DIGITAL LOGIC DESIGN

(R22A0402)

LECTURE NOTES

B.TECH

(II YEAR – I SEM)

(2024-25)

Prepared By

Mr. E.MAHENDER REDDY (Assistant Professor)

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Maisammguda, Dhulapally Post, Secunderabad 500100

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech. ECE- I Sem **L/T/P/C**

3/-/-/3

(R22A0402) DIGITAL LOGIC DESIGN

COURSE OBJECTIVES:

1. To understand common forms of number representation in digital electroniccircuits and to be able to convert between different representations.
2. To implement simple logical operations using combinational logic circuits
3. To design combinational logic circuits, sequential logic circuits.
4. To impart to student the concepts of sequential circuits, enabling themto analyze sequential systems.
5. Understanding of the different technologies related to HDLs, construct, compile andexecute Verilog HDL programs using provided.
6. Designing digital circuits, behavior and RTL modeling of digital circuits using Verilog HDL.

UNIT –I:

Number Systems, Boolean Algebra and Switching Functions:

Number Systems, Base Conversion Methods, Complements of Numbers, Codes- Binary Codes, Binary Coded Decimal Code, Unit Distance Codes, Error Detecting and Error Correcting Codes, Hamming Code.

Boolean Algebra:

Basic Theorems and Properties, Switching Functions, Canonical and Standard Forms, Algebraic Simplification of Digital Logic Gates, Properties of XOR Gates, Universal Logic Gates.

UNIT –II

Minimization and Design of Combinational Circuits:

K- Map Method, up to Five variable K- Maps, Don't Care Map Entries, Combinational Design, Arithmetic Circuits, Comparator, decoder, Encoder, Multiplexers, De-Multiplexers, Code Converters.

UNIT –III:

Sequential Machines Fundamentals:

Introduction, Basic Architectural Distinctions between Combinational and Sequentialcircuits, classification of sequential circuits, The binary cell, The S-R-Latch Flip-Flop The D-Latch Flip-Flop, The "Clocked T" Flip-Flop, The " Clocked J-K" Flip-Flop, Conversion from one type of Flip-Flop to another.

UNIT –IV:

INTRODUCTION TO VERILOG HDL: Verilog as HDL, Levels of Design Description,Concurrency, Simulation and Synthesis, Programming Language Interface, Module.

Language Constructs and Conventions: Introduction, Keywords, Identifiers, White Space, Characters, Comments, Numbers, Strings, Logic Values, Data Types, Operators.

UNIT –V:

GATE LEVEL MODELING: Introduction, AND Gate Primitive, Module Structure, Other Gate Primitives, Illustrative Examples, Design of Flip- Flops with Gate Primitives, Delay.
MODELING AT DATAFLOW LEVEL: Introduction, Continuous Assignment Structure, Delays and Continuous Assignments.
BEHAVIORAL MODELING: Introduction, Operations and Assignments, 'Initial' Construct, 'always' construct, , Design at Behavioral Level, The 'Case' Statement, 'If' and 'if-Else' Constructs

TEXT BOOKS:

1. Digital Design- Morris Mano, PHI, 3rd Edition.
2. Switching Theory and Logic Design-A. Anand Kumar, PHI, 2nd Edition.
3. T.R. Padmanabhan, B Bala Tripura Sundari, Design through Verilog HDL, Wiley 2009.
4. Verilog HDL - Samir Palnitkar, 2nd Edition, Pearson Education, 2009.

REFERENCE BOOKS:

1. Introduction to Switching Theory and Logic Design – Fredriac J. Hill, Gerald R. Peterson, 3rdEd,John Wiley & Sons Inc.
2. Digital Fundamentals – A Systems Approach – Thomas L. Floyd, Pearson, 2013.
3. Switching Theory and Logic Design – Bhanu Bhaskara –Tata McGraw Hill Publication,2012
4. Fundamentals of Logic Design- Charles H. Roth, Cengage Learning, 5th, Edition, 2004.
5. Fundamentals of Digital Logic with Verilog Design - Stephen Brown,Zvonkoc Vranesic, TMH, 2nd Edition.
6. Advanced Digital Design with Verilog HDL - Michel D. Ciletti, PHI, 2009.

COURSE OUTCOMES:

Upon completion of the course, student should possess the following skills:

1. Be able to manipulate numeric information in different forms
2. Be able to manipulate simple Boolean expressions using the theorems and postulates of Boolean algebra and to minimize combinational functions.
3. Be able to design and analyze small combinational circuits and to use standard combinational functions to build larger more complexcircuits.
4. Be able to design and analyze Digital circuits
5. Verify behavior and Implement RTL models on FPGAs.

UNIT I

Number System and Boolean Algebra

If base or radix of a number system is 'r', then the numbers present in that number system are ranging from zero to r-1. The total numbers present in that number system is 'r'. So, we will get various number systems, by choosing the values of radix as greater than or equal to two.

In this chapter, let us discuss about the **popular number systems** and how to represent a number in the respective number system. The following number systems are the most commonly used.

- Decimal Number system
- Binary Number system
- Octal Number system
- Hexadecimal Number system

Decimal Number System

The **base** or radix of Decimal number system is **10**. So, the numbers ranging from 0 to 9 are used in this number system. The part of the number that lies to the left of the **decimal point**is known as integer part. Similarly, the part of the number that lies to the right of the decimal point is known as fractional part.

In this number system, the successive positions to the left of the decimal point having weights of 10^0, 10^1, 10^2, 10^3 and so on. Similarly, the successive positions to the right of the decimal point having weights of 10^{-1}, 10^{-2}, 10^{-3} and so on. That means, each position has specific weight, which is **power of base 10**

Example

Consider the **decimal number 1358.246**. Integer part of this number is 1358 and fractional part of this number is 0.246. The digits 8, 5, 3 and 1 have weights of 100, 101, 10^2 and 10^3respectively. Similarly, the digits 2, 4 and 6 have weights of 10^{-1}, 10^{-2} and 10^{-3} respectively.

Mathematically, we can write it as

$1358.246 = (1 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) +$

$(4 \times 10^{-2}) + (6 \times 10^{-3})$

After simplifying the right hand side terms, we will get the decimal number, which is on left hand side.

Binary Number System

All digital circuits and systems use this binary number system. The **base** or radix of this number system is **2**. So, the numbers 0 and 1 are used in this number system.

The part of the number, which lies to the left of the **binary point** is known as integer part. Similarly, the part of the number, which lies to the right of the binary point is known as fractional part.

In this number system, the successive positions to the left of the binary point having weights of 2^0, 2^1, 2^2, 2^3 and so on. Similarly, the successive positions to the right of the binary point having weights of 2^{-1}, 2^{-2}, 2^{-3} and so on. That means, each position has specific weight, which is **power of base 2**.

Example

Consider the **binary number 1101.011**. Integer part of this number is 1101 and fractional part of this number is 0.011. The digits 1, 0, 1 and 1 of integer part have weights of 2^0, 2^1, 2^2, 2^3respectively. Similarly, the digits 0, 1 and 1 of fractional part have weights of 2^{-1}, 2^{-2}, 2^{-3} respectively.

Mathematically, we can write it as

$1101.011 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) +$

$(1 \times 2^{-2}) + (1 \times 2^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of binary number on left hand side.

