on the size and type of the device, these digital computers are classified into four categories.

- Microcomputer
- Minicomputer
- Mainframe computer
- Supercomputer

Microcomputer

A Microcomputer is not really expensive and it comes with a microprocessor as its Central processing unit and input/output devices.

These computers are generally called personal computers and a few of the examples are IBM pc, Apple, Dell etc.

Minicomputers

Minicomputers are known as mid-range computers that contain one or more processors.

They support multiprocessing which means these multiple processors share the same computer memory and other required peripheral devices to perform the given task. Minicomputers are generally used for processing transactions, file handling, managing database.

Mainframe computers

Mainframe computers are generally large size computers mainly used for storing large amounts of data and processing. It is known for its high level of reliability.

These machines are used by an organization that requires crucial applications such as census, customer statistics for large calculations which require a high volume of data processing.

Super Computers

Supercomputers are very expensive and the world's fastest computers are available.

These computers have thousands of processors that perform trillions of calculations per second and hence the fastest known ever. Supercomputers are used extensively in enterprises and organizations that require massive calculations.

Integrated Circuit (IC)

An integrated circuit (IC) is a small semiconductor-based electronic device consisting of fabricated transistors, resistors and capacitors. Integrated circuits are the building blocks of most electronic devices and equipment.

An integrated circuit is built with the primary objective of embedding as many transistors as possible on a single semiconductor chip.

According to their design assembly, integrated circuits have several generations of advancements and developments such as:

- Small Scale Integration (SSI): Ten to hundreds of transistors per chip
- Medium Scale Integration (MSI): Hundreds to thousands of transistors per chip
- Large Scale Integration (LSI): Thousands to several hundred thousand transistors per chip
- Very Large Scale Integration (VLSI): Up to 1 million transistors per chip
- Ultra Large Scale Integration (ULSI): This represents a modern IC with millions and billions of transistors per chip

An IC can be further classified as digital, analog or a combination of both. The most common example of a modern IC is the computer processor, which consists of billions of fabricated transistors, logic gates and other digital circuits.

Number System:

Number systems are the technique to represent numbers in the computer system architecture, every value that you are saving or getting into/from computer memory has a defined number system.

Computer architecture supports following number systems.

- Binary number system
- Octal number system
- Decimal number system
- Hexadecimal (hex) number system

Binary Number System

A Binary number system has only two digits that are **0 and 1**. Every number (value) represents with 0 and 1 in this number system. The base of binary number system is 2, because it has only two digits.

For Example: $(110010010)_2$

Octal Number System

Octal number system has only eight (8) digits from **0 to 7**. Every number (value) represents with 0,1,2,3,4,5,6 and 7 in this number system. The base of octal number system is 8, because it has only 8 digits. For Example: $(12567)_8$

Decimal Number System

Decimal number system has only ten (10) digits from **0 to 9**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8 and 9 in this number system. The base of decimal number system is 10, because it has only 10 digits.

For Example: $(127839)_{10}$

Hexadecimal Number System

A Hexadecimal number system has sixteen (16) alphanumeric values from **0 to 9** and **A to F**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E and F in this number system. The base of hexadecimal number system is 16,

because it has 16 alphanumeric values. Here **A = 10**, **B = 11**, **C = 12**, **D = 13**, **E = 14** and **F = 15**.

For Example: $(178AFD)_{16}$

Number System Conversion:

There are three types of conversion:

Decimal Number System to Other Base:

- Decimal to Binary
- Decimal to Octal
- Decimal to Hexadecimal

Other Base to Decimal Number System:

- Binary to Decimal
- Octal to Decimal
- Hexadecimal to Decimal

Other Base to Other Base:

- Binary to Octal and Vice-versa
- Binary to Hexadecimal Vice-versa
- Octal to Hexadecimal Vice-versa

Decimal Number System to Other Base

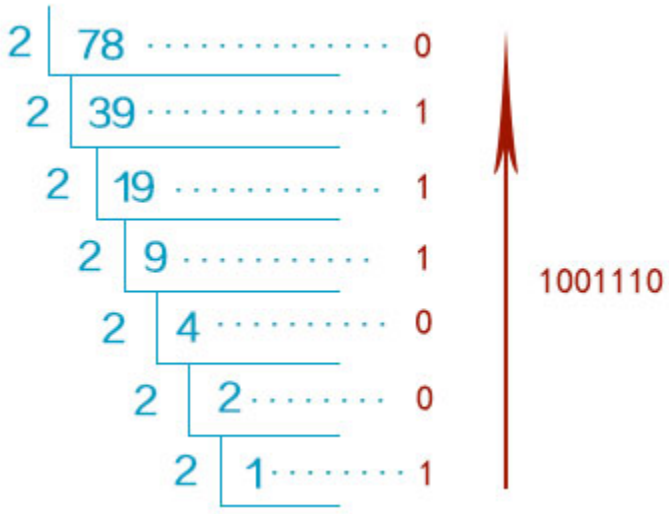
To convert Number system from **Decimal Number System** to **Any Other Base** is quite easy; you have to follow just two steps:

a) Divide the given Number (*Decimal Number*) by the base of target base system (*in which you want to convert the number: Binary (2), octal (8) and Hexadecimal (16)*).

b) Write the remainder from Most Signification Bit (MSB) to Least Significant Bit (MSB) Step last as a MSB. Or Bottom to up.

Decimal to Binary: Divide given decimal number by 2 and write remainder from MSB to LSB.

Example1: $(78)_{10}$ to $(?)_2$



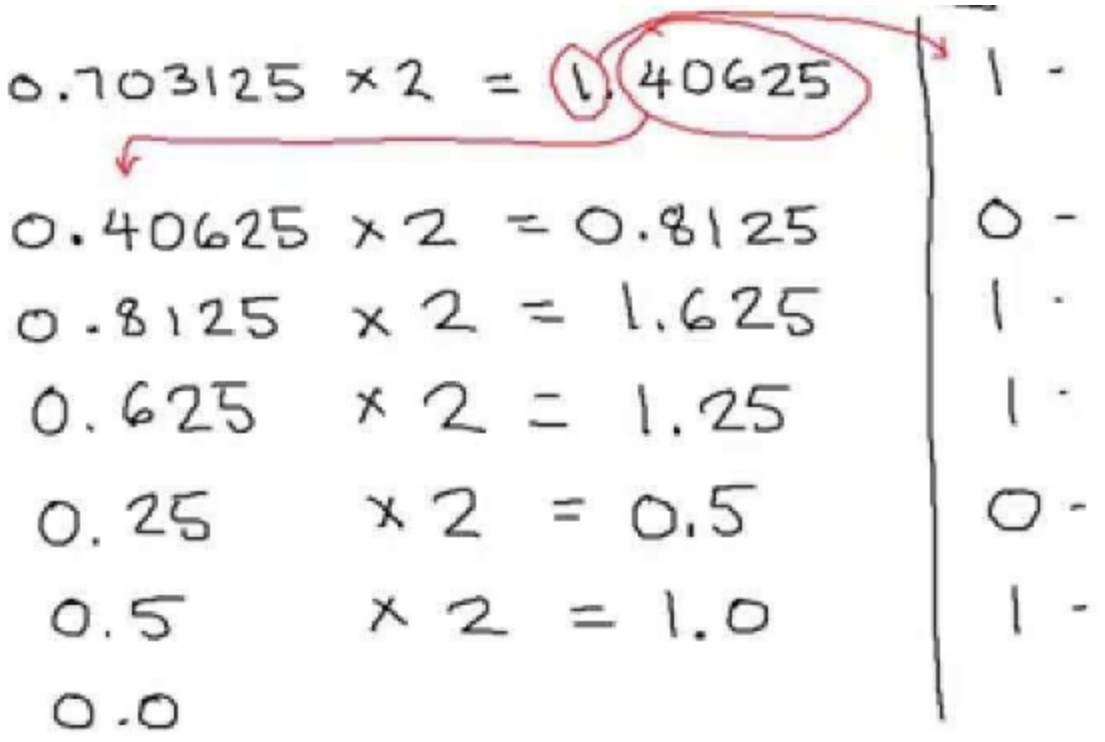
$(78)_{10} = (1001110)_2$

Example2: $(3.703125)_{10}$ to $(?)_2$

First convert integer value i.e. 3 in to binary

$(3)_{10} = (11)_2$

Then, Convert fraction value i.e. 0.703125 in to binary



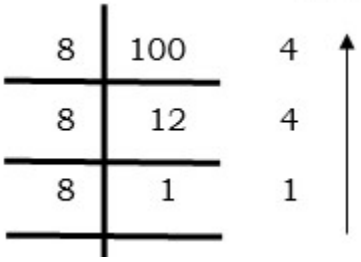
= 0.101101

Now concatenate integer and fraction value

$(3.703125)_{10} = (11.101101)_2$

Decimal to Octal: Divide given decimal number by 8 and write remainder from MSB to LSB.

Example2: $(100)_{10}$ to $(?)_8$



$(100)_{10} = (144)_8$

Example2: $(100.342)_{10}$ to $(?)_8$

First, convert integer value i.e. 100 to Octal,

$(100)_{10} = (144)_8$

Then, convert fraction value i.e. 0.342 to Octal

$(0.342)_{10} = (?)_8$

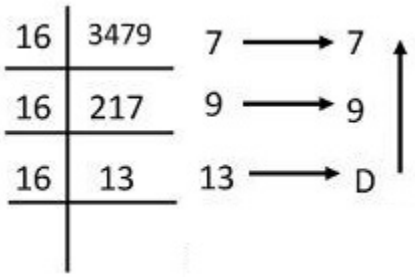
0.342 x 8 = 2.736
0.736 x 8 = 5.888
0.888 x 8 = 7.104
0.104 x 8 = 0.832

$(0.342)_{10} = (0.2570)_8$

Hence, $(100.342)_{10} = (144.2570)_8$

Decimal to Hexadecimal: Divide given decimal number by 16 and write remainder from MSB to LSB

Example1: $(3479)_{10}$ to $(?)_{16}$



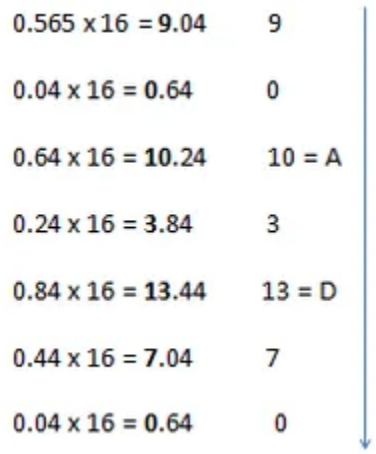
$(3479)_{10} = (D97)_{16}$

Example2: $(3479.565)_{10}$ to $(?)_{16}$

First convert integer number i.e. 3479 in to hexadecimal,

$(3479)_{10} = (D97)_{16}$

Then, convert fraction number i.e. 0.565 in to hexadecimal



$(0.565)_{10} = (0.90A3D70)_{16}$

Hence, $(3479.565)_{10} = (D97.90A3D70)$

Other to Decimal

To convert Number System from **Any Other Base System** to **Decimal Number System**, you have to follow just three steps:

a) Determine the base value of source Number System (*that you want to convert*), and also determine the position of digits from LSB (*first digit's position – 0, second digit's position – 1 and so on*).

b) Multiply each digit with its corresponding multiplication of position value and Base of Source Number System's Base.

c) Add the resulted value of step b

Binary to Decimal

Example1: $(101)_2$ to $(?)_{10}$

$1*2^2+0*2^1+1*2^0$

4+0+1

$(5)_{10}$

Example2: $(101.110)_2$ to $(?)_{10}$

$1*2^2+0*2^1+1*2^0+1*2^{-1}+1*2^{-2}+0*2^{-3}$

$4+0+1+0.5+0.25+0.125$

$(101.110)_2 = (5.875)_{10}$

Octal to Decimal

Example1: $(274)_8$ to $(?)_{10}$

$2*8^2+7*8^1+4*8^0$

$128+56+4$

$(188)_{10}$

Example2: $(274.267)_8$ to $(?)_{10}$

$2*8^2+7*8^1+4*8^0+2*8^{-1}+6*8^{-2}+7*8^{-3}$

$128+56+4+0.25+0.094+0.014$

$(274.267)_8 = (188.358)_{10}$

Hexadecimal to Decimal

Example1: $(1A5)_{16}$ to $(?)_{10}$

$1*16^2+10*16^1+5*16^0$

$256+160+5$

$(421)_{10}$

Example2: $(1A5.B1)_{16}$ to $(?)_{10}$

$1*16^2+10*16^1+5*16^0+11*16^{-1}+1*16^{-2}$

$156+160+5+0.688+0.004$

$(1A5.B1)_{16} = (421.692)_{10}$

Other Base to Other Base:

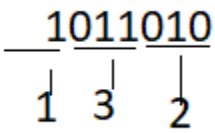
Binary to Octal

Step1: separate the given binary in the group of 3 digits from LSB

Step2: find the corresponding value of binary digits

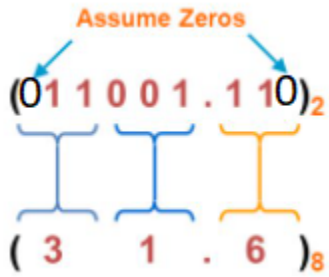
Step3: write value from MSB to LSB

Example1: (1011010) to $(?)_8$



$(1011010)_2 = (132)_8$

Example2: $(11001.11)_2$ to $(?)_8$



$(11001.11)_2 = (31.6)_8$

Octal to Binary

Step1: Separate each digit of given number

Step2: find binary value of given number

Step3: write value from MSB to LSB

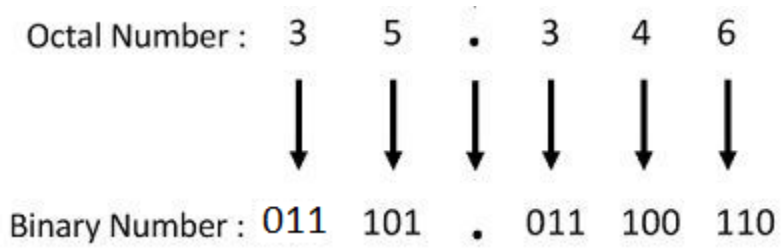
Example1: $(172)_8$ to $(?)_2$

1	7	2
001	111	010

$(172)_8 = (001111010)_2$

Or, $(172)_8 = (1111010)_2$

Example2: $(35.346)_8$ to $(?)_2$



$(35.346)_8 = (011\ 101.\ 011\ 100\ 110)_2$

Or, $(35.346)_8 = (11\ 101.\ 011\ 100\ 11)_2$

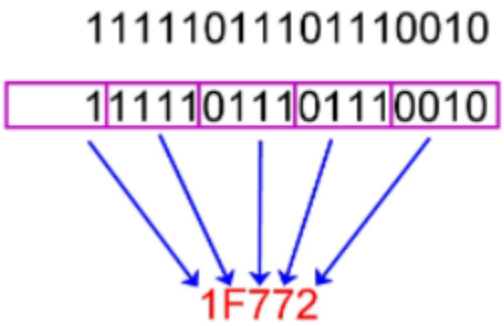
Binary to Hexadecimal

Step1: separate the given binary in the group of 4 digits from LSB

Step2: find the corresponding value of binary digits

Step3: write value from MSB to LSB

Example: $(1111101110111 0010)_2$ to $(?)_{16}$



$(1111101110111 0010)_2 = (1F772)_{16}$

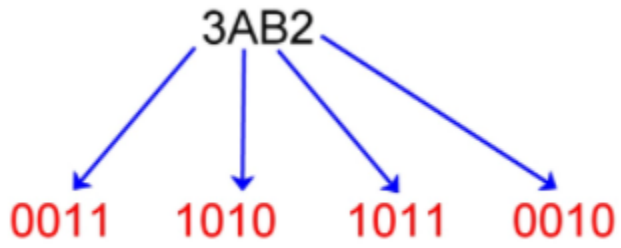
Hexadecimal to Binary

Step1: Separate each digit of given number

Step2: find binary value of given number

Step3: write value from MSB to LSB

Example: $(3AB2)_{16}$ to $(?)_2$



$(3AB2)_{16} = (11101010110010)_2$

Octal to Hexadecimal

Option1: *Convert Octal to Decimal and Decimal to Hexadecimal*

Option2: *Convert Octal to Binary and Binary to Hexadecimal*

Hexadecimal to Octal

Option1: *Hexadecimal to Decimal and Decimal to Octal*

Option2: *Hexadecimal to Binary and Binary to Octal*

Representation of Signed and Unsigned Numbers

Unsigned Numbers:

Unsigned numbers don't have any sign, these can contain only magnitude of the number. So, representation of unsigned binary numbers are all positive numbers only.

For example, representation of positive decimal numbers are positive by default. We always assume that there is a positive sign symbol in front of every number.

Representation of Unsigned Binary Numbers:

Since there is no sign bit in this unsigned binary number, so N bit binary number represent its magnitude only. Zero (0) is also unsigned number. This representation has only one zero (0), which is always positive. Every number in unsigned number representation has only one unique binary equivalent form, so this is unambiguous representation technique. The range of unsigned binary number is from 0 to $(2^n - 1)$.

Example 1: Represent decimal number 92 in unsigned binary number.

Simply convert it into Binary number, it contains only magnitude of the given number.

$= (92)_{10}$

$= (1x2^6+0x2^5+1x2^4+1x2^3+1x2^2+0x2^1+0x2^0)_{10}$

$= (1011100)_{10}$

It's 7 bit binary magnitude of the decimal number 92.

Example 2: Find range of 5 bit unsigned binary numbers. Also, find minimum and maximum value in this range.

Since, range of unsigned binary number is from 0 to (2^n-1). Therefore, range of 5 bit unsigned binary number is *from* 0 to (2^5-1) which is equal from minimum value 0 (i.e., 00000) to maximum value 31 (i.e., 11111).

Signed Numbers:

Signed numbers contain sign flag, this representation distinguish positive and negative numbers. This technique contains both sign bit and magnitude of a number. For example, in representation of negative decimal numbers, we need to put negative symbol in front of given decimal number.

Representation of Signed Binary Numbers:

There are three types of representations for signed binary numbers. Because of extra signed bit, binary number zero has two representation, either positive (0) or negative (1), so ambiguous representation. But 2's complementation representation is unambiguous representation because of there is no double representation of number 0. These are: Sign-Magnitude form, 1's complement form, and 2's complement form which are explained as below.

1. ***Sign-Magnitude form***
   In this form, a binary number has a bit for a sign symbol. If this bit is set to 1, the number will be negative else the number will be positive if it is set to 0. Apart from this sign-bit, the n-1 bits represent the magnitude of the number.
   Syntax:

Sign Bit	Actual binary

   E.g. +7 = 0111
   -7 = 1111

2. ***1's Complement***
   By inverting each bit of a number, we can obtain the 1's complement of a number. The negative numbers can be represented in the form of 1's complement. In this form, the binary number also has an extra bit for sign representation as a sign-magnitude form.
   Syntax:

Sign Bit	1's complement of actual binary

   E.g. 7 = 111
   1's complement = 000
   - 7 = 1000

3. ***2's Complement***
   By inverting each bit of a number and adding plus 1 to its least significant bit, we can obtain the 2's complement of a number. The negative numbers can also be represented in the form of 2's complement. In this form, the binary number also has an extra bit for sign representation as a sign-magnitude form.
   Syntax:

Sign Bit	2's complement of actual binary

   E.g. 7 = 111
   1's complement = 000
   2's complement = 000 + 1 = 001
   - 7 = 1001

Binary Arithmetic Operations

How to find binary of given integer number or vice-versa?

Use the pattern $2^n+.....+2^5+2^4+2^3+2^2+2^1+2^0$

i.e. n+…….+32+16+8+4+2+1

For Integer to binary: *calculate binary digits of any given number, add the numbers from above pattern whose sum is equals to given integer number and replace by 1 in the place of used number and 0 by unused number.*

For Binary to Integer: *take equal digits of pattern as given binary digit. Then, add the digits from pattern in the place of 1 in the binary digits.*

Note: *For Octal take 3 digits, For Decimal and Hexadecimal take 4 digits*

Find binary value of given integer number

Example1: *Find binary value of 6*

First find the value whose sum is 6 from the pattern (4+2+ 1) which is 4 and 2

Place 1 for used value and 0 for remaining

So, 110 which is the binary value of 6

Example2: *Find binary value of 60*

First find the value whose sum is 60 from the pattern (32+16+8+ 4+2+ 1) which is 4 and 2

Place 1 for used value and 0 for remaining

So, 111100 which is the binary of 60

Find integer number from binary value

Example3: find the integer number of binary digit 101

Then, take vale in the place of 1 and ignore place of 0 from the pattern (4+ 2+1)

4+1=5 which is the integer of 101

Example3: find the integer number of binary digit 101101

Then, take vale in the place of 1 and ignore place of 0 from the pattern (32+16+8+ 4+ 2+1)

i.e. (32+8+4+1) = 45 which is the integer number of 101101

Complement Rule:

Base 2: Binary number

- ***1's complement*** – toggle the given binary number i.e. 1 to 0 and 0 to 1, then we can get 1's complement
  Eg: fined the 1's complement of $(1101010)_2$
  1's complement of given number is **$(0010101)_2$**
- ***2's complement*** – first take 1's complement and add 1 to the result, then we can get 2's complement
  Eg: find the 2's complement of $(11011)_2$
  1's complement of given number is **00100**
  Now add 1 to the result
  00100
  +1

  $(00101)_2$ this is 2's complement

Base 8: Octal number

- ***7's complement*** – take highest number of n bits and subtract given number from that highest number, then we can get 7's complement
  Eg: find 7's complement of $(2345)_8$
  First take highest number of 4 bit i.e. 7777
  New, subtract 2345 from 7777
  7777
  -2345

  $(5432)_8$ this is 7's complement

- **8's complement –** add 1 to the 7's complement

Eg: find 8's complement of $(6573)_8$
First find 7's complement of 6573
7777
-6573

1204 *this is 7's complement*
Now, add 1 to the result
1204
+1

$(1205)_8$ this is 8's complement

Base 10: Decimal Number

- ***9's complement*** – take highest number of n bits and subtract given number from that highest number, then we can get 9's complement
  Eg: find 9's complement of $(3489)_{10}$
  First take highest number of 4 bit i.e. 9999
  New, subtract 3489 from 9999
  9999
  -3489

  $(6510)_8$ this is 9's complement

- **10's complement –** add 1 to the 9's complement
  Eg: find 10's complement of $(6579)_{10}$
  First find 9's complement of 6579
  9999
  -6579

  3420 *this is 9's complement*

  Now, add 1 to the result
  3420
  +1

$(3421)_{10}$ this is 10's complement

Base 16: Hexadecimal

- ***15's complement*** – take highest number of n bits and subtract given number from that highest number, then we can get 15's complement
  Eg: find 15's complement of $(233A)_{16}$
  First take highest number of 4 bit i.e. FFFF
  New, subtract 233A from FFFF
  FFFF
  -233A

  $(DCC5)_{16}$ this is 15's complement

- **16's complement –** add 1 to the 15's complement
  Eg: find 16's complement of $(6573)_{16}$
  First find 15's complement of 6573
  FFFF
  -6573

  9A8C *this is 15's complement*
  Now, add 1 to the result
  9A8C
  +1

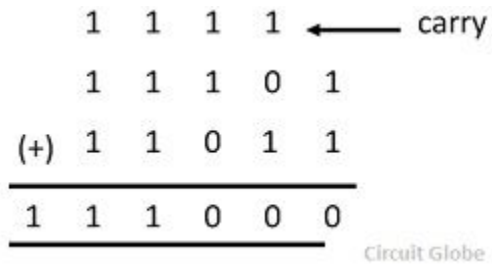
  $(9A8D)_{16}$ this is 16's complement

Binary Addition

The binary number system uses only two digits 0 and 1 due to which their addition is simple. There are four basic operations for binary addition, as mentioned above.

0+0=0
0+1=1
1+0=1
1+1=10

Example: Add 11101 and 11011.



The above sum is carried out by following step

1 + 1 = 10 = 0 with a carry of 1

1+0+1 = 10 = 0 with a carry of 1

1+1+0 = 10 = 10 = 0 with a carry of 1

1+1+1= 10+1 = 11= 1 with a carry of 1

1 +1 +1 = 11

Note carefully that 10 + 1 = 11, which is equivalent to two + one = three (the next binary number after 10)

Thus the required result is 111000.

Binary Subtraction

The subtraction of the binary digit depends on the four basic operations

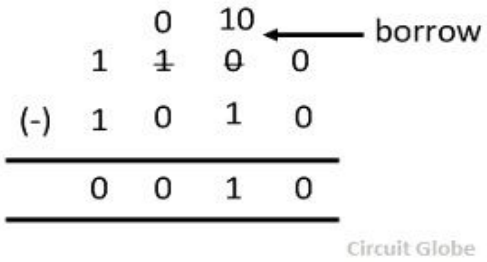
0 – 0 = 0

1 – 0 = 1

1 – 1 = 0

10 – 1 = 1

Example: Subtract 1010 from 1100



The above subtraction is carried out through the following steps.

0 – 0 = 0

For 0 – 1 = 1, taking borrow 1 and then 10 – 1 = 1

For 1 – 0 , since 1 has already been given, it becomes 0 – 0 = 0

1 – 1 = 0

Therefore the result is 0010.

Binary subtraction using 1's and 2's complement

1's complement Rule:

Step1: *Take 1's complement of subtrahend (which is subtracted)*

i.e. 11001 - 1100

minuend subtrahend

Step2: *Add the result i.e. 1's complement of subtrahend to the minuend*

If, carry occurs add that carry to the result

If, carry not occurs, which means result is negative,

Then, do 1's complement of result

Example1: Subtract 1010 from 1100 using 1's complement

Solution: Here, minuend = 1100, subtrahend = 1010

Take 1's complement of subtrahend i.e. 1010 = 0101

Then, add result of 1's complement of subtrahend to the minuend

```
        1
      1100
    + 0101
    -------
Carry—(1)0001
add carry  + 1
    -------
      0010
    -------
```

Hence, Result is (0010)

Example2: Subtract 1100 from 1010 using 1's complement

Solution: Here, minuend = 1010, subtrahend = 1100

Take 1's complement of subtrahend i.e. 1100 = 0011

Then, add result of 1's complement of subtrahend to the minuend

```
      1010
    + 0011
    -------
no carry 1101
    -------
```

There is no carry so, result is negative.

Then, take 1's complement of result i.e. 1101 = 0010

Hence, Result is (- 0010)

2's Complement Rule:

Step1: *Take 2's complement of subtrahend*

Step2: *add result of 2's complement of subtrahend to the minuend*

If, carry occurs, discard the carry and you got the result.

If, carry not occurs, result is negative

Then, perform 2's complement of the result and you got final result.

2's Complement rule for sign number:

2's complement of –x = +x

2's complement of +x = -x

Example: find the binary value of -8

Solution: here, -8 is equals to 2's complement of +8

+8 = 01000
1's comp 10111
2's comp +1

11000
Hence, -8 = 11000

Example 1: Subtract 0010 from 0111 using 2's complement

Solution: Here, Subtrahend = 0010, Minuend = 0111

Take 2's complement of subtrahend i.e. 0010

First, take 1's complement of 0010 = 1101

Add 1 to the result

```
 1101
   + 1
------
 1110
------
```

New, add result to the minuend

```
    0111
  + 1110
-------
Carry (1)0101
```

Here, carry occurred i.e. 1, discard it

Hence, result is (0101)

Example 2: Subtract 0111 from 0010 using 2's complement

Solution: Here, Subtrahend = 0111, Minuend = 0010

Take 2's complement of subtrahend i.e. 0111

First, take 1's complement of 0111 = 1000

Add 1 to the result

```
 1000
   +1
 ----
 1001
 ----
```

Now, add result to the minuend

```
          0010
        + 1001
          ----
no carry  1011
          ----
```

Here, no carry so result is negative

Then, perform 2's complement of result

First, take 1's complement of result i.e. 1011 = 0100

Add 1 to the result

```
 0100
   +1
 ----
 0101
 ----
```

Hence, Result is $-(0101)_2$

Example 3: If A = 37 and B = -18 then perform B-A using 2's complement

Solution: B-A = (-18) -37 = (-18) + (-37) = -55

Now, find the binary value of -18 and -37

-18 = 2's complement of +18

+18 => 010010

1's comp => 101101
+1

2's comp => 101110

So, -18 => 101110
Similarly, -37 = 2's complement of +37
+37 => 0100101
1's comp => 1011010
+1

2's comp => 1011011

So, -37 => 1011011
Now add binary value of -18 and -37

```
  0101110
  1011011
  -------
1| 0001001
```

Here first 1 is carry so discard it, and final answer is (001001) which is equivalent to -55

Let's check, 2's complement of 001001 should be 55.

1's complement of 001001

110110
+1

110111 => 55
Hence, 001001 => -55

Subtraction using 9's and 10's complement

9's Complement Rule:

Step1: *Take 9's complement of subtrahend (which is subtracted)*

i.e. $(225)_{10}$ - $(115)_{10}$

minuend subtrahend

Step2: *Add the result i.e. 9's complement of subtrahend to the minuend*

If, carry occurs add that carry to the result

If, carry not occurs, which means result is negative,

Then, do 9's complement of result

Example 1: Subtract $(225)_{10}$ - $(115)_{10}$ using 9's complement

Solution: Here, Subtrahend = 115, Minuend = 225

First, take 9's complement of subtrahend i.e. 115

```
 999
-115
-------
 884 this is 9's complement
```

Now, add this result to the minuend

```
  225
 +884
-------
1 109
```

Here, 1 is carry add it to the result

```
 109
  +1
--------
 110
```

Hence, result of subtraction is $(110)_{10}$

Example 2: Subtract $(145)_{10} - (150)_{10}$ using 9's complement

First, take 9's complement of subtrahend i.e. 150

```
 999
-150
-------
```

849 *this is 9's complement*

Now, add this result to the minuend

```
  145
+849
-------
```

894 no carry occurs

Here, No carry occurs, hence result is negative, then take 9's complement of result

```
 999
-894
--------
```

005

Hence, result of subtraction is $-(5)_{10}$

10's Complement Rule:

Step1: *Take 10's complement of subtrahend*

Step2: *add result of 10's complement of subtrahend to the minuend*

If, carry occurs, discard the carry and you got the result.

If, carry not occurs, result is negative

Then, perform 10's complement of the result and you got final result.

Example 1: subtract $(115)_{10} - (105)_{10}$

Solution: take 10's complement of subtrahend i.e. 105

First, take 9's complement of 105

```
 999
-105
-------
```

894 this is 9's complement

Add 1 to the result

```
 894
  +1
-------
```

895 this is 10's complement

Now, add result to the minuend

```
 115
+895
--------
```

1 010 here 1 is carry discard it

Hence, result is $(10)_{10}$

Example 2: subtract $(115)_{10} - (220)_{10}$

Solution: take 10's complement of subtrahend i.e. 220

First, take 9's complement of 220

```
 999
-220
-------
```

779 this is 9's complement

Add 1 to the result

```
 779
  +1
-------
```

780this is 10's complement

Now, add result to the minuend

```
  115
+780
--------
 895
```

here no carry occurs, hence result is negative and take 10's complement of result

```
  999
-895
--------
 104
```

New add 1 to the result

```
  104
   +1
--------
  105
```

Result of subtraction is $(105)_{10}$

Binary Multiplication

For the binary multiplication we have some rules as:

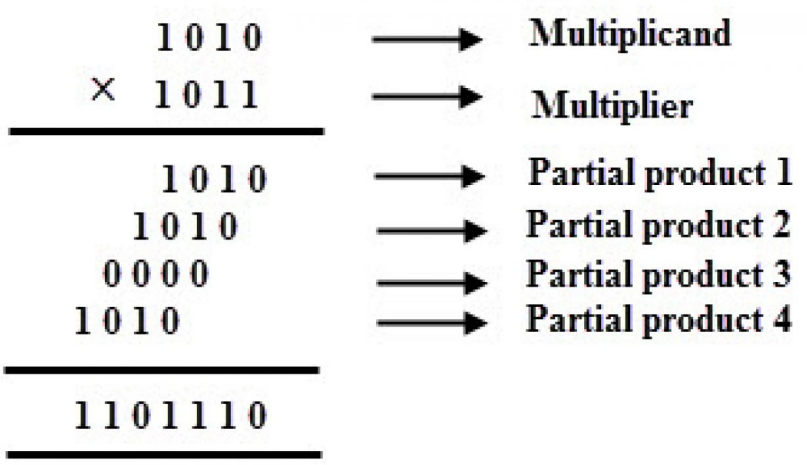
0 x 0 = 0

0 x 1 = 0

1 x 0 = 0

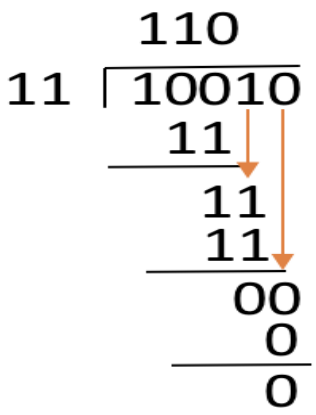
1 x 1 = 1

Example: Multiply 1010 and 1011



Binary Division

Example: Divide 10010 by 11



Result is 110

Binary Codes:

In the coding, when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code. The digital data is represented, stored and transmitted as group of binary bits. This group is also called as binary code. The binary code is represented by the number as well as alphanumeric letter.

Advantages of Binary Code

Following is the list of advantages that binary code offers.

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Binary codes make the analysis and designing of digital circuits if we use the binary codes.
- Since only 0 & 1 are being used, implementation becomes easy.

Classification of binary codes

The codes are broadly categorized into following four categories.

- Binary Coded Decimal (BCD) Code
- Alphanumeric Codes
- Weighted Codes
- Non-Weighted Codes

Binary Coded Decimal (BCD) code:

In this code each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

Representation of BCD:

0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
10 = 0001 0000

11 = 0001 0001
12 = 0001 0010
13 = 0001 0011
255 = 0010 0101 0101
And so on….

Advantages of BCD Codes

- It is very similar to decimal system.
- We need to remember binary equivalent of decimal numbers 0 to 9 only.

Disadvantages of BCD Codes

- The addition and subtraction of BCD have different rules.
- The BCD arithmetic is little more complicated.
- BCD needs more number of bits than binary to represent the decimal number. So BCD is less efficient than binary.

BCD Conversion:

- BCD to Decimal
- Decimal to BCD
- BCD to Binary
- Binary to BCD

BCD to Decimal:

Divide the given BCD code to 8421(group of 4-bit) group and convert into corresponding decimal values of each group.

Example: convert the following BCD code to decimal

a. $(100100110111)_{BCD}$

Solution: first make 4-bit group of given BCD code

1001 0011 0111
 9 3 7

Hence decimal value is $(937)_{10}$

Decimal to BCD:

Convert each digit of given decimal number in to their corresponding 4-bit BCD code.

Example: Convert following decimal value to BCD code

a. $(2345)_{10}$

Solution: BCD code to each digits of given decimal value is

2 = 0010 3 = 0011 4 = 0100 5 = 0101

Hence BCD code is $(0010001101000101)_{BCD}$

BCD to Binary:

Step 1: Concert given BCD code to decimal value

Step 2: Convert resultant decimal value to binary code

Binary to BCD:

Step 1: Convert given binary code to decimal value

Step 2: Convert resultant decimal value to BCD code

BCD Addition:

When we perform the BCD addition following scenario can occurs then we have to deal with them.

Case 1: sum <=9 and final carry = 0, answer is correct.

Case 2: sum<=9 and final carry = 1, answer is incorrect,

Then, add 6 (0110) to the result.

Case 3: sum>9, answer is incorrect,

Then add 6 (0110) to the result.

Example: Perform BCD addition to following

a. 5 + 3 b. 8 + 9 c. 7 + 3

Solution:

a. 5 = 0101 3 = 0011

0101
0011

1000 <9 and carry =0 so answer is correct
Hence result is $(1000)_{BCD}$

b. 8 = 1000 9 = 1001

1000
1001
---
1 0001<9 and carry = 1 so answer is incorrect so add 0110 to the answer

10001
0110

10111
Hence the final result is $(10111)_{BCD}$

Alphanumeric codes:

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other symbols.

The alphanumeric codes are the codes that represent numbers and alphabetic characters. Mostly such codes also represent other characters such as symbol and various instructions necessary for conveying information. An alphanumeric code should at least represent 10 digits and 26 letters of alphabet i.e. total 36 items. The following three alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).

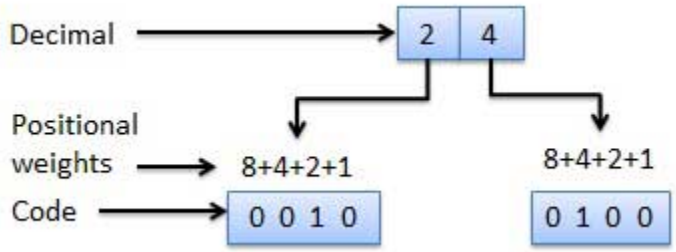
ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

Some of the character and their corresponding ASCII and Binary codes are:

Character	ASCII Code	7 bit Binary Code
A	65	1000001
B	66	1000010
C	67	1000011
D	68	1000100
a	97	1100001
b	98	1100010
c	99	1100011
@	64	1000000
+	43	0101011
%	37	0100101
&	38	0100110

Weighted Codes:

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits.



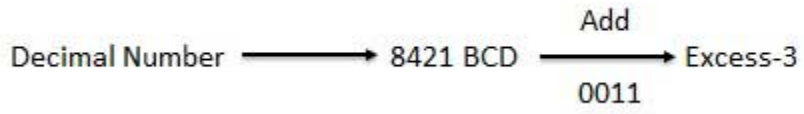
Non-Weighted Codes

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are

- Excess-3 code
- Gray code

Excess-3 code

The Excess-3 code is also called as XS-3 code. It is non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding $(0011)_2$ or $(3)_{10}$ to each code word in 8421. The excess-3 codes are obtained as follows:



Example: Following table shows the BCD and corresponding XS-3 codes

Decimal	BCD 8 4 2 1	Excess-3 BCD + 0011
0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 1 0 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 0 0
6	0 1 1 0	1 0 0 1
7	0 1 1 1	1 0 1 0
8	1 0 0 0	1 0 1 1
9	1 0 0 1	1 1 0 0

Gray Code:

It is the non-weighted code and it is not arithmetic codes. That means there are no specific weights assigned to the bit position. It has a very special feature that, only

one bit will change each time the decimal number is incremented as shown in fig. As only one bit changes at a time, the gray code is called as a unit distance code. The gray code is a cyclic code. Gray code cannot be used for arithmetic operation.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

Application of Gray code

- Gray code is popularly used in the shaft position encoders.
- A shaft position encoder produces a code word which represents the angular position of the shaft.

Conversion:

- Binary to Gray
- Gray to Binary
- BCD to Gray
- Gray to BCD

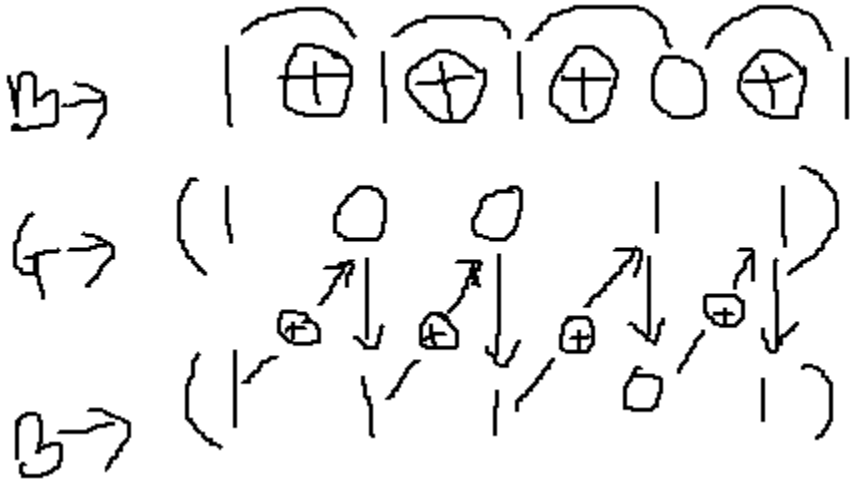
Binary to Gray / BCD to Gray

To convert Binary to Gray: Take first bit as it is and perform XOR operation of adjacent digits of given binary or BCD codes.

Gray Code = B_i XOR B_{i+i}

To convert Gray to binary: Take first bit as it is and perform XOR operation of resultant binary bit and preceding gray code

Binary code = Result XOR G_{i+1}



Assignment:

1. What is digital computer? Explain components of digital computer with block diagram.
2. If A = $(101)_2$ and B = $(11)_2$ then perform A-B using 2's complement method.
3. If A = 61 and B =32 then perform B-A using 2's complement.
4. If A = 37 and B = -18 then perform B-A using 2's complement
5. If A = 127 and B= 15 then calculate (-A) + (-B) using 2's complement.
6. Convert the following
    a. $(1893.22)_{10}$ to Octal
    b. $(603.25)_8 = (?)_{BCD}$
    c. (1110)gray=(?)BCD
    d. $(1430)_{10}$=(?)Excess-3
    e. $(101001001)_2$=$(?)_{gray}$
    f. $(93)_{10}$=(?)Excess-3

g. $(AB.0F)_{16}$ to Binary
h. $(67.51)_8$ to Hexadecimal
i. $(1001.011)_{10}$ to Binary
j. $(2040.0001953125)_{10}$ to binary, octal and hexadecimal

7. Define positional number system. Subtract 21 from 35 using 2's complement method.
8. Subtract 675.6 – 456.4 using both 10^{th} and 9^{th} complement.
9. Subtract 1010.110-101.101 using both 2's and 1's complement.
10. Convert $(62.75)_{10}$ into single precision floating point format.
11. Find binary, hexadecimal and BCD equivalents of $(45.3125)_{10.}$
12. Subtract$(1110.111)_2$ - (1010.101) using 2's complement
13. Subtract using 9's complement$(453.35)_{10}$ - $(321.17)_{10}$
14. Subtract using 1's complement$(1010000)_2 – (1000100)_2$
15. What is the weight of 0 in binary number 10111?
16. Decimal numbers are weighted number. Justify it.
17. Why alphabets are used to represent number above 9 in hexadecimal number system?