Octal Number System

The **base** or radix of octal number system is **8**. So, the numbers ranging from 0 to 7 are used in this number system. The part of the number that lies to the left of the **octal point** is known as integer part. Similarly, the part of the number that lies to the right of the octal point is known as fractional part.

In this number system, the successive positions to the left of the octal point having weights of 8^0, 8^1, 8^2, 8^3 and so on. Similarly, the successive positions to the right of the octal point having weights of 8^{-1}, 8^{-2}, 8^{-3} and so on. That means, each position has specific weight, which is **power of base 8**.

Example

Consider the **octal number 1457.236**. Integer part of this number is 1457 and fractional part of this number is 0.236. The digits 7, 5, 4 and 1 have weights of 8^0, 8^1, 8^2 and 8^3respectively. Similarly, the digits 2, 3 and 6 have weights of 8^{-1}, 8^{-2}, 8^{-3} respectively.

Mathematically, we can write it as

$1457.236 = (1 \times 8^3) + (4 \times 8^2) + (5 \times 8^1) + (7 \times 8^0) + (2 \times 8^{-1}) +$

$(3 \times 8^{-2}) + (6 \times 8^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of octal number on left hand side.

Hexadecimal Number System

The **base** or radix of Hexa-decimal number system is **16**. So, the numbers ranging from 0 to 9 and the letters from A to F are used in this number system. The decimal equivalent of Hexa-decimal digits from A to F are 10 to 15.

The part of the number, which lies to the left of the **hexadecimal point** is known as integer part. Similarly, the part of the number, which lies to the right of the Hexa-decimal point is known as fractional part.

In this number system, the successive positions to the left of the Hexa-decimal point having weights of 16^0, 16^1, 16^2, 16^3 and so on. Similarly, the successive positions to the right of the Hexa-decimal point having weights of 16^{-1}, 16^{-2}, 16^{-3} and so on. That means, each position has specific weight, which is **power of base 16**.

Example

Consider the **Hexa-decimal number 1A05.2C4**. Integer part of this number is 1A05 and fractional part of this number is 0.2C4. The digits 5, 0, A and 1 have weights of 16^0, 16^1, 16^2 and 16^3respectively. Similarly, the digits 2, C and 4 have weights of 16^{-1}, 16^{-2} and 16^{-3} respectively.

Mathematically, we can write it as

$1A05.2C4 = (1 \times 16^3) + (10 \times 16^2) + (0 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1}) +$

$(12 \times 16^{-2}) + (4 \times 16^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of Hexa-decimal number on left hand side.

In previous chapter, we have seen the four prominent number systems. In this chapter, let us convert the numbers from one number system to the other in order to find the equivalent value.

Decimal Number to other Bases Conversion

If the decimal number contains both integer part and fractional part, then convert both the parts of decimal number into other base individually. Follow these steps for converting the decimal number into its equivalent number of any base 'r'.

- Do **division** of integer part of decimal number and **successive quotients** with base 'r' and note down the remainders till the quotient is zero. Consider the remainders in reverse order to get the integer part of equivalent number of base 'r'. That means, first and last remainders denote the least significant digit and most significant digit respectively.

- Do **multiplication** of fractional part of decimal number and **successive fractions** with base 'r' and note down the carry till the result is zero or the desired number of equivalent digits is obtained. Consider the normal sequence of carry in order to get the fractional part of equivalent number of base 'r'.

Decimal to Binary Conversion

The following two types of operations take place, while converting decimal number into its equivalent binary number.

- Division of integer part and successive quotients with base 2.
- Multiplication of fractional part and successive fractions with base 2.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 − Division of 58 and successive quotients with base 2.

Operation	Quotient	Remainder
58/2	29	**0 (LSB)**
29/2	14	**1**
14/2	7	**0**
7/2	3	**1**
3/2	1	**1**
1/2	0	**1(MSB)**

$\Rightarrow (58)_{10} = (111010)_2$

Therefore, the **integer part** of equivalent binary number is **111010**.

Step 2 − Multiplication of 0.25 and successive fractions with base 2.

Operation	Result	Carry
0.25 x 2	0.5	0
0.5 x 2	1.0	1
-	0.0	-

$$\Rightarrow(.25)_{10} = (.01)_2$$

Therefore, the **fractional part** of equivalent binary number is **.01**

$$\Rightarrow(58.25)_{10} = (111010.01)_2$$

Therefore, the **binary equivalent** of decimal number 58.25 is 111010.01.

Decimal to Octal Conversion

The following two types of operations take place, while converting decimal number into its equivalent octal number.

- Division of integer part and successive quotients with base 8.
- Multiplication of fractional part and successive fractions with base 8.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 − Division of 58 and successive quotients with base 8.

Operation	Quotient	Remainder
58/8	7	**2**
7/8	0	**7**

$$\Rightarrow(58)_{10} = (72)_8$$

Therefore, the **integer part** of equivalent octal number is **72**.

Step 2 − Multiplication of 0.25 and successive fractions with base 8.

Operation	Result	Carry
0.25 x 8	2.00	2
-	0.00	-

$$\Rightarrow (.25)_{10} = (.2)_8$$

Therefore, the **fractional part** of equivalent octal number is .2

$$\Rightarrow (58.25)_{10} = (72.2)_8$$

Therefore, the **octal equivalent** of decimal number 58.25 is 72.2.

Decimal to Hexa-Decimal Conversion

The following two types of operations take place, while converting decimal number into its equivalent hexa-decimal number.

- Division of integer part and successive quotients with base 16.
- Multiplication of fractional part and successive fractions with base 16.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and decimal part is 0.25.

Step 1 − Division of 58 and successive quotients with base 16.

Operation	Quotient	Remainder
58/16	3	10=A
3/16	0	3

$$\Rightarrow (58)_{10} = (3A)_{16}$$

Therefore, the **integer part** of equivalent Hexa-decimal number is 3A.

Step 2 − Multiplication of 0.25 and successive fractions with base 16.

Operation	Result	Carry
0.25 x 16	4.00	4
-	0.00	-

$\Rightarrow (.25)_{10} = (.4)_{16}$

Therefore, the **fractional part** of equivalent Hexa-decimal number is .4.

$$\mathbf{\Rightarrow (58.25)_{10} = (3A.4)_{16}}$$

Therefore, the **Hexa-decimal equivalent** of decimal number 58.25 is 3A.4.

Binary Number to other Bases Conversion

The process of converting a number from binary to decimal is different to the process of converting a binary number to other bases. Now, let us discuss about the conversion of a binary number to decimal, octal and Hexa-decimal number systems one by one.

Binary to Decimal Conversion

For converting a binary number into its equivalent decimal number, first multiply the bits of binary number with the respective positional weights and then add all those products.

Example

Consider the **binary number 1101.11**.

Mathematically, we can write it as

$(1101.11)_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) +$

(1×2^{-2})

$\Rightarrow (1101.11)_2 = 8 + 4 + 0 + 1 + 0.5 + 0.25 = 13.75$

$\Rightarrow (1101.11)_2 = (13.75)_{10}$

Therefore, the **decimal equivalent** of binary number 1101.11 is 13.75.

Binary to Octal Conversion

We know that the bases of binary and octal number systems are 2 and 8 respectively. Three bits of binary number is equivalent to one octal digit, since $2^3 = 8$.

Follow these two steps for converting a binary number into its equivalent octal number.

- Start from the binary point and make the groups of 3 bits on both sides of binary point. If one or two bits are less while making the group of 3 bits, then include required number of zeros on extreme sides.
- Write the octal digits corresponding to each group of 3 bits.

Example

Consider the **binary number 101110.01101**.

Step 1 − Make the groups of 3 bits on both sides of binary point.

101 110.011 01

Here, on right side of binary point, the last group is having only 2 bits. So, include one zero on extreme side in order to make it as group of 3 bits.

⇒ 101 110.011 010

Step 2 − Write the octal digits corresponding to each group of 3 bits.

$$\Rightarrow (101\ 110.011\ 010)_2 = (56.32)_8$$

Therefore, the **octal equivalent** of binary number 101110.01101 is 56.32.

Binary to Hexa-Decimal Conversion

We know that the bases of binary and Hexa-decimal number systems are 2 and 16 respectively. Four bits of binary number is equivalent to one Hexa-decimal digit, since $2^4 = 16$.

Follow these two steps for converting a binary number into its equivalent Hexa-decimal number.

- Start from the binary point and make the groups of 4 bits on both sides of binary point. If some bits are less while making the group of 4 bits, then include required number of zeros on extreme sides.
- Write the Hexa-decimal digits corresponding to each group of 4 bits.

Example

Consider the **binary number 101110.01101**

Step 1 − Make the groups of 4 bits on both sides of binary point.

10 1110.0110 1

Here, the first group is having only 2 bits. So, include two zeros on extreme side in order to make it as group of 4 bits. Similarly, include three zeros on extreme side in order to make the last group also as group of 4 bits.

⇒ 0010 1110.0110 1000

Step 2 − Write the Hexa-decimal digits corresponding to each group of 4 bits.

$$\Rightarrow (0010\ 1110.0110\ 1000)_2 = (2E.68)_{16}$$

Therefore, the **Hexa-decimal equivalent** of binary number 101110.01101 is (2E.68).

Octal Number to other Bases Conversion

The process of converting a number from octal to decimal is different to the process of converting an octal number to other bases. Now, let us discuss about the conversion of an octal number to decimal, binary and Hexa-decimal number systems one by one.

Octal to Decimal Conversion

For converting an octal number into its equivalent decimal number, first multiply the digits of octal number with the respective positional weights and then add all those products.

Example

Consider the **octal number 145.23**.

Mathematically, we can write it as

$(145.23)_8 = (1 \times 8^2) + (4 \times 8^1) + (5 \times 8^0) + (2 \times 8^{-1}) + (3 \times 8^{-2})$

$\Rightarrow (145.23)_8 = 64 + 32 + 5 + 0.25 + 0.05 = 101.3$

$\Rightarrow (145.23)_8 = (101.3)_{10}$

Therefore, the **decimal equivalent** of octal number 145.23 is 101.3.

Octal to Binary Conversion

The process of converting an octal number to an equivalent binary number is just opposite to that of binary to octal conversion. By representing each octal digit with 3 bits, we will get the equivalent binary number.

Example

Consider the **octal number 145.23**.

Represent each octal digit with 3 bits.

$$(145.23)_8 = (001\ 100\ 101.010\ 011)_2$$

The value doesn't change by removing the zeros, which are on the extreme side.

$$\Rightarrow (145.23)_8 = (1100101.010011)_2$$

Therefore, the **binary equivalent** of octal number 145.23 is 1100101.010011.

Octal to Hexa-Decimal Conversion

Follow these two steps for converting an octal number into its equivalent Hexa-decimal number.

- Convert octal number into its equivalent binary number.

- Convert the above binary number into its equivalent Hexa-decimal number.

Example

Consider the **octal number 145.23**

In previous example, we got the binary equivalent of octal number 145.23 as 1100101.010011.

By following the procedure of binary to Hexa-decimal conversion, we will get

$$(1100101.010011)_2 = (65.4C)16$$

$$\Rightarrow (145.23)_8 = (65.4C)_{16}$$

Therefore, the **Hexa-decimal equivalent** of octal number 145.23 is 65.4*C*.

Hexa-Decimal Number to other Bases Conversion

The process of converting a number from Hexa-decimal to decimal is different to the process of converting Hexa-decimal number into other bases. Now, let us discuss about the conversion of Hexa-decimal number to decimal, binary and octal number systems one by one.

Hexa-Decimal to Decimal Conversion

For converting Hexa-decimal number into its equivalent decimal number, first multiply the digits of Hexa-decimal number with the respective positional weights and then add all those products.

Example

Consider the **Hexa-decimal number 1A5.2**

Mathematically, we can write it as

$$(1A5.2)_{16} = (1 \times 16^2) + (10 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1})$$

$\Rightarrow (1A5.2)_{16} = 256 + 160 + 5 + 0.125 = 421.125$

$\Rightarrow (1A5.2)_{16} = (421.125)_{10}$

Therefore, the **decimal equivalent** of Hexa-decimal number 1A5.2 is 421.125.

Hexa-Decimal to Binary Conversion

The process of converting Hexa-decimal number into its equivalent binary number is just opposite to that of binary to Hexa-decimal conversion. By representing each Hexa-decimal digit with 4 bits, we will get the equivalent binary number.

Example

Consider the **Hexa-decimal number 65.4C**

Represent each Hexa-decimal digit with 4 bits.

$$(65.4C)_6 = (0110\ 0101.0100\ 1100)_2$$

The value doesn't change by removing the zeros, which are at two extreme sides.

$\Rightarrow (65.4C)_{16} = (1100101.010011)_2$

Therefore, the **binary equivalent** of Hexa-decimal number 65.4C is 1100101.010011.

Hexa-Decimal to Octal Conversion

Follow these two steps for converting Hexa-decimal number into its equivalent octal number.

- Convert Hexa-decimal number into its equivalent binary number.
- Convert the above binary number into its equivalent octal number.

Example

Consider the **Hexa-decimal number 65.4C**

In previous example, we got the binary equivalent of Hexa-decimal number 65.4C as 1100101.010011.

By following the procedure of binary to octal conversion, we will get

$$(1100101.010011)_2 = (145.23)_8$$

$$\Rightarrow (65.4C)_{16} = (145.23)_8$$

Therefore, the **octal equivalent** of Hexa-decimal number 65.4*C*is 145.23.

We can make the binary numbers into the following two groups − **Unsigned numbers** and **Signed numbers**.

Unsigned Numbers

Unsigned numbers contain only magnitude of the number. They don't have any sign. That means all unsigned binary numbers are positive. As in decimal number system, the placing of positive sign in front of the number is optional for representing positive numbers. Therefore, all positive numbers including zero can be treated as unsigned numbers if positive sign is not assigned in front of the number.

Signed Numbers

Signed numbers contain both sign and magnitude of the number. Generally, the sign is placed in front of number. So, we have to consider the positive sign for positive numbers and negative sign for negative numbers. Therefore, all numbers can be treated as signed numbers if the corresponding sign is assigned in front of the number.

If sign bit is zero, which indicates the binary number is positive. Similarly, if sign bit is one, which indicates the binary number is negative.