End of Unit-1

Unit-2: Boolean Algebra and Logic Gates

Digital Logic:

Digital logic is the underlying logic system that drives electronic circuit board design. Digital logic is the manipulation of binary values through printed circuit board technology that uses circuits and logic gates to construct the implementation of computer operations.

Basic Operation

Digital logic has three basic operators, the AND, the OR and the NOT. These three operators form the basis for everything in digital logic. In fact, almost everything your computer does can be described in terms of these three operations. All most all operations can perform in digital logic by using these basic operators.

AND Operator:

The symbol for an AND operator is a single dot (.) sign. A mathematical expression using AND is (A.B).

The output of an AND expression is 1 only if both input values are 1. Otherwise, the output is 0.

Truth Table of AND Logic

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR Operator:

The symbol for the OR operator is a plus (+) sign. A mathematical expression using OR is (A+B).

The output of an OR expression is 0 only if both the input values are 0. Otherwise the output is 1.

Truth Table for the OR Logic

A	B	A+B
0	0	0
0	1	1

1	0	1
1	1	1

NOT Operator:

NOT is a unary operator, which requires only single input, while as AND and OR are binary operators as they require two values as input. Symbol for NOT operator or **A'**. The output of a NOT expression is the opposite of the input value.

Truth Table of NOT Logic

A	A’
0	1
1	0

NAND and NOR Operator:

If the AND, OR and NOT operators are combined, then the NOR and NAND can be created:

NAND Operator:

A NAND **B** denoted by expression **(A.B)’** this is the inverted (*complement*) output of the AND gate.

NOR Operator:

A NOR **B** denoted by expression **(A+B)’** this is the inverted (*complement*) output of an OR gate.

Truth Table of NAND and NOR Logics

NOR

NAND

X	Y	Q (x.y)'
0	0	1
0	1	1
1	0	1
1	1	0

X	Y	Q (x+y)'
0	0	1
0	1	0
1	0	0
1	1	0

XOR and XNOR Operator:

Two other important logical operators are the ***Exclusive-OR (XOR)*** and ***Exclusive-NOR (XNOR)***. This is also denoted by a plus (+) sign in a circle for ***XOR*** and dot (.) sign in a circle for ***XNOR***.

XOR Operator:

A XOR **B** denoted by expression $A \oplus B$.This is true only if exactly one of the inputs is one. In another word it gives output as zero when two inputs are same; otherwise it gives output as one.

Above expression is simplified as

$$A \oplus B = A\overline{B} + \overline{A}B$$

XNOR Operator:

A XNOR **B** denoted by expression $A \odot B$. This is the inverted (*complement*) output of XOR logic. It produces output as one if both input are the same; otherwise it produces output as zero.

Above expression is simplified as

$$A \odot B = AB + \bar{A}\bar{B}$$

Truth Table of XOR and XNOR logic

XOR

X	Y	Q
0	0	0
0	1	1
1	0	1

XNOR

X	Y	Q
0	0	1
0	1	0
1	0	0
1	1	1

1	1	0

Class work: Prove following equation using truth table

$$A \oplus B = A\overline{B} + \overline{A}B$$

$$A \odot B = AB + \bar{A}\bar{B}$$

Digital Logic Gates:

Logic gates are used to develop digital logic circuits. Logic gates can be classified in three categories:

- Basic Gates
- Universal Gates
- Special Gates

Basic Gates:

Basic gates are used to perform basic operation like AND, OR, and NOT. There are three types of basic gates:

- AND Gate
- OR Gate
- NOT Gate

AND Gate:

AND Gate perform AND operation. In this operation output of logic gate is high only when both inputs are high, otherwise output will be low.

Symbol of AND Gate:

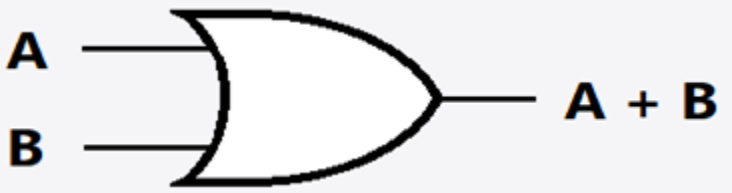
Truth Table of AND Gate:

Inputs		Output
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate:

OR Gate perform OR operation. In this operation output of logic gate is low only when both inputs are low, otherwise output will be high.

Symbol of OR Gate:



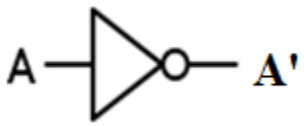
Truth Table of OR Gate:

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

NOT Gate:

NOT gate perform the NOT operation. In this operation output of logic gate is inverse (opposite) of the input. This is also called the invertor.

Symbol of NOT Gate:



Truth Table of NOT Gate:

Input	Output
A	A'
0	1
1	0

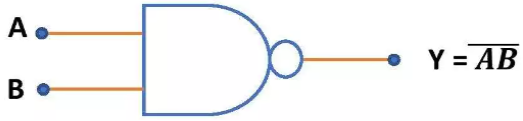
Universal Gate:

NAND and NOR gate are called universal gates because we can design any kind of gates using these two gates.

NAND Gate:

The Logic NAND Gate is a combination of AND gate and a NOT gate connected together in series. The output of NAND gate is low only when both inputs are high otherwise output will be high. This is the inverse of AND gate.

Symbol of NAND Gate:



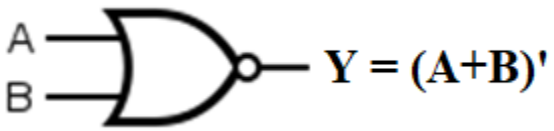
Truth Table:

Inputs		Output
A	B	(A.B)'
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate:

The Logic NOR Gate is a combination of OR gate and a NOT gate connected together in series. The output of NOR gate is high only when both inputs are low otherwise output will be low. This is the inverse of OR gate.

Symbol of NOR Gate:



Truth Table:

Inputs		Output
A	**B**	**(A+B)'**
0	0	1
0	1	0
1	0	0
1	1	0

Special Gate:

There two types of other gates which we called special gates. Those are X-OR and X-NOR

Exclusive-OR (X-OR):

An X-OR gate is a digital logic gate with two or more inputs and one output that performs exclusive disjunction.

The output of an X-OR gate is high only when exactly one of its inputs is high. In other word, if both of an X-OR gate's inputs are same i.e. either 0 or 1, then the output of the X-OR gate is low.

Symbol of X-OR Gate:

Logical Expression:

$$A \oplus B = A\overline{B} + \overline{A}B$$

Truth Table:

Inputs		Output
A	**B**	**A X-OR B**
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR (X-NOR):

An X-NOR is a digital logic gate with two or more inputs and one output. The X-NOR gate output will be HIGH when both the inputs are same and will become LOW for different combination of input. This is the inverse of X-OR gate.

Symbol of X-NOR Gate:

Logical Expression:

$$A \odot B = AB + \bar{A}\bar{B}$$

Truth table:

Inputs		Output
A	**B**	**A X-NOR B**

0	0	1
0	1	0
1	0	0
1	1	1

Logic Circuit Design using Basic Gates:

1. XOR Gate: A'B + AB'
2. XNOR Gate: A'B' + AB
3. AB + A'B + A'B'
4. (A+B) (A'+B')
5. A+BC

Logic Circuit Design using Universal Gates:

The minimum numbers of **NAND** and **NOR** Gates required to design the gates are:

Gates	NAND	NOR
NOT	1	1
AND	2	3
OR	3	2
XOR	4	5
XNOR	5	4
NAND	-	4
NOR	4	-

Design all the Gates using NAND Gate only

1. NOT
2. AND
3. OR
4. NOR
5. XOR
6. XNOR

Design all the Gates using NOR Gate only

1. NOT
2. OR
3. AND
4. NAND
5. XOR
6. XNOR

Boolean algebra:

Boolean algebra is the category of algebra in which the variable's values are the truth values, true and false, ordinarily denoted 1 and 0 respectively. It is used to analyze and simplify digital circuits or digital gates. It is also called Binary Algebra or logical Algebra. It has been fundamental in the development of digital electronics and is provided for in all modern programming languages.

Boolean algebra differs from both general mathematical algebra and binary number systems. In Boolean algebra, **A+A =A** and **A.A = A**, because the variable **A** has only logical value. It doesn't have any numerical significance.

In ordinary mathematical algebra, **A+A = 2A** and **A.A = A^2**, because the variable **A** has some numerical value here. In Binary Number System **1+1 = 10**, and in general mathematical algebra **1+1 = 2,** but in **Boolean algebra 1+1 = 1**. Unlike ordinary algebra and Binary Number System here is no subtraction or division in Boolean algebra. We only use **AOI (AND, OR and NOT/INVERT)** logic operations to perform calculations in Boolean algebra.

Boolean Algebra Terminologies:

Now, let us discuss the important terminologies covered in Boolean algebra.

Boolean algebra: Boolean algebra is the branch of algebra that deals with logical operations and binary variables.

Boolean Variables: A Boolean variable is defined as a variable or a symbol, generally an alphabet that represents the logical quantities such as 0 or 1.

Boolean Function: A Boolean function consists of binary variables, logical operators, constants such as 0 and 1, equal to operator, and the parenthesis symbols.

Literal: A literal may be a variable or a complement of a variable.

Complement: The complement is defined as the inverse of a variable, which is represented by a bar over the variable.

Truth Table: The truth table is a table that gives all the possible values of logical variables and the combination of the variables. It is possible to convert the Boolean equation into a truth table. The number of rows in the truth table should be equal to 2^n, where "n" is the number of variables in the equation.

For example, if a Boolean equation consists of 3 variables, then the number of rows in the truth table is 8 (i.e.,) $2^3 = 8$.

Rules in Boolean Algebra:

Following are the important rules used in Boolean algebra.

- Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
- Complement of a variable is represented by an overbar (-), this is NOT Operation. Thus, complement of variable B is represented as $\overline{B}$. Thus if B = 0 then $\overline{B}$ = 1 and B = 1 then $\overline{B}$ = 0.
- ORing of the variables is represented by a plus (+) sign between them. For example ORing of A, B, C is represented as A + B + C.
- Logical ANDing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometime the dot may be omitted like ABC.

Axiomatic Definition of Boolean algebra:

There are some set of logical expressions which we accept as true and upon which we can build a set of useful theorems. These sets of logical expressions are known as **Axioms or postulates of Boolean algebra**. An axiom is nothing more than the definition of three basic logic operations (AND, OR and NOT). All axioms defined

in Boolean algebra are the results of an operation that is performed by a logical gate.

Some basic axioms or postulates of Boolean algebra are:

Axiom 1: 0.0 = 0

Axiom 2: 0.1 = 0

Axiom 3: 1.0 = 0

Axiom 4: 1.1 = 1

Axiom 5: 0+0 = 0

Axiom 6: 0+1 = 1

Axiom 7: 1+0 = 1

Axiom 8: 1+1 = 1

Axiom 9: 0 = 1

Axiom 10: 1 = 0

Basic Theorems and Properties of Boolean algebra:

There are six types of Boolean algebra laws. They are:

- Commutative law
- Associative law
- Distributive law
- AND law
- OR law
- Inversion law

Commutative law:

Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

First Law: A+B = B+A

Second Law: A.B = B.A

Truth table for First Law i.e. A+B = B+A

Inputs		Outputs	
A	**B**	**A+B**	**B+A**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

This truth table proves that A+B = B+A

Logic Circuit:

Truth table for Second Law i.e. A.B = B.A

Inputs		Outputs	
A	B	A.B	B.A
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

This truth table proves that A.B = B.A

Logic Circuit:

Associative Law:

It states that the order in which the logic operations are performed is irrelevant as their effect is the same.

First Law: (A+B)+C = A+(B+C)

Second Law: (A.B).C = A.(B.C)

Truth table for First Law i.e. (A+B)+C = A+(B+C)

Inputs			Outputs			
A	B	C	A+B	(A+B)+C	B+C	A+(B+C)
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1

1	1	0	1	1	1	1
1	1	1	1	1	1	1

This truth table proves that (A+B)+C = A+(B+C)

Logic Circuit:

Truth table for Second Law i.e. (A.B).C = A.(B.C)

Inputs			Outputs			
A	**B**	**C**	**A.B**	**(A.B).C**	**B.C**	**A.(B.C)**
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

This truth table proves that (A.B).C = A.(B.C)

Logic Circuit:

Distributive Law:

Distributive law states the following conditions:

First Law: A. (B + C) = (A. B) + (A. C)

Second Law: A + (B. C) = (A + B) . (A + C)

Truth table of First Law i.e. A.(B + C) = (A. B) + (A. C)

Inputs			Outputs				
A	**B**	**C**	**B+C**	**A.(B+C)**	**A.B**	**A.C**	**(A.B)+(A.C)**
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0

0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Hence, truth table proved that A.(B + C) = (A. B) + (A. C)

Logic Circuit:

Truth table of Second Law i.e. A + (B. C) = (A + B) . (A + C)

Inputs			Outputs				
A	**B**	**C**	**B.C**	**A+(B.C)**	**A+B**	**A+C**	**(A+B).(A+C)**
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Hence, truth table proved that A + (B. C) = (A + B) . (A + C)

Logic Circuit:

AND Law:

These laws use the AND operation. Therefore they are called AND laws.

- A .0 = 0
- A . 1 = A
- A. A = A
- $A.\bar{A} = 0$

A.0 = 0 (If A = 0, then 0.0 = 0 and when A=1, 1.0 = 0, Hence the expression will always be 0 regardless of the value of A)

A.1 = A (If A = 0 then 0.1 = 0 and when A=1, 1.1=1.Hence the expression will be equal to value of A always)

A.A = A (If A = 0 then 0.0 = 0 and when A = 1 then 1.1 = 1)

A.A' = 0 (if A = 0 then 0.1 = 0 and when A = 1 then 1.0 = 0)

OR Law:

These laws use the OR operation. Therefore they are called OR laws.

- A + 0 = A
- A + 1 = 1
- A + A = A
- $A + \bar{A} = 1$

0+A = A (If A = 0. then 0+0 = 0 and when A=1, 0+1=1. Hence the expression will be equal to value of A always)

A+1 = 1 (If A = 0 then 0+1 =1 and when A=1, 1+1 =1,Hence the expression will always be 1 regardless of the value of A)

A+A = A (If A = 0 then 0+0 = 0 and when A =1 then 1+1 = 1)

A+A' = 1 (if A = 0 then 0+1 = 1 and when A = 1 then 1+0 = 1)

Inversion Law:

In Boolean algebra, the inversion law states that double inversion of variable results in the original variable itself.

- $\bar{\bar{A}} = A$

Boolean Algebra Theorems:

Theorem 1:

DeMorgan's 1st Law:

The complement of the product of variables is equal to the sum of their individual complements.

$\overline{A.B} = \overline{A} + \overline{B}$

The truth table that shows the verification of De Morgan's First law is given as follows:

A	B	A'	B'	(A.B)'	A'+B'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

The last two columns show that (A.B)' = A'+B'

Hence, De Morgan's First Law is proved.

Logic Circuit:

DeMorgan's 2nd Law:

The complement of the sum of variables is equal to the product of their individual complements.

$$\overline{A+B} = \overline{A} \cdot \overline{B}$$

The following truth table shows the proof for De Morgan's second law.

A	B	A'	B'	(A+B)'	A'. B'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

The last two columns show that (A+B)' = A'. B'.

Hence, De Morgan's second law is proved.

Logic Circuit:

Theorem 2:

Absorption Law:

This law states that,

First Law: A+AB = A

Second Law: A (A+B) = A

Proof of First Law: A+AB = A

LHS: A+AB

= A.1 + AB [A.1 = A : AND Law]

= A (1+B)

= A.1 [A+1 = 1 : OR Law]

= A

LHS = RHS hence proved

Truth Table of First Law: A+AB = A

A	B	AB	A+AB
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

This truth table proved A+AB = A

Logical Circuit:

Proof of Second Law: A (A+B) = A

LHS = A(A+B)

= AA +AB

= A+AB [AA = A : AND Law]

= A(1+B)

= A.1 [B+1 = 1 : OR Law]

= A

LHS = RHS hence proved

Truth Table of First Law: A(A+B) = A

A	B	A+B	A(A+B)
0	0	0	0
0	1	1	0
1	0	1	1

1	1	1	1

This truth table proved A(A+B) = A

Logic Circuit:

Theorem 3:

Redundancy or Common Identity Law:
First Law: A+ĀB = A+B

Second Law: A.(Ā+B) = AB

Proof of First Law: A+ĀB = A+B

LHS = A+A'B

= (A+A') (A+B) [A+BC = (A+B)(A+C) : Distributive Law]

= 1(A+B) [A+A' = 1 : OR Law]

= A+B

LHS = RHS hence proved

Truth Table:

Logic Circuit:

Proof of Second Law: A.(Ā+B) = AB

LHS = A(A'+B)

= AA' + AB

= 0 + AB [A.A' = 0 : AND Law]

= AB

LHS = RHS hence proved

Truth Table:

Logic Circuit:

Principle of Duality:

The principle of duality states that every valid Boolean expression remain valid if the operators and identity elements are interchanged as follows

+ ⇝ > *

1 ⇝ > 0

For Example:

1+0 = 0.1

X+1 = X.0

A+B.C = A.(B+C)

Boolean Simplification:

Simplify the following Boolean expression using postulates and theorems
Q1. (A+B).(A+B')

= (A+B).(A+B')

= AA +AB'+BA+BB'

=A+AB'+BA+0

=A+A(B'+B)

=A+A.1

=A+A

=A

Q2. ABC+AB'C+ABC'

= ABC+AB'C+ABC'

=AC(B+B')+ABC'

=AC.1+ABC'

=AC+ABC'

=A(C+BC')

=A(C+B) [A+A'B= A+B: Redundancy Law]

Q3. ABC+AB'C+ABC'+AB'C'

= ABC+AB'C+ABC'+AB'C'

= AC(B+B') + AC'(B+B')

= AC.1+AC'.1

= AC+AC'

= A(C+C')

= A.1

= A

Q4. AB +A'C + BC

= AB +A'C + BC

= AB+A'C+BC(A+A')

= AB+A'C+ABC+A'BC

= AB+ABC+A'C+A'BC

= AB(1+C)+A'C(1+B)

= AB.1+A'C.1 [1+A = 1 : OR Law]

= AB+A'C

Q5. AB'C'+AB'C'D+AC'

= AB'C'+AB'C'D+AC'

= AB'C'(1+D)+AC'

= AB'C'.1+AC'

= AB'C'+AC'

= AC'(B'+1)

= AC'.1

= AC'

Prove the following expression using Boolean postulates and theorems

Q1. X+X = X

LHS = X+X

= (X+X).1

= (X+X).(X+X')

= X+XX' [A+BC =(A+B)(A+C) : Distributive Law]

= X+0

= X

LHS = RHS proved

Q2. X.X = X

LHS = X.X

= X.X + 0

= X.X + X.X'

= X (X+X')

= X. 1

= X

LHS = RHS proved

Q3. X+1 = 1

LHS = X+1

= (X+1).1

= (X+1) (X+X')

= X+1X' [A+BC = (A+B)(A+C) : Distributive Law]

= X+X'

= 1

LHS = RHS proved

Q4. AB+AB'C+A'BC = AB+AC+BC

LHS = AB+AB'C+A'BC

= A(B+B'C)+A'BC

= A (B+C)+A'BC [A+A'B=A+B: Redundancy Law]

= AB+AC+A'BC

= AB+C(A+A'B)

= AB+C(A+B)

= AB+AC+BC

LHS=RHS proved

Q5. (X+Y)(X+Z) = X+YZ

LHS = (X+Y)(X+Z)

= XX+XZ+XY+YZ

= X+XZ+XY+YZ

= X(1+Z)+XY+YZ

= X.1+XY+YZ

= X+XY+YZ

= X(1+Y)+YZ

= X.1+YZ

= X+YZ

LHS = RHS proved

Q6. A+AB = A

LHS = A+AB

= A(1+B)

= A.1

= A

LHS = RHS proved

Q7. A+A'B = A+B

LHS = A+A'B

= A+AB+A'B [A+AB = A : Absorption Law]

= A+B(A+A')

= A+B.1 [A+A' = 1 : OR Law]

= A+B

LHS = RHS proved

Q8. XY+XZ+YZ' = XZ+YZ'

LHS = XY+XZ+YZ'

= XY(Z+Z')+XZ(Y+Y')+YZ'(X+X')

= XYZ+XYZ'+XYZ+XY'Z+XYZ'+X'YZ'

= XYZ+XYZ'+XY'Z+X'YZ' [Eliminating the repeat term]

= XYZ+XY'Z+XYZ'+X'YZ'

= XZ(Y+Y')+YZ'(X+X')

= XZ.1+YZ'.1

= XZ+YZ'

LHS = RHS proved

Q9. A(A'+B) = AB

LHS = A(A’+B)

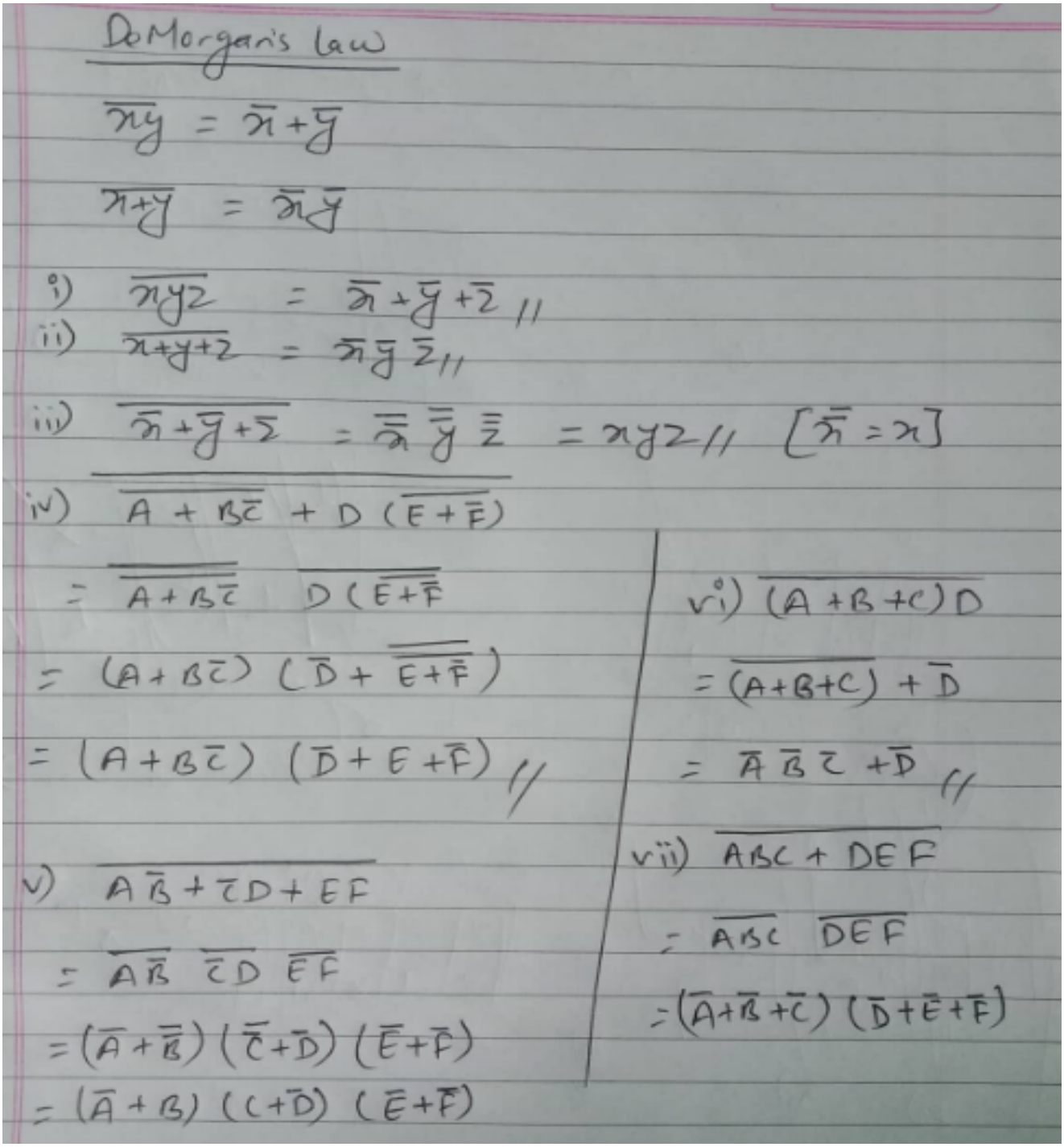
= AA’+AB

= 0+AB

= AB

LHS = RHS proved

Apply DeMorgan's Law to following expressions



Boolean Functions and Expressions

The binary variables and logic operations are used in Boolean algebra. The algebraic expression is known as ***Boolean Expression,*** is used to describe the ***Boolean Function.*** The Boolean expression consists of the constant value 1 and 0, logical operation symbols, and binary variables.

Example 1: f=xy'z+p

We defined the Boolean function f=xy'z+p in terms of four binary variables x, y, z, and p. This function will be equal to 1 when x=1, y=0, z=1 or z=1.

Example 2:

$$F(A,B,C,D) = A + \overline{BC} + ACD \qquad Equation\ No.\ 1$$

Boolean Function Boolean Expression

The output Y is represented on the left side of the equation. So,

$$Y = A + \overline{BC} + ACD$$

Apart from the algebraic expression, the Boolean function can also be described in terms of the truth table. We can represent a function using multiple algebraic expressions. They are their logically equivalents. But for every function, we have only one unique truth table.

In truth table representation, we represent all the possible combinations of inputs and their result. We can convert the switching equations into truth tables.

Example: F(A,B,C,D)=A+BC'+D

The output will be high when A=1 or BC'=1 or D=1 or all are set to 1. The truth table of the above example is given below. The 2^n is the number of rows in the truth table. The n defines the number of input variables. So the possible input combinations are 2^3=8.

Inputs				Output
A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Canonical and Standard Forms:

Min Term:

Product of all input literals are appearing either in normal or complement form in output expression is called Min Term. This is represents in Sum of Product (SOP) form. We will get four Boolean product terms by combining two variables x and y with logical AND operation.

The min terms are x'y', x'y, xy' and xy.

In Min Term Boolean variables are represents by 1 and complement variables are represents by 0. Min Term is denoted by sign Sigma or lowercase letter 'm'.

The output of the Min Term should be 1.

Example:

F(AB) = A'B+AB

Min term representation of above expression is

= m(01)+m(11)

F = m0+m1

OR

F = □m(0,1)

Max Term:

Sum of all input literals are appearing either in normal or complement form in output expression is called Max Term. This is represents in Product of Sum (POS) form. We will get four Boolean sum terms by combining two variables x and y with logical OR operation.

The Max terms are x + y, x + y', x' + y and x' + y'.

In Max Term Boolean variables are represents by 0 and complement variables are represent by 1. Max Term is denoted by Sign Pie or uppercase letter M.

Output of the Max Term should be 0.

Example:

F(AB) = (A+B).(A'+B')

Max Term representation of above expression is

= M(00).M(11)

F = M0.M1

OR

F= πM(0,1)

The following table shows the representation of min terms and MAX terms for 2 variables.

x	Y	Min terms	Max terms
0	0	m0=x'y'	M0=x + y
0	1	m1=x'y	M1=x + y'
1	0	m2=xy'	M2=x' + y
1	1	m3=xy	M3=x' + y'

From the above table, we can easily notice that Min terms and Max terms are complement of each other. If there are 'n' Boolean variables, then there will be 2^n Min terms and 2^n Max terms.

SOP (Sum of Product):

F(A,B) = AB+AB'+A'B+A'B'

F(ABC) = ABC+AB+ABC'

POS (Product of Sum):

F(AB) = (A+B)(A'+B)(A+B')

F(ABC) = (A+B+C)(A'+B'+C)(A+B)

Literal:

All the Boolean variables including complement

Eg: if A is a variables the its literals are A and A'

Canonical Form:

To be a canonical form all the input literals (variable or complement of variable) should be present in output expression.

Example:

F(AB) = AB+B'A+A'B' this is canonical form, all literals are included

F(AB) = AB+A'B+A this is not a canonical form because B is missing in last term

We can express each output variable in following two ways.

- Canonical SoP (Sum of Product) form
- Canonical PoS (Product of Sum) form

Canonical SoP form:

Canonical SoP form means Canonical Sum of Products form. In this form, each product term contains all literals. So, these product terms are nothing but the min terms. Hence, canonical SoP form is also called as **sum of min terms** form.

Example:

F(pqr) = pqr + pqr' + pq'r + p'qr

This is the canonical SOP form

This can be express as:

f=m(111)+m(110)+m(101)+m(011)

f=m7+m6+m5+m3

f=m3+m5+m6+m7 ---------- Equation 1

f=∑m(3,5,6,7)f=∑m(3,5,6,7) ---------- Equation 2

In one equation, we represented the function as sum of respective min terms. In other equation, we used the symbol for summation of those min terms.

Canonical PoS form:

Canonical PoS form means Canonical Product of Sums form. In this form, each sum term contains all literals. So, these sum terms are nothing but the Max terms. Hence, canonical PoS form is also called as **product of Max terms** form.

Example:

Therefore, the Boolean function of output is,

$F(pqr) = (p+q+r).(p+q+r').(p+q'+r).(p'+q+r)$

This is the **canonical PoS form** of

We can also represent this function in following two notations.

$f=M_0.M_1.M_2.M_4$

$f=\pi M(0,1,2,4)$

In one equation, we represented the function as product of respective Max terms. In other equation, we used the symbol for multiplication of those Max terms.

The Boolean function,

$F(pqr) = (p+q+r).(p+q+r').(p+q'+r).(p'+q+r)$ …… Equation 1

$F(pqr) = pqr + pqr' + pq'r + p'qr$ …….. Equation 2

Here we can see Equation 1 is dual of Equation 2.

Therefore, both canonical SoP and canonical PoS forms are **Dual** to each other. Functionally, these two forms are same. Based on the requirement, we can use one of these two forms.

Conversion of SOP to Canonical SOP:

Step 1: Identify the missing literal in product term

Step 2: Multiply by missing variable as (Variable + Complement) form

Step 3: Eliminate the repeat terms

Convert the following SOP expression in to Canonical SOP form

F(A,B) = AB+B

Add missing variable as (A+A')

= AB+B(A+A')

= AB+BA+BA'

Eliminate repeated combination because (x+x = x)

= AB+BA'

Now this is in Canonical SOP form.

We can represent is as Min Term

= m(11)+m(10)

= m3+m2

= m2+m3

= □m(2,3)

Conversion of POS to Canonical POS:

Step 1: Identify the missing variable

Step 2: Add all possible combination of literals of missing variables separately

Step 3: Eliminate the repeated terms

Convert the following POS expression in to Canonical POS form

Q1. F(AB) = (A+B).A'

Here B is missing in second term, so possible combination of literals of variable B is B and B' so add B and B' separately to A'

= (A+B)(A'+B)(A'+B')

Now this is in Canonical POS

We can represent it in Max Term

= 00 10 11

= M0.M2.M3

= πM(0,2,3)

Q2. F(ABC) = A(A+C)

Here BC is missing in first term and B is missing in second term. Possible combination of literals of variable BC are B+C, B'+C, B+C', B'+C' and possible combination of literals of variable C is C and C'. Then add those combination of literals to first and second term respectively.

= A(A+C)

= (A+B+C)(A+B'+C)(A+B+C')(A+B'+C')(A+C+B)(A+C+B')

Now eliminate the repeated term

= (A+B+C)(A+B'+C)(A+B+C')(A+B'+C')

This is the canonical POS

We can represent it in Max Term

= 000 010 001 011

= M0 M2 M1 M3

= M0.M1.M2.M3

= πM(0,1,2,3)

Standard forms:

Standard forms are the simplified version of canonical form. Standard output term may or may not contains all the input literals.

We discussed two canonical forms of representing the Boolean outputs. Similarly, there are two standard forms of representing the Boolean outputs. These are the simplified version of canonical forms.

- Standard SoP form
- Standard PoS form

The main **advantage** of standard forms is that the number of inputs applied to logic gates can be minimized. Sometimes, there will be reduction in the total number of logic gates required.

Standard SoP form:

Standard SoP form means **Standard Sum of Products** form. In this form, each product term need not contain all literals. So, the product terms may or may not be the min terms. Therefore, the Standard SoP form is the simplified form of canonical SoP form.

We will get Standard SoP form of output variable in two steps.

- Get the canonical SoP form of output variable
- Simplify the above Boolean function, which is in canonical SoP form

Sometimes, it may not possible to simplify the canonical SoP form. In that case, both canonical and standard SoP forms are same.

Example: Convert the following Boolean function into Standard SoP form.

F(pqr) = p'qr + pq'r + pqr' + pqr

The given Boolean function is in canonical SoP form. Now, we have to simplify this Boolean function in order to get standard SoP form.

= p'qr + pq'r + pqr' + pqr + pqr + pqr

= p’qr + pqr + q’pr + qpr + r’pq + rpq

= qr(p’+p) + pr(q’+q) + pq(r’+r)

= qr+pr+pq

This is the simplified Boolean function.

Therefore, the **standard SoP form** corresponding to given canonical SoP form is

F(pqr) = pq + qr + pr

Standard PoS form:

Standard PoS form means **Standard Product of Sums** form. In this form, each sum term need not contain all literals. So, the sum terms may or may not be the Max terms. Therefore, the Standard PoS form is the simplified form of canonical PoS form.

We will get Standard PoS form of output variable in two steps.

- Get the canonical PoS form of output variable
- Simplify the above Boolean function, which is in canonical PoS form.

Sometimes, it may not possible to simplify the canonical PoS form. In that case, both canonical and standard PoS forms are same.

Example:

Convert the following Boolean function into Standard PoS form.

F(pqr) = (p+q+r).(p+q+r′).(p+q′+r).(p′+q+r)

The given Boolean function is in canonical PoS form. Now, we have to simplify this Boolean function in order to get standard PoS form.

Apply AND law x.x = x this means product of n number of variable is equals to that variable

= (p+q+r) (p+q+r) (p+q+r).(p+q+r′).(p+q′+r).(p′+q+r)

= (p+q+r) (p+q+r’) (p+r+q) (p+r+q’) (q+r+p) (q+r+p’)

Applying distributive law i.e. x+yz = (x+y)(x+z)

= (p+q+rr')(p+r+qq')(q+r+pp')

Applying AND law xx' = 0

= (p+q)(p+r)(q+r)

This is the simplified Boolean function.

Therefore, the **standard PoS form** corresponding to given canonical PoS form is

F = (p+q)(p+r)(q+r)

This is the **dual** of the Boolean function, f = pq + qr + pr

Therefore, both Standard SoP and Standard PoS forms are Dual to each other.

Assignments:

1. Explain the basic logic gates with its logic diagram, truth table and Boolean expression
2. What are universal logic gates? Realize NAND and NOR as universal logic gates with graphical symbol, truth table, algebraic expression.
3. Draw the logic circuit of the expression
    a. A+BC+AB
    b. A'B+CA+BC
    c. (A'+B')(A+B)
    d. (AB+BC)(AC+B'C)
4. Draw the logic circuit of the expression using NAND and NOR only
    a. A+B+C
    b. AB+BC
    c. A'B+B'C
    d. (A'+B)(A+B)

5. What is the difference between canonical and standard form? Justify with an example.
6. Convert the following

a. F(abc) = ab+bc'+a in to Canonical SOP
b. F(abc) = (a+b')(a+b+c)(a+c) in to Canonical POS

7. Write Short Notes on:
   a. Min Term
   b. Max Term
   c. Literals
   d. SOP
   e. POS

End of Unit - 2

Unit-3: Simplification of Boolean Function

Introduction:

A Boolean function is a function that has '**n'** input variables or entries, so it has 2^n possible combinations of the output expression. These functions will assume only **0** or **1** in its output.

An example of a Boolean function is,

f(a,b,c) = abc + a'bc + ab'c + a'b'c'

These functions are implemented with the logic gates.

Boolean function can simplify by using following two methods.

- Algebraic Method
- Map Method

Algebraic Method:

In this approach, one Boolean expression is minimized into an equivalent expression by applying Boolean postulates, laws, and theorems.

Example:

F(A,B,C)=(A+B)(A+C)

Solution:

= (A+B)(A+C)

= A.A+A.C+B.A+B.C [Applying distributive Rule]

= A+A.C+B.A+B.C [Applying AND Law]

= A(1+C)+B.A+B.C [Applying distributive Law]

= A.1+B.A+B.C [Applying OR Law]

= A+ B.A+B.C [Applying AND Law]

= A(1+B)+B.C [Applying distributive Law]

= A.1+B.C [Applying OR Law]

= A+B.C [Applying AND Law]

So, F(A,B,C)=A+BC is the minimized form.

Map Method (K-Map):

The Karnaugh map (K–map), introduced by Maurice Karnaughin in 1953, is a grid-like representation of a truth table which is used to simplify boolean algebra expressions. A Karnaugh map has zero and one entries at different positions. It provides grouping together Boolean expressions with common factors and eliminates unwanted variables from the expression.

Types of K-Maps

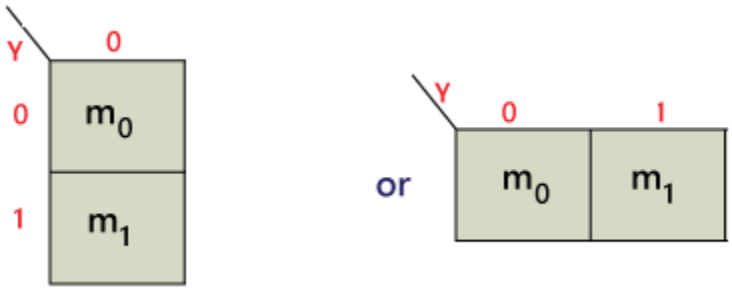
- 1-Variable K-Map
- 2-Variable K-Map
- 3-Variable K-Map
- 4-Variable K-Map

Total numbers of Cells in K-Map is the 2^n, where n is the numbers of input variables.

Eg: if input variables are 2 then cell in k-map will be 2^2 i.e. 4.

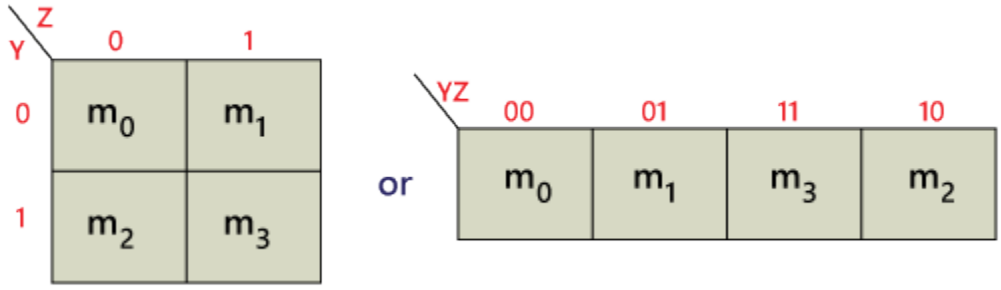
1-Variable K-map:

The number of cells in 1 variable K-map is two (2^1), since the number of variables is one. The following figure shows the structure of 1-variable K-map



2-Variable K-map:

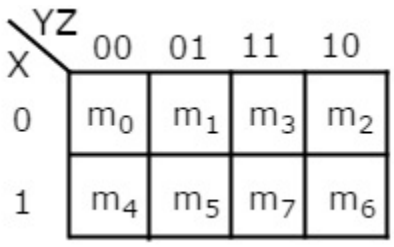
The number of cells in 2 variable K-map is four (2^2), since the number of variables is two. The following figure shows the structure of 2-variable K-map



- In the above figure, there is only one possibility of grouping four adjacent minterms which his (m0, m1, m2, m3).
- The possible combinations of grouping 2 adjacent minterms are {(m_0, m_1), (m_2, m_3), (m_0, m_2) and (m_1, m_3)}.

3-variable K-map:

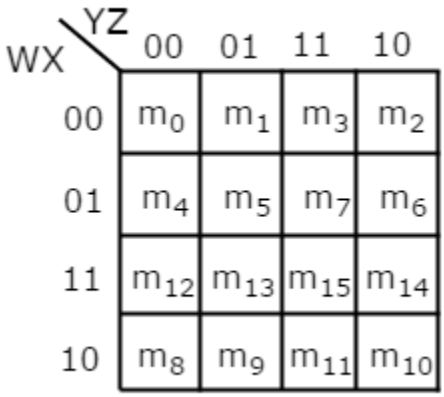
The number of cells in 3 variable K-map is eight (2^3), since the number of variables is three. The following figure shows the structure of 3-variable K-map:



- There is only one possibility of grouping 8 adjacent min terms (m0, m1, m2, m3, m4, m5, m6).
- The possible combinations of grouping 4 adjacent min terms are {(m0, m1, m3, m2), (m4, m5, m7, m6), (m0, m1, m4, m5), (m1, m3, m5, m7), (m3, m2, m7, m6) and (m2, m0, m6, m4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m1, m3), (m3, m2), (m2, m0), (m4, m5), (m5, m7), (m7, m6), (m6, m4), (m0, m4), (m1, m5), (m3, m7) and (m2, m6)}.