Representation of Un-Signed Binary Numbers

The bits present in the un-signed binary number holds the **magnitude** of a number. That means, if the un-signed binary number contains **'N'** bits, then all **N** bits represent the magnitude of the number, since it doesn't have any sign bit.

Example

Consider the **decimal number 108**. The binary equivalent of this number is **1101100**. This is the representation of unsigned binary number.

$$(108)_{10} = (1101100)_2$$

It is having 7 bits. These 7 bits represent the magnitude of the number 108.

Representation of Signed Binary Numbers

The Most Significant Bit (MSB) of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as **sign bit**. The positive sign is represented by placing '0' in the sign bit. Similarly, the negative sign is represented by placing '1' in the sign bit.

If the signed binary number contains 'N' bits, then (N-1) bits only represent the magnitude of the number since one bit (MSB) is reserved for representing sign of the number.

There are three **types of representations** for signed binary numbers

- Sign-Magnitude form
- 1's complement form
- 2's complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example

Consider the **positive decimal number +108**. The binary equivalent of magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$$(+108)_{10} = (01101100)_2$$

Therefore, the **signed binary representation** of positive decimal number +108 is $\mathbf{01101100}$. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing **sign**of the number and the remaining bits represent the **magnitude**of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude.

$$(-108)_{10} = (11101100)_2$$

Therefore, the sign-magnitude representation of -108 is **11101100**.

1's complement form

The 1's complement of a number is obtained by **complementing all the bits** of signed binary number. So, 1's complement of positive number gives a negative number. Similarly, 1's complement of negative number gives a positive number.

That means, if you perform two times 1's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number. Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number.

$$(-108)_{10} = (10010011)_2$$

Therefore, the **1's complement of $(108)_{10}$** is **$(10010011)_2$**.

2's complement form

The 2's complement of a binary number is obtained by **adding one to the 1's complement** of signed binary number. So, 2's complement of positive number gives a negative number. Similarly, 2's complement of negative number gives a positive number.

That means, if you perform two times 2's complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**.

We know the 1's complement of $(\mathbf{108})_{\mathbf{10}}$ is $(\mathbf{10010011})_{\mathbf{2}}$

2's compliment of $(108)_{10}$ = 1's compliment of $(108)_{10}$ + 1.

= 10010011 + 1

= 10010100

Therefore, the **2's complement of $(108)_{10}$** is **$(10010100)_2$**.

In this chapter, let us discuss about the basic arithmetic operations, which can be performed on any two signed binary numbers using 2's complement method. The **basic arithmetic operations** are addition and subtraction.

Addition of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We can perform the **addition** of these two numbers, which is similar to the addition of two unsigned binary numbers. But, if the resultant sum contains carry out from sign bit, then discard (ignore) it in order to get the correct value.

If resultant sum is positive, you can find the magnitude of it directly. But, if the resultant sum is negative, then take 2's complement of it in order to get the magnitude.

Example 1

Let us perform the **addition** of two decimal numbers **+7 and +4** using 2's complement method.

The **2's complement** representations of +7 and +4 with 5 bits each are shown below.

$$(+7)_{10} = (00111)_2$$

$$(+4)_{10} = (00100)_2$$

The addition of these two numbers is

$$(+7)_{10} + (+4)_{10} = (00111)_2 + (00100)_2$$

$$\Rightarrow (+7)_{10} + (+4)_{10} = (01011)_2.$$

The resultant sum contains 5 bits. So, there is no carry out from sign bit. The sign bit '0' indicates that the resultant sum is **positive**. So, the magnitude of sum is 11 in decimal number system. Therefore, addition of two positive numbers will give another positive number.

Example 2

Let us perform the **addition** of two decimal numbers **-7** and **-4**using 2's complement method.

The **2's complement** representation of -7 and -4 with 5 bits each are shown below.

$$(-7)_{10} = (11001)_2$$

$$(-4)_{10} = (11100)_2$$

The addition of these two numbers is

$$(-7)_{10} + (-4)_{10} = (11001)_2 + (11100)_2$$

$$\Rightarrow (-7)_{10} + (-4)_{10} = (110101)_2.$$

The resultant sum contains 6 bits. In this case, carry is obtained from sign bit. So, we can remove it

Resultant sum after removing carry is $(-7)_{10} + (-4)_{10} = \mathbf{(10101)_2}$.

The sign bit '1' indicates that the resultant sum is **negative**. So, by taking 2's complement of it we will get the magnitude of resultant sum as 11 in decimal number system. Therefore, addition of two negative numbers will give another negative number.

Subtraction of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We know that 2's complement of positive number gives a negative number. So, whenever we have to subtract a number B from number A, then take 2's complement of B and add it to A. So, **mathematically**we can write it as

A - B = A + ***(2's complement of B)***

Similarly, if we have to subtract the number A from number B, then take 2's complement of A and add it to B. So, **mathematically** we can write it as

B - A = B + ***(2's complement of A)***

So, the subtraction of two signed binary numbers is similar to the addition of two signed binary numbers. But, we have to take 2's complement of the number, which is supposed to be subtracted. This is the **advantage** of 2's complement technique. Follow, the same rules of addition of two signed binary numbers.

Example 3

Let us perform the **subtraction** of two decimal numbers **+7 and +4** using 2's complement method.

The subtraction of these two numbers is

$$(+7)_{10} - (+4)_{10} = (+7)_{10} + (-4)_{10}.$$

The **2's complement** representation of +7 and -4 with 5 bits each are shown below.

$$(+7)_{10} = (00111)_2$$

$$(+4)_{10} = (11100)_2$$

$\Rightarrow (+7)_{10} + (+4)_{10} = (00111)_2 + (11100)_2 = (00011)_2$

Here, the carry obtained from sign bit. So, we can remove it. The resultant sum after removing carry is

$(+7)_{10} + (+4)_{10} =$ **$(00011)_2$**

The sign bit '0' indicates that the resultant sum is **positive**. So, the magnitude of it is 3 in decimal number system. Therefore, subtraction of two decimal numbers +7 and +4 is +3.

Example 4

Let us perform the **subtraction of** two decimal numbers **+4** and **+7** using 2's complement method.

The subtraction of these two numbers is

$$(+4)_{10} - (+7)_{10} = (+4)_{10} + (-7)_{10}.$$

The **2's complement** representation of +4 and -7 with 5 bits each are shown below.

$$(+4)_{10} = (00100)_2$$

$$(-7)_{10} = (11001)_2$$

$\Rightarrow (+4)_{10} + (-7)_{10} = (00100)_2 + (11001)_2 = (11101)_2$

Here, carry is not obtained from sign bit. The sign bit '1' indicates that the resultant sum is **negative**. So, by taking 2's complement of it we will get the magnitude of resultant sum as 3 in decimal number system. Therefore, subtraction of two decimal numbers +4 and +7 is -3.

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

Binary codes can be classified into two types.

- Weighted codes

- Unweighted codes

If the code has positional weights, then it is said to be **weighted code**. Otherwise, it is an unweighted code. Weighted codes can be further classified as positively weighted codes and negatively weighted codes.

Binary Codes for Decimal digits

The following table shows the various binary codes for decimal digits 0 to 9.

Decimal Digit	8421 Code	2421 Code	84-2-1 Code	Excess 3 Code
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

We have 10 digits in decimal number system. To represent these 10 digits in binary, we require minimum of 4 bits. But, with 4 bits there will be 16 unique combinations of zeros and ones. Since, we have only 10 decimal digits, the other 6 combinations of zeros and ones are not required.

8 4 2 1 code

- The weights of this code are 8, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- This code is also called as **natural BCD** (Binary Coded Decimal) **code**.

Example

Let us find the BCD equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the BCD (8421) codes of 7, 8 and 6 are 0111, 1000 and 0110 respectively.

$$\therefore (786)_{10} = (011110000110)_{BCD}$$

There are 12 bits in BCD representation, since each BCD code of decimal digit has 4 bits.

2 4 2 1 code

- The weights of this code are 2, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- It is an **unnatural BCD** code. Sum of weights of unnatural BCD codes is equal to 9.
- It is a **self-complementing** code. Self-complementing codes provide the 9's complement of a decimal number, just by interchanging 1's and 0's in its equivalent 2421 representation.

Example

Let us find the 2421 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 2421 codes of 7, 8 and 6 are 1101, 1110 and 1100 respectively.

Therefore, the 2421 equivalent of the decimal number 786 is **110111101100**.

8 4 -2 -1 code

- The weights of this code are 8, 4, -2 and -1.
- This code has negative weights along with positive weights. So, it is a **negatively weighted code**.
- It is an **unnatural BCD** code.
- It is a **self-complementing** code.

Example

Let us find the 8 4-2-1 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 8 4 -2 -1 codes of 7, 8 and 6 are 1001, 1000 and 1010 respectively.

Therefore, the 8 4 -2 -1 equivalent of the decimal number 786 is **100110001010**.

Excess 3 code

- This code doesn't have any weights. So, it is an **un-weighted code**.
- We will get the Excess 3 code of a decimal number by adding three (0011) to the binary equivalent of that decimal number. Hence, it is called as Excess 3 code.
- It is a **self-complementing** code.

Example

Let us find the Excess 3 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the Excess 3 codes of 7, 8 and 6 are 1010, 1011 and 1001 respectively.

Therefore, the Excess 3 equivalent of the decimal number 786 is **101010111001**

Gray Code

The following table shows the 4-bit Gray codes corresponding to each 4-bit binary code.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110

5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

- This code doesn't have any weights. So, it is an **un-weighted code**.
- In the above table, the successive Gray codes are differed in one bit position only. Hence, this code is called as **unit distance** code.

Binary code to Gray Code Conversion

Follow these steps for converting a binary code into its equivalent Gray code.

- Consider the given binary code and place a zero to the left of MSB.
- Compare the successive two bits starting from zero. If the 2 bits are same, then the output is zero. Otherwise, output is one.

- Repeat the above step till the LSB of Gray code is obtained.

Example

From the table, we know that the Gray code corresponding to binary code 1000 is 1100. Now, let us verify it by using the above procedure.

Given, binary code is 1000.

Step 1 − By placing zero to the left of MSB, the binary code will be 01000.

Step 2 − By comparing successive two bits of new binary code, we will get the gray code as **1100**.

Error Detection & Correction Codes

We know that the bits 0 and 1 corresponding to two different range of analog voltages. So, during transmission of binary data from one system to the other, the noise may also be added. Due to this, there may be errors in the received data at other system.

That means a bit 0 may change to 1 or a bit 1 may change to 0. We can't avoid the interference of noise. But, we can get back the original data first by detecting whether any errors*s* present and then correcting those errors. For this purpose, we can use the following codes.

- Error detection codes
- Error correction codes

Error detection codes − are used to detect the errors*s* present in the received data bitstream. These codes contain some bits*s*, which are included appended to the original bit stream. These codes detect the error, if it is occurred during transmission of the original data bitstream.**Example** − Parity code, Hamming code.

Error correction codes − are used to correct the errors*s* present in the received data bitstream so that, we will get the original data. Error correction codes also use the similar strategy of error detection codes.**Example** − Hamming code.

Therefore, to detect and correct the errors, additional bits*s* are appended to the data bits at the time of transmission.

Parity Code

It is easy to include append one parity bit either to the left of MSB or to the right of LSB of original bit stream. There are two types of parity codes, namely even parity code and odd parity code based on the type of parity being chosen.

Even Parity Code

The value of even parity bit should be zero, if even number of ones present in the binary code. Otherwise, it should be one. So that, even number of ones present in **even parity code**. Even parity code contains the data bits and even parity bit.

The following table shows the **even parity codes** corresponding to each 3-bit binary code. Here, the even parity bit is included to the right of LSB of binary code.

Binary Code	Even Parity bit	Even Parity Code
000	0	0000
001	1	0011
010	1	0101
011	0	0110
100	1	1001
101	0	1010
110	0	1100
111	1	1111

Here, the number of bits present in the even parity codes is 4. So, the possible even number of ones in these even parity codes are 0, 2 & 4.

- If the other system receives one of these even parity codes, then there is no error in the received data. The bits other than even parity bit are same as that of binary code.
- If the other system receives other than even parity codes, then there will be an error*s* in the received data. In this case, we can't predict the original binary code because we don't know the bit position*s* of error.

Therefore, even parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Odd Parity Code

The value of odd parity bit should be zero, if odd number of ones present in the binary code. Otherwise, it should be one. So that, odd number of ones present in **odd parity code**. Odd parity code contains the data bits and odd parity bit.

The following table shows the **odd parity codes** corresponding to each 3-bit binary code. Here, the odd parity bit is included to the right of LSB of binary code.

Binary Code	Odd Parity bit	Odd Parity Code
000	1	0001
001	0	0010
010	0	0100
011	1	0111
100	0	1000
101	1	1011
110	1	1101
111	0	1110

Here, the number of bits present in the odd parity codes is 4. So, the possible odd number of ones in these odd parity codes are 1 & 3.

- If the other system receives one of these odd parity codes, then there is no error in the received data. The bits other than odd parity bit are same as that of binary code.
- If the other system receives other than odd parity codes, then there is an errors$s$ in the received data. In this case, we can't predict the original binary code because we don't know the bit positions$s$ of error.

Therefore, odd parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Hamming Code

Hamming code is useful for both detection and correction of error present in the received data. This code uses multiple parity bits and we have to place these parity bits in the positions of powers of 2.

The **minimum value of 'k'** for which the following relation is correct valid$valid$ is nothing but the required number of parity bits.

$$2^k \geq n + k + 1$$

Where,

'n' is the number of bits in the binary code information$information$

'k' is the number of parity bits

Therefore, the number of bits in the Hamming code is equal to n + k.

Let the **Hamming code** is $b_{n+k}b_{n+k-1}$..... & parity bits $p_k,p_{k-1},....p_1$ $pk,pk-1,....p1$. We can place the 'k' parity bits in powers of 2 positions only. In remaining bit positions, we can place the 'n' bits of binary code.

Based on requirement, we can use either even parity or odd parity while forming a Hamming code. But, the same parity technique should be used in order to find whether any error present in the received data.

Follow this procedure for finding **parity bits**.