4-Variable K-Map:

The number of cells in 4-variable K-map is sixteen (2^4), since the number of variables is four. The following figure shows structure of 4-variable K-map



- There is only one possibility of grouping 16 adjacent min terms.
- Let R1, R2, R3 and R4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C1, C2, C3 and C4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R1, R2), (R2, R3), (R3, R4), (R4, R1), (C1, C2), (C2, C3), (C3, C4), (C4, C1)}.
- Similarly, we can make combination of 4 and 2 adjacent min terms.

Various Implicants in K-Map:

Implicant is a product/minterm term in Sum of Products (SOP) or sum/maxterm term in Product of Sums (POS) of a Boolean function. In other word it is a group of maximum possible group of 1 or 0 in K-map.

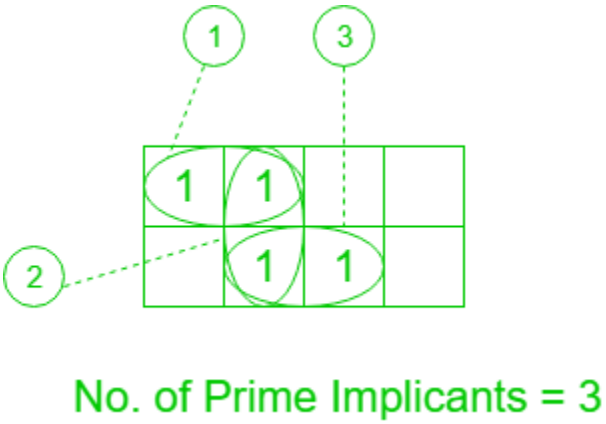
Example: consider a boolean function, F = AB + ABC + BC. Implicants are AB, ABC, and BC.

Types of implicants are discuss bellow:

Prime Implicants (PI):

A group of maximum possible adjacent minterms which is allowed by the definition of K-Map are called prime implecants.

Example:

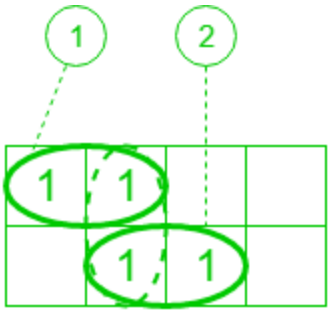


No. of Prime Implicants = 3

Essential Prime Implicants (EPI):

It is a PI where at least one member that can't be covered by any other prime implicant. EPI are those prime implicants that always appear in the final solution.

Example:



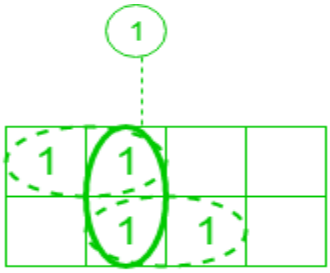
No. of Essential Prime Implicants = 2

In the PI 1 and 2 there are one member in each PI which can't combine with other implecants, so these are EPI.

Redundant Prime Implicants (RPI):

The prime implicants for which each of its minterm is covered by (repeated) some essential prime implicant are **redundant prime implicants (RPI)**. This prime implicant never appears in the final solution.

Example:

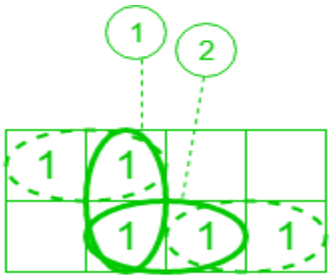


No. of Redundant Prime Implicants = 1

Here in the PI 1, all the members are covered by other EPI.

Selective Prime Implicants (SPI): The prime implicants for which are neither essential nor redundant prime implicants are called **selective prime implicants (SPI)**. These are also known as non-essential prime implicants. They may appear in some solution or may not appear in some solution.

Example:

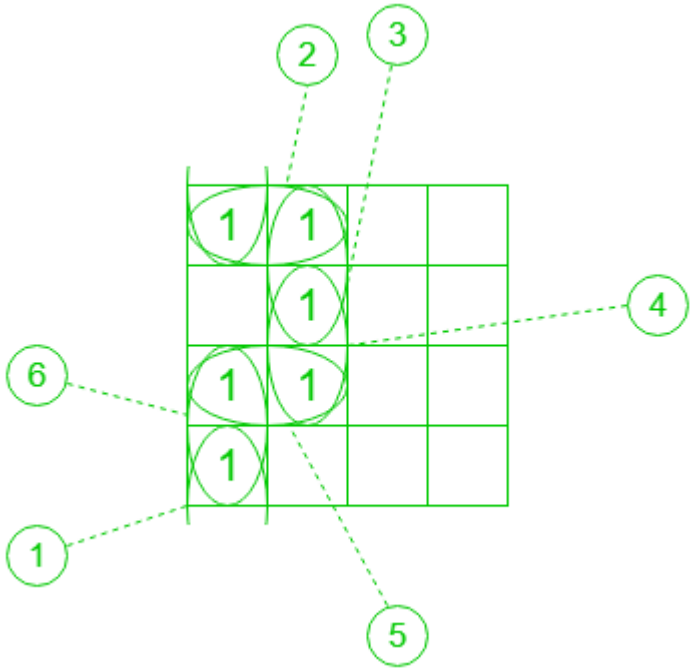


No. of Selective Prime Implicants = 2

Here PI 1 and 2 are, not EPI because all members are covered by other PI, and these are also not RPI because all members are not covered by other EPI. So there are neither EPI nor RPI so these are SPI.

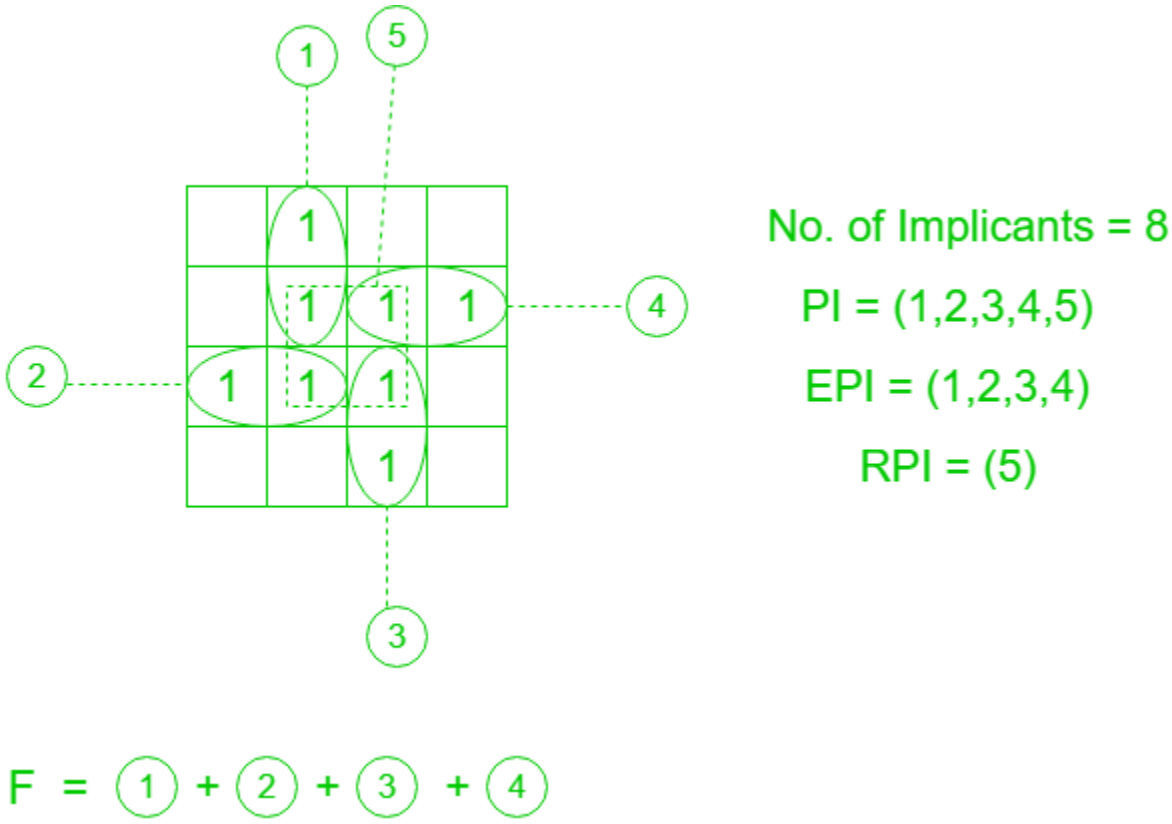
Cyclic Prime Implicant (CPI):

If there is no EPI then this is called CPI. In other word if all the members of PI are covered by other PI then this is called CPI.



Here all the members of PI are covered by other PI.

Example 1: Given F = ∑(1, 5, 6, 7, 11, 12, 13, 15) find number of implicant, PI, EPI, RPI and SPI.



Expression: BD + A'C'D + A'BC+ ACD+ABC'

No. of Implicants = 8 (Because there are 8 minterms)

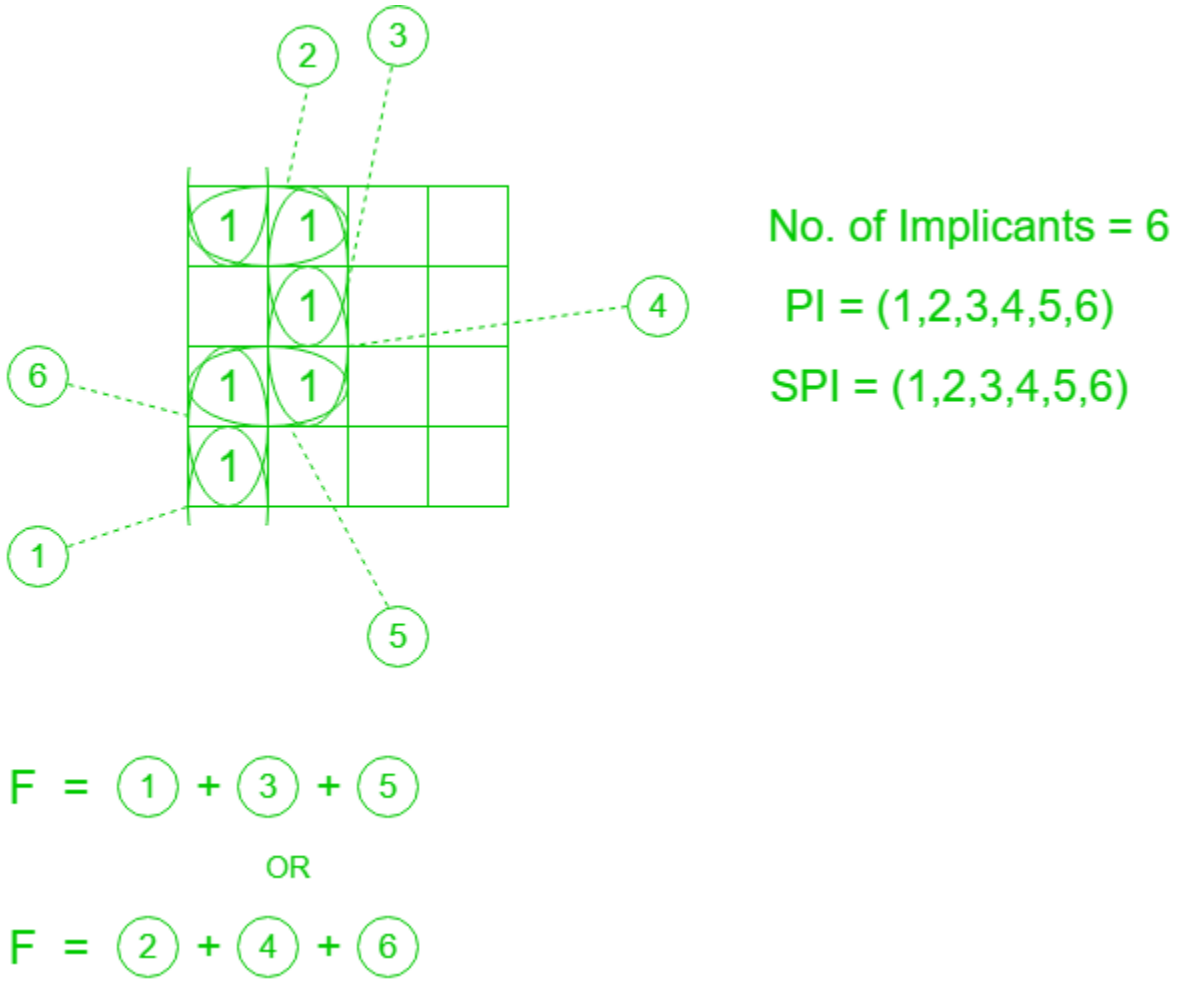
No. of Prime Implicants(PI) = 5

No. of Essential Prime Implicants(EPI) = 4

No. of Redundant Prime Implicants(RPI) = 1

No. of Selective Prime Implicants(SPI) = 0

Example 2: Given F = ∑(0, 1, 5, 8, 12, 13), find number of implicant, PI, EPI, RPI and SPI.



Expression: A'B'C'+ C'DB + C'D'A

No. of Implicants = 6

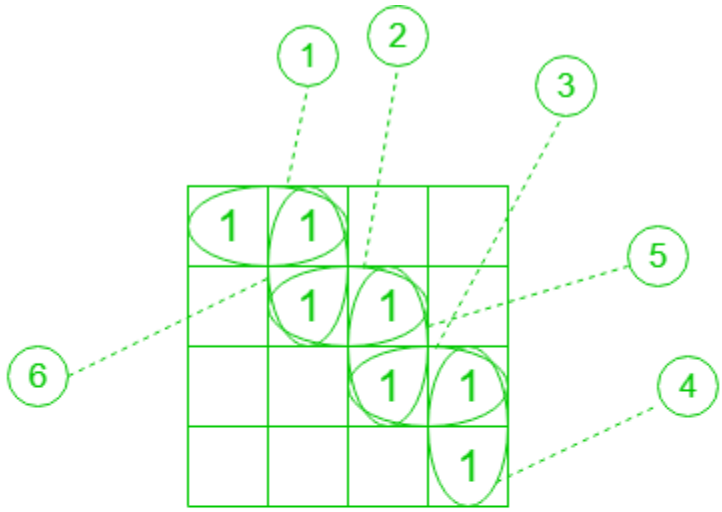
No. of Prime Implicants(PI) = 6

No. of Essential Prime Implicants(EPI) = 0

No. of Redundant Prime Implicants(RPI) = 0

No. of Selective Prime Implicants(SPI) = 6 (If there is no EPI the all are SPI)

Example-3: Given F = ∑(0, 1, 5, 7, 15, 14, 10) find number of implicant, PI, EPI, RPI and SPI.



No. of Implicants = 7

PI = (1,2,3,4,5,6)

EPI = (1,4)

SPI = (2,3,5,6)

F = 1 + 2 + 3 + 4

OR

F = 1 + 5 + 6 + 4

No. of Implicants = 7

No. of Prime Implicants (PI) = 6

No. of Essential Prime Implicants (EPI) = 2

No. of Redundant Prime Implicants (RPI) = 0

No. of Selective Prime Implicants(SPI) = 4

Steps to solve expression using K-map

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1's in blocks of K-map respective to the minterms and (0's elsewhere).
4. For POS put 0's in blocks of K-map respective to the maxterms (1's elsewhere).
5. Make rectangular groups containing maximum number of 1s for SOP and 0s for POS. Group contains 1 or 0 in power of two like 2,4,8… and also can be a group of single 1 or 0 if there is no any other value to combine.
6. After making group,
    a. For the SOP fine the unchanged variable in the group in row and column and write 1 for standard variable and 0 for complement variables.
    b. For the POS fine the unchanged variable in the group in row and column and write 0 for standard variable and 1 for complement variables.

Unit-4: Combinational Circuit

Parallel Adder:

The Parallel binary adder is a combinational circuit consists of various full adders in parallel structure so that when more than 1-bit numbers are to be added, then there can be full adder for every column for the addition. The number of full adders in a parallel binary adder depends on the number of bits present in the number for the addition. If 4-bits numbers are to be added, then there will be 4-full adder in the parallel binary adder.

The Binary Adder is formed with the help of the Full-Adder circuit. The Full-Adders are connected in series, and the output carry of the first Adder will be treated as the input carry of the next Full-Adder.

2-bit Parallel Adder:

2-bit parallel adder used to add two 2-bit numbers. For the addition of 2-bit numbers we need two full adders.

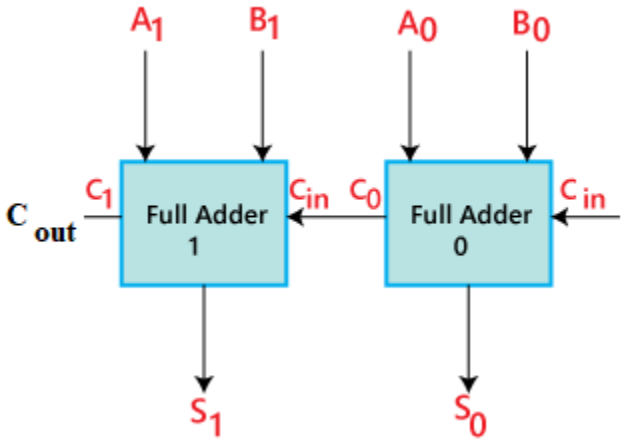


Fig: 2-Bit Parallel Binary Adder

When we start addition of two numbers, the first step we follow is the addition of LSB (Least Significant Bits) of two numbers. After this, if we have any carry we forward it to higher order columns. Now, the adder performs the similar task; it adds the LSBs of both the numbers and if any carry bit is there it passes it to the carry-in terminal of another.

You may use half adder for the addition of LSBs of both the numbers as for the addition of LSBs there is no previous carry from previous addition. But for the addition of bits present in higher order column, you must use full adder because there may be or may not be a carry from previous addition.

4-Bit Parallel Adder:

4-bit parallel adder is used to add two 4-bit numbers. In the 4-bit parallel adder we need to combine 4 full adders in series.

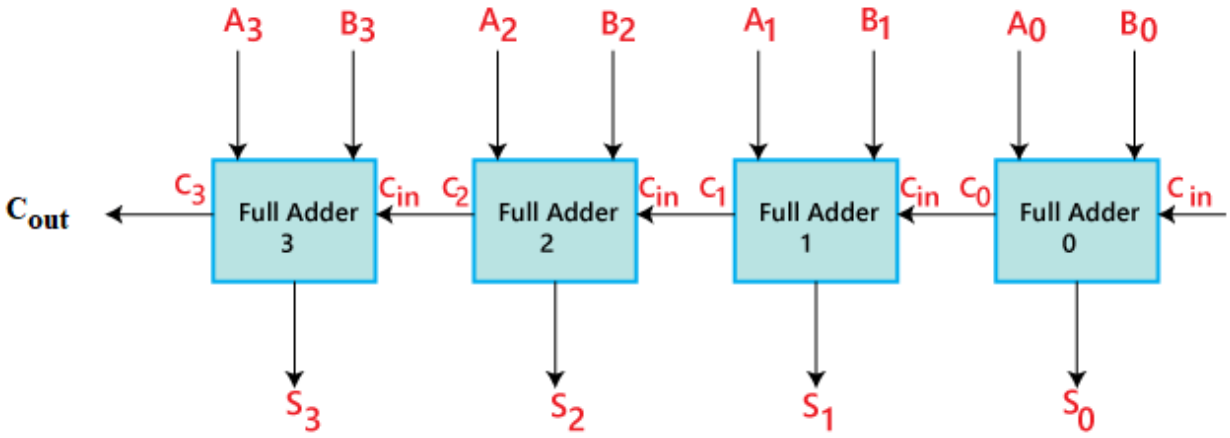


Fig: 4-bit Parallel Binary Adder

When we start addition of two numbers, the first step we follow is the addition of LSB (Least Significant Bits) of two numbers. After this, if we have any carry we forward it to higher order columns. Now, the adder performs the similar task; it adds the LSBs of both the numbers and if any carry bit is there it passes it to the carry-in terminal of another.

N-bit Parallel Adder:

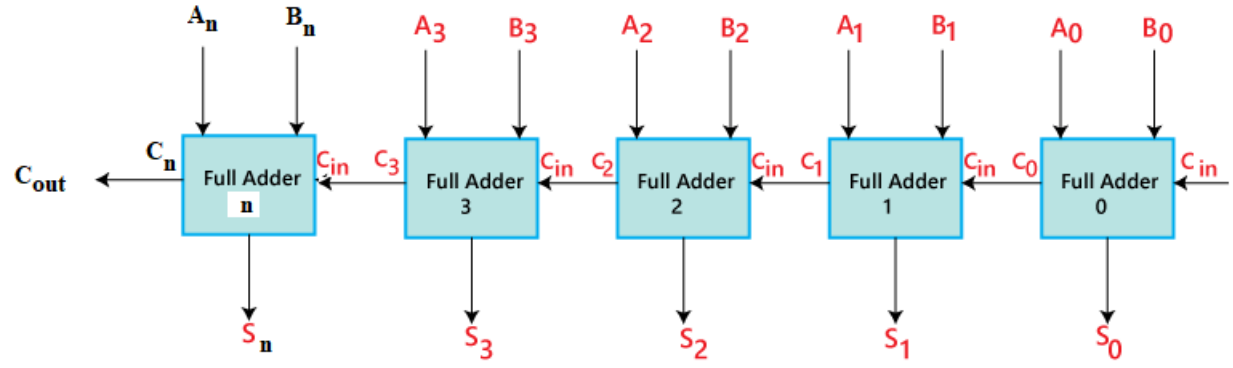


Fig: N-Bit Parallel Adder

Parallel Substractor:

A Parallel Subtractor is a digital circuit capable of finding the arithmetic difference of two binary numbers which has multiple bits by operating on corresponding pairs of bits in parallel. The parallel subtractor can be designed in several ways including combination of half and full subtractors, all full subtractors or all full adders with subtrahend complement input.

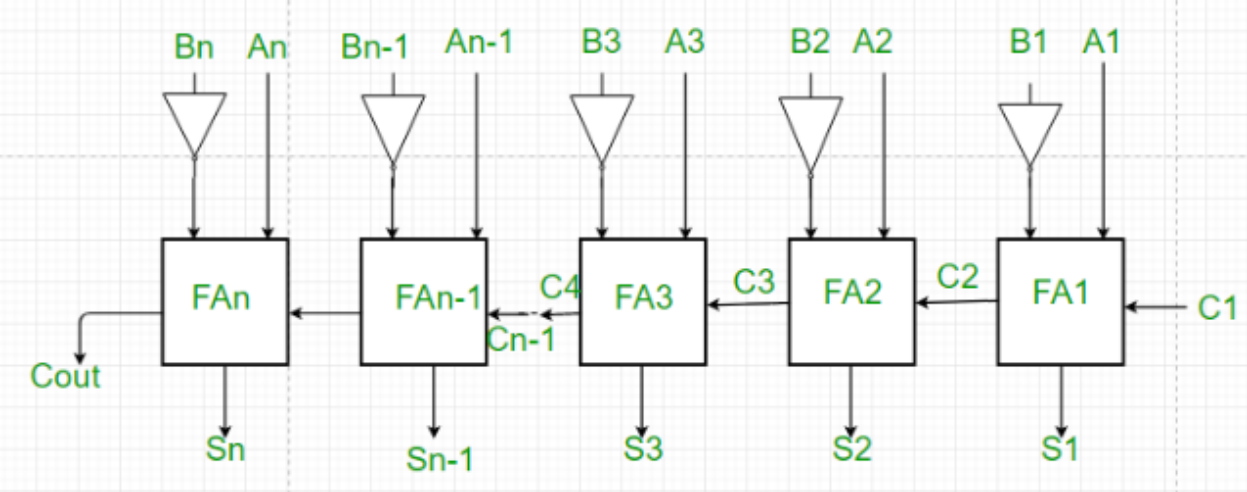


Fig: N-bit parallel subtractor using full adders.

Working of Parallel Subtractor

1. As shown in the figure, the parallel binary subtractor is formed by combination of all full adders with subtrahend complement input.
2. This operation considers that the addition of minuend along with the 2's complement of the subtrahend is equal to their subtraction.
3. Firstly the 1's complement of B is obtained by the NOT gate and 1 can be added through the carry to find out the 2's complement of B. This is further added to A to carry out the arithmetic subtraction.
4. The process continues till the last full adder FAn uses the carry bit Cn to add with its input An and 2's complement of Bn to generate the last bit of the output along last carry bit Cout.

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**.

Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

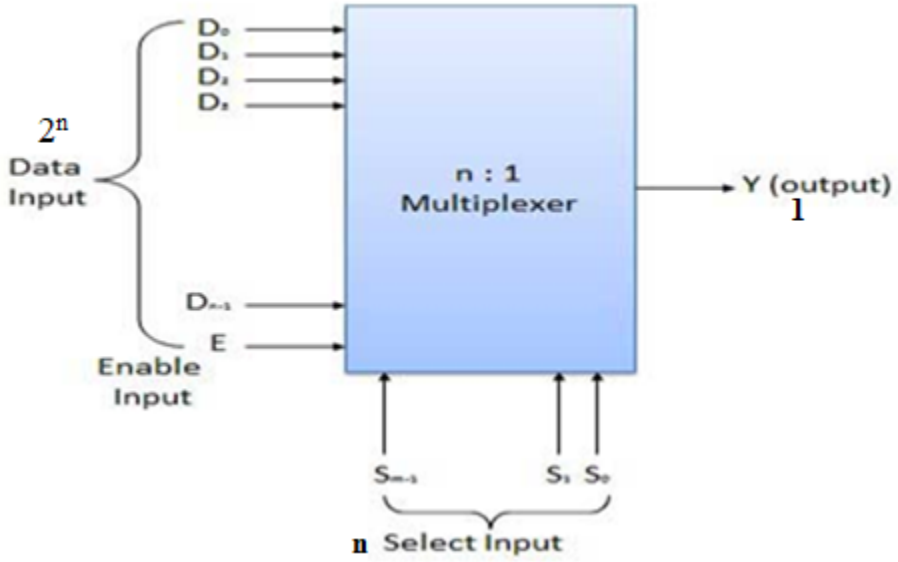


Fig: Block Diagram of Multiplexer

Here in diagram we can see there are 2^n input lines, n select lines and 1 output lines. Depending on value of n select line one of the 2^n input line will be selected as an output line.

There are various types of multiplexer some of them are:

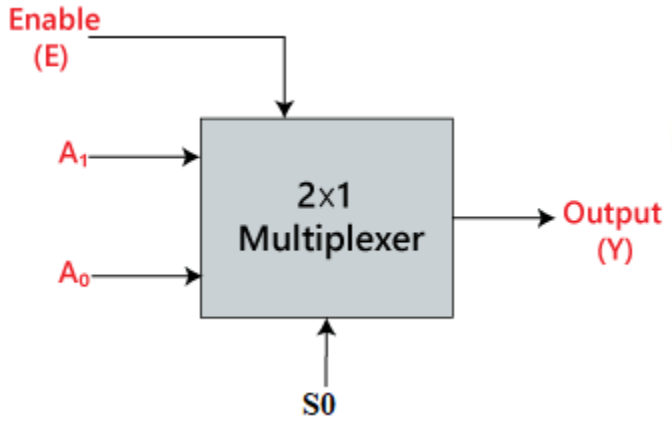
- 2x1 mux
- 4x1 mux
- 8x1 mux
- 16x1 mux

2×1 Multiplexer:

In 2×1 multiplexer, there are only two inputs, i.e., A_0 and A_1, 1 selection line, i.e., S_0 and single outputs, i.e., Y. On the basis of the combination of input selection line S_0, one of these 2 inputs will be connected to the output.

The block diagram and the truth table of the 2✕1 multiplexer are given below.

Block Diagram:



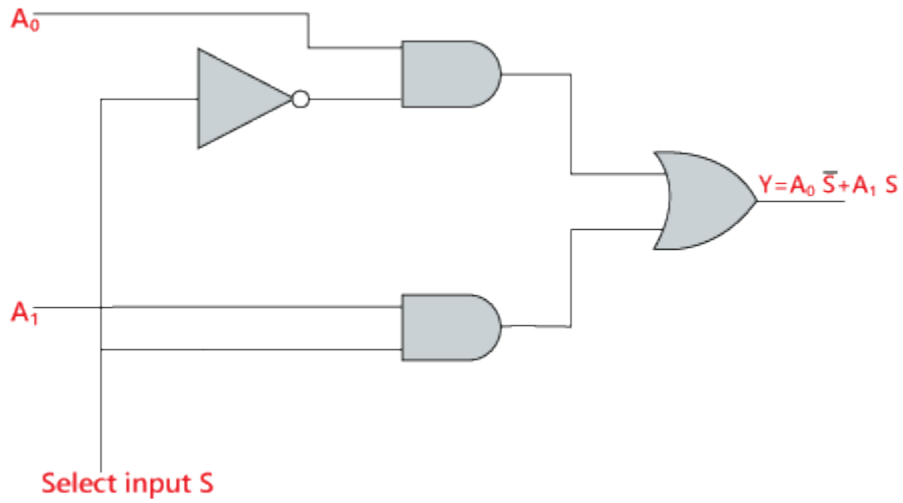
Truth Table:

INPUTS	Output
S_0	Y
0	A_0
1	A_1

Logical Expression:

$Y=S_0'.A_0+S_0.A_1$

Logical Circuit:

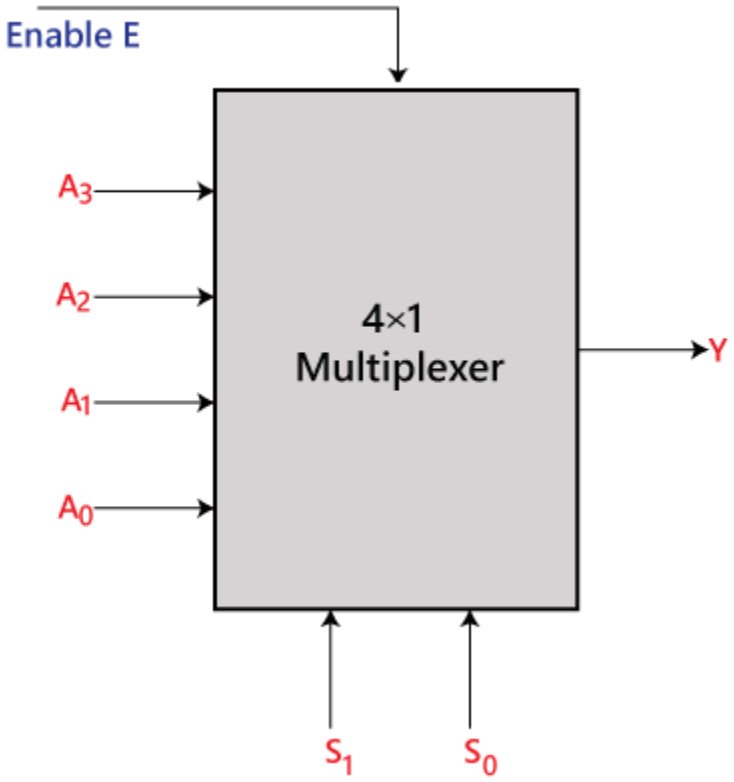


4×1 Multiplexer:

In the 4×1 multiplexer, there is a total of four inputs, i.e., A_0, A_1, A_2, and A_3, 2 selection lines, i.e., S_0 and S_1 and single output, i.e., Y. On the basis of the combination of input selection lines S_0 and S_1, one of these 4 inputs are connected to the output.

The block diagram and the truth table of the 4✕1 multiplexer are given below.

Block Diagram:



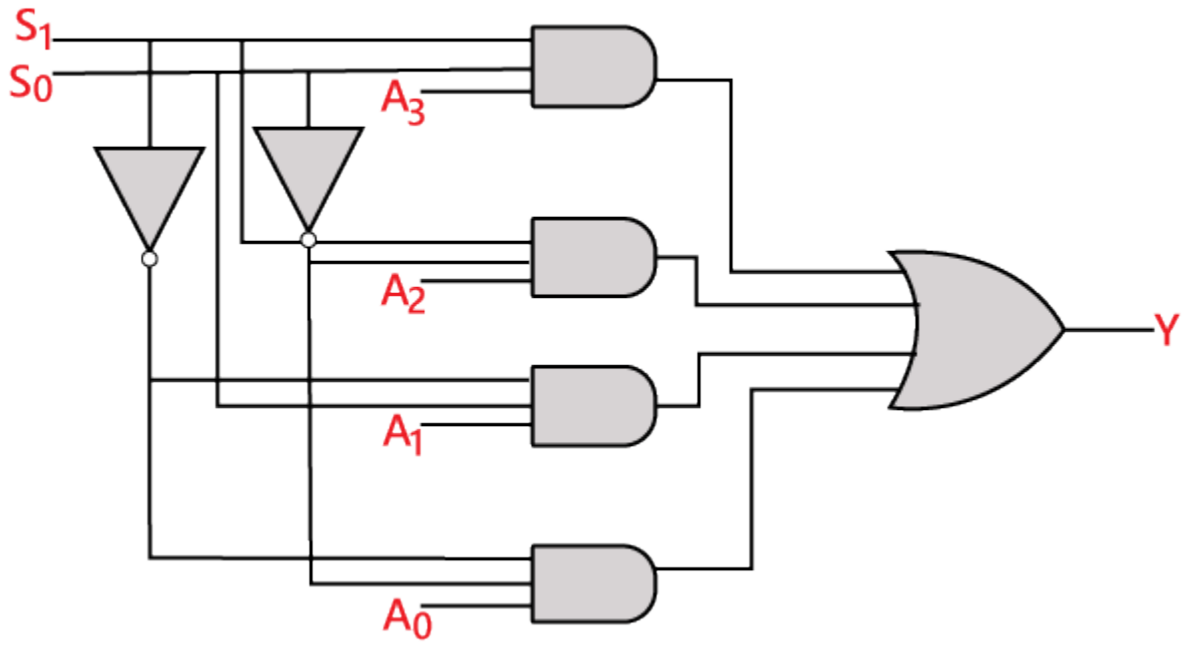
Truth Table:

INPUTS		Output
S_1	S_0	Y
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3

Logical Expression:

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

Logical Circuit:

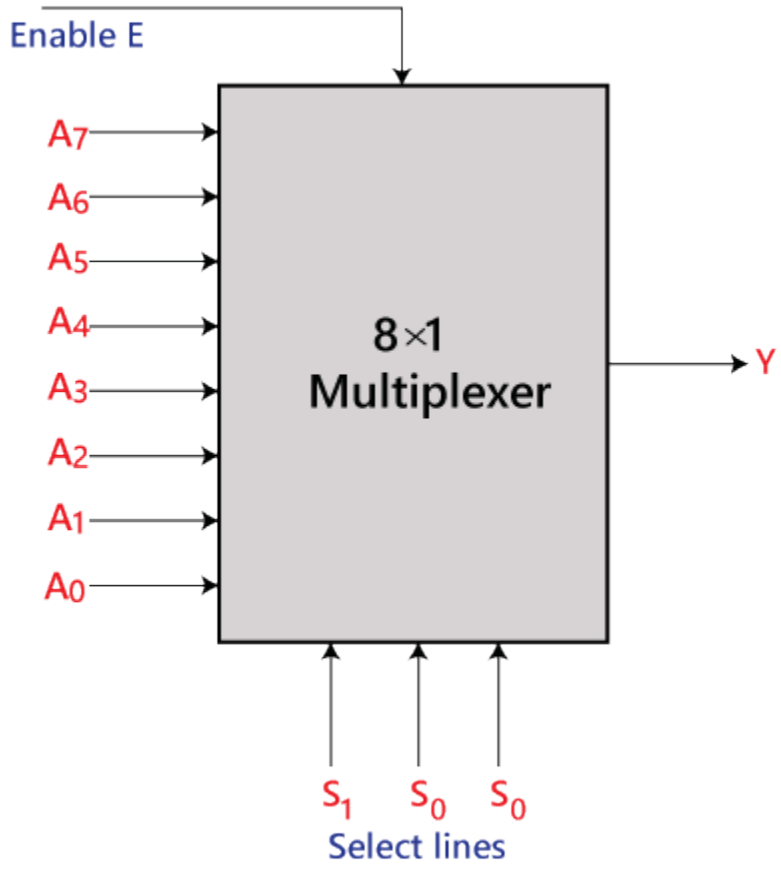


8 to 1 Multiplexer:

In the 8 to 1 multiplexer, there are total eight inputs, i.e., A_0, A_1, A_2, A_3, A_4, A_5, A_6, and A_7, 3 selection lines, i.e., S_0, $S_1$and S_2 and single output, i.e., Y. On the basis of the combination of input selection lines S_0, S_1, and S_2, one of these 8 inputs are connected to the output.

The block diagram and the truth table of the 8✕1 multiplexer are given below.

Block Diagram:



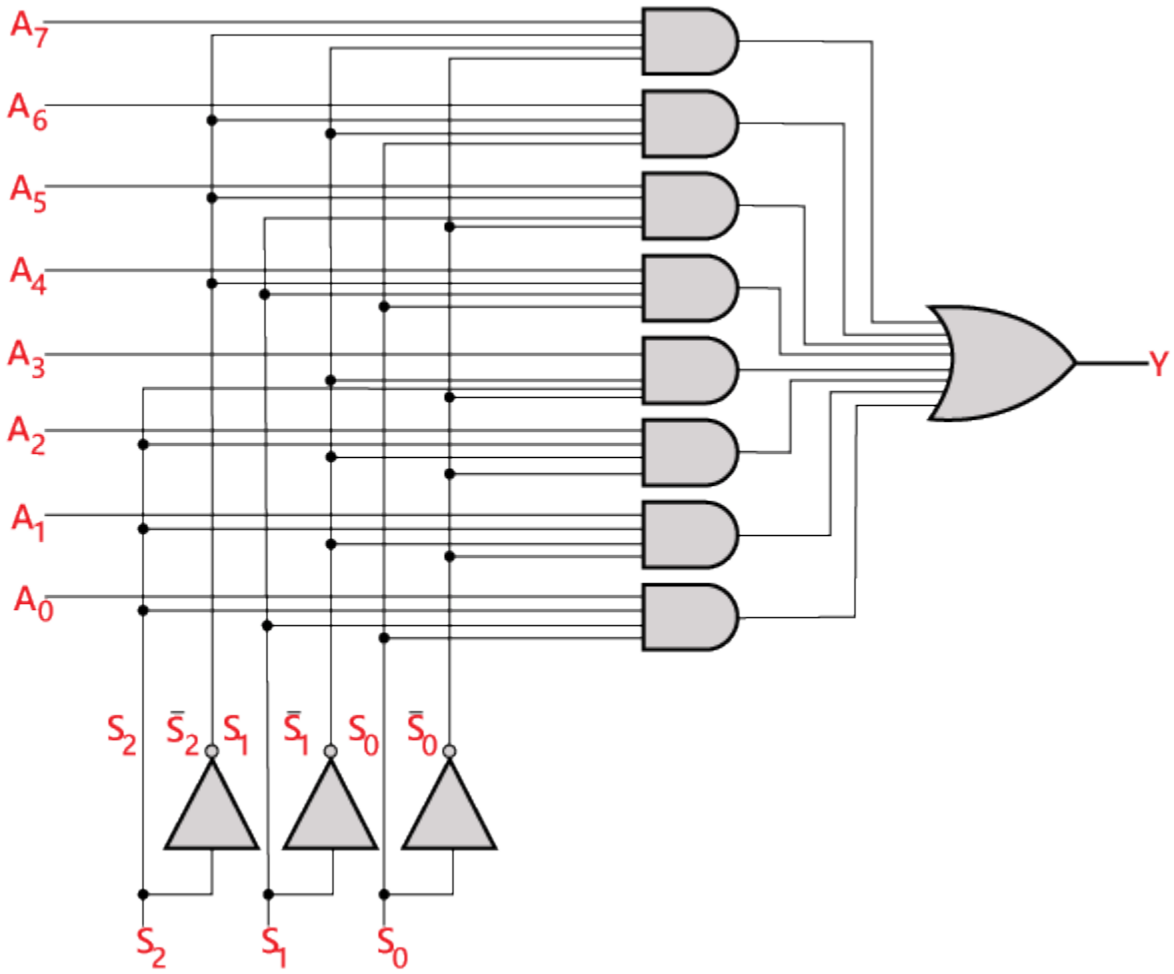
Truth Table:

INPUTS			Output
S_2	S_1	S_0	Y
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

Logical Expression:

$Y=S_0'.S_1'.S_2'.A_0+S_0.S_1'.S_2'.A_1+S_0'.S_1.S_2'.A_2+S_0.S_1.S_2'.A_3+S_0'.S_1'.S_2\ A_4+S_0.S_1'.S_2\ A_5+S_0'.S_1.S_2\ .A_6+S_0.S_1.S_3.A_7$

Logical Circuit:

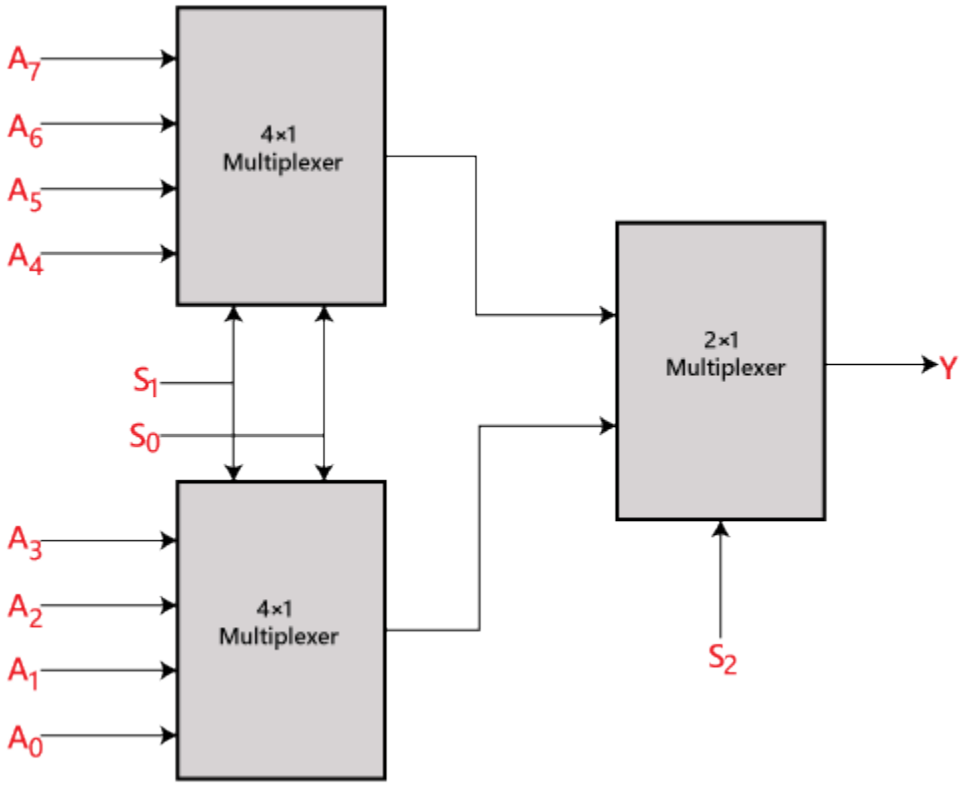


8 ×1 multiplexer using lower order mux (4×1 and 2×1 mux)

We can implement the 8✕1 multiplexer using a lower order multiplexer. To implement the 8✕1 multiplexer, we need two 4✕1 multiplexers and one 2✕1 multiplexer. The 4✕1 multiplexer has 2 selection lines, 4 inputs, and 1 output. The 2✕1 multiplexer has only 1 selection line.