- Find the value of $\mathbf{p_1}$, based on the number of ones present in bit positions b_3, b_5, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^0.
- Find the value of $\mathbf{p_2}$, based on the number of ones present in bit positions b_3, b_6, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^1.
- Find the value of $\mathbf{p_3}$, based on the number of ones present in bit positions b_5, b_6, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^2.
- Similarly, find other values of parity bits.

Follow this procedure for finding **check bits**.

- Find the value of c_1, based on the number of ones present in bit positions b_1, b_3, b_5, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^0.
- Find the value of c_2, based on the number of ones present in bit positions b_2, b_3, b_6, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^1.
- Find the value of c_3, based on the number of ones present in bit positions b_4, b_5, b_6, b_7 and so on. All these bit positions $suffixes$ in their equivalent binary have '1' in the place value of 2^2.
- Similarly, find other values of check bits.

The decimal equivalent of the check bits in the received data gives the value of bit position, where the error is present. Just complement the value present in that bit position. Therefore, we will get the original binary code after removing parity bits.

Example 1

Let us find the Hamming code for binary code, $d_4d_3d_2d_1$ = 1000. Consider even parity bits.

The number of bits in the given binary code is n=4.

We can find the required number of parity bits by using the following mathematical relation.

$$2^k \geq n + k + 1$$

Substitute, n=4 in the above mathematical relation.

$$\Rightarrow 2^k \geq 4 + k + 1$$

$$\Rightarrow 2^k \geq 5 + k$$

The minimum value of k that satisfied the above relation is 3. Hence, we require 3 parity bits p_1, p_2, and p_3. Therefore, the number of bits in Hamming code will be 7, since there are 4 bits in binary code and 3 parity bits. We have to place the parity bits and bits of binary code in the Hamming code as shown below.

The **7-bit Hamming code** is b7b6b5b4b3b2b1=d4d3d2p3d1p2bp1$b7b6b5b4b3b2b1=d4d3d2p3d1p2bp1$

By substituting the bits of binary code, the Hamming code will be b7b6b5b4b3b2b1=100p3Op2p1$b7b6b5b4b3b2b1=100p3Op2p1$. Now, let us find the parity bits.

p1=b7⊕b5⊕b3=1⊕0⊕0=1$p1=b7⊕b5⊕b3=1⊕0⊕0=1$

p2=b7⊕b6⊕b3=1⊕0⊕0=1$p2=b7⊕b6⊕b3=1⊕0⊕0=1$

p3=b7⊕b6⊕b5=1⊕0⊕0=1

By substituting these parity bits, the **Hamming code** will be $b_7b_6b_5b_4b_3b_2b_1$=1001011$b7b6b5b4b3b2b1=1001011$.

Example 2

In the above example, we got the Hamming code as $b_7b_6b_5b_4b_3b_2b_1$=1001011$b7b6b5b4b3b2b1=1001011$. Now, let us find the error position when the code received is $b_7b_6b_5b_4b_3b_2b_1$=1001111$b7b6b5b4b3b2b1=1001111$.

Now, let us find the check bits.

$$c_1=b_7\oplus b_5\oplus b_3\oplus b_1=1\oplus 0\oplus 1\oplus 1=1 c1=b7⊕b5⊕b3⊕b1=1⊕0⊕1⊕1=1$$

$$c_2=b_7\oplus b_6\oplus b_3\oplus b_2=1\oplus 0\oplus 1\oplus 1=1 c2=b7⊕b6⊕b3⊕b2=1⊕0⊕1⊕1=1$$

$$c_3=b_7\oplus b_6\oplus b_5\oplus b_4=1\oplus 0\oplus 0\oplus 1=0 c3=b7⊕b6⊕b5⊕b4=1⊕0⊕0⊕1=0$$

The decimal value of check bits gives the position of error in received Hamming code.

$$c_3c_2c_1=(011)_2=(3)_{10} c3c2c1=(011)2=(3)10$$

Therefore, the error present in third bit (b_3) of Hamming code. Just complement the value present in that bit and remove parity bits in order to get the original binary code.

Boolean Algebra

is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables.

The range of voltages corresponding to Logic 'High' is represented with '1' and the range of voltages corresponding to logic 'Low' is represented with '0'.

Postulates and Basic Laws of Boolean Algebra

In this section, let us discuss about the Boolean postulates and basic laws that are used in Boolean algebra. These are useful in minimizing Boolean functions.

Boolean Postulates

Consider the binary numbers 0 and 1, Boolean variable (x) and its complement (x'). Either the Boolean variable or complement of it is known as **literal**. The four possible **logical OR** operations among these literals and binary numbers are shown below.

$$x + 0 = x$$

$$x + 1 = 1$$

$$x + x = x$$

$$x + x' = 1$$

Similarly, the four possible **logical AND** operations among those literals and binary numbers are shown below.

$$x.1 = x$$

$$x.0 = 0$$

$$x.x = x$$

$$x.x' = 0$$

These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with '0' or '1'.

Note− The complement of complement of any Boolean variable is equal to the variable itself. i.e., (x')'=x.

Basic Laws of Boolean Algebra

Following are the three basic laws of Boolean Algebra.

- Commutative law
- Associative law
- Distributive law

Commutative Law

If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be **Commutative**. The logical OR & logical AND operations of two Boolean variables x & y are shown below

$$x + y = y + x$$

$$x.y = y.x$$

The symbol '+' indicates logical OR operation. Similarly, the symbol '.' indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.

Associative Law

If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be **Associative**. The logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$$x + (y + z) = (x + y) + z$$

$$x.(y.z) = (x.y).z$$

Associative law obeys for logical OR & logical AND operations.

Distributive Law

If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be **Distributive**. The distribution of logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$$x.(y + z) = x.y + x.z$$

$$x + (y.z) = (x + y).(x + z)$$

Distributive law obeys for logical OR and logical AND operations.

These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with '0' or '1'.

Theorems of Boolean Algebra

The following two theorems are used in Boolean algebra.

- Duality theorem
- DeMorgan's theorem

Duality Theorem

This theorem states that the **dual** of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.

Let us make the Boolean equations (relations) that we discussed in the section of Boolean postulates and basic laws into two groups. The following table shows these two groups.

Group1	Group2
x + 0 = x	x.1 = x
x + 1 = 1	x.0 = 0
x + x = x	x.x = x
x + x' = 1	x.x' = 0
x + y = y + x	x.y = y.x
x + (y + z) = (x + y) + z	x.(y.z) = (x.y).z
x.(y + z) = x.y + x.z	x + (y.z) = (x + y).(x + z)

In each row, there are two Boolean equations and they are dual to each other. We can verify all these Boolean equations of Group1 and Group2 by using duality theorem.

DeMorgan's Theorem

This theorem is useful in finding the **complement of Boolean function**. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable.

DeMorgan's theorem with 2 Boolean variables x and y can be represented as

$$(x + y)' = x'.y'$$

The dual of the above Boolean function is

$$(x.y)' = x' + y'$$

Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan's theorem for more than 2 Boolean variables also.

Simplification of Boolean Functions

Till now, we discussed the postulates, basic laws and theorems of Boolean algebra. Now, let us simplify some Boolean functions.

Example 1

Let us **simplify** the Boolean function, f = p'qr + pq'r + pqr' + pqr

We can simplify this function in two methods.

Method 1

Given Boolean function, f = p'qr + pq'r + pqr' +pqr.

Step 1 − In first and second terms r is common and in third and fourth terms pq is common. So, take the common terms by using **Distributive law**.

$$\Rightarrow f = (p'q + pq')r + pq(r' + r)$$

Step 2 − The terms present in first parenthesis can be simplified to Ex-OR operation. The terms present in second parenthesis can be simplified to '1' using **Boolean postulate**

$$\Rightarrow f = (p \oplus q)r + pq(1)$$

Step 3 − The first term can't be simplified further. But, the second term can be simplified to pq using **Boolean postulate**.

$$\Rightarrow f = (p \oplus q)r + pq$$

Therefore, the simplified Boolean function is **$f = (p \oplus q)r + pq$**

Method 2

Given Boolean function, f = p'qr + pq'r + pqr' + pqr.

Step 1 − Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 − Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

$$\Rightarrow f = qr(p' + p) + pr(q' + q) + pq(r' + r)$$

Step 3 − Use **Boolean postulate**, x + x' = 1 for simplifying the terms present in each parenthesis.

$$\Rightarrow f = qr(1) + pr(1) + pq(1)$$

Step 4 − Use **Boolean postulate**, x.1 = x for simplifying the above three terms.

$$\Rightarrow f = qr + pr + pq$$

$$\Rightarrow f = pq + qr + pr$$

Therefore, the simplified Boolean function is **f = pq + qr + pr**.

So, we got two different Boolean functions after simplifying the given Boolean function in each method. Functionally, those two Boolean functions are same. So, based on the requirement, we can choose one of those two Boolean functions.

Example 2

Let us find the **complement** of the Boolean function, f = p'q + pq'.

The complement of Boolean function is f' = (p'q + pq')'.

Step 1 − Use DeMorgan's theorem, (x + y)' = x'.y'.

$$\Rightarrow f' = (p'q)'.(pq')'$$

Step 2 − Use DeMorgan's theorem, (x.y)' = x' + y'

$$\Rightarrow f' = \{(p')' + q'\}.\{p' + (q')'\}$$

Step3 − Use the Boolean postulate, (x')'=x.

$$\Rightarrow f' = \{p + q'\}.\{p' + q\}$$

$$\Rightarrow f' = pp' + pq + p'q' + qq'$$

Step 4 − Use the Boolean postulate, xx'=0.

$$\Rightarrow f = 0 + pq + p'q' + 0$$

$$\Rightarrow f = pq + p'q'$$

Therefore, the **complement** of Boolean function, p'q + pq' is **pq + p'q'**.

We will get four Boolean product terms by combining two variables x and y with logical AND operation. These Boolean product terms are called as **min terms** or **standard product terms**. The min terms are x'y', x'y, xy' and xy.

Similarly, we will get four Boolean sum terms by combining two variables x and y with logical OR operation. These Boolean sum terms are called as **Max terms** or **standard sum terms**. The Max terms are x + y, x + y', x' + y and x' + y'.

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

x	y	Min terms	Max terms
0	0	$m_0=x'y'$	$M_0=x + y$
0	1	$m_1=x'y$	$M_1=x + y'$
1	0	$m_2=xy'$	$M_2=x' + y$
1	1	$m_3=xy$	$M_3=x' + y'$

If the binary variable is '0', then it is represented as complement of variable in min term and as the variable itself in Max term. Similarly, if the binary variable is '1', then it is represented as complement of variable in Max term and as the variable itself in min term.

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.

Canonical SoP and PoS forms

A truth table consists of a set of inputs and output(s). If there are 'n' input variables, then there will be 2^n possible combinations with zeros and ones. So the value of each output variable depends on the combination of input variables. So, each output variable will have '1' for some combination of input variables and '0' for some other combination of input variables.

Therefore, we can express each output variable in following two ways.

- Canonical SoP form
- Canonical PoS 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.

First, identify the min terms for which, the output variable is one and then do the logical OR of those min terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of sum of min terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the following **truth table**.

Inputs		**Output**	
p	**q**	**r**	**f**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1

1	1	1	1

Here, the output (f) is '1' for four combinations of inputs. The corresponding min terms are p'qr, pq'r, pqr', pqr. By doing logical OR of these four min terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, f = p'qr + pq'r + pqr' + pqr. This is the **canonical SoP form** of output, f. We can also represent this function in following two notations.

$$f=m3+m5+m6+m7f=m3+m5+m6+m7$$

$$f=\sum m(3,5,6,7)f=\sum m(3,5,6,7)$$

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.

First, identify the Max terms for which, the output variable is zero and then do the logical AND of those Max terms in order to get the Boolean expression (function) corresponding to that output variable. This Boolean function will be in the form of product of Max terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the same truth table of previous example. Here, the output (f) is '0' for four combinations of inputs. The corresponding Max terms are p + q + r, p + q + r', p + q' + r, p' + q + r. By doing logical AND of these four Max terms, we will get the Boolean function of output (f).

Therefore, the Boolean function of output is, f = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r). This is the **canonical PoS form**of output, f. We can also represent this function in following two notations.

$$f=M0.M1.M2.M4f=M0.M1.M2.M4$$

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

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 = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r) is the dual of the Boolean function, f = p'qr + pq'r + pqr' + pqr.

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.

Standard SoP and PoS forms

We discussed two canonical forms of representing the Boolean output(s). Similarly, there are two standard forms of representing the Boolean output(s). These are the simplified version of canonical forms.

- Standard SoP form
- Standard PoS form

We will discuss about Logic gates in later chapters. 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.

Follow the same procedure for other output variables also, if there is more than one output variable. 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.

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

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

Step 1 − Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 − Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

$$\Rightarrow f = qr(p' + p) + pr(q' + q) + pq(r' + r)$$

Step 3 − Use **Boolean postulate**, $x + x' = 1$ for simplifying the terms present in each parenthesis.

$$\Rightarrow f = qr(1) + pr(1) + pq(1)$$

Step 4 − Use **Boolean postulate**, $x.1 = x$ for simplifying above three terms.

$$\Rightarrow f = qr + pr + pq$$

$$\Rightarrow f = pq + qr + pr$$

This is the simplified Boolean function. Therefore, the **standard SoP form** corresponding to given canonical SoP form is **f = 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.

Follow the same procedure for other output variables also, if there is more than one output variable. 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.

$$f = (p + q + r).(p + q + r').(p + q' + r).(p' + q + r)$$

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

Step 1 − Use the **Boolean postulate**, $x.x = x$. That means, the Logical AND operation with any Boolean variable 'n' times will be equal to the same variable. So, we can write the first term p+q+r two more times.

$$\Rightarrow f = (p + q + r).(p + q + r).(p + q + r).(p + q + r').(p +q' + r).(p' + q + r)$$

Step 2 − Use **Distributive law,** x + (y.z) = (x + y).(x + z) for 1st and 4th parenthesis, 2nd and 5th parenthesis, 3rd and 6thparenthesis.

$$\Rightarrow f = (p + q + rr').(p + r + qq').(q + r + pp')$$

Step 3 − Use **Boolean postulate**, x.x'=0 for simplifying the terms present in each parenthesis.

$$\Rightarrow f = (p + q + 0).(p + r + 0).(q + r + 0)$$

Step 4 − Use **Boolean postulate**, x + 0 = x for simplifying the terms present in each parenthesis

$$\Rightarrow f = (p + q).(p + r).(q + r)$$

$$\Rightarrow f = (p + q).(q + r).(p + r)$$

This is the simplified Boolean function. Therefore, the **standard PoS form** corresponding to given canonical PoS form is **f = (p + q).(q + r).(p + 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.