For getting 8 data inputs, we need two 4×1 multiplexers. The two 4×1 multiplexer produces two outputs. So, in order to get the final one output, we need a 2×1 multiplexer.

The block diagram of 8×1 multiplexer using 4×1 and 2×1 multiplexer is given below.



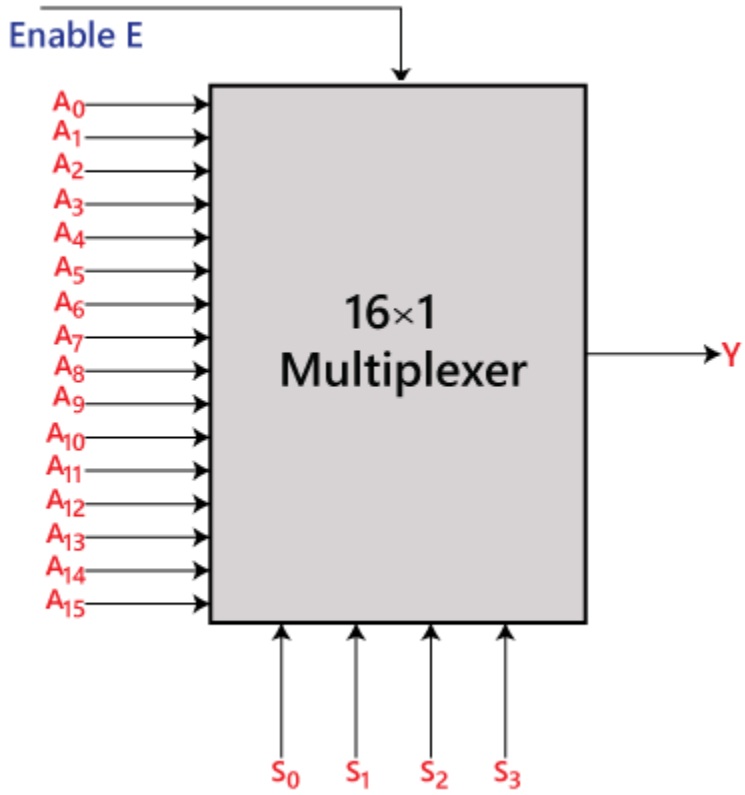
Note: Truth Table, Logical Expression and Logic Circuit are same as 8x1 mux.

16 to 1 Multiplexer:

In the 16 to 1 multiplexer, there are total of 16 inputs, i.e., A_0, A_1, …, A_{16}, 4 selection lines, i.e., S_0, S_1, S_2, and S_3 and single output, i.e., Y. On the basis of the combination of input selection lines S_0, S_1, S_2 and S_3, one of these 16 inputs will be connected to the output.

The block diagram and the truth table of the 16×1

Block Diagram:



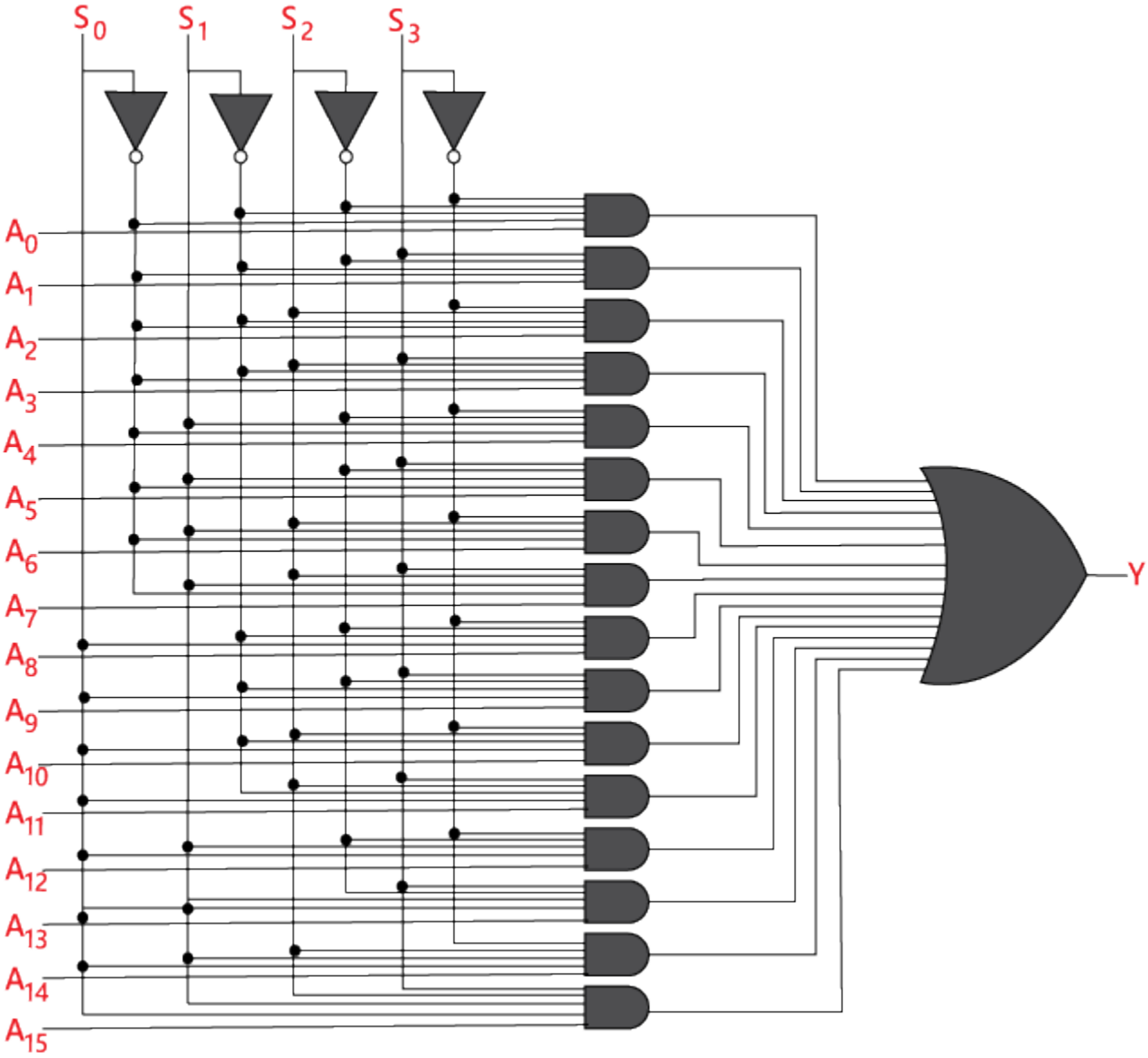
Truth Table:

INPUTS				Output
S_0	S_1	S_2	S_3	Y
0	0	0	0	A_0
0	0	0	1	A_1
0	0	1	0	A_2
0	0	1	1	A_3
0	1	0	0	A_4
0	1	0	1	A_5
0	1	1	0	A_6
0	1	1	1	A_7
1	0	0	0	A_8
1	0	0	1	A_9
1	0	1	0	A_{10}
1	0	1	1	A_{11}
1	1	0	0	A_{12}
1	1	0	1	A_{13}
1	1	1	0	A_{14}
1	1	1	1	A_{15}

Logical Expression:

$Y=A_0.S_0'.S_1'.S_2'.S_3'+A_1.S_0'.S_1'.S_2\ '.S_3+A_2.S_0'.S_1'.S_2.S_3'+A_3.S_0'.S_1\ '.S_2.S_3+A_4.S_0'.S_1.S_2'$
$.S_3'+A_5.S_0\ '.S_1.S_2'.S_3+A_6.S_1.S_2.S_3'+A_7.S_0\ '.S_1.S_2.S_3+A_8.S_0.S_1'.S_2'.S_3'+A_9\ .S_0.S_1'.S_2'.S_3$
$+Y_10.S_0.S_1'.S_2.S_3\ '+A_11.S_0.S_1'.S_2.S_3+A_12$
$S_0.S_1.S_2\ '.S_3'+A_13.S_0.S_1.S_2'.S_3+A_14.S_0.S_1\ .S_2.S_3'+A_15.S_0.S_1.S_2'.S_3$

Logical Circuit:

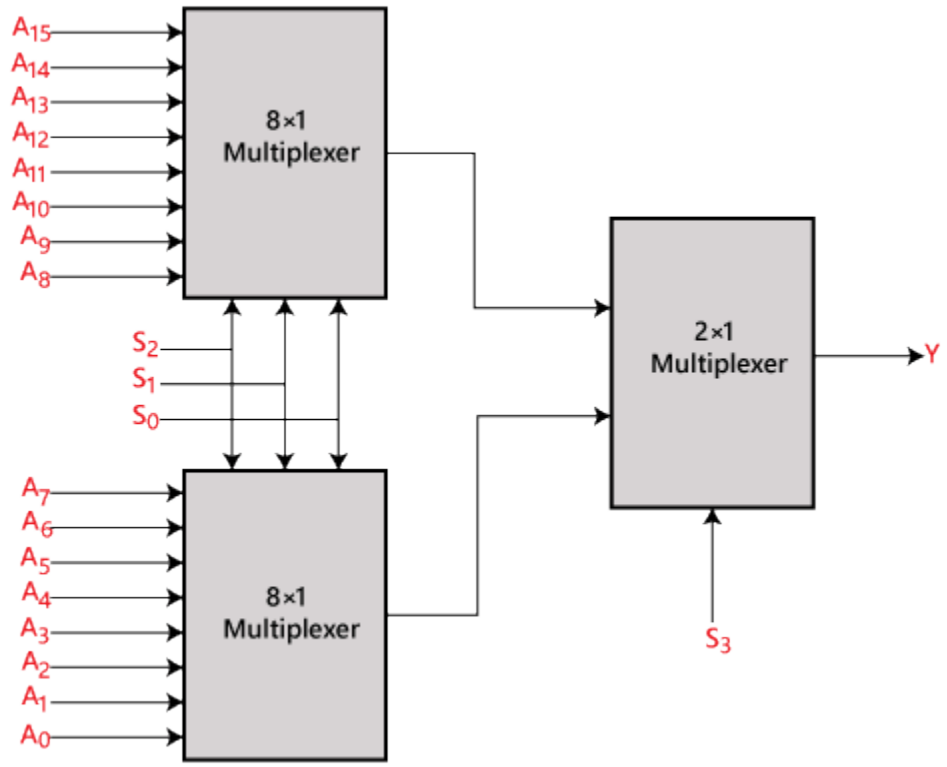


<u>16×1 multiplexer using lower order mux (8×1 and 2×1 mux)</u>

We can implement the 16✕1 multiplexer using a lower order multiplexer. To implement the 8✕1 multiplexer, we need two 8✕1 multiplexers and one 2✕1 multiplexer. The 8✕1 multiplexer has 3 selection lines, 4 inputs, and 1 output. The 2✕1 multiplexer has only 1 selection line.

For getting 16 data inputs, we need two 8 ×1 multiplexers. The two 8✕1 multiplexer produces two outputs. So, in order to get the final one output, we need a 2✕1 multiplexer.

The block diagram of 16✕1 multiplexer using 8✕1 and 2✕1 multiplexer is given below.



Note: Truth Table, Logical Expression and Logic Circuit are same as 16x1 mux.

De-multiplexer

A De-multiplexer is a combinational circuit that has only 1 input line and 2^n output lines. Simply, the multiplexer is a single-input and multi-output combinational circuit. The information is received from the single input lines and directed to the output line. On the basis of the values of the selection lines, the input will be connected to one of these outputs. De-multiplexer is opposite to the multiplexer. De-multiplexer is also treated as De-mux.

There are various types of De-multiplexer which are as follows:

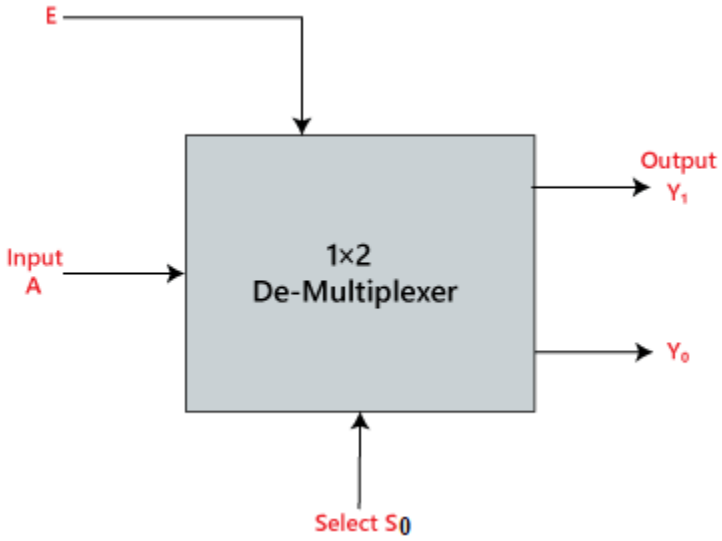
- 1x2 De-Mux
- 1x4 De-Mux
- 1x8 De-Mux
- 1x16 De-Mux

1×2 De-multiplexer:

In the 1 to 2 De-multiplexer, there are only two outputs, i.e., Y_0, and Y_1, 1 selection lines, i.e., S_0, and single input, i.e., A. On the basis of the combination of input selection lines S_0, the input will be connected to one of the outputs.

The block diagram and the truth table of the 1✕2 multiplexer are given below.

Block Diagram:



Truth Table:

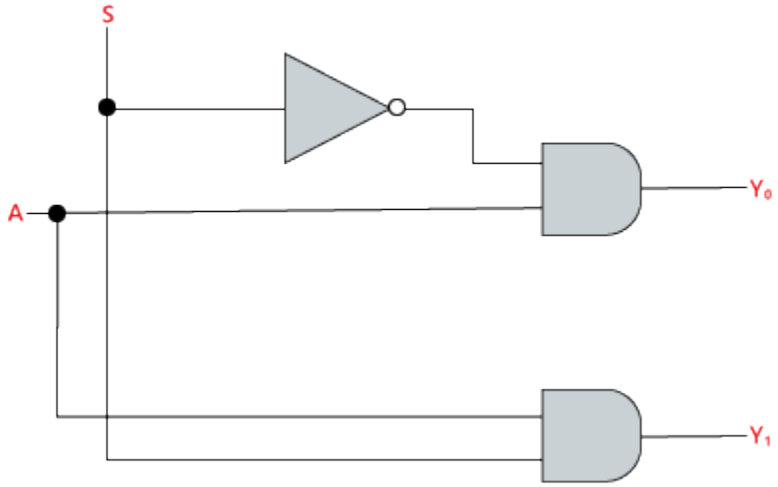
INPUTS	Output	
S_0	Y_1	Y_0
0	0	A
1	A	0

Logical Expression:

$Y_0=S_0'.A$

$Y_1=S_0.A$

Logical Circuit:

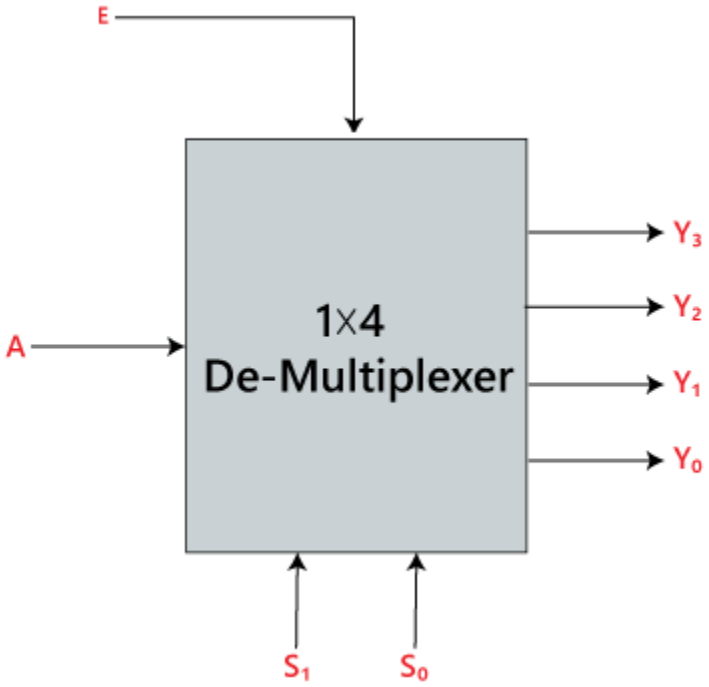


1×4 De-multiplexer:

In 1 to 4 De-multiplexer, there are total of four outputs, i.e., Y_0, Y_1, Y_2, and Y_3, 2 selection lines, i.e., S_0 and S_1 and single input, i.e., A. On the basis of the combination of input selection lines S_0 and S_1, the input will be connected to one of the outputs.

The block diagram and the truth table of the 1✕4 multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS		Output			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0

Logical Expression:

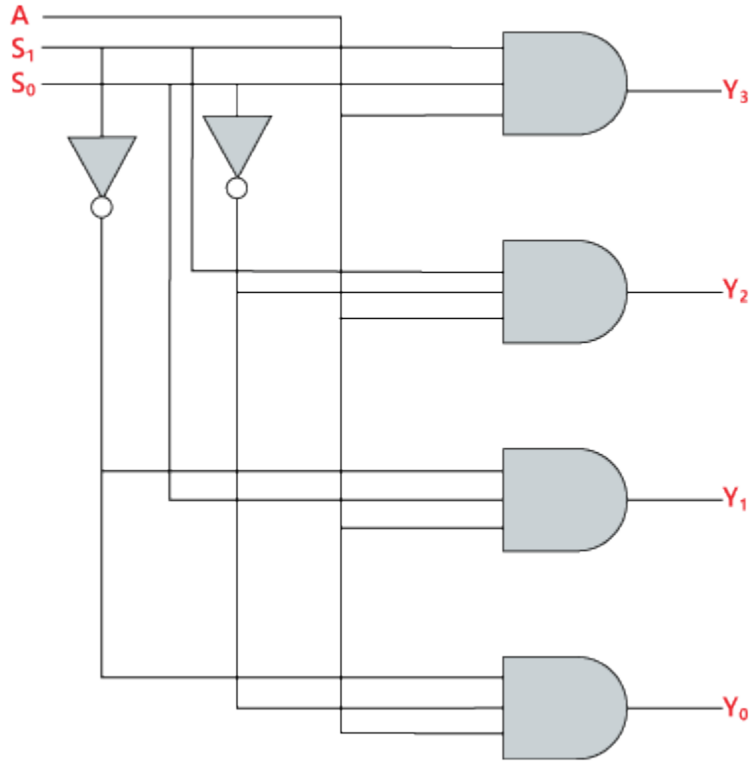
$Y_0 = S_1' \, S_0' \, A$

$y_1 = S_1' \, S_0 \, A$

$y_2 = S_1 \, S_0' \, A$

$y_3 = S_1 \, S_0 \, A$

Logical Circuit:

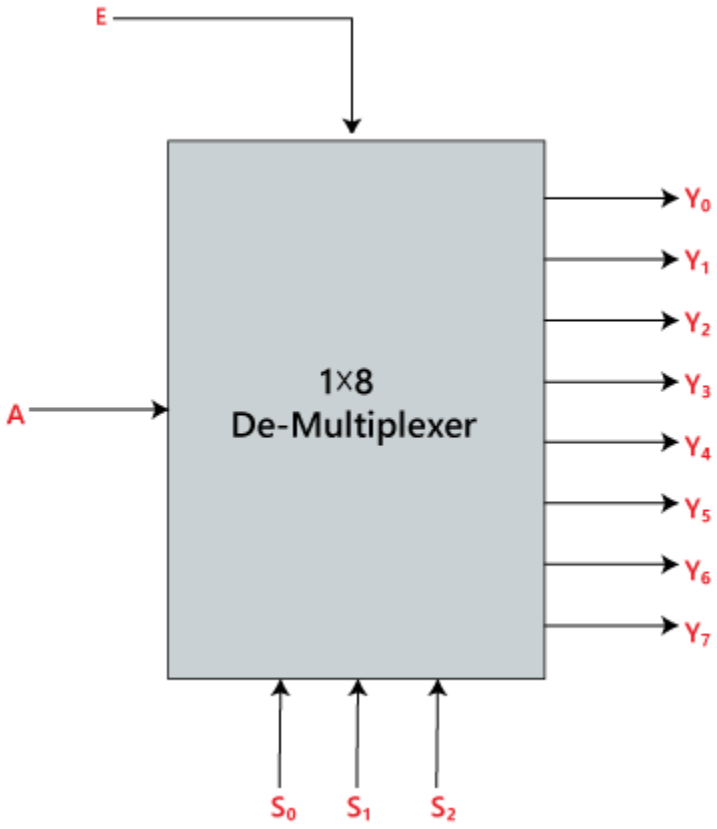


1×8 De-multiplexer

In 1 to 8 De-multiplexer, there are total of eight outputs, i.e., Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, and Y_7, 3 selection lines, i.e., S_0, $S_1$and S_2 and single input, i.e., A. On the basis of the combination of input selection lines S_0, S_1 and S_2, the input will be connected to one of these outputs.

The block diagram and the truth table of the 1✕8 de-multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS			Output							
S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	A
0	0	1	0	0	0	0	0	0	A	0
0	1	0	0	0	0	0	0	A	0	0
0	1	1	0	0	0	0	A	0	0	0
1	0	0	0	0	0	A	0	0	0	0
1	0	1	0	0	A	0	0	0	0	0
1	1	0	0	A	0	0	0	0	0	0
1	1	1	A	0	0	0	0	0	0	0

Logical Expression:

$Y_0=S_0'.S_1'.S_2'.A$

$Y_1=S_0.S_1'.S_2'.A$

$Y_2=S_0'.S_1.S_2'.A$

$Y_3=S_0.S_1.S_2'.A$

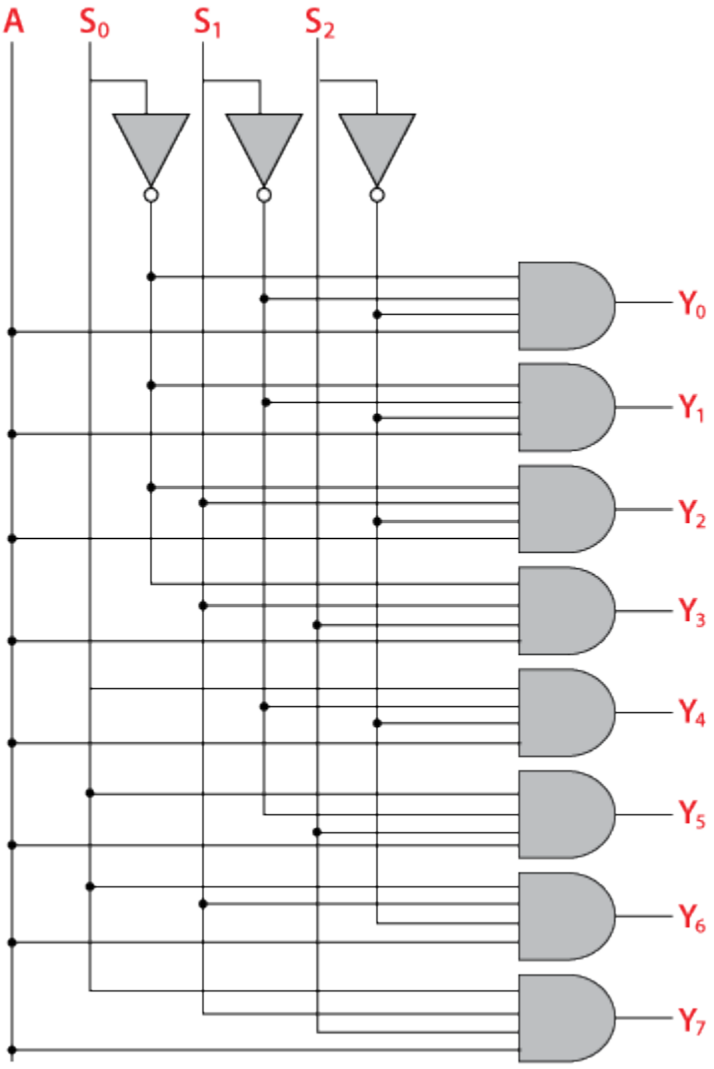
$Y_4=S_0'.S_1'.S_2\ A$

$Y_5=S_0.S_1'.S_2\ A$

$Y_6=S_0'.S_1.S_2\ A$

$Y_7=S_0.S_1.S_3.A$

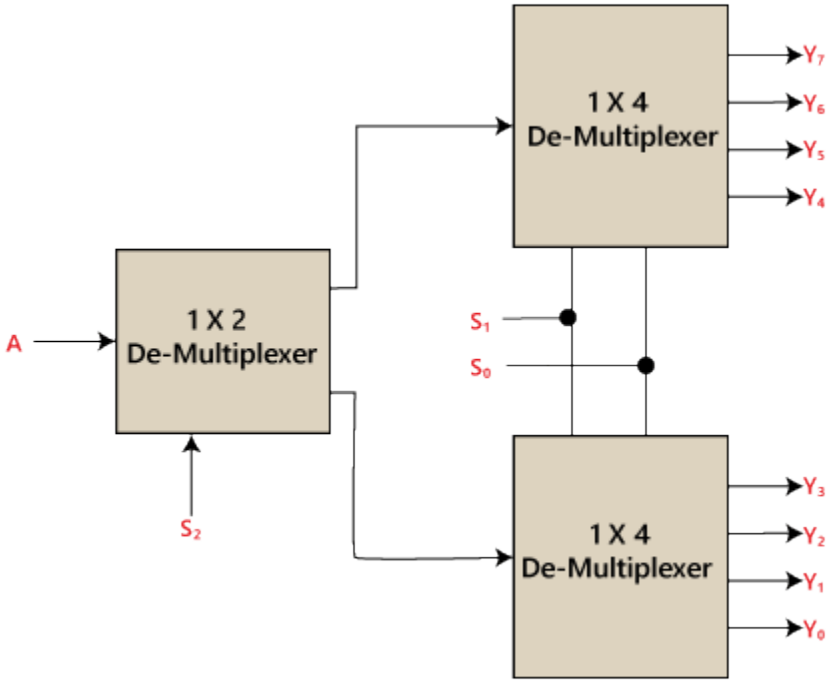
Logical Circuit:



1×8 De-multiplexer using 1×4 and 1×2 de-multiplexer

We can implement the 1✕8 de-multiplexer using a lower order de-multiplexer. To implement the 1✕8 de-multiplexer, we need two 1✕4 de-multiplexer and one 1✕2 de-multiplexer. The 1✕4 multiplexer has 2 selection lines, 4 outputs, and 1 input. The 1✕2 de-multiplexer has only 1 selection line, 2 outputs and 1 input.

For getting 8 data outputs, we need two 1✕4 de-multiplexers. The 1×2 de-multiplexer produces two outputs. So, in order to get the final output, we have to pass the outputs of 1×2 de-multiplexer as an input of both the 1✕4 de-multiplexer. The block diagram of 1✕8 de-multiplexer using 1✕4 and 1✕2 de-multiplexer is given below.

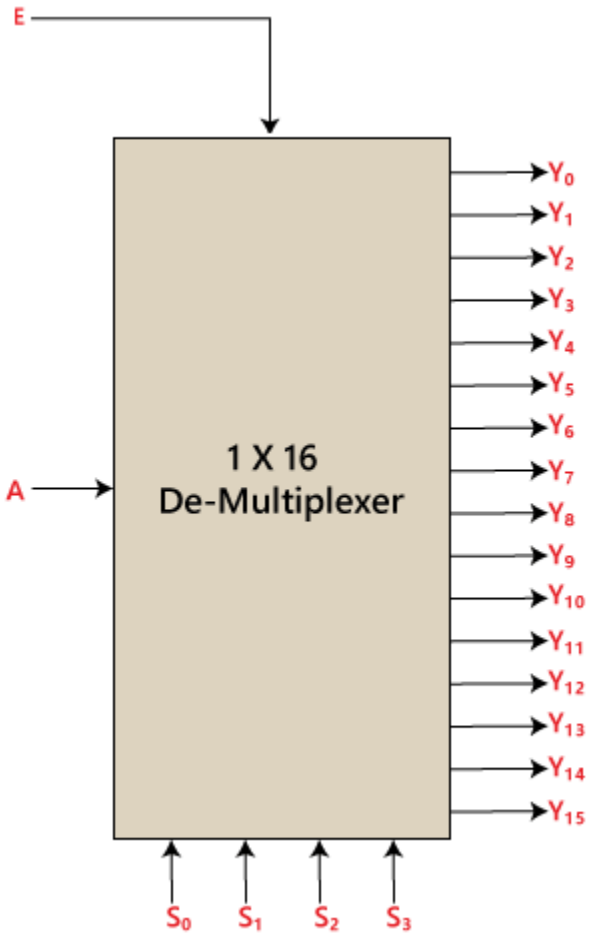


1 x 16 De-multiplexer

In 1×16 de-multiplexer, there are total of 16 outputs, i.e., Y_0, Y_1, …, Y_{16}, 4 selection lines, i.e., S_0, S_1, S_2, and S_3 and single input, i.e., A. On the basis of the combination of input selection lines S_0, S_1, and S_2, the input will be connected to one of these outputs.

The block diagram and the truth table of the 1✕16 de-multiplexer are given below.

Block Diagram:



Truth Table:

INPUTS				OUTPUTS															
S_3	S_2	S_1	S_0	Y_{15}	Y_{14}	Y_{13}	Y_{12}	Y_{11}	Y_{10}	Y_9	Y_8	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Logical Expression:

$Y_0=A.S_0'.S_1'.S_2'.S_3'$

$Y_1=A.S_0'.S_1'.S_2'.S_3$

$Y_2=A.S_0'.S_1'.S_2.S_3'$

$Y_3=A.S_0'.S_1'.S_2.S_3$

$Y_4=A.S_0'.S_1.S_2'.S_3'$

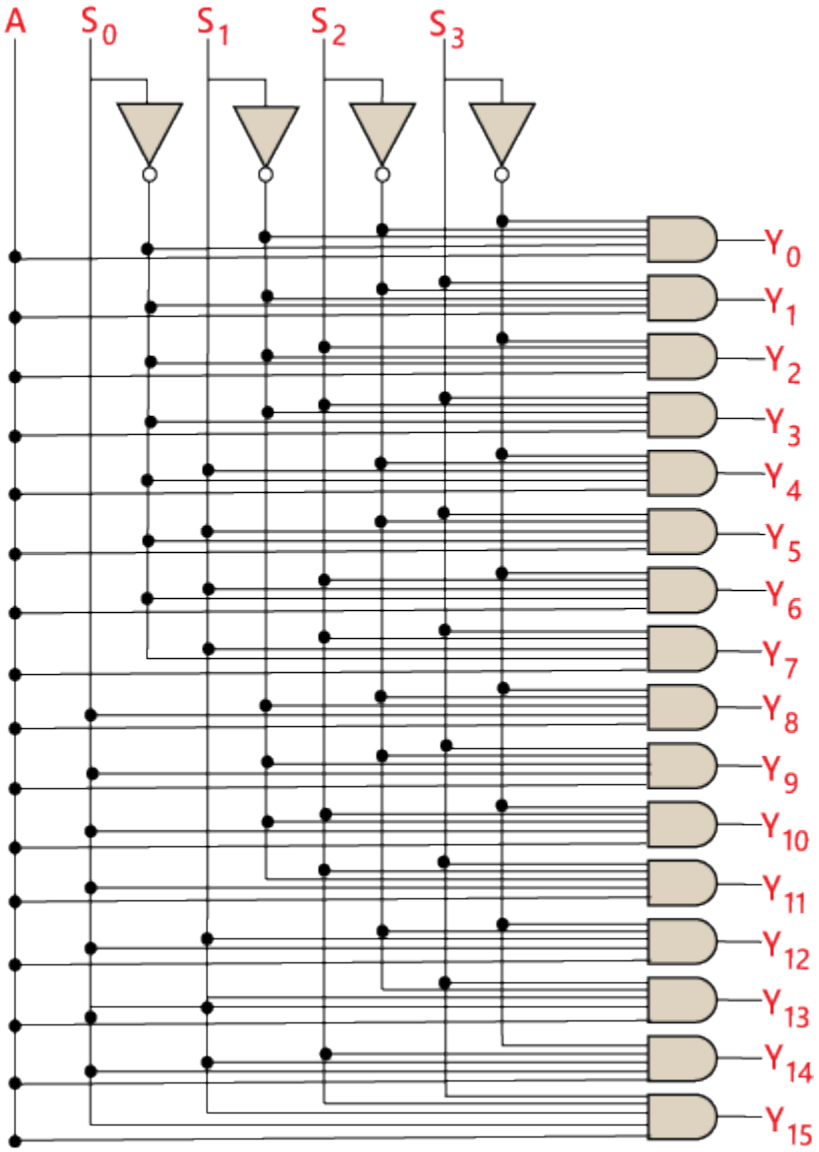
$Y_5=A.S_0'.S_1.S_2'.S_3$

$Y_6=A.S_0'.S_1.S_2.S_3'$

$Y_7=A.S_0'.S_1.S_2.S_3$

$Y_8=A.S_0.S_1'.S_2'.S_3'$
$Y_9=A.S_0.S_1'.S_2'.S_3$
$Y_{10}=A.S_0.S_1'.S_2.S_3'$
$Y_{11}=A.S_0.S_1'.S_2.S_3$
$Y_{12}=A.S_0.S_1.S_2'.S_3'$
$Y_{13}=A.S_0.S_1.S_2'.S_3$
$Y_{14}=A.S_0.S_1.S_2.S_3'$
$Y_{15}=A.S_0.S_1.S_2'.S_3$

Logical Circuit:

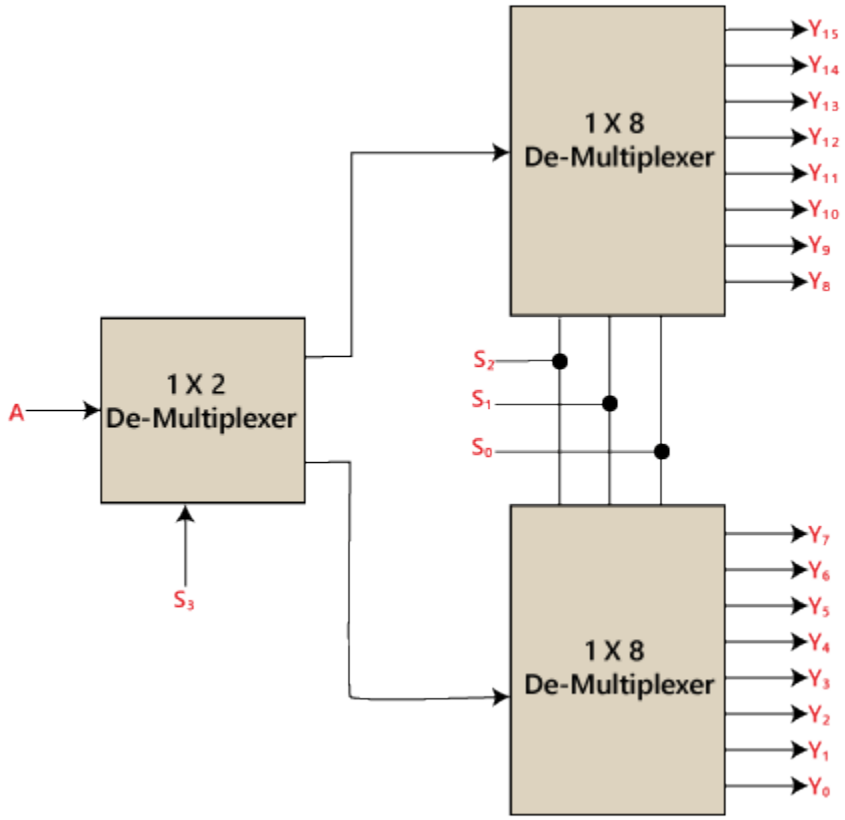


1×16 de-multiplexer using 1×8 and 1×2 de-multiplexer

We can implement the 1✕16 de-multiplexer using a lower order de-multiplexer. To implement the 1✕16 de-multiplexer, we need two 1✕8 de-multiplexer and one 1✕2 de-multiplexer. The 1✕8 multiplexer has 3 selection lines, 1 input, and 8 outputs. The 1✕2 de-multiplexer has only 1 selection line, 2 outputs and 1 input.

For getting 16 data outputs, we need two 1×8 de-multiplexers. The 1✕8 de-multiplexer produces eight outputs. So, in order to get the final output, we need a 1✕2 de-multiplexer to produce two outputs from a single input. Then we pass these outputs into both the de-multiplexer as an input.

The block diagram of 1✕16 de-multiplexer using 1✕8 and 1✕2 de-multiplexer is given below.



Basic Concept of programmable logic

Programmable logic is the logic which is used to control the programmable devices. It consists of different integrated circuits and logical gates.

Programmable Logic Devices (PLDs)

Programmable Logic Devices PLDs are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

- Read Only Memory (ROM)
- Programmable Array Logic
- Programmable Logic Array

The process of entering the information into these devices is known as **programming**. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Read Only Memory (ROM)

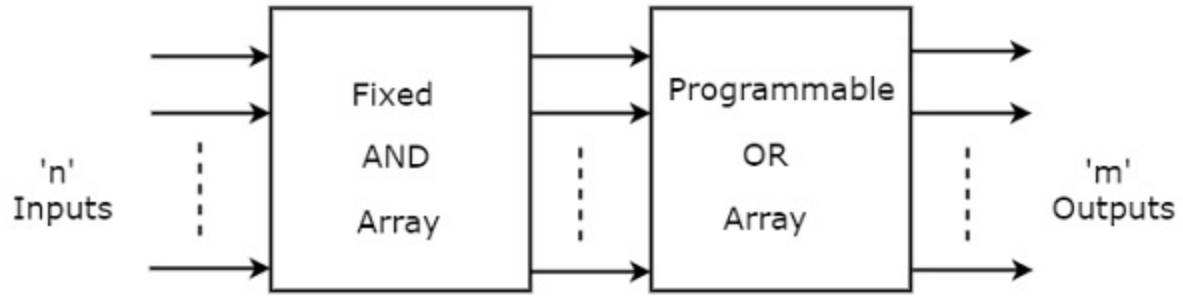
Read Only Memory ROM is a memory device, which stores the binary information permanently. That means, we can't change that stored information by any means later. ROM can further classified in terms of programmable capability as

- PROM
- EPROM

PROM:

If the ROM has programmable feature, then it is called as **Programmable ROM (PROM)**. The user has the flexibility to program the binary information electrically once by using PROM programmer.

PROM is a programmable logic device that has fixed AND array & Programmable OR array. The **block diagram** of PROM is shown in the following figure.



Here, the inputs of AND gates are not of programmable type. So, we have to generate 2^n product terms by using 2^n AND gates having n inputs each. We can implement these product terms by using $nx2^n$ decoder and this decoder generates 'n' **min terms**.

Here, the inputs of OR gates are programmable that means, all the outputs of AND gates are accessible to each OR gate as input but we can program only the required numbers of product terms. Therefore, the outputs of PROM will be in the form of **sum of min terms or SOP**.

Example 1: Implement the following **Boolean functions** using PROM.

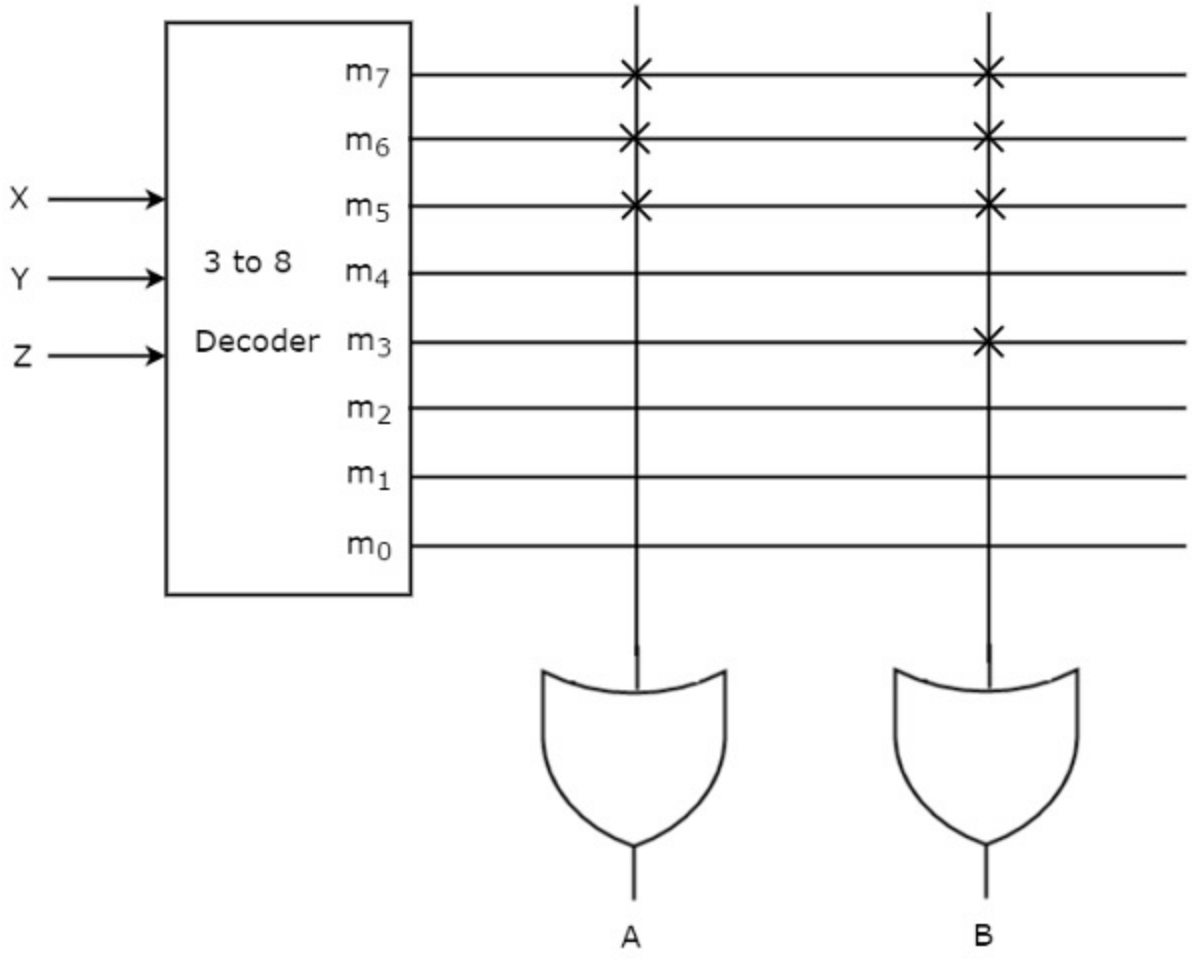
$A(X,Y,Z) = \sum m(5,6,7)$

$B(X,Y,Z) = \sum m(3,5,6,7)$

Solutions:

The given two functions are in SOP form and each function is having three variables X, Y & Z. So, we require a 3 to 8 decoder and two programmable OR gates for producing these two functions.

The corresponding **PROM** is shown in the following figure.



Here, 3 to 8 decoder generates eight min terms. The two programmable OR gates have the access of all these min terms. But, only the required min terms are programmed in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

Example 2: Implement the following combinational logic function using ROM

A1	A0	F1	F2
0	0	1	0
0	1	0	1
1	0	1	1
1	1	1	0

Solution:

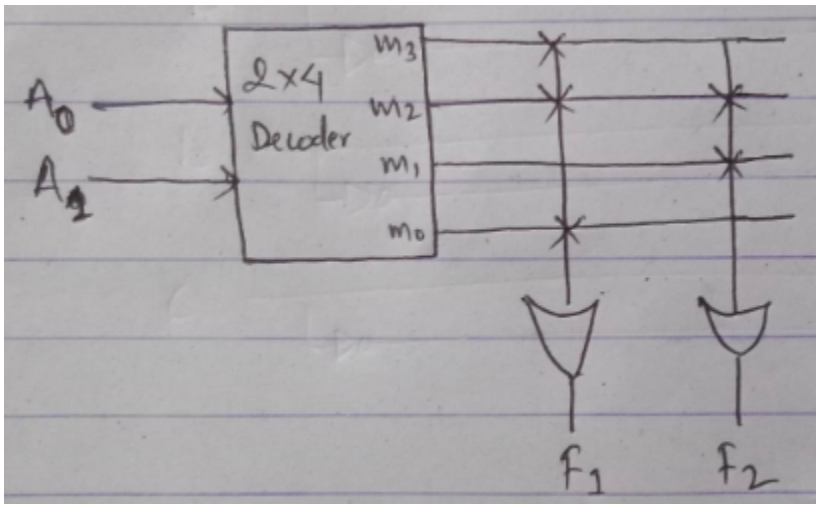
Let's make Boolean function from above table,

$F1 = A_1' A_0' + A_1 A_0' + A_1 A_0 = \sum m(0,2,3)$

$F2 = A_1' A_0 + A_1 A_0' = \sum m(1,2)$

Now Draw ROM for above Boolean function, we need one 2x4 decoder and two OR gates.

Circuit Diagram:



Example 3: Design a combinational circuit using ROM that accept 3-bit input number and generate an output binary number equals to the square of the input number with circuit diagram, truth table and block diagram.

Solution:

Here Given,

Input data is 3 bit

Output is square of input i.e. $Y=A^2$ where A is input Y is output

3 bit input value can be 0,1,2,3,4,5,6, and 7

Output range is square of input i.e. 0, 1, 2, 9, 16, 25, 36, and 49 respectively

Now we need 6 bit output to represent maximum output value 49.

i.e $2^6 = 64$, which includes 49.

Truth Table:

Input			Output					
A_0	A_1	A_2	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0
0	1	1	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	0	0	1

Output functions are:

$Y_0 = A_0\ A_1A_2' + A_0\ A_1A_2 = \square m(6,7)$

$Y_1 = A_0\ A_1'A_2' + A_0\ A_1'A_2 + A_0\ A_1A_2 = \square m(4,5,7)$

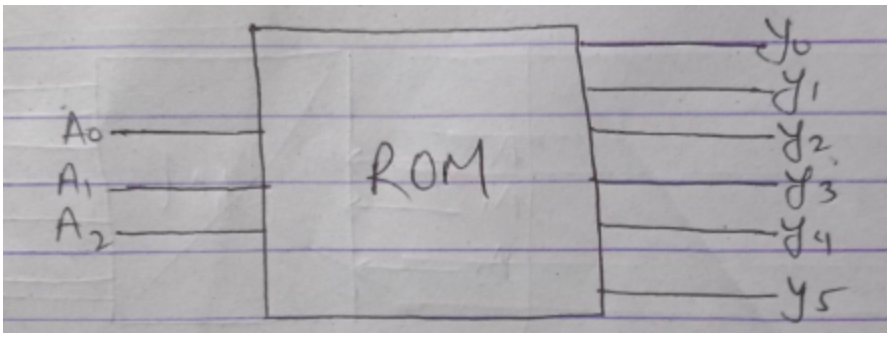
$Y_2 = A_0'A_1A_2 + A_0\ A_1'A_2 = \square m(3,5)$

$Y_3 = A_0'A_1A_2' + A_0\ A_1A_2' = \square m(2,6)$

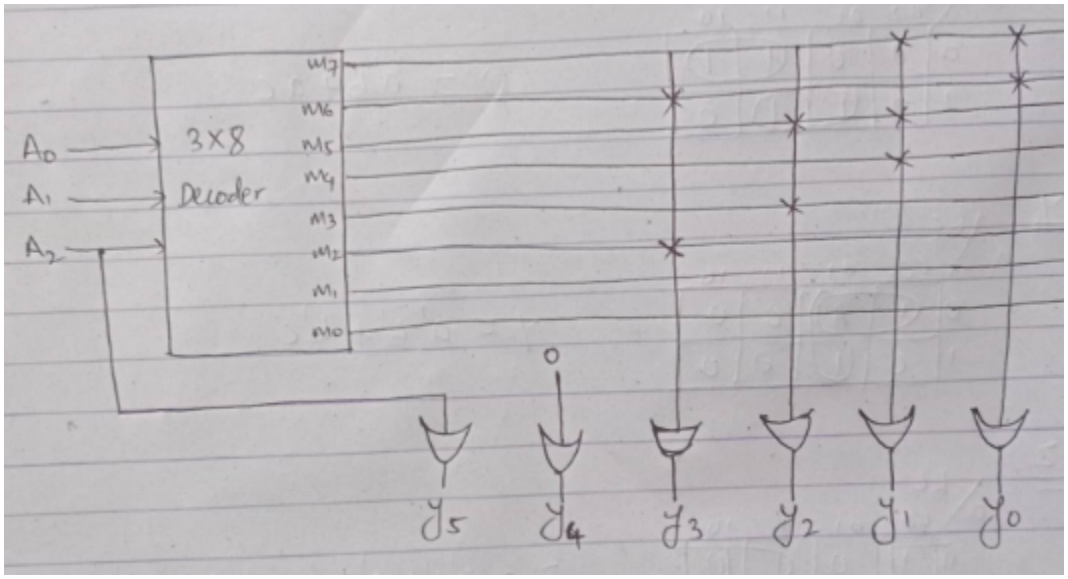
$Y_4 = 0$

$Y_5 = A_2$

Block Diagram:



Circuit Diagram:



EPROM:

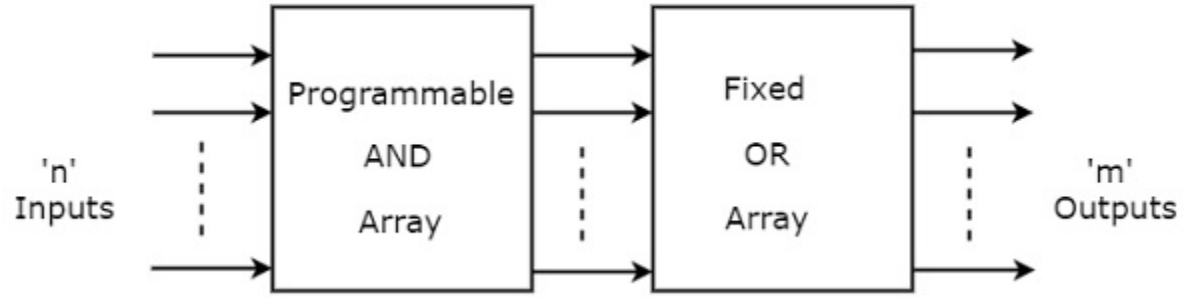
EPROM (**erasable programmable read-only memory**) is a form of computer memory that does not lose its content when the power supply is cut off and that can be erased and reused. EPROMs are generally implements for programs designed for repeated use and that can be upgraded with a later version of a program. EPROMs are erased with ultraviolet light.

The capabilities of EPROMs were extended with EEPROM (electrically erasable programmable read-only memory); flash memory, which is extensively used in computers in the early 21st century, is an example of EEPROM.

Programmable Array Logic (PAL)

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required number of product terms of Boolean function instead of generating all the min terms (SOP) by using programmable AND gates.

The **block diagram** of PAL is shown in the following figure.



Here, the inputs of AND gates are programmable that means each AND gate has both normal and complemented forms of input variables. So, based on the requirement, we can program any of those input literals to generate only the required **product terms** by using these AND gates.

The inputs of OR gates are not of programmable type that means the number of inputs to each OR gate will be fixed. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products (SOP).**

Example 1: Implement the following **Boolean functions** using PAL.

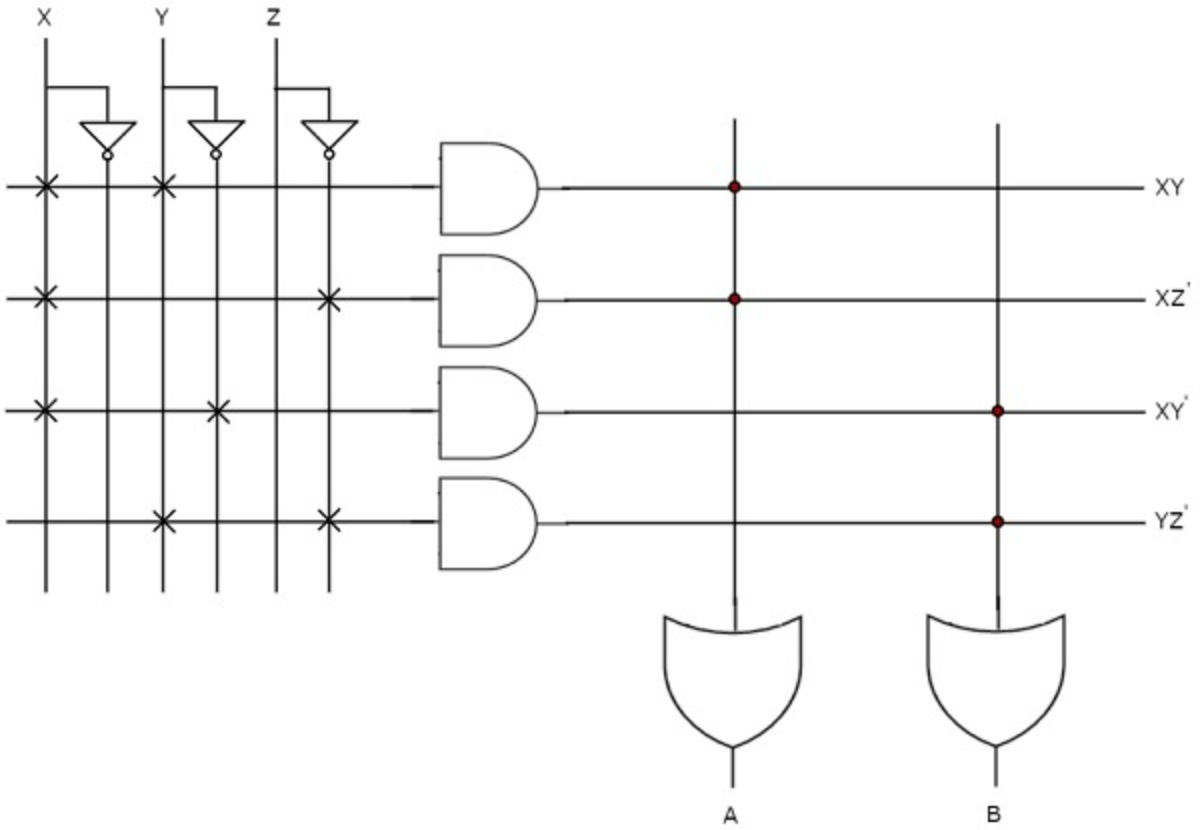
$A = XY+XZ'$

$B = XY'+YZ'$

Solution:

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions.

The corresponding PAL logic circuit is shown in the following figure.



The programmable AND gates have the access of both normal and complemented forms of inputs variables. In the above figure, the inputs X, X′, Y, Y′, Z & Z′, are available at the inputs of each AND gate. But the program uses only those literals which are required in order to generate one product term by each AND gate. The symbol 'X' is used for programmable connections.

The inputs of OR gates are fixed type. So, the necessary product terms are connected to inputs of each OR gate to produce the respective Boolean functions. The symbol **'.'** is used for fixed connections.

Example 2: Design PAL Circuit and Programmable table with given functions:

X(a,b,c) = □m(2,3,5,7)

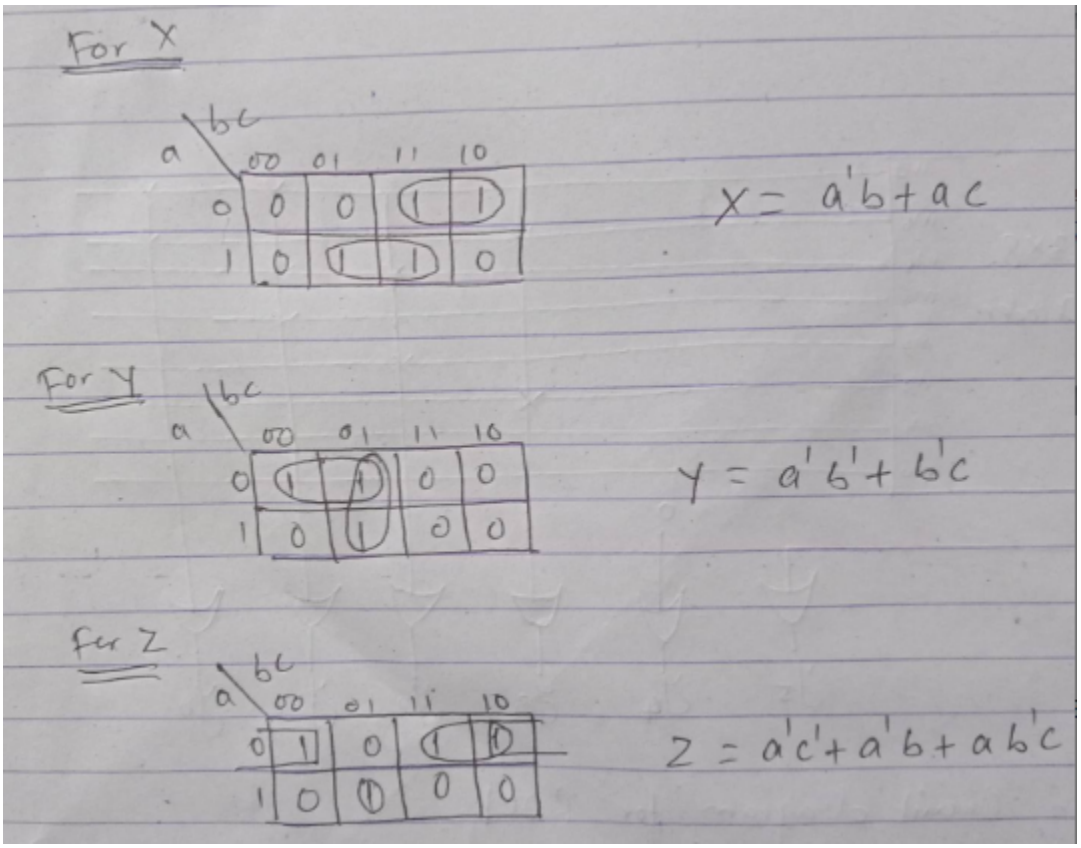
Y(a,b,c) = □m(0,1,5)

Z(a,b,c) = □m(0,2,3,5)

Solution:

In the PAL, AND array is programmable and OR array is fixed, so for AND array we have to find the minterm or Product Term for all the function.

Now, first find the simplified product term from the given function

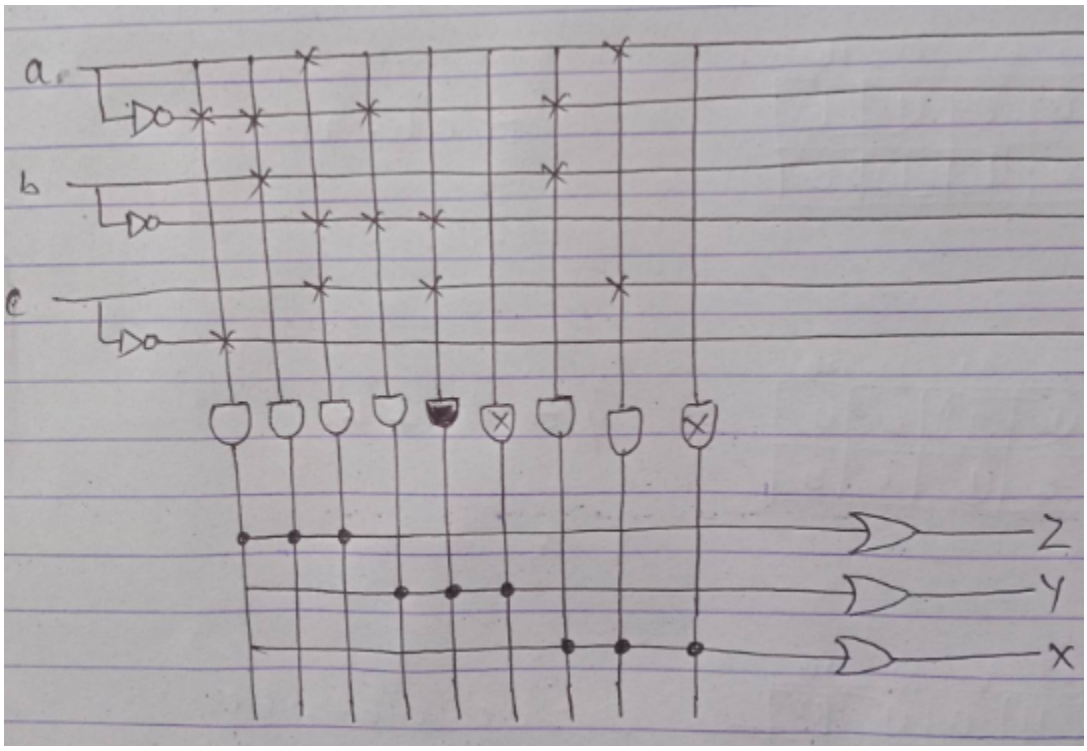


Programmable Table:

Input	Output					
Product Term	**a**	**b**	**C**	**X**	**Y**	**Z**
1	0	1	-	1	-	-
2	1	-	1	1	-	-
3	0	0	-	-	1	-
4	-	0	1	-	1	-
5	0	-	0	-	-	1
6	0	1	-	-	-	1
7	1	0	1	-	-	1

Circuit Diagram:

In the PAL circuit, all function should contain equal numbers of AND gate. In the above Boolean functions the maximum number of AND gate present is in function Z i.e. 3, so we need 3 AND gate for each function. Therefore we need total 9 AND gates and 3 OR gates to construct PAL logic circuit.

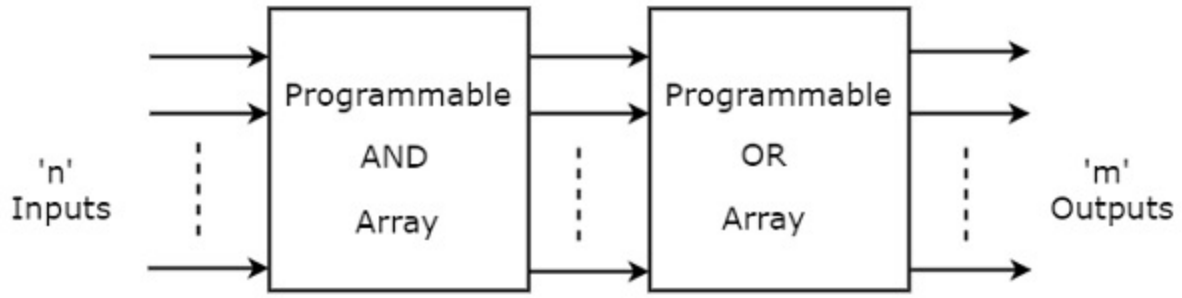


Here in the PAL logic circuit, AND gate with (X) symbol is not used gate.

Programmable Logic Array (PLA)

PLA is a programmable logic device that has Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD.

The **block diagram** of PLA is shown in the following figure.



Here, the inputs of AND gates are programmable that means each AND gate has both normal and complemented forms of inputs variables. So, based on the requirement, we can program any of these inputs to generate only the required product terms by using these AND gates.

The inputs of OR gates are also programmable that means all the outputs of AND gates are applied as inputs to each OR gate. So, we can program any number of required product terms to generate required output functions. Therefore, the outputs of PLA will be in the form of **sum of products (SOP)**.

Example 1: Implement the following **Boolean functions** using PLA.

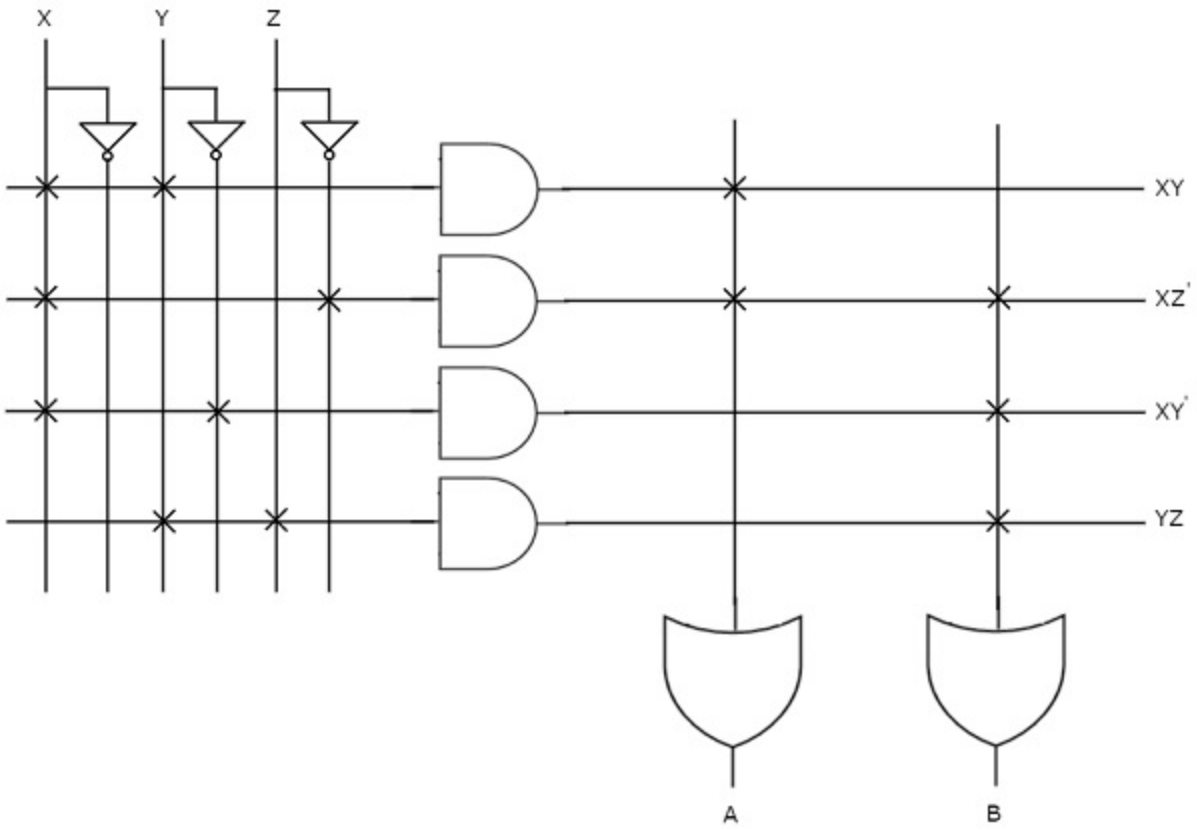
$A=XY+XZ'$

$B=XY'+YZ+XZ'$

Solution:

The given two functions are in sum of products form. The number of product terms present in the given Boolean functions A & B are two and three respectively. One product term, XZ' is common in each function. So, we require total numbers of four programmable AND gates & two programmable OR gates for producing those two functions.

The corresponding PLA circuit diagram is shown in the following figure.



The programmable AND gates have the access of both normal and complemented forms of input variables but we can program only those literals which are required in order to generate one product term by each AND gate. In the above figure, the inputs X, X′, Y, Y′, Z & Z′, are available at the inputs of each AND gate.

All these product terms are available at the inputs of each programmable OR gate. But, program use only those product term which are required in order to produce the respective Boolean functions by each OR gate.

The symbol ‘X’ is used for programmable connections.

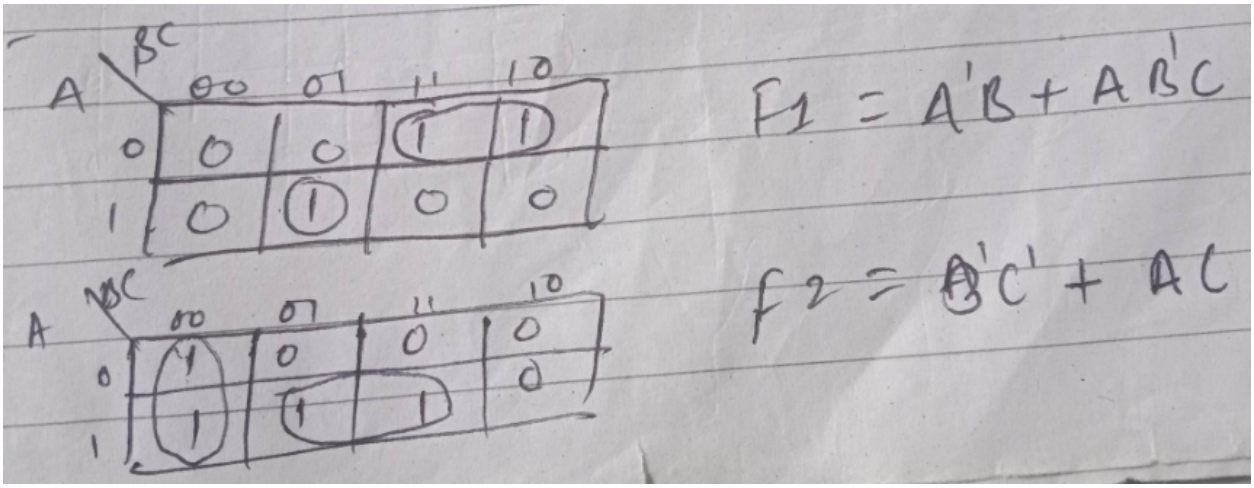
Example 2: Design PLA circuit and PLA programmable table with given functions.

$F_1(a,b,c) = \square m(2,3,5)$

$F_2(a,b,c) = \square m(0,4,5,7)$

Solution:

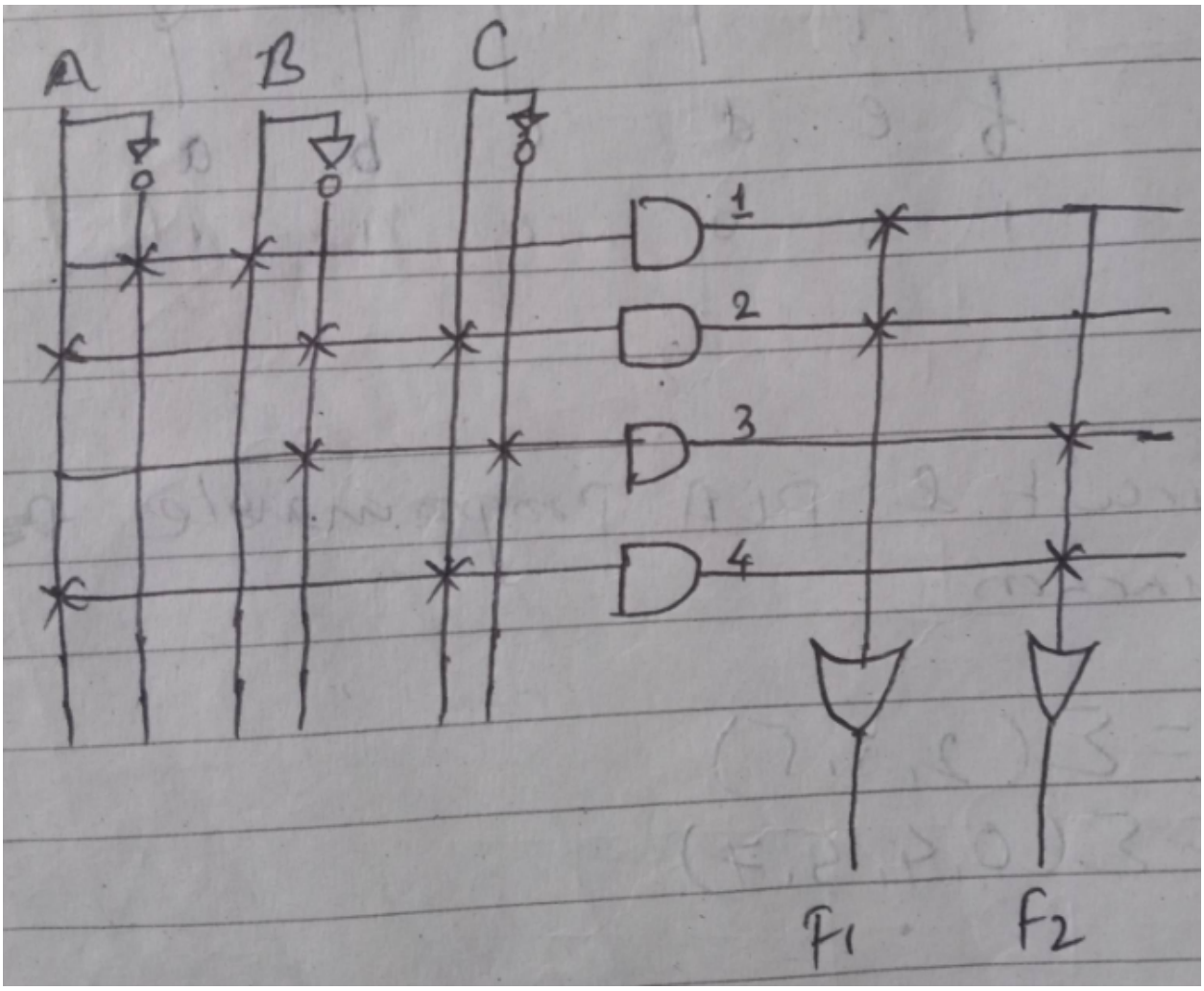
First of all let's find the simplified function of given minterms using k-map



Programmable table:

Product Term	A	B	C	F_1	F_2
1	0	1	-	1	-
2	1	0	1	1	-
3	-	0	0	-	1
4	1	-	1	-	1

PLA Circuit:



BCD to Access-3 Code Converter:

A BCD digit can be converted to its corresponding Excess-3 code by simply adding 3 to it. Since we have only 10 digits (0 to 9) in decimal, we don't care about the rest and marked them with a cross (X).

Let **A, B, C,** and **D** be the bits representing the binary numbers, where **D** is the LSB and **A** is the MSB.

Let **w, x, y,** and **z** be the bits representing the gray code of the corresponding binary numbers, where **z** is the LSB and **w** is the MSB.

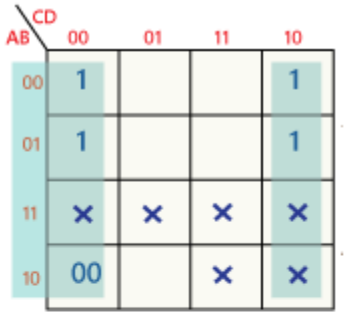
The truth table for the conversion is given below. The X mark is don't care condition.

Truth Table:

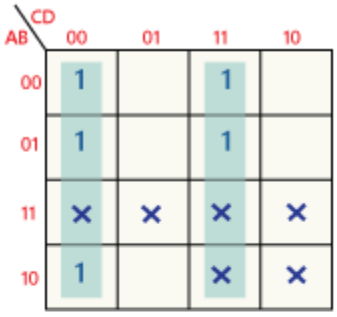
Decimal Number	**BCD Code**				**Excess-3 Code**			
	A	**B**	**C**	**D**	**W**	**x**	**y**	**z**
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X

12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

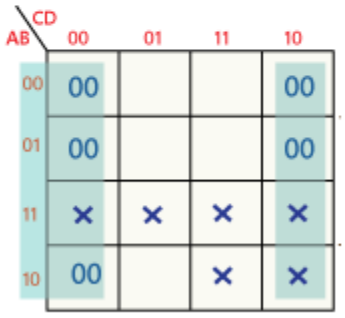
Now, we will use the K-map method to design the logical circuit for the conversion of BCD to Excess-3 code as:



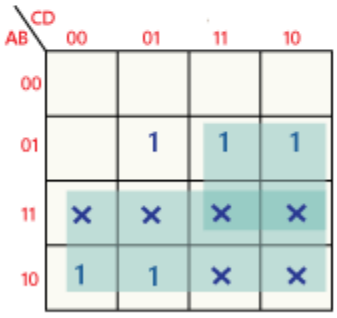
z=D′



y=CD+C′D′



x=B′C+B′D+BC′D′

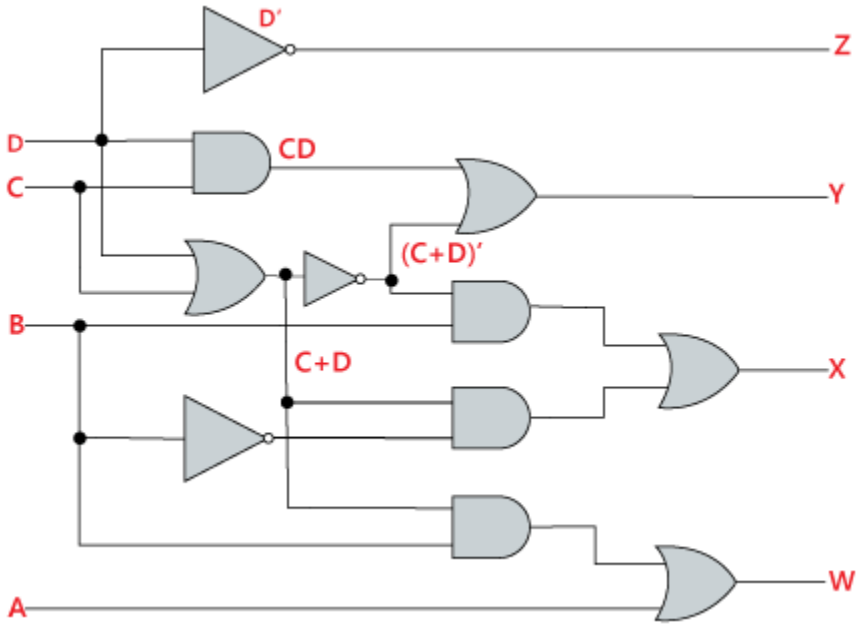


w=A+BC+BD

Boolean Equations are:

w=A+BC+BD
  =A+B(C+D)
x=B'C+B'D+BC'D'
 = B’(C+D)+BC’D’
y=CD+C'D'
z=D'

Logic Circuit:



Excess-3 to BCD code converter:

The process of converting Excess-3 to BCD is opposite to the process of converting BCD to Excess-3. The BCD code can be calculated by subtracting 3, i.e., 0011 from each four-digit Excess-3 code. Below is the truth table for the conversion of Excess-3 code to BCD.

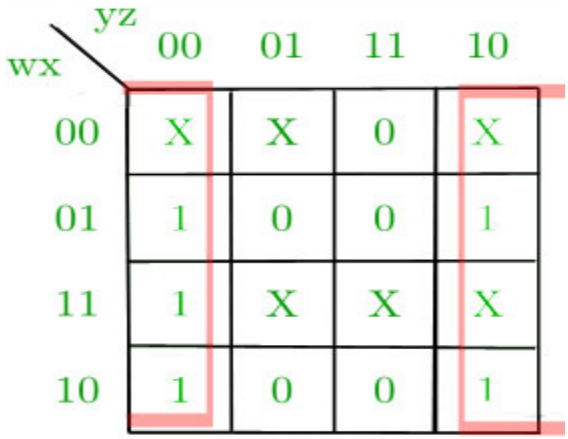
In the below table, the variables w, x, y, and z represent the bits of the Excess-3 code. The variable 'z' represents the LSB, and the variable 'w' represents the MSB. In the same way, the variables A, B, C, and D represent the bits of the binary

numbers. The variable 'D' represents the LSB, and the variable 'A' represents the MSB. The 'don't care conditions' is defined by the variable 'X'.