Digital electronic circuits operate with voltages of **two logic levels**namely Logic Low and Logic High. The range of voltages corresponding to Logic Low is represented with '0'. Similarly, the range of voltages corresponding to Logic High is represented with '1'.

The basic digital electronic circuit that has one or more inputs and single output is known as **Logic gate**. Hence, the Logic gates are the building blocks of any digital system. We can classify these Logic gates into the following three categories.

- Basic gates
- Universal gates
- Special gates

Now, let us discuss about the Logic gates come under each category one by one.

Basic Gates

In earlier chapters, we learnt that the Boolean functions can be represented either in sum of products form or in product of sums form based on the requirement. So, we can implement these Boolean functions by using basic gates. The basic gates are AND, OR & NOT gates.

AND gate

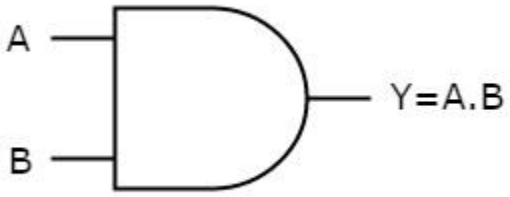
An AND gate is a digital circuit that has two or more inputs and produces an output, which is the **logical AND** of all those inputs. It is optional to represent the **Logical AND** with the symbol '.'.

The following table shows the **truth table** of 2-input AND gate.

A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output of two input AND gate. If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'.

The following figure shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.



This AND gate produces an output (Y), which is the **logical AND**of two inputs A, B. Similarly, if there are 'n' inputs, then the AND gate produces an output, which is the logical AND of all those inputs. That means, the output of AND gate will be '1', when all the inputs are '1'.

OR gate

An OR gate is a digital circuit that has two or more inputs and produces an output, which is the logical OR of all those inputs. This **logical OR** is represented with the symbol '+'.

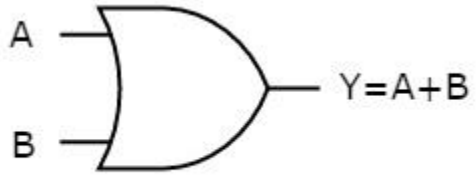
The following table shows the **truth table** of 2-input OR gate.

A	B	Y = A + B
0	0	0

0	1	1
1	0	1
1	1	1

Here A, B are the inputs and Y is the output of two input OR gate. If both inputs are '0', then only the output, Y is '0'. For remaining combinations of inputs, the output, Y is '1'.

The following figure shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.



This OR gate produces an output (Y), which is the **logical OR** of two inputs A, B. Similarly, if there are 'n' inputs, then the OR gate produces an output, which is the logical OR of all those inputs. That means, the output of an OR gate will be '1', when at least one of those inputs is '1'.

NOT gate

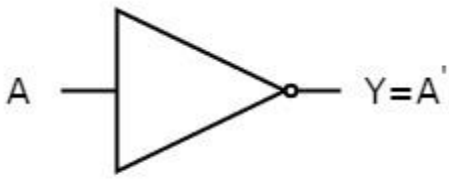
A NOT gate is a digital circuit that has single input and single output. The output of NOT gate is the **logical inversion** of input. Hence, the NOT gate is also called as inverter.

The following table shows the **truth table** of NOT gate.

A	Y = A'
0	1
1	0

Here A and Y are the input and output of NOT gate respectively. If the input, A is '0', then the output, Y is '1'. Similarly, if the input, A is '1', then the output, Y is '0'.

The following figure shows the **symbol** of NOT gate, which is having one input, A and one output, Y.



This NOT gate produces an output (Y), which is the **complement** of input, A.

Universal gates

NAND & NOR gates are called as **universal gates**. Because we can implement any Boolean function, which is in sum of products form by using NAND gates alone. Similarly, we can implement any Boolean function, which is in product of sums form by using NOR gates alone.

NAND gate

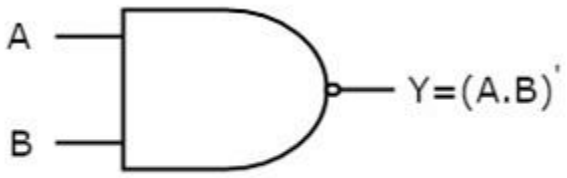
NAND gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical AND** of all those inputs.

The following table shows the **truth table** of 2-input NAND gate.

A	B	Y = (A.B)'
0	0	1
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input NAND gate. When both inputs are '1', the output, Y is '0'. If at least one of the input is zero, then the output, Y is '1'. This is just opposite to that of two input AND gate operation.

The following image shows the **symbol** of NAND gate, which is having two inputs A, B and one output, Y.



NAND gate operation is same as that of AND gate followed by an inverter. That's why the NAND gate symbol is represented like that.

NOR gate

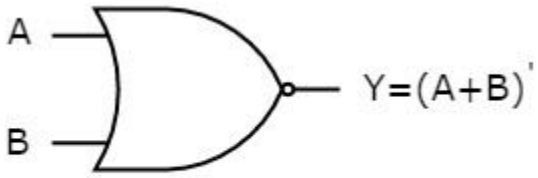
NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.

The following table shows the **truth table** of 2-input NOR gate

A	B	Y = (A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

Here A, B are the inputs and Y is the output. If both inputs are '0', then the output, Y is '1'. If at least one of the input is '1', then the output, Y is '0'. This is just opposite to that of two input OR gate operation.

The following figure shows the **symbol** of NOR gate, which is having two inputs A, B and one output, Y.



NOR gate operation is same as that of OR gate followed by an inverter. That's why the NOR gate symbol is represented like that.

Special Gates

Ex-OR & Ex-NOR gates are called as special gates. Because, these two gates are special cases of OR & NOR gates.

Ex-OR gate

The full form of Ex-OR gate is **Exclusive-OR** gate. Its function is same as that of OR gate except for some cases, when the inputs having even number of ones.

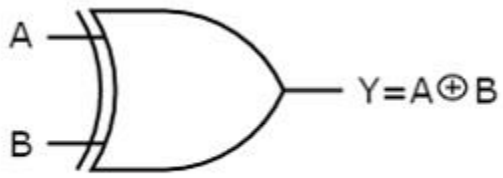
The following table shows the **truth table** of 2-input Ex-OR gate.

A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input Ex-OR gate. The truth table of Ex-OR gate is same as that of OR gate for first three rows. The only modification is in the fourth row. That means, the output (Y) is zero instead of one, when both the inputs are one, since the inputs having even number of ones.

Therefore, the output of Ex-OR gate is '1', when only one of the two inputs is '1'. And it is zero, when both inputs are same.

Below figure shows the **symbol** of Ex-OR gate, which is having two inputs A, B and one output, Y.



Ex-OR gate operation is similar to that of OR gate, except