Truth Table:

Decimal Number	Excess-3 Code				BCD Code			
	w	**x**	**y**	**z**	**A**	**B**	**C**	**D**
0	0	0	0	0	X	X	X	X
1	0	0	0	1	X	X	X	X
2	0	0	1	0	X	X	X	X
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	1
11	1	0	1	1	1	0	0	0
12	1	1	0	0	1	0	0	1
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

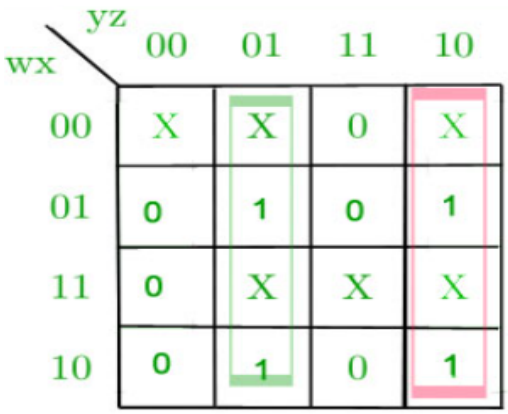
K-Map for D:



wx \ yz	00	01	11	10
00	X	X	0	X
01	1	0	0	1
11	1	X	X	X
10	1	0	0	1

D = z’

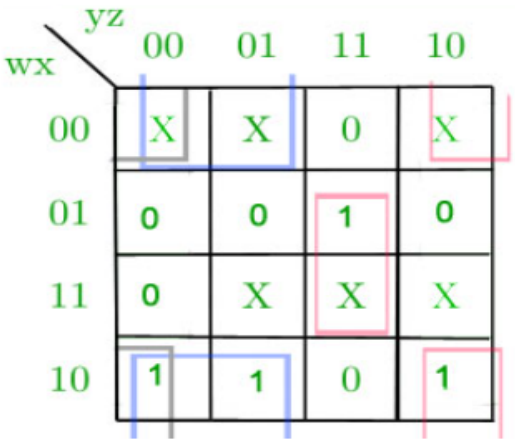
K-Map for C



wx \ yz	00	01	11	10
00	X	X	0	X
01	0	1	0	1
11	0	X	X	X
10	0	1	0	1

C = y’z+yz’

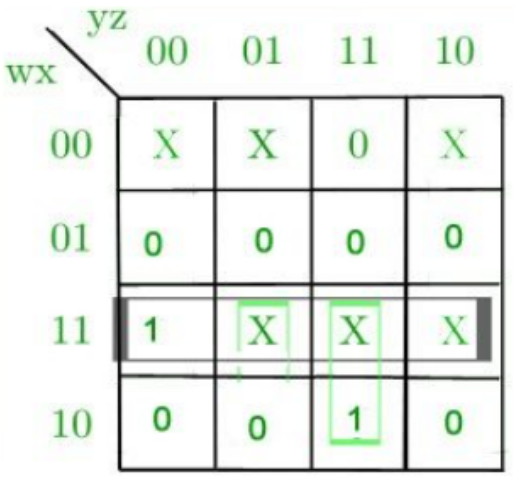
K-Map for B:



wx \ yz	00	01	11	10
00	X	X	0	X
01	0	0	1	0
11	0	X	X	X
10	1	1	0	1

B = x’z’+x’y’+xyz

K-map for A:



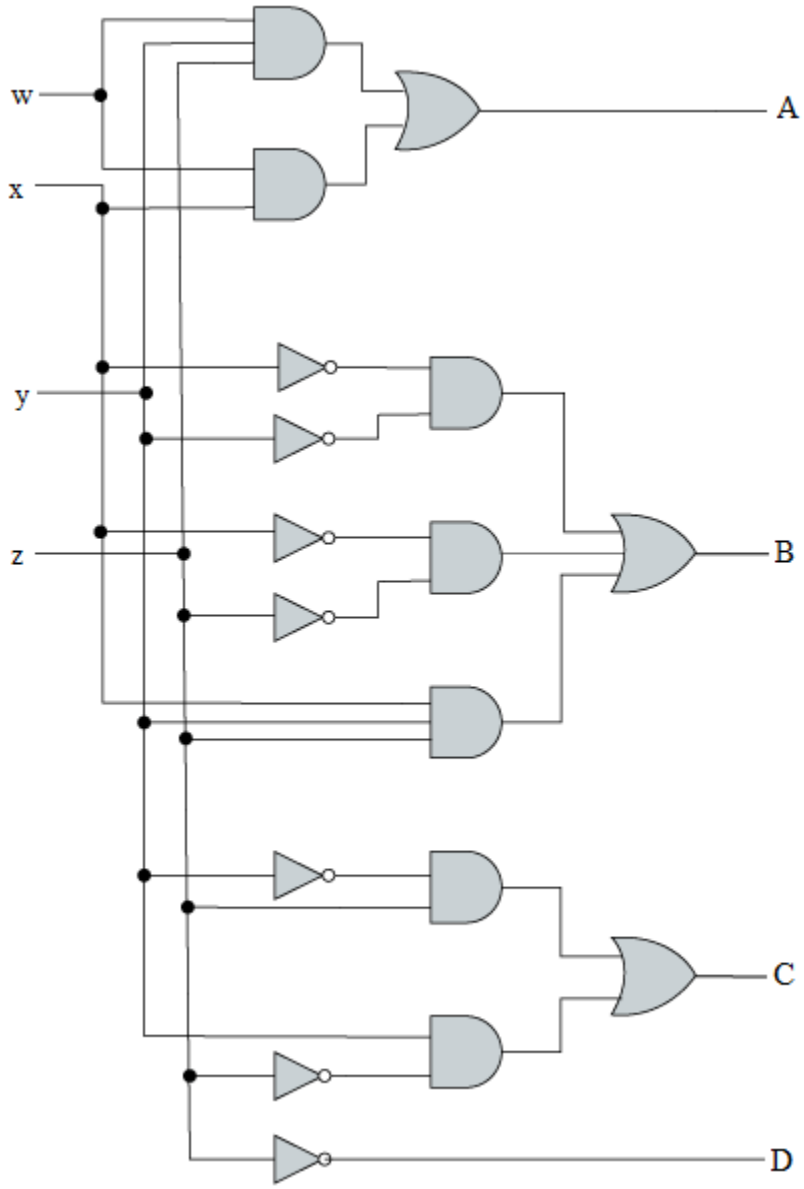
wx \ yz	00	01	11	10
00	X	X	0	X
01	0	0	0	0
11	1	X	X	X
10	0	0	1	0

A = wx+wyz

Boolean Equations:

A = wx+wyz
B = x’z’+x’y’+xyz
C = y’z+yz’
D = z’

Logic Circuit:



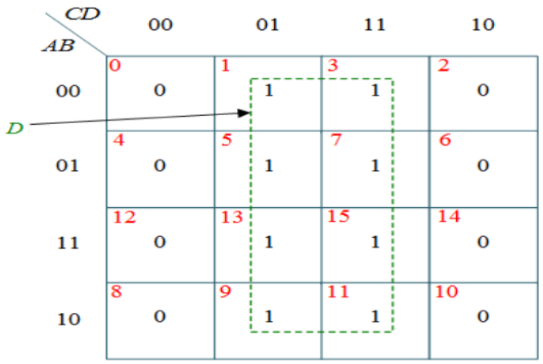
Binary to BCD code converter:

BCD is binary coded decimal number, where each digit of a decimal number is respected by its equivalent binary number. In BCD number is represent 0 to 9 which means 4 bit is required to represent the BCD. But only 0 to 9 are considered and remaining 6 bits are don't care.

Truth Table:

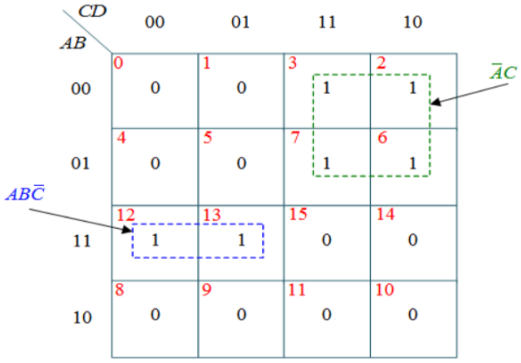
Binary Code	Decimal Number	BCD Code
A B C D		**B_4 :$B_3B_2B_1B_0$**
0 0 0 0	0	0 : 0 0 0 0
0 0 0 1	1	0 : 0 0 0 1
0 0 1 0	2	0 : 0 0 1 0
0 0 1 1	3	0 : 0 0 1 1
0 1 0 0	4	0 : 0 1 0 0
0 1 0 1	5	0 : 0 1 0 1
0 1 1 0	6	0 : 0 1 1 0
0 1 1 1	7	0 : 0 1 1 1
1 0 0 0	8	0 : 1 0 0 0
1 0 0 1	9	0 : 1 0 0 1
1 0 1 0	10	1 : 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

K-map for B0



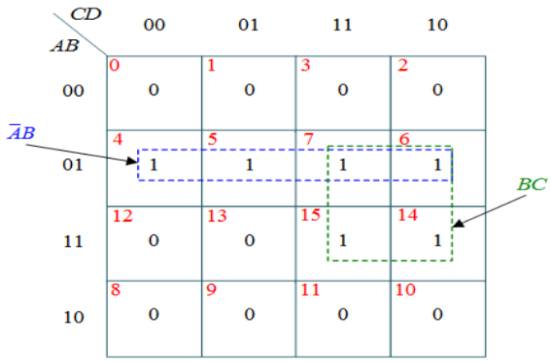
B0 = D

K-map for B1



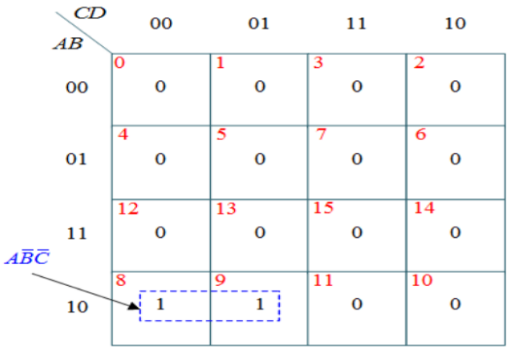
B1 = ABC'+A'C

K-map for B2



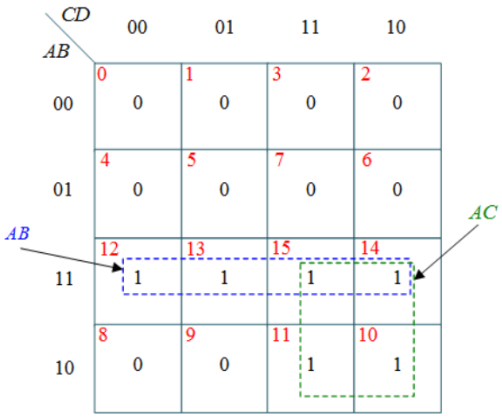
B2 = A'B+BC

K-map for B3



B3 =AB'C'

K-map for B4



B4 = AB+AC

Boolean Equations:

B0 = D
B1 = ABC'+A'C
B2 = A'B+BC
B3 =AB'C'
B4 = AB+AC

End of Unit-4

Unit-5: Sequential Logic

Introduction to Sequential Circuit:

In a sequential circuit, the output produced depends not only on the applied input but also on the past history of the outputs.

The sequential circuit consists of a combinational logic, which gets the external input and the previous state output through the inbuilt memory unit as feedback. The memory unit may be a latch or a flip flop.

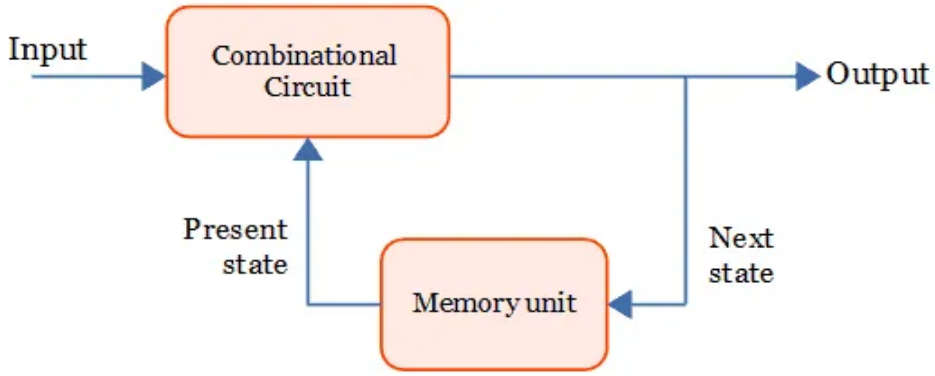


Fig: Block Diagram of Sequential Circuit

The present state and the external inputs determine the outputs and the next state of the sequential circuit. So, a sequential circuit works on a time sequence of external inputs, internal states and outputs.

Designing a sequential circuit involves the representation of sequential circuit models. It includes a state diagram, state table, reduced state table, reduced state diagram.

The sequential circuits are classified based on the clock pulses given to the memory units. There are two types as follows

1. Synchronous sequential circuit
2. Asynchronous sequential circuit.

Synchronous sequential circuit

In these circuits, change in input can affect the memory elements only upon the activation of the clock pulse. The memory units are clocked flip-flops.

The circuit will change its state for every clock pulse. Hence it is also called clocked sequential circuits.

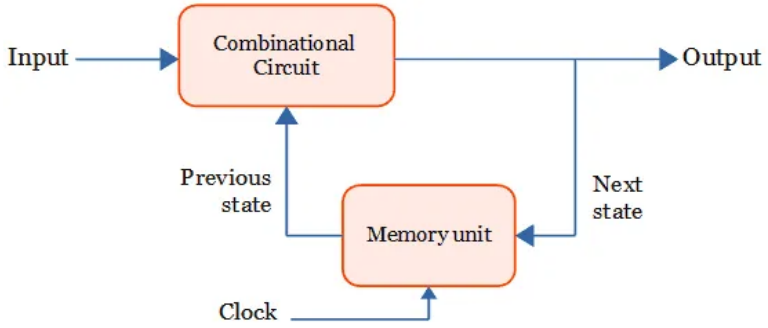


Fig: Synchronous Clocked Sequential Circuit

Basic Models of Clocked Sequential Machines

The synchronous sequential circuits are represented by two models. These models have a finite number of states and are hence called finite state machine models.

- Moore Model/ Moore State Machine
- Mealy Model/ Mealy State Machine

Moore model

In this sequential model, the output depends only on the present state of the flip flops, the sequential circuit is called Moore circuit or Moore Machine.

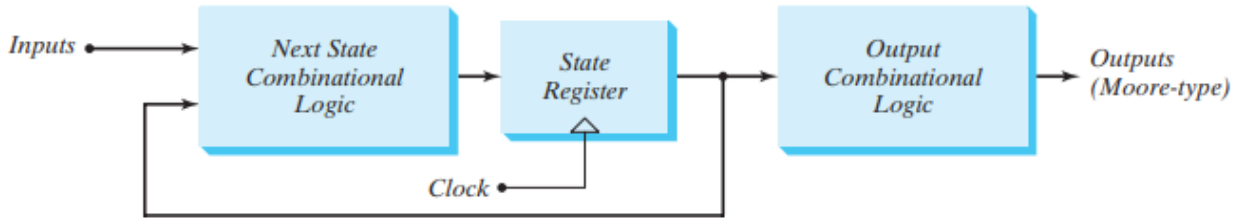


Fig: Block Diagram of Moore Model

An example of Moore model circuit is shown in the below diagram. In this circuit, two JK flip flops and AND gates. Two JK flip-flops are connected together. Circuit has one input X and one output Y.

The input is connected only to the first JK flip flop. The output is the multiplication of the next state output, but not of the input.

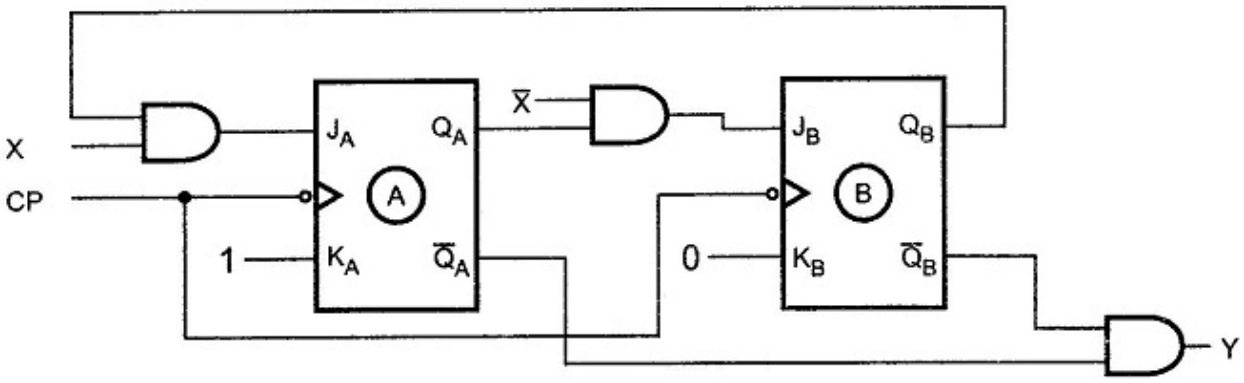


Fig: Circuit Diagram of Moore Model

As shown in the above figure, input is used to determine the inputs of the flip-flops. It is not used to determine the output. The output is derived using only present states of the flip-flops or combination of it (in this case $Y=Q_AQ_B$).

Mealy model

The output depends on both the present state of the flip flops and the input, the sequential circuit is called Mealy Model circuit or Mealy Machine.

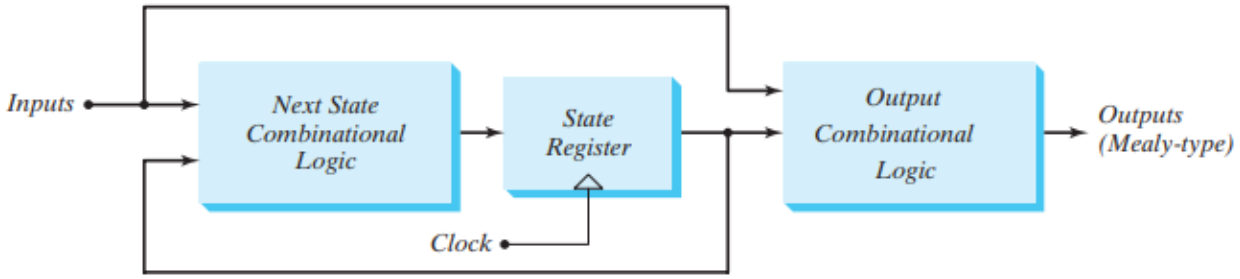


Fig: Block diagram of Mealy Model

The circuit shown below is an example of Mealy circuit model. As you can see, the output is the product of the present state of flip flop and the input.

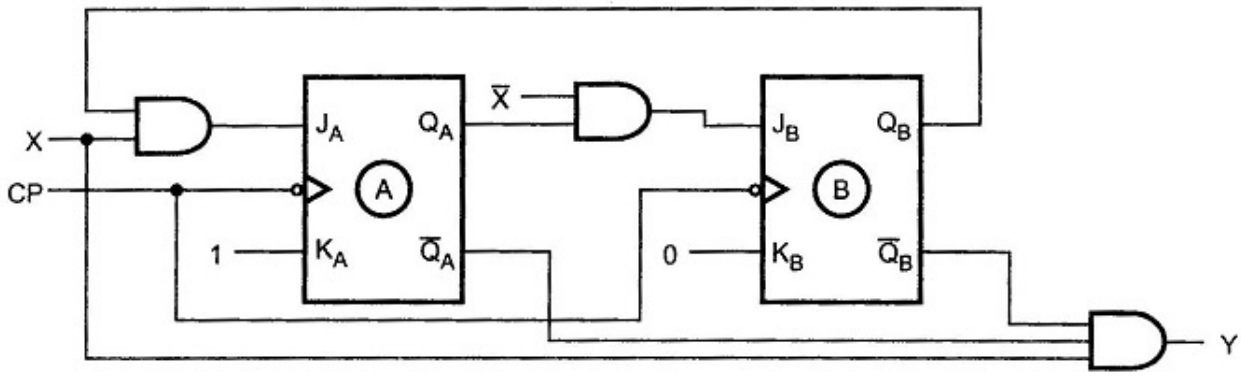


Fig: Circuit Diagram of Mealy Model

Looking at above figure, we can easily realize that, changes in the input within the clock pulses cannot affect the state of the flip-flop. However, they can affect the output of the circuit. Due to this, if the input variations are not synchronized with the clock, the derived output will also not be synchronized with the clock and we get false output. The false outputs can be eliminated by allowing input to change only at the active transition of the clock (in our example HIGH-to-LOW).

Asynchronous sequential circuit

The clock signals are not used by the Asynchronous sequential circuits. So, the changes in the input can change the state of the circuit. The asynchronous circuits do not use clock pulses. The internal state is changed when the input variable is changed. The un-clocked flip-flops or time-delayed are the memory elements of asynchronous sequential circuits. The asynchronous sequential circuit is similar to the combinational circuits with feedback.

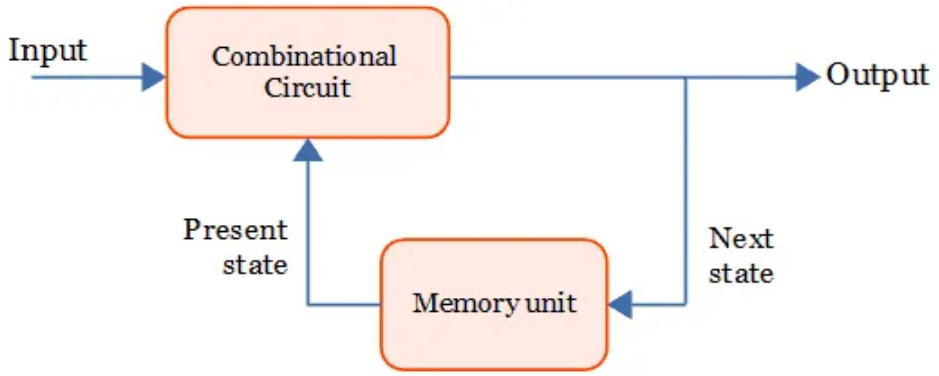


Fig: Block Diagram of Asynchronous Sequential Circuit

What is a clock pulse?

A clock pulse is a continuously changing signal that oscillates between a high state and a low state. The common type of clock signal is a square type, which has 50% duty cycle with a fixed and constant frequency. A clock pulse is shown below.

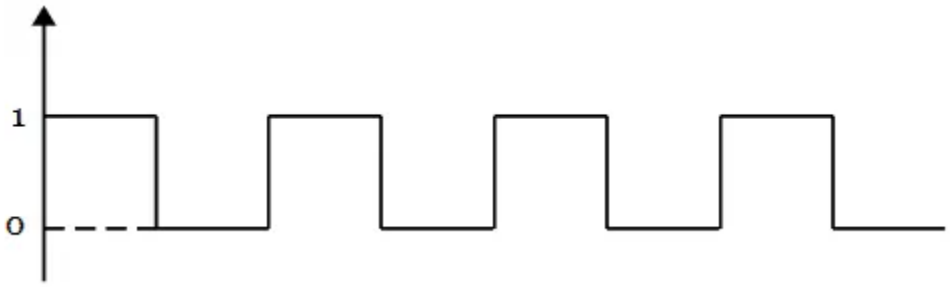


Fig: Clock Pulse

While applying the clock pulse to the flip flop, it gets triggered by two ways, Level triggering and edge triggering.

Triggering:

The state of a flip-flop is changed by a momentary change in the input signal. This change is called a trigger and the transition it causes is said to trigger of flip-flop. The triggering input is called clock. At the specific interval of clock, triggering of flip-flop is takes place.

There are two types of triggering

- Level Triggering
- Edge Triggering

Level triggering

In this, the flip flop is triggered (change its state) only during the high-level or the low-level of the clock pulse. In other words, the flip-flop changes its state, when active low or active high level is maintained at the clock signal.

Based on the level of triggering, it is of two types

- Positive Level Triggering
- Negative Level Triggering

Positive level triggering:

If the flip flop is triggered (change its state) at the positive level of the clock pulse, then it is said to be a positive level triggering.

Block Diagram:

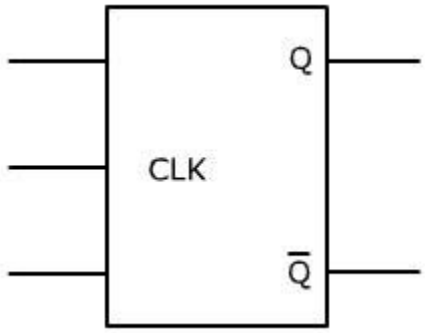


Fig: Positive level triggering

Timing Diagram:

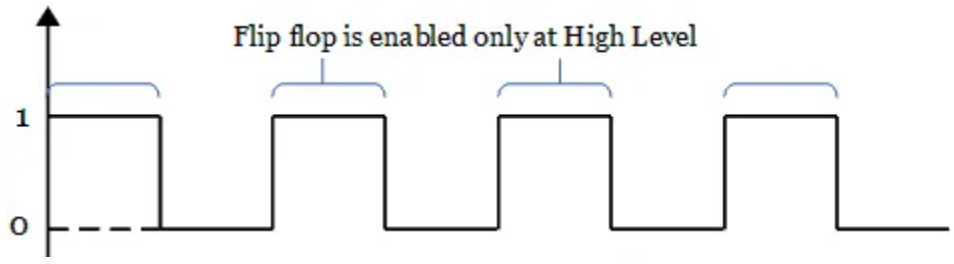


Fig: Positive level triggering

Negative level triggering:

If the flip flop is triggered (change its state) at the negative level of the clock pulse, then it is said to be negative level triggering.

Block Diagram:

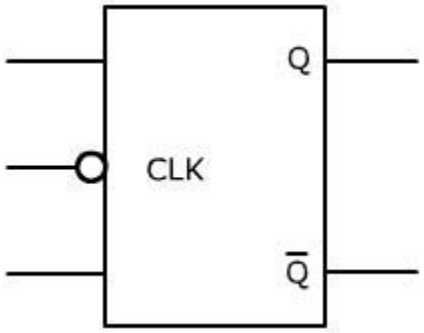


Fig: Negative level triggering

Timing Diagram:

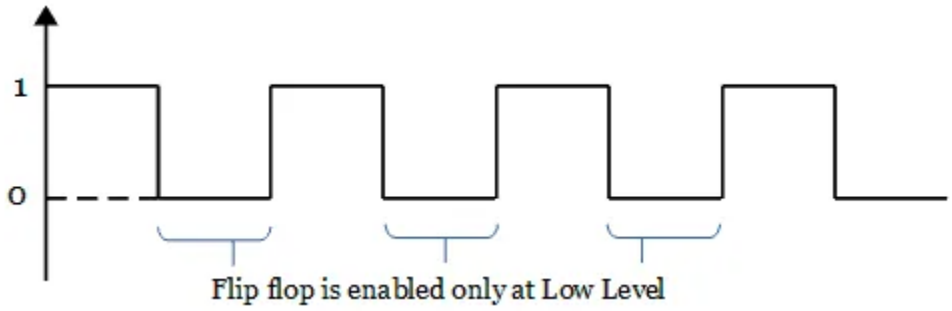


Fig: Negative level triggering

Edge triggering

In edge triggering, the flip flop changes its state during the positive edge or negative edge of the clock pulse.

There are two types of edge triggering.

- Positive Edge Triggering
- Negative Edge Triggering

Positive edge triggering:

If the flip flop is triggered (change its state) only at the positive edge of the clock pulse, then it is said to be a positive edge triggering.

Block Diagram:

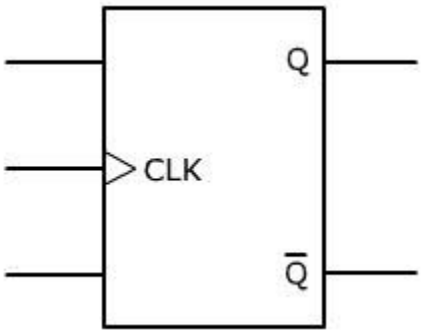


Fig: Positive edge triggering

Timing Diagram:

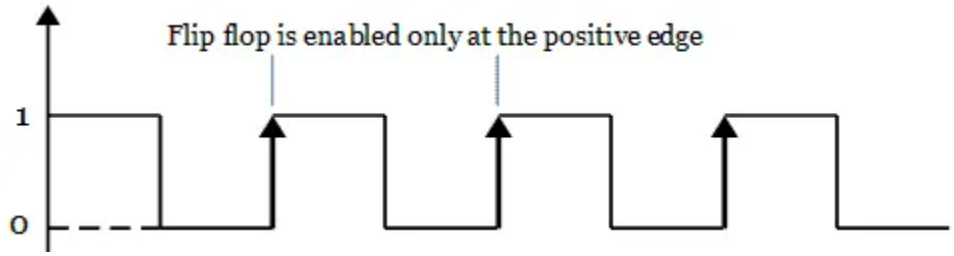


Fig: Positive edge triggering

Negative edge triggering:

If the flip flop is triggered (change its state) only at the negative edge of the clock pulse, then it is said to be a negative edge triggering.

Block Diagram:

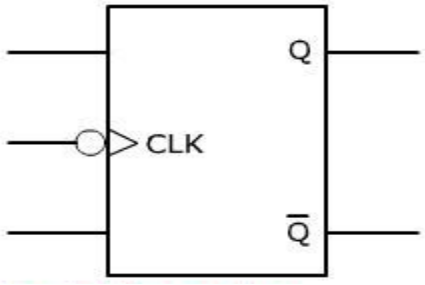


Fig: Negative edge triggering

Timing Diagram:

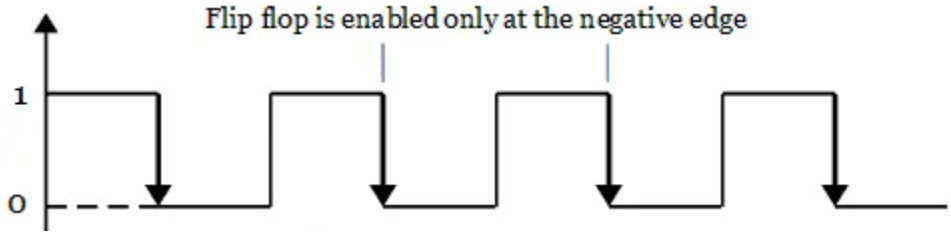


Fig: Negative edge triggering

Difference between Combinational and Sequential Circuit

Combinational Circuit	Sequential Circuit
The output always depends on the combination of input variables.	The Output produced depends on both the present state and the next state variable.
A memory unit is not necessary.	Memory is necessary to store the next state variables.
The propagation delay is less due to the absence of memory units and so they are faster.	The speed is slower than combinational circuits.

Combinational Circuit	Sequential Circuit
Easy to design.	They are comparatively harder to design.

Flip-Flops:

J-K Flip Flop

The JK flip flop is one of the most used flip flops in digital circuits. The JK flip flop is also called universal flip flop having two inputs 'J' and 'K' and two outputs, Q and Q', which are also known as the true and complement outputs.

The JK flip flop work in the same way as the SR flip flop. 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 JK flip flop, there are no invalid states even if both 'J' and 'K' flip flops are set to 1.

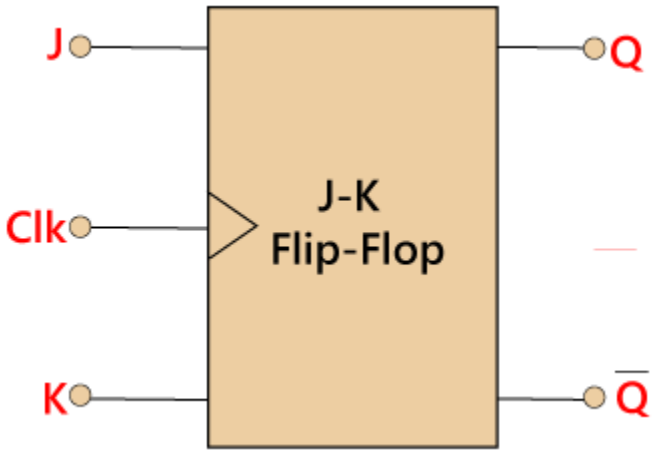
The JK flip-flop is used in a variety of applications, including storing and transferring data, controlling the timing of circuits, and implementing counters and state machines. It is often preferred over the simpler SR flip-flop because it has the ability to toggle its state, which makes it more versatile and easier to use in certain types of circuits.

The JK flip-flop has four possible input and output combinations:

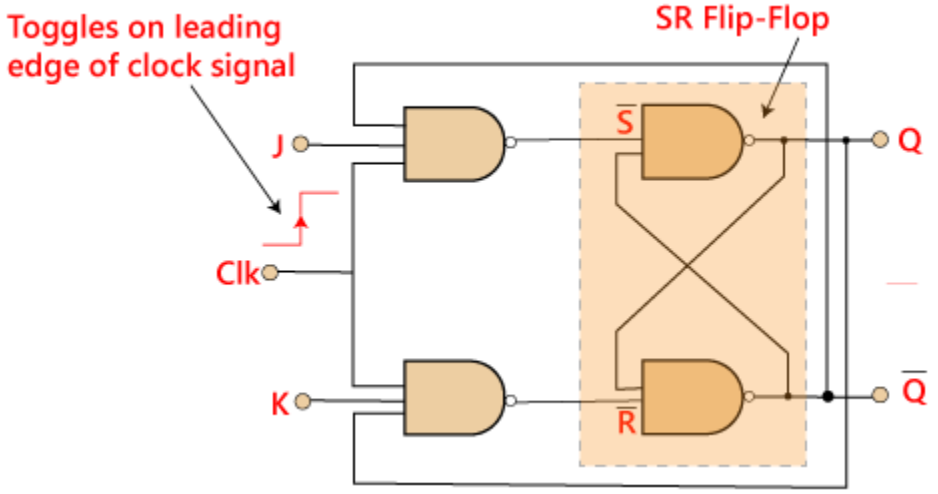
- J=0, K=0: The flip-flop remains in its current state.
- J=0, K=1: The Q output becomes 0, and the Q' output becomes 1.

- J=1, K=0: The Q output becomes 1, and the Q' output becomes 0.
- J=1, K=1: The flip-flop toggles its state, meaning that if the Q output was 1, it becomes 0, and if the Q output was 0, it becomes 1.

Block Diagram:



Circuit Diagram:



Truth Table:

Inputs			Output
Clk	**J**	**K**	**Q_{n+1}**
0	X	X	Q_n (Memory/Halt)
1	0	0	Q_n (Memory)
1	0	1	0

1	1	0	1
1	1	1	Q'_n (Toggle)

Note: Testing

Input: *J = K = 0, (Assume Q = 1, Q' = 0 in present state)*

Output: *Q = 1, Q' = 0, it is same as previous state which means output halts or memory.*

Input: *J = 0, K = 1 (Q=0, Q' = 1 in present state)*

Output*: Q = 0, Q' = 1, so, next state* $Q_{n+1} = 0$

Input: *J = 1, K = 0 (Q = 0, Q' = 1 in present state)*

Output: *Q = 1, Q' = 0, so, next state* $Q_{n+1} = 1$

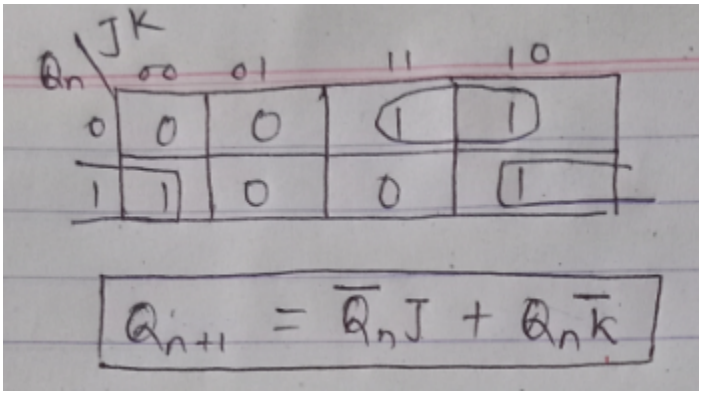
Input: *J=K=1 (Q = 1, Q' = 0 in present state)*

Output: *Q = 0,1,0,1,0,1,……, Q' = 1,0,1,0,1,0,……, so, next state* $Q_{n+1} = Q_n'$ *this is called race around condition.*

Characteristics Table:

Q_n	J	K	Q_{n+1}
0	0	0	0 (Q_n)
0	0	1	0
0	1	0	1
0	1	1	1 (Q'_n)
1	0	0	1 (Q_n)
1	0	1	0
1	1	0	1
1	1	1	0 (Q'_n)

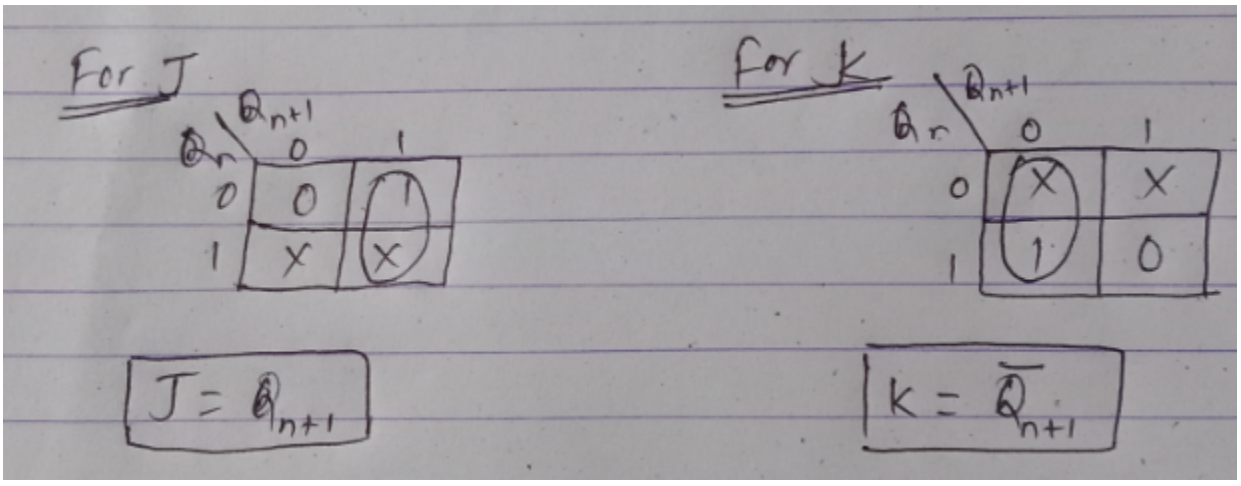
Now, derive the equation for Q_{n+1} from above table using k-map



Excitation table:

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

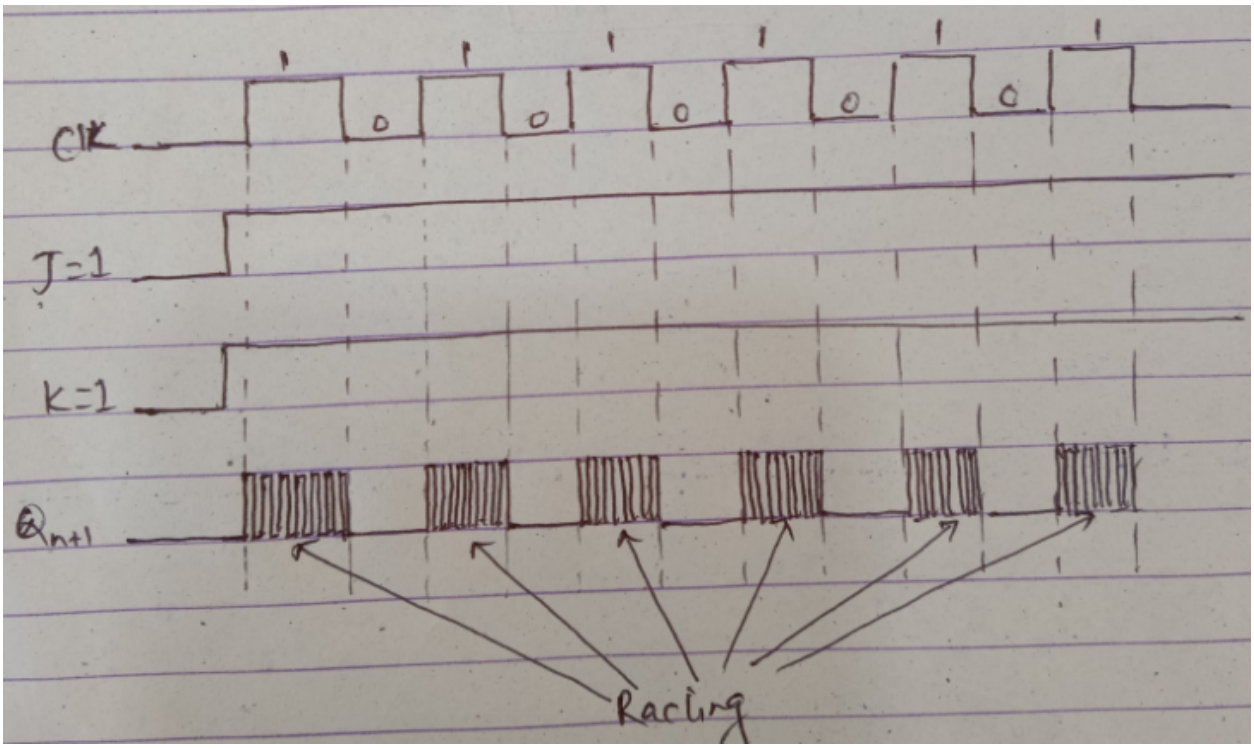
Now, derive the equation for J and K from above table using k-map



Race-around Condition in JK Flip-flop:

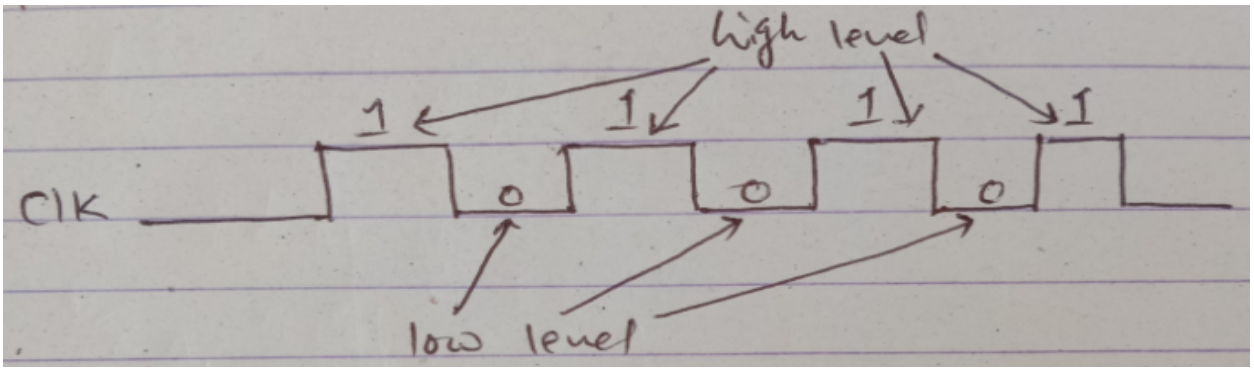
For J-K flip-flop, if J=K=1, and if clk=1 for a long period of time, then Q output will toggle as long as CLK is high, which makes the output of the flip-flop unstable or uncertain. This problem is called race around condition in J-K flip-flop. This is the main drawback of JK flip flop.

Timing diagram of J-K flip flop:



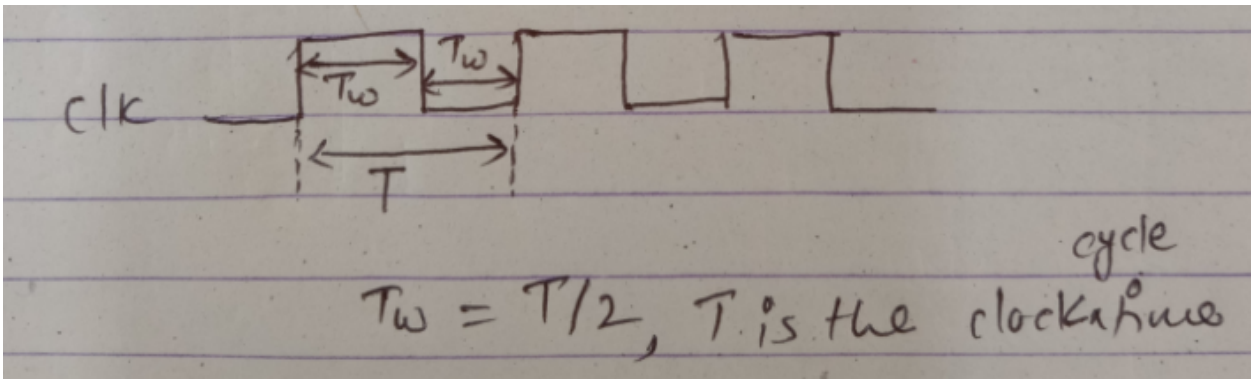
Condition for Race-around occurrence:

Condition 1: Level Triggered JK flip flop



Condition 2: When J=K=1 (Toggle Mode)

Condition 3: $T_w > T_d$, where T_w is time of clock level (high or low), T_d is time taken by filp flop to calculation (Propagation delay).



Methods to avoid the Race-around conditions:

1. T/2 <Propagation delay
2. Edge Triggering
3. Master-Slave Flip flop

T/2 < Propagation Delay:

When the time period of clock level (*where flip flop is being triggered or flip flop is in active state*) is less than the propagation delay (*which is the execution time period of a flip flop*) then clock will goes to low level (*where flip flop will be deactivated*) before getting the feedback signal from fillip flop output (*which leads the race-around condition)*. This will avoid the race around condition.

This is the theoretical condition but in practical this is very hard to implement.

Edge Triggering:

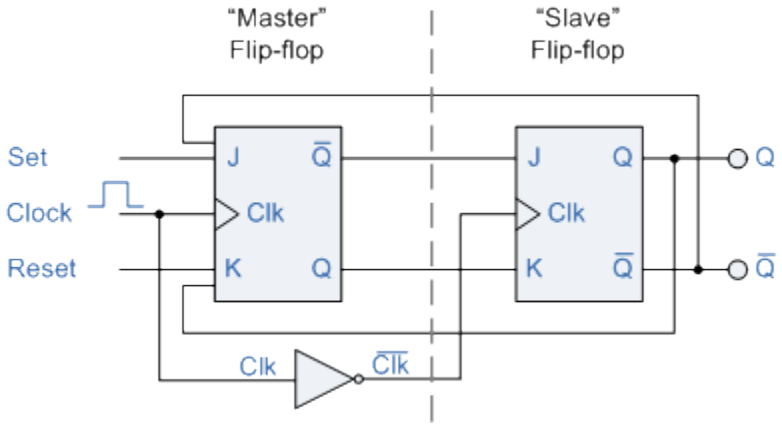
In Edge triggering flip flop changes its state either at positive edge or negative edge of the clock pulse. The time period of edge of clock pulse is very low than the propagation delay so flip flop goes the next state before getting the feedback. This will avoid the race-around condition.

Master-Slave Flip Flop:

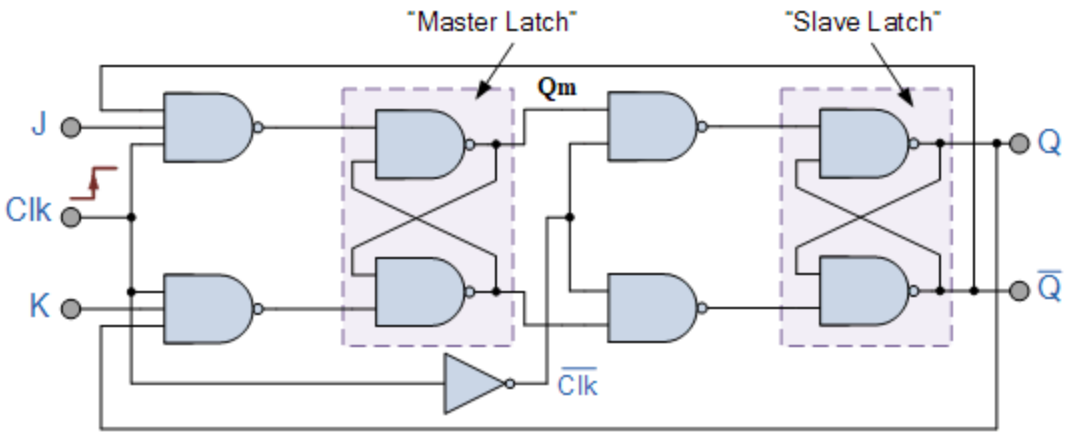
The Master-Slave Flip-Flop is basically a combination of two JK flip-flops connected together in a series configuration. Out of these, one acts as the **"master"** and the other as a **"slave"**. The output from the master flip flop is connected to the two inputs of the slave flip flop whose output is fed back to inputs of the master flip flop.

In addition to these two flip-flops, the circuit also includes an **inverter**. The inverter is connected to clock pulse in such a way that the inverted clock pulse is given to the slave flip-flop. In other words if clk=0 for a master flip-flop, then clk=1 for a slave flip-flop and if clk=1 for master flip flop then clk=0 for slave flip flop.

Block Diagram of Master-Slave FF:



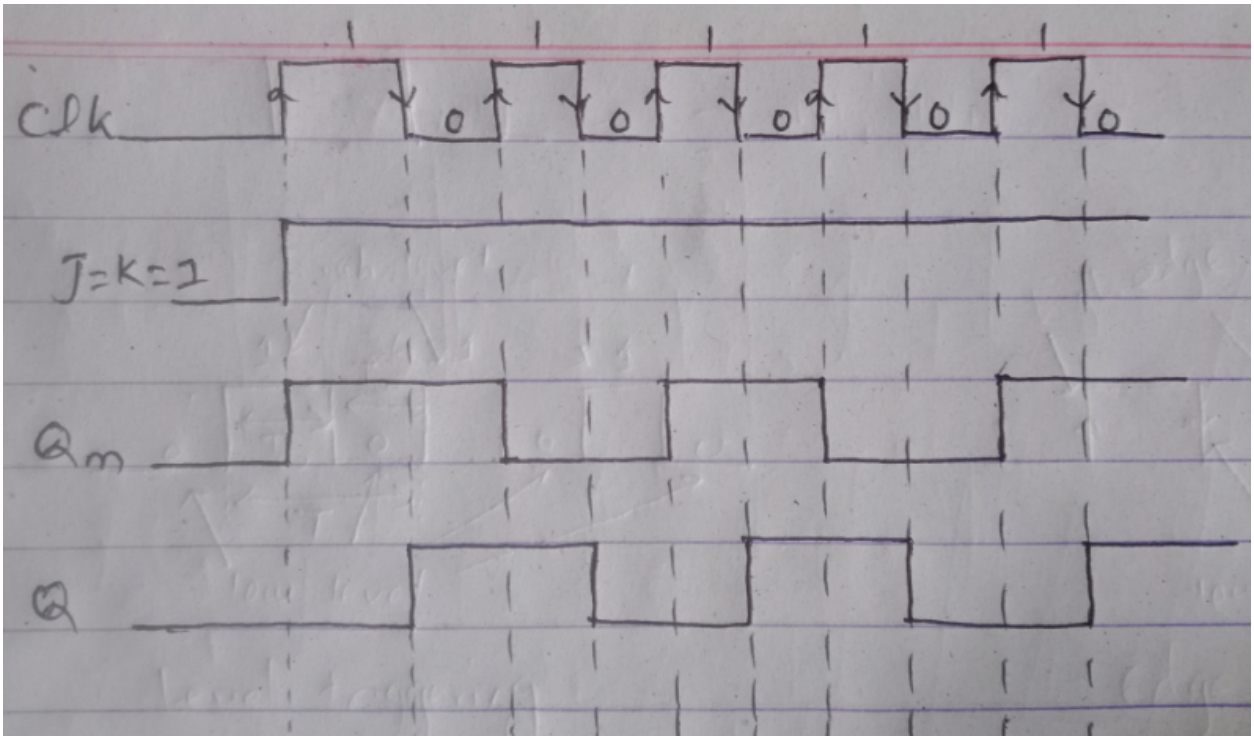
Circuit Diagram of Master-Slave FF:



Here,

J=K=1, clk = 1, Q_m = Active, Q = Memory
J=K=1, clk = 0, Q_m = Memory Q = Active

Timing Diagram of Master-Slave FF



The Master and Slave both FF are positive edge triggering that means, to activate the FF both need high clock pulse, but due to use to inverter in clock pulse for

Slave FF it's impossible to make clock pulse high at same time for both FF, so they can't activate simultaneously.

When Master FF is activate it needs current output feedback from slave to occur race-around condition but at that time Slave FF is deactivated so it can't get feedback from Slave FF hence race-around condition is avoided.

T Flip Flop:

We can construct the "T Flip Flop" by making changes in the "JK Flip Flop". The "T Flip Flop" has only one input, which is constructed by connecting the input of JK flip flop. This single input is called T. The "T Flip Flop" is also called single input "JK Flip Flop".

If the toggle input is HIGH, the T flip-flop changes state (toggles) when the clock signal is applied. If the toggle input is LOW, the T flip-flop holds the previous state.

Block Diagram:

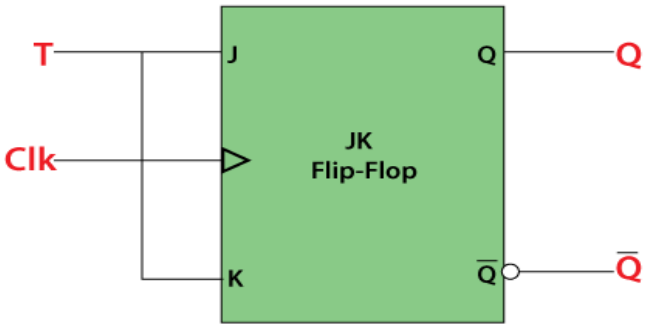


Fig: Block Diagram of T FF

Circuit Diagram:

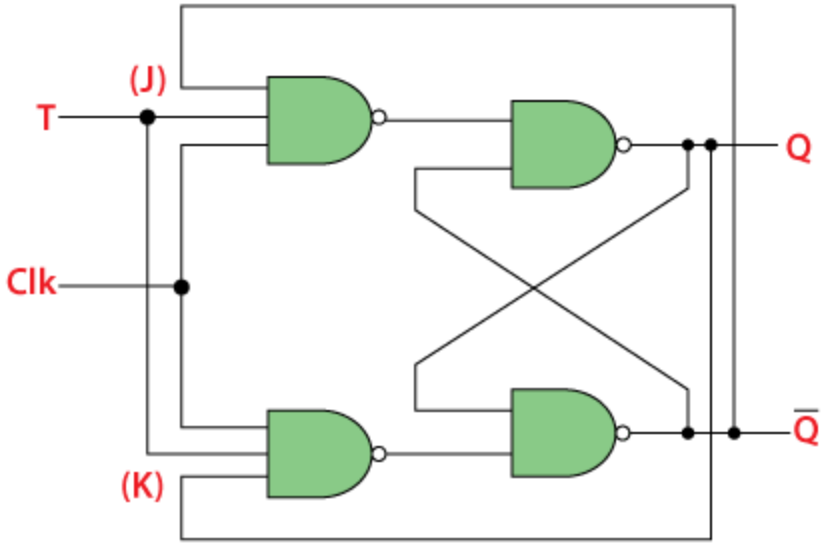


Fig: Circuit Diagram of T FF

Truth Table:

Clk	T	Q_{n+1}
0	x	Q_n (Memory)
1	0	Q_n (Memory)
1	1	Q'_n (Toggle)

Characteristic Table:

Q_n	T	Q_{n+1}
0	0	0 (Q_n)
0	1	1 (Q'_n)
1	0	1 (Q_n)
1	1	0 (Q'_n)

This is the XOR combination so,

$Q_{n+1} = Q_n$ xor T

Excitation Table:

Q_n	Q_{n+1}	T
0	0	0

0	1	1
1	0	1
1	1	0

$T = Q_n$ xor Q_{n+1}

State Diagram and State Table:

State Diagram:

The state diagram is the pictorial representation of the behavior of sequential circuits. It clearly shows the transition of states from the present state to the next state and output for a corresponding input.

- In this diagram, each present state is represented inside a circle.
- The transition from the present state to the next state is represented by a directed line connecting the circles.
- If the directed line connects the circle itself, which indicates that there is no change in the state (the next state is the same as the present state).

State Table:

State table consist of all the binary value of present state, next state, inputs and outputs of the state diagram.

State Diagram and State Table for Mealy Model:

For **Mealy Model Circuit,** the directed line is labeled with binary numbers separated with '/', as shown in the below diagram.

The input value, which causes the transition to occur is labeled at before '/'

i.e. **'x/'.**

The output produced for the corresponding input is labeled at after '/'

i.e. **'/x'.**

State Diagram for Mealy Model:

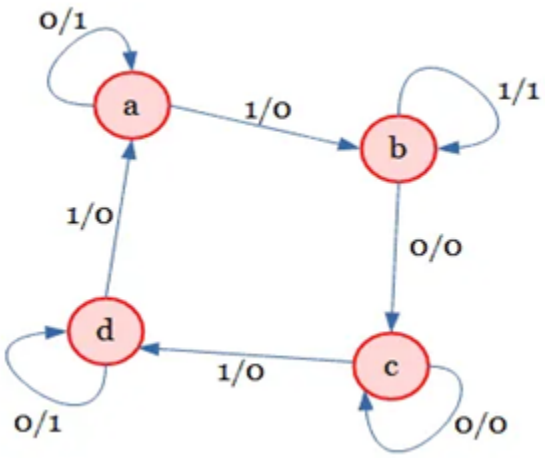


Fig: State Diagram for Mealy Model

State Table for Mealy Model:

Here, x=0 x=1 are inputs in both NS and Output

Present State (PS)	Next State (NS) x=0	x=1	Output x=0	x=1
a	a	b	1	0
b	c	b	0	1
c	c	d	0	0
d	d	a	1	0

State Diagram and State Table for Moore Model:

For **Moore Model Circuit,** the directed lines are labeled with only one binary number. It is nothing but the input value which causes the transition.

The output value is indicated inside the circle below the present state. It is because, in Moore model, the output depends on the present state but not on the input.

State diagram of Moore Model:

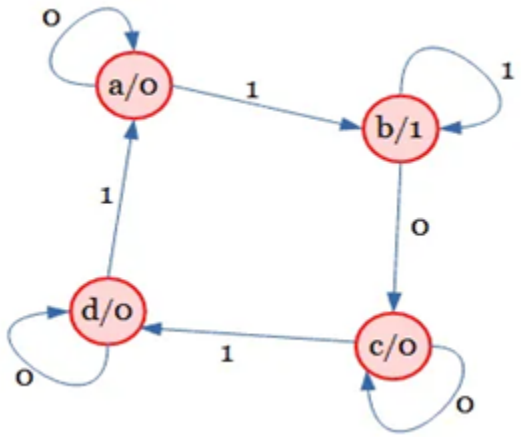


Fig: State Diagram of Moore Model

State Table for Moore Model:

Here, x=0 x=1 are inputs in both NS and Output

Present State (PS)	Next State (NS) x=0	x=1	Output x=0	x=1
a	a	b	0	1
b	c	b	0	1
c	c	d	0	0
d	d	a	0	0

State Reduction

The design of a sequential circuit starts from a set of specifications and culminates (reaches) in a logic diagram. Two sequential circuits may exhibit the same input-output behavior, but have a different number of internal states in their state diagram.

Here we are going to discuss certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flip-flops reduces the cost of a circuit.

State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external, input-output requirements unchanged.

Example:
Since **n** flip- flops produce $\mathbf{2^n}$ stales, a reduction in the number of states may result in a reduction in the number of flip-flops. We will illustrate the state-reduction procedure with an example. As shown in following state diagram.

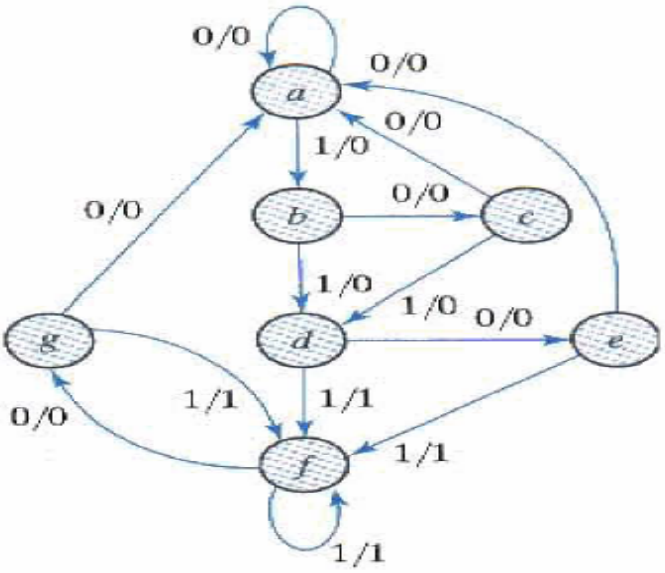


Fig: State diagram of Mealy Model

State Table:

Present State (PS)	Next State (NS) x=0	x=1	Output x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1

e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Table above is obtained directly from the state diagram, so In order to reduce the states the following algorithm should be followed:

"Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state."

When two states are equivalent, one of them can be removed without altering the input-output relationships.

Here, the PS **e** and **g** have same NS i.e**. a** and **f** and same output i.e. **0 1** for x=0 and x=1 respectively.

Therefore, states **e** and **g** are equivalent and one of them states can be removed. Let's remove **g,** the procedure of removing a state and replacing it by its equivalent is demonstrated in Table below.

Present State (PS)	**Next State (NS)**		**Output**	
	x=0	x=1	x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

Here the row with present state **g** is removed, and state **g** is replaced by state **e** each time it occurs in the columns headed "Next State." Present state **f** now has next states **e** and **f** and outputs 0 and 1 for x = 0 and x = 1, respectively.

The same next states and outputs appear in the row with present state **d**. Therefore, states **d** and **f** are equivalent, and state **f** can be removed and replaced by **d.** The final reduced table is shown in Table below.

Reduced State Table:

Present State (PS)	Next State (NS) x=0	x=1	Output x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

The state diagram for the reduced table consists of only five states and is shown in figure bellow:

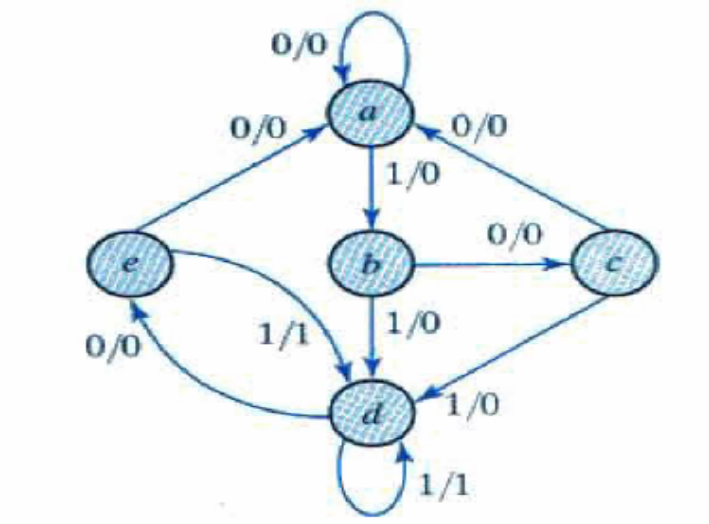


Fig: Reduced state diagram

Analysis of Clocked Sequential Circuit:

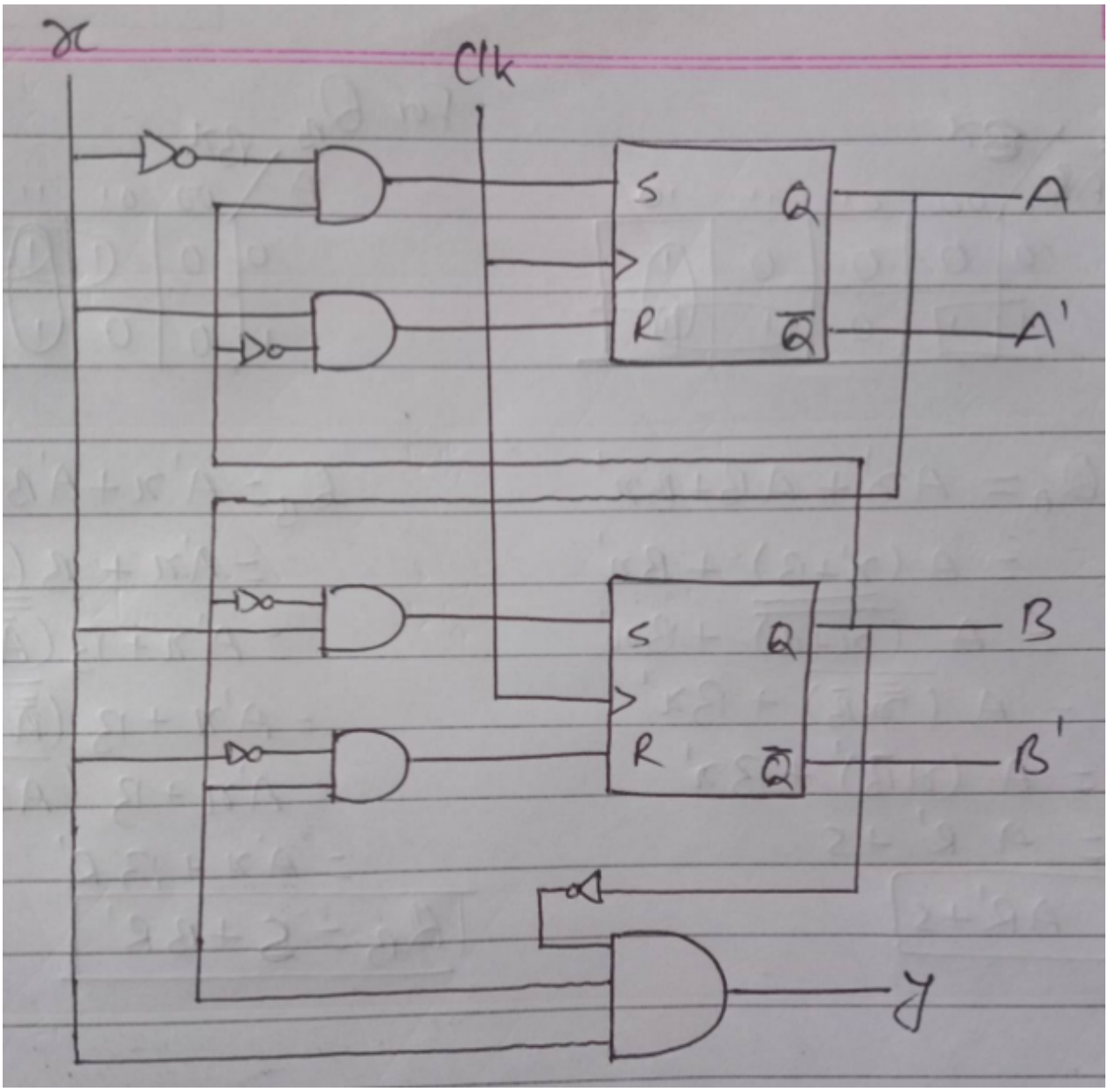
To analyze the clocked sequential circuit we have to find

- Input equations of FF
- Output equations (this is the combination of inputs and present state)

- Circuit diagram if not given
- State table
- Next state equation
- State diagram

Analysis of SR FF:

Analyze the following circuit diagram and draw the state table and state diagram.



Here, A and B are the present state of FF A and B

Step 1: Input equation of FF:

$S_A = Bx'$ $R_A = B'x$

$S_B = A'x$ $R_B = Ax'$

Step 2: Output equation

$y = AB'x$

Step 3: State table

To find the next state we have to consider the characteristics table of SR FF

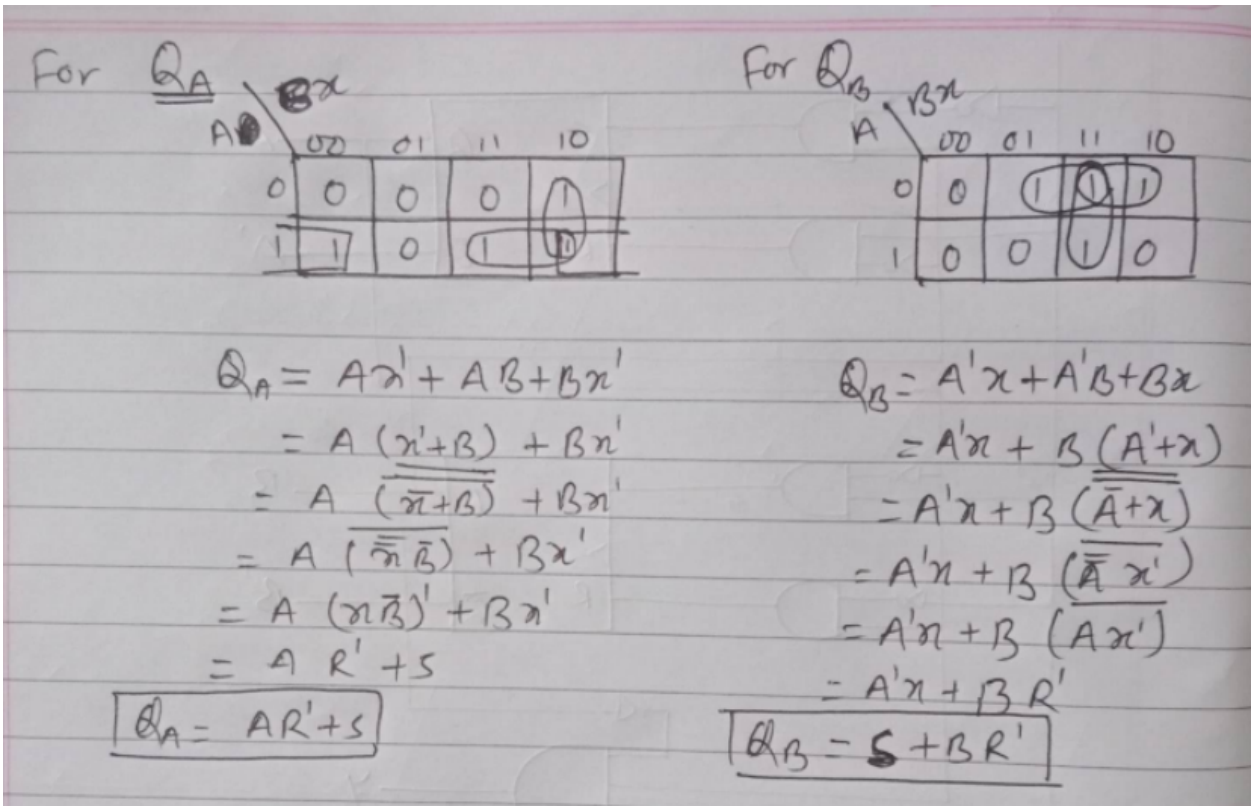
Q_n	S	R	Q_{n+1}
0	0	0	0 (Q_n)
0	0	1	0
0	1	0	1
0	1	1	Invalid
1	0	0	1 (Q_n)
1	0	1	0
1	1	0	1
1	1	1	Invalid

State Table

Present State		Input	FF Input				Next State		Output
A	B	X	S_A	R_A	S_B	R_B	$Q_{n+1}A$	$Q_{n+1}B$	y
0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	1	0
0	1	0	1	0	0	0	1	1	0
0	1	1	0	0	1	0	0	1	0
1	0	0	0	0	0	1	1	0	0
1	0	1	0	1	0	0	0	0	1
1	1	0	1	0	0	1	1	0	0
1	1	1	0	0	0	0	1	1	0

Step 4: Next state equation from state table

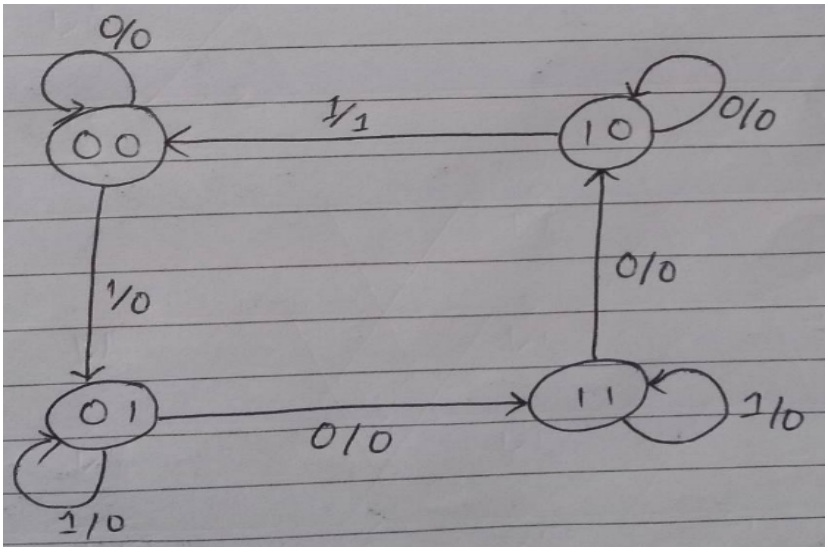
Suppose, Q_A and Q_B are the next state of FF A and B



Step 5: State Diagram

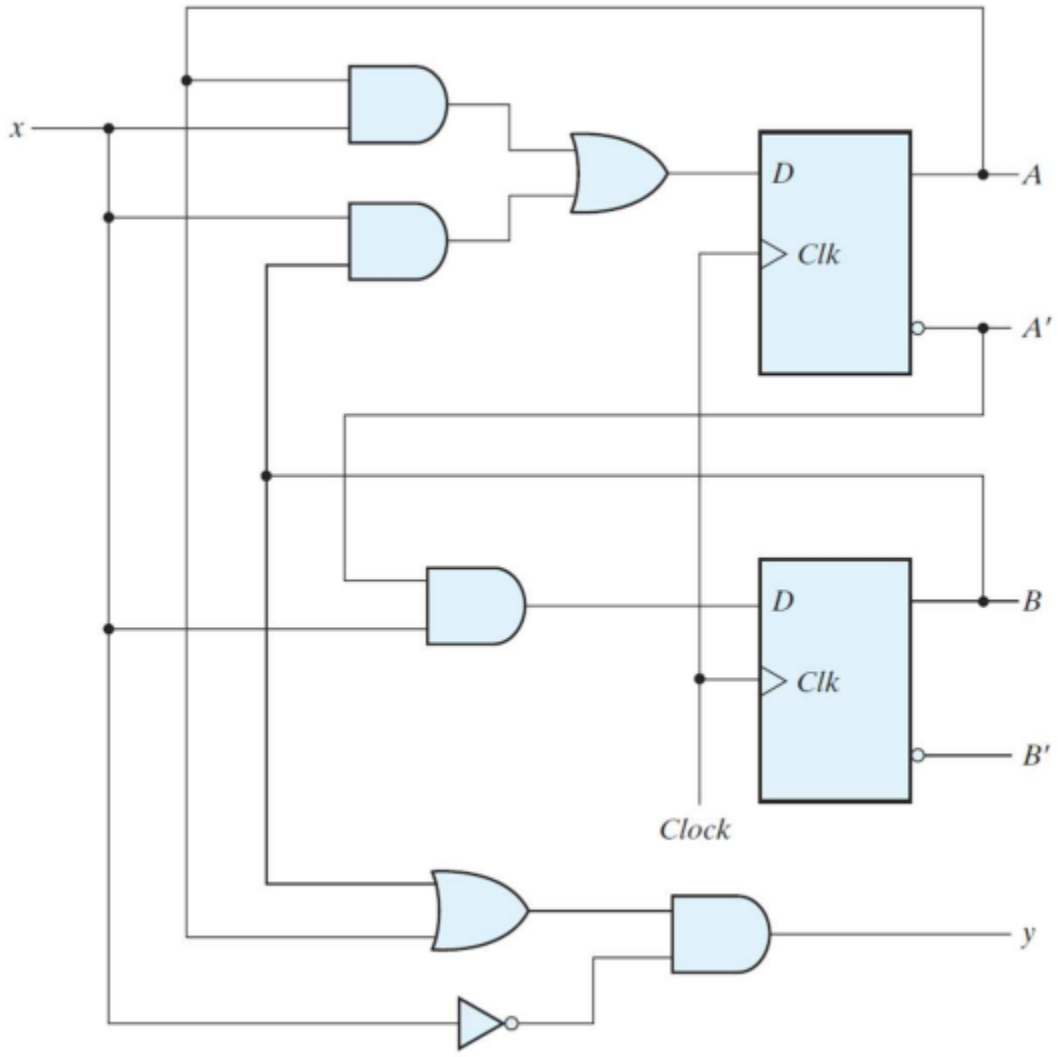
Here, we have two FF So total number of state is $2^2 = 4$

i.e. S0 = 00, S1 = 01, S2 = 10, S3 = 11



Analysis of D FF:

Analyze the following circuit diagram and draw the state table and state diagram.



Solution:

Here, A and B are the present state of FF A and B

Step 1: Input equations of FF

$D_A = Ax + Bx$

$D_B = A'x$

Step 2: Output equation of given circuit

$y = (A+B)x'$

Step 3: State table

In D FF next state is equals to input of FF.

Present State		Input	FF Inputs		Next State		Output
A	B	X	D_A	D_B	$Q_{n+1}A$	$Q_{n+1}B$	Y
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	1
1	1	1	1	0	1	0	0

Step 4: Next state equation from state table

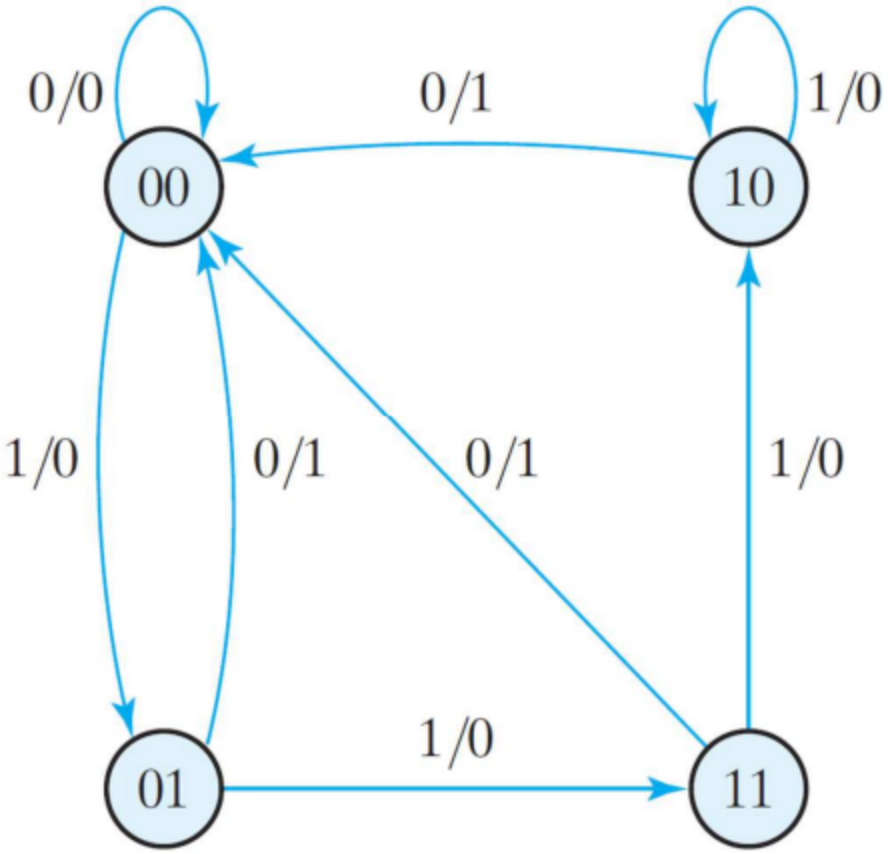
$Q_{n+1}A = D_A$

$Q_{n+1}B = D_B$

State 5: State Diagram

Here, we have two FF so number of state is $2^2 = 4$ states

i.e. S0 = 00, S1 = 01, S2 = 10, S3 = 11



Unit-6: Registers, Counters and Memory Units

Introduction to Register:

One flip-flop can store one-bit of information. In order to store multiple bits of information, we require multiple flip-flops. A group of flip-flops, which is used to hold or store the sequence of binary information, is known as **register**.

An *n*-bit register has a group of *n* flip-flops and is capable of storing any *n*-bit binary information. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops and gates that effect their transition. The

flip-flops hold binary information and the gates control when and how new information is transferred into the register.

There are various types of registers are available, the simplest possible register is one that consists of only flip-flops without any external gates. Following figure shows such a 4-bit register constructed with four *D*-type flip-flops and a common clock pulse input.

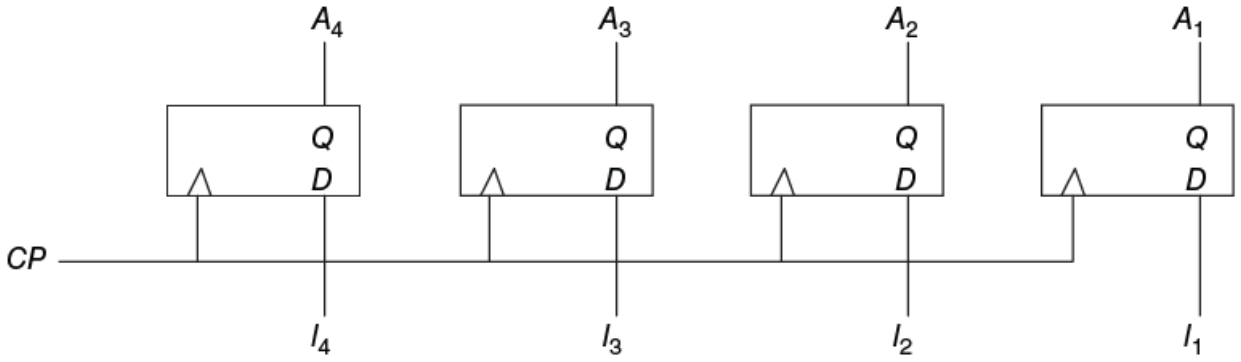


Fig: 4-bit Register without external gates

The clock pulse input, *CP,* enables all flip-flops so that the information presently available at the four inputs I_1, I_2, I_3, I_4 can be transferred into the 4-bit register. The four outputs A_1, A_2, A_3, A_4 of flip flops are the information presently stored in the register.

The information present at a data (*D*) input is transferred to the *Q* output when the enable (*CP*) is 1, and the *Q* output follows the input data as long as the *CP* signal remains 1. When *CP* goes to 0, the information that was present at the data input just before the transition is retained at the *Q* output. In other words, the flip-flops are sensitive to the pulse duration, and the register is enabled for as long as *CP* = 1.

Register with Parallel Load:

The transfer of new information into a register is referred to as *loading* the register. If all the bits of the register are loaded simultaneously with a single clock pulse, we say that the loading is done in parallel.

A register with parallel load can be constructed with *D* flip-flops as shown in figure bellow:

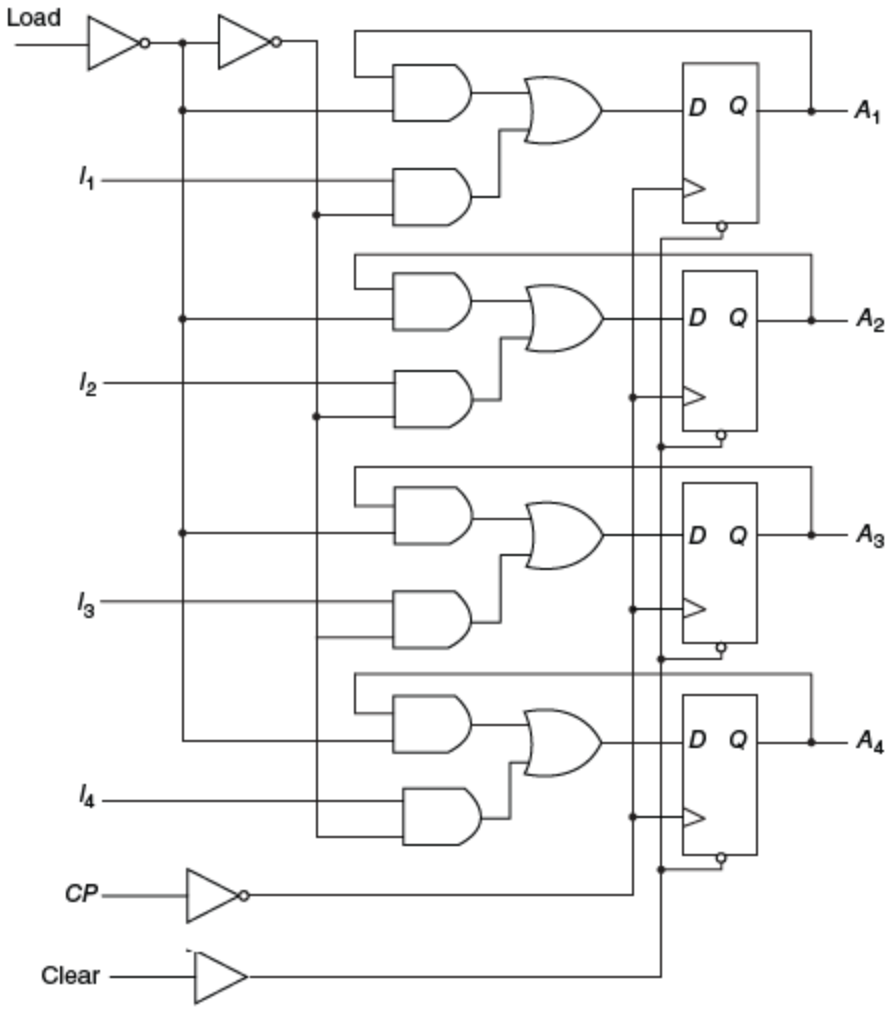


Fig: Register with parallel load using D FF

The ***CP*** input acts as an enable signal which controls the loading of new information into the register. The inverter in the *CP* path causes all flip-flops to be triggered by the negative edge of the incoming pulses. When *CP* goes to 0, the input information is loaded into the register. If *CP* remains at 1, the content of the register is not changed.

The purpose of the inverter is to reduce the loading of the master-clock generator. This is because the *CP* input is connected to only one inverter instead of the four-gate inputs that would have been required if the connections were made directly into the flip-flop clock inputs.

The ***clear*** input goes to a special terminal in each, flip-flop through a non-inverting buffer gate. It is useful for clearing the register to all 0's prior to its clocked operation. When this terminal goes to 0, the flip-flop is cleared asynchronously. The clear input must be maintained at 1 during normal clocked operations.

When the ***load*** input is 1, the I_1 I_2 I_3 *and* I_4 inputs are transferred into the register on the next clock pulse (i.e. next negative edge). When the load input is 0, the circuit inputs are inhibited (copied) and the *D* flip-flops are reloaded with their present value, thus maintaining the content of the register. With each negative clock pulse, the *D* input determines the next state of the output. To leave the output unchanged, it is necessary to make the *D* input equal to the present *Q* output in each flip-flop.

The feedback connection in each Flip-flop is necessary when *D* type is used because a *D* flip-flop does not have a "no-change" input condition.

Shift Registers:

A register capable of shifting its binary information either to the right or to the left is called a *shift register.* The logical configuration of a shift register consists of a chain of flip-flops connected in cascade (sequence), with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive a common clock pulse which causes the shift from one stage to the next.

Left Shift Register: Shift Register which can shift bits of information towards one bit left is called Left Shift Register. This is the multiply by 2 counter.

Example: 110 = 6 perform left shift of given bits
1100 = 12 this is the 2 times of 6.

Right Shift Register: Shift Register which can shift bits of information towards one bit right is called left shift Register. This is the divide by 2 counter.

Example: 110 = 6 perform right shift of given bits
11 = 3 this is the fraction of 2.

Bidirectional Shift Register: Shift Register which can shift bits of information towards both side Left and Right is called Bidirectional Shift Register. This will shift the bit one side at a time either left or right not in both side simultaneously.

Universal Shift Register: Shit Register which can shift bits of information bidirectional with parallel load is called Universal Shift Register.

Types of Shift Register:

Based on the data input and output there are four types of shift registers

- Serial In Serial Out (SISO)
- Serial In Parallel Out (SIPO)
- Parallel In Serial Out (PISO)
- Parallel In Parallel Out (PIPO)

Serial In − Serial Out (SISO) Shift Register

The Serial In Serial Out shift register accepts data serially that is, one bit at a time on a single line. It produces the stored information on its output also in serial form. Figure bellow shows a 4-bit right shift SISO register implemented with D flip-flops. With four stages, this register can store up to four bits of data.

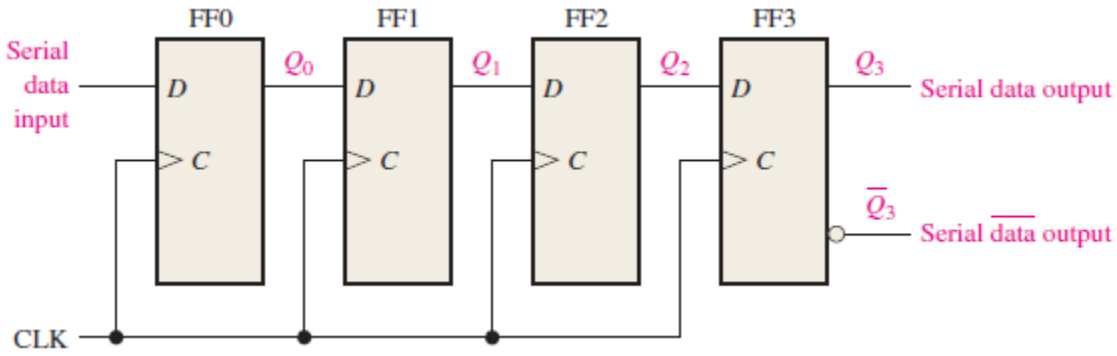


Fig: Serial In Serial Out (SISO) right shift register

This block diagram consists of 4 D flip-flops, which are cascaded (*arrange in sequential manner*). That means, output of one D flip-flop is connected as the input

of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop i.e. FF0 Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next toward right. So, we can receive the bits serially from the output of right most D flip-flop i.e. FF3 Hence, this output is also called as **serial output**.

Example:

Let us see the working of 4-bit SISO shift register by sending the binary information **"1010"** from LSB to MSB serially at the input.

Assume, initial status of the D flip-flops in the absence of clock signal from leftmost to rightmost is $Q_0Q_1Q_2Q_3$ =0000.

We can understand the **working of 4-bit SISO shift register** from the following table.

No of positive edge of Clock	Serial Input	Q_0	Q_1	Q_2	Q_3
0	-	0	0	0	0
1	0 (LSB)	0	0	0	0
2	1	1	0	0	0
3	0	0	1	0	0
4	1 (MSB)	1	0	1	0 (LSB)

5	-	-	1	0	1
6	-	-	-	1	0
7	-	-	-	-	1 (MSB)

Here, the serial output is coming from Q_3 is **1010**. So, the LSB 0 is received at 4th positive edge of clock and the MSB 1 is received at 7th positive edge of clock.

Therefore, the 4-bit SISO shift register requires seven clock pulses in order to produce the valid output. Similarly, the **N-bit SISO shift register** requires **2N-1** clock pulses in order to shift 'N' bit information.

Timing Diagram:

Serial In - Parallel Out (SIPO) Shift Register

The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out (SIPO) shift register. The **block diagram** of 4-bit SIPO shift register is shown in the following figure.

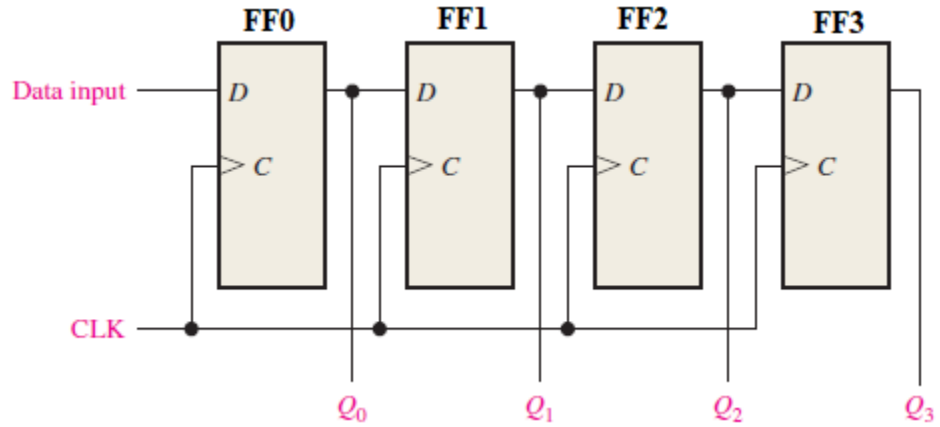


Fig: Serial In Parallel Out (SIPO) Shift Register

This circuit consists of 4 D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop i.e. FF0. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next towards right. In this case, we can access the outputs of each D flip-flop in parallel i.e. FF0, FF1, FF2, FF3 simultaneously. So, this output is called **parallel outputs**.

Example:

Let us see the working of 4-bit SIPO shift register by sending the binary information **"1010"** from LSB to MSB serially at the input.

Assume, initial status of the D flip-flops in the absence of clock signal from leftmost to rightmost is $Q_0Q_1Q_2Q_3$=0000. Here, Q_0 and Q_3 are MSB & LSB respectively. We can understand the **working of 4-bit SIPO shift register** from the following table.

No of positive edge of Clock	Serial Input	Q_0 MSB	Q_1	Q_2	Q_3 LSB
0	-	0	0	0	0
1	0 (LSB)	0	0	0	0
2	1	1	0	0	0
3	0	0	1	0	0
4	1 (MSB)	1 (MSB)	0	1	0 (LSB)

The binary information **"1010"** is obtained in parallel at the outputs of D flip-flops for 4th positive edge of clock.

So, the 4-bit SIPO shift register requires 4 clock pulses in order to produce the valid output. Similarly, the **N-bit SIPO shift register** requires **N** clock pulses in order to shift 'N' bit information.

Timing Diagram:

Parallel In – Serial Out (PISO) Shift Register

The shift register, which allows parallel input and produces serial output is known as Parallel In – Serial Out (PISO) shift register. The **block diagram** of 4-bit PISO shift register is shown in the following figure.

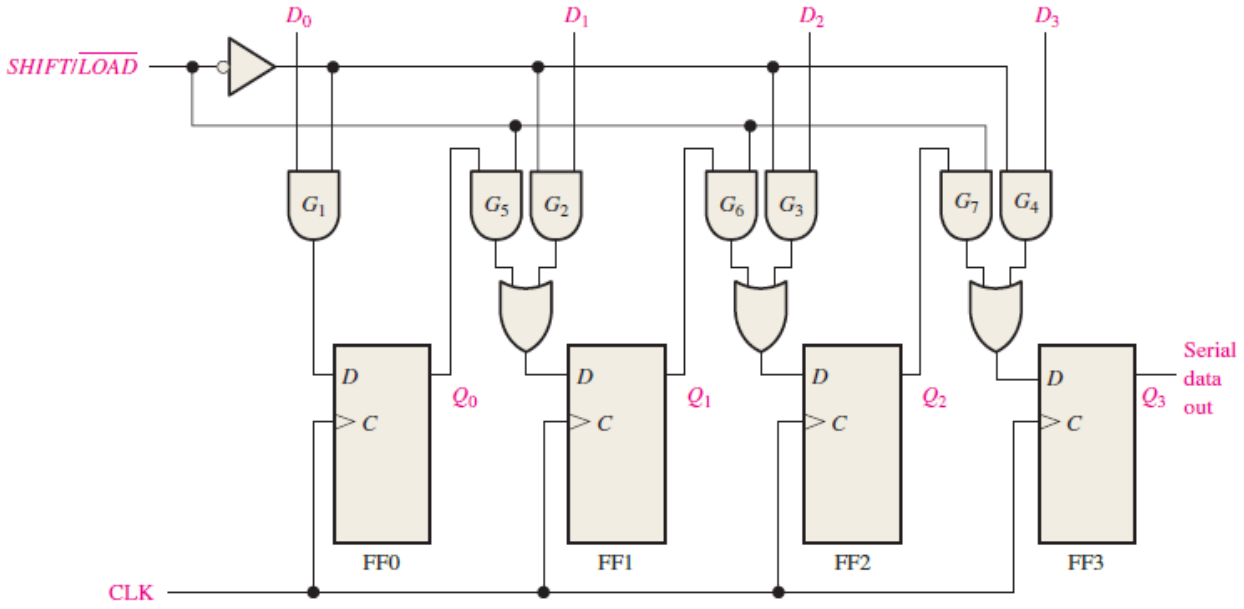


Fig: Parallel In Serial Out (PISO) Shift Register

This circuit consists of 4 D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

Above figure illustrates a 4-bit parallel in/serial out shift register and a typical logic symbol. There are four data-input lines, D0, D1, D2, and D3, and a $SHIFT/\overline{LOAD}$ input, which allows four bits of data to **load** in parallel into the register. When $SHIFT/\overline{LOAD}$ is LOW, gates **G1, G2, G3 and G4** are enabled and rest of all are disabled, allowing each data bit to be applied (loaded) to the **D** input of its respective flip-flop. When a clock pulse is applied, the flip-flops with ***D = 1*** will set and those with ***D = 0*** will reset, thereby storing all four bits simultaneously. When $SHIFT/\overline{LOAD}$ is HIGH, gates **G5, G6 and G7** are enabled and rest of all are disabled, allowing the data bits to shift right from one stage to the next.

The OR gates allow either the normal shifting operation or the parallel data-entry operation, depending on which AND gates are enabled by the level on the $SHIFT/\overline{LOAD}$ input.

Notice that FF0 has a single AND to disable or enable the parallel input, **D0.** It does not require an AND/OR arrangement because there is no serial data in.

Example:
Let us see the working of 4-bit PISO shift register by applying the binary information **"1010"** in parallel through preset inputs.

Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be $Q_0Q_1Q_2Q_3$=1010. We can understand the **working of 4-bit PISO shift register** from the following table.

No of positive edge of Clock	Q_0	Q_1	Q_2	Q_3
0	1 (MSB)	0	1	0 (LSB)
1	-	1	0	1
2	-	-	1	0

3	-	-	-	1 (MSB)

Here, the serial output is coming from Q_3. So, the LSB 0 is received before applying positive edge of clock and the MSB 1 is received at 3rd positive edge of clock.

Therefore, the 4-bit PISO shift register requires 3 clock pulses in order to produce the valid output. Similarly, the **N-bit PISO shift register** requires **N-1** clock pulses in order to shift 'N' bit information.

Timing Diagram:

Parallel In - Parallel Out (PIPO) Shift Register

The shift register, which allows parallel input and produces parallel output is known as Parallel In − Parallel Out (PIPO) shift register. The **block diagram** of 4-bit PIPO shift register is shown in the following figure.

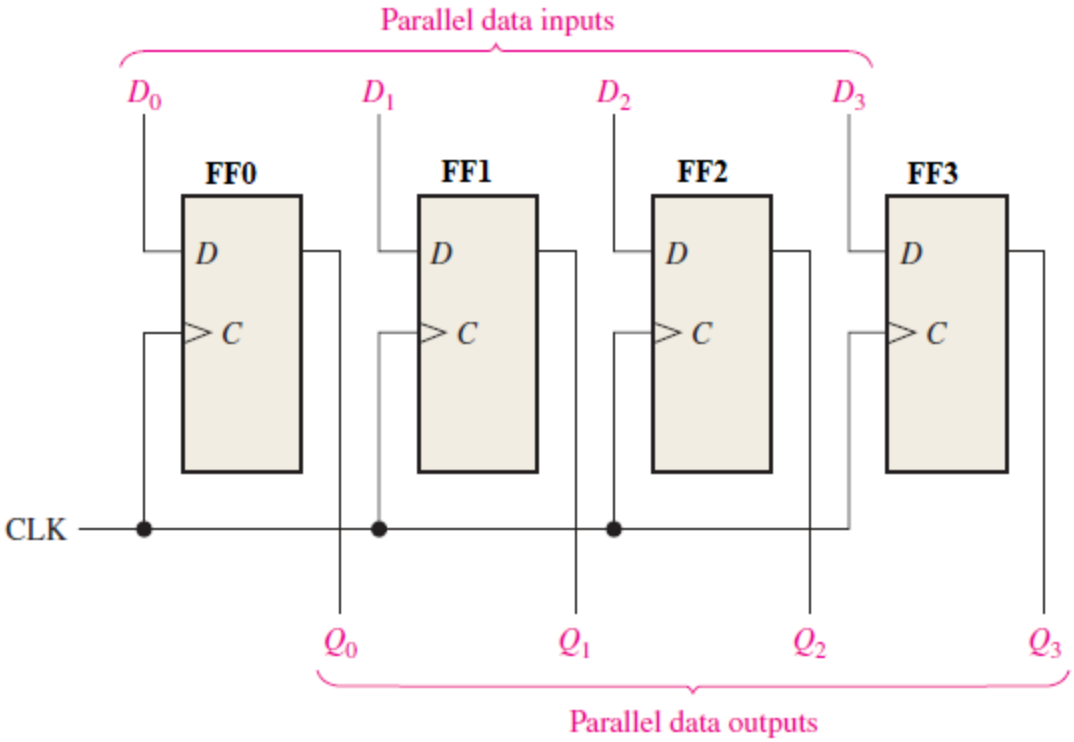


Fig: Parallel In Parallel Out (PIPO) Shift Register

This circuit consists of 4 D flip-flops, which are placed in parallel manner. That means, output of one D flip-flop is not connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop. The flip-flops produce the corresponding outputs, based on the values of asynchronous inputs. In this case, the effect of outputs is independent of clock transition. So, we will get the **parallel outputs** from each D flip-flop without clock pulse.

Example:

Let us see the working of 4-bit PIPO shift register by applying the binary information **"1010"** in parallel through preset inputs.

Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be $Q_0Q_1Q_2Q_3$=1010.

We can understand the **working of 4-bit PISO shift register** from the following table.

No of positive edge of Clock	Q_0	Q_1	Q_2	Q_3
0	1 (MSB)	0	1	0 (LSB)

The binary information **"1010"** is obtained in parallel at the outputs of D flip-flops before applying positive edge of clock.

Therefore, the 4-bit PIPO shift register requires zero clock pulses in order to produce the valid output. Similarly, the **N-bit PIPO shift register** doesn't require any clock pulse in order to shift 'N' bit information.

Timing Diagram:
No timing diagram required because there is no clock pulse is applied in register.

Bidirectional Shift Register:

A bidirectional shift register is one in which the data can be shifted either left or right. It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left, depending on the level of a control line.

A 4-bit bidirectional shift register is shown in Figure bellow:

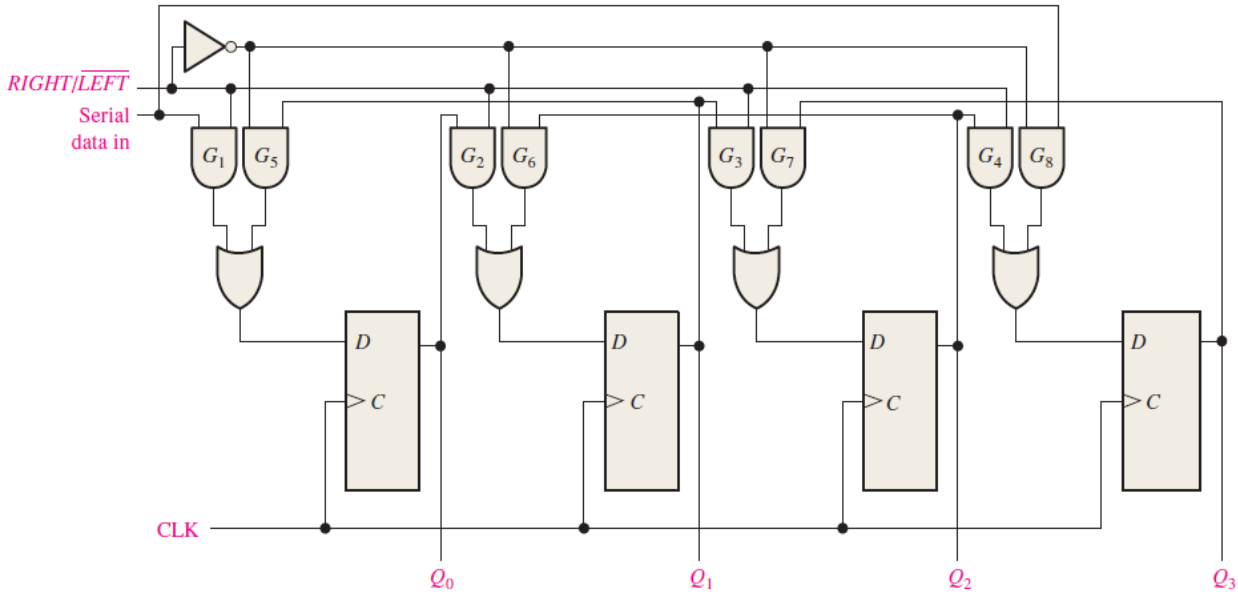


Fig: 4-bit Bidirectional Shift Register

When clock pulse occurs and $RIGHT/\overline{LEFT}$ control input is HIGH, gates G1, G2, G3 and G4 are enabled, and the Q output of each flip-flop is passed to the D input of next flip-flop towards right which is right shift.

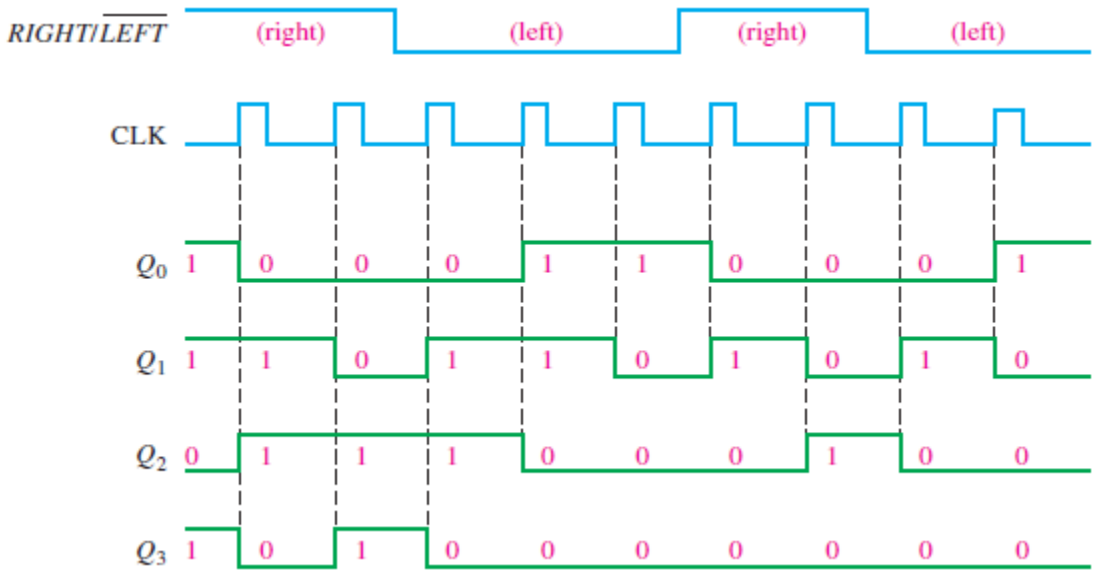
When clock pulse occurs and $RIGHT/\overline{LEFT}$ control input is LOW, gates G5, G6 and G7 are enabled, and the Q output of each flip-flop is passed to the D input of next flip-flop towards left which is left shift.

Example: The data in the Bidirectional Shift Register is Q_0 Q_1 Q_2 Q_3 = 1101 respectively at the initial phase. Perform the RIGHT/LEFT shift operation depending on control signal $RIGHT/\overline{LEFT}$ input.

R/L control	Clk	Q_0	Q_1	Q_2	Q_3
Load	0	1	1	0	1
R	1	0	1	1	0
R	2	0	0	1	1
L	3	0	1	1	0
L	4	1	1	0	0
L	5	1	0	0	0
R	6	0	1	0	0

R	7	0	0	1	0
L	8	0	1	0	0
L	9	1	0	0	0

Timing Diagram:



Universal Shift Register:

The register has both Left Shift and Right Shift with parallel-load capability then it is called a *Universal shift register or Shift Register with Parallel Load.*

The diagram of a 4-bit universal shift register is shown in figure bellow which consists of four *D* flip-flops, four 4x1 multiplexers (MUX) with select lines S_0 and S_1, one clear input, one clock input, four parallel inputs I_0 I_1 I_2 I_2, four parallel 0utputs A_0 A_1 A_2 and A_3 , one serial input in both sides.

Register perform the operation according to the selection line of MUX

Mode control		Register operation
s_1	s_0	
0	0	No change
0	1	Shirt right
1	0	Shift left
1	1	Parallel load

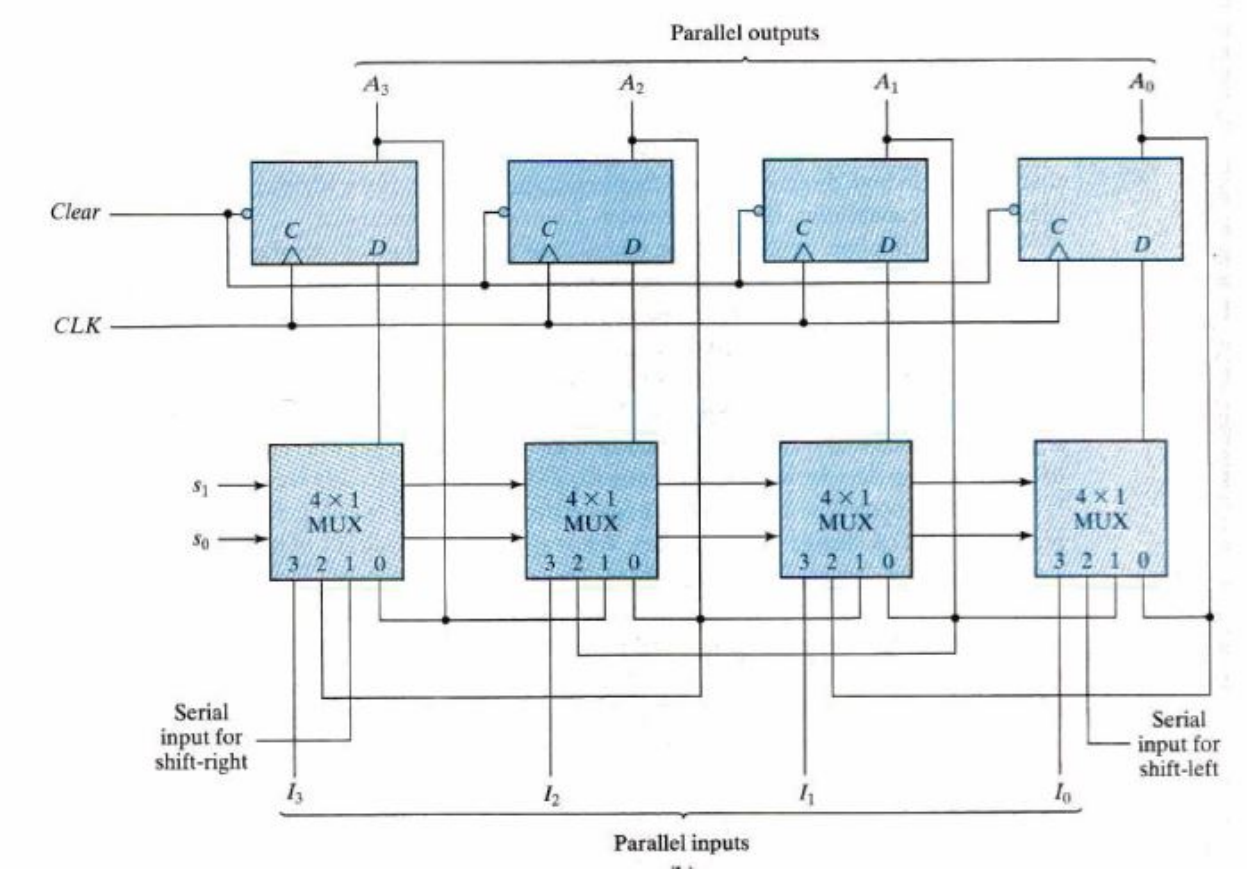


Fig: Universal Shift Register

Counters:

Synchronous Counter:

The synchronous counter is a type of counter in which the clock signal is simultaneously provided to each flip-flop present in the counter circuit. More specifically, we can say that each flip-flop is triggered in synchronism with the clock input.

Synchronization leads to variation in each output bit at the same time with a common clock signal. Thereby eliminating the ripple effects and so the propagation delay.

Memory Unit:

A Memory unit stores binary information in group of bits called word. Each word in memory is assigned an identification number, called an address, staring from 0 to 2^k -1 where k is the number of address lines.

Special input lines called address lines select one particular word. The address consists of k lines which specify which word (among the 2^k words available) to be selected for reading or writing.

Data input line provides the information to be stored (written) into the memory, while data output lines carry the information out (read) from the memory. The control line Read/Write specifies the direction of transfer of the data either in or out. A decoder inside the memory accepts this address and opens the paths needed to select the bits of the specified word.

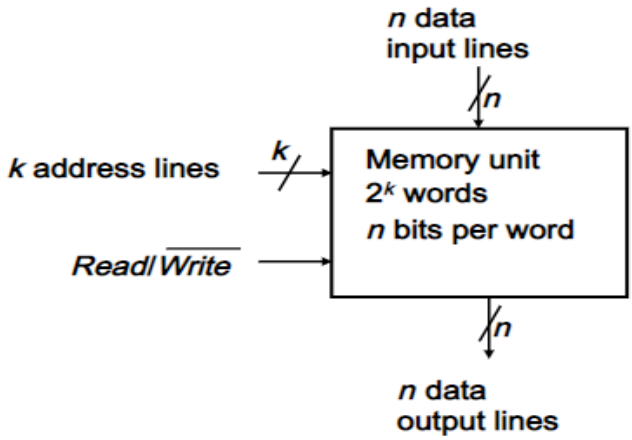


Fig: Block Diagram of Memory Unit

Content of a 1024x16 bit memory:

Memory address		Memory content
binary	decimal	
0000000000	0	1011010111011101
0000000001	1	1010000110000110
0000000010	2	0010011101110001
:	:	:
:	:	:
1111111101	1021	1110010101010010
1111111110	1022	0011111010101110
1111111111	1023	1011000110010101

Note: *Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits. It is customary to refer to the number of words (or bytes) in a memory with one of the letters:*

- ***K(Kilo)*** *is equal to 2^{10} words*
- ***M(Mega)*** *is equal to 2^{20} words*
- ***G(Giga)*** *is equal to 2^{30} words*

Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations.

Write Operation:

The write signal specifies a transfer-in operation. When write control signal is applied, the memory unit will then take the bits from the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Transfer the binary address of the desired word to the address lines.
2. Transfer the data bits that must be stored in memory to the data input lines.
3. Activate the write control input.

Read Operation:

The read signal specifies a transfer-out operation. When read control signal is applied, the memory unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Transfer the binary address of the desired word to the address lines.
2. Activate the read control input.

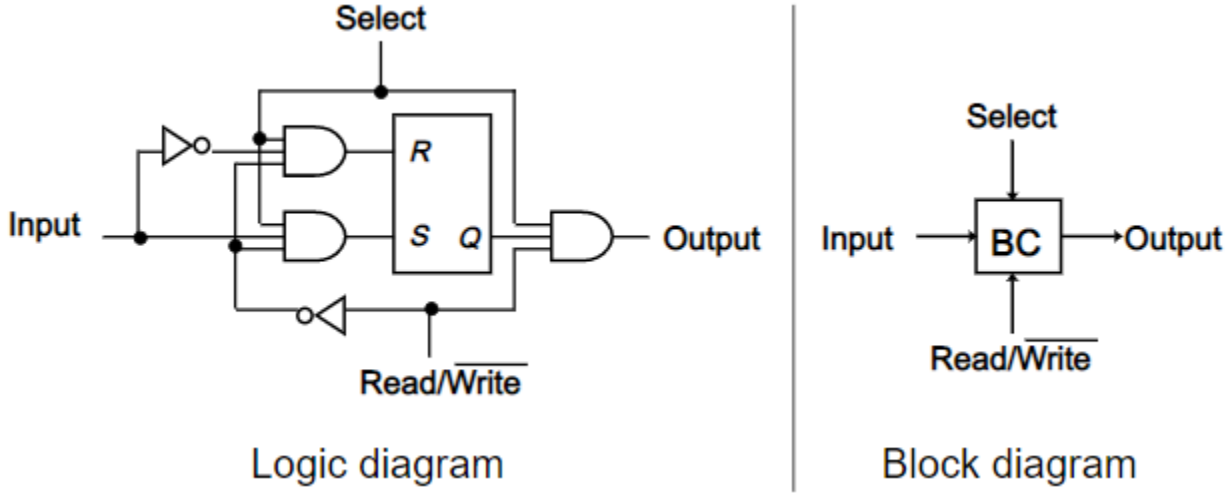
Control Input to Memory Cell:

Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

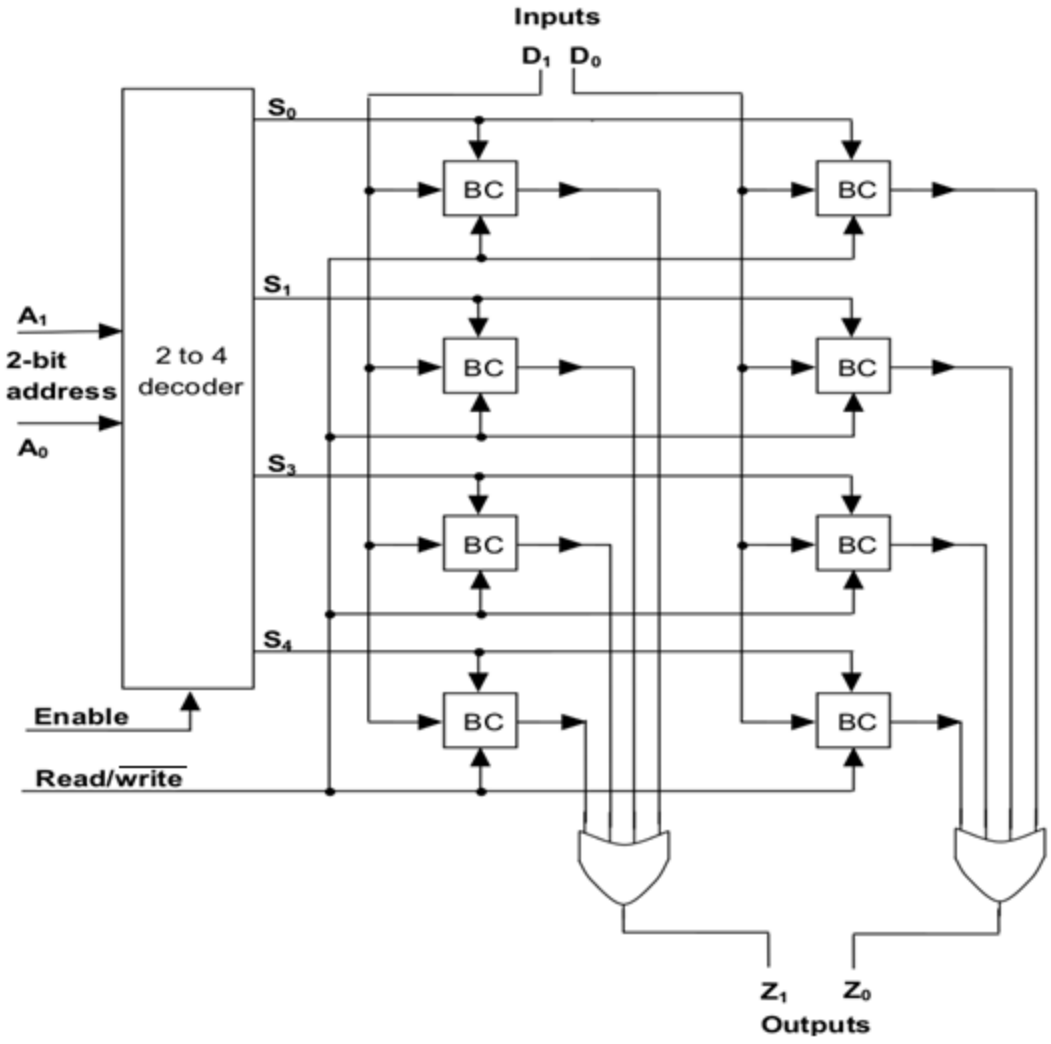
Binary Cell / Memory Cell:

A binary cell in RAM is a single unit of storage capable of holding a binary digit (bit), which can either be 0 or 1. The combination of multiple binary cells forms a word, which is the basic unit of data stored.

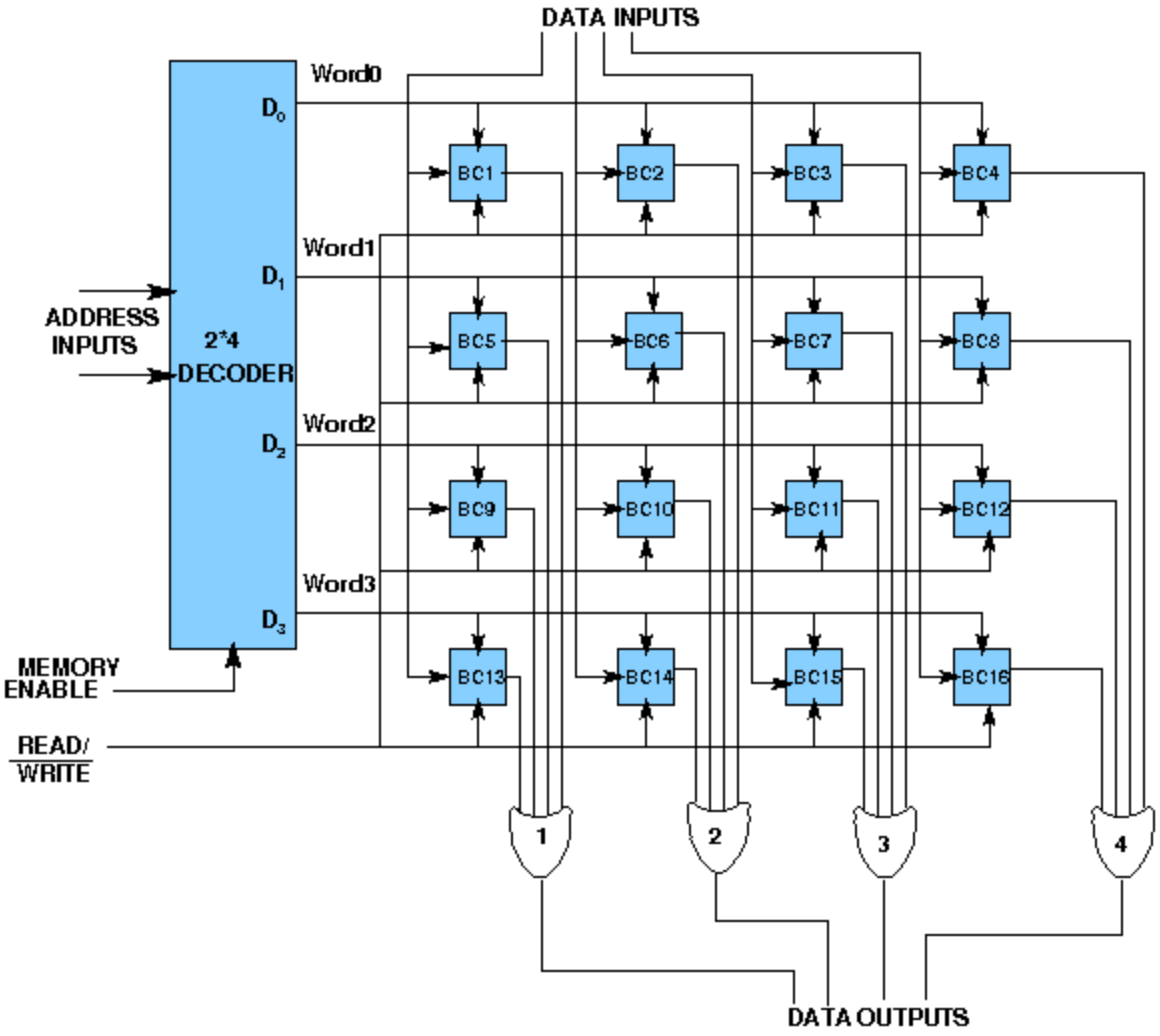
Block and Logic Diagram of Binary Cell:



Example 1: Construct 4x2 RAM using decoder and binary cell



Example 2: Construct 4x4 RAM


DATA INPUTS
Word0
D0
Word1
D1
ADDRESS INPUTS
2*4 DECODER
Word2
D2
Word3
D3
MEMORY ENABLE
READ/ WRITE
BC1
BC2
BC3
BC4
BC5
BC6
BC7
BC8
BC9
BC10
BC11
BC12
BC13
BC14
BC15
BC16
1
2
3
4
DATA OUTPUTS

End of Unit-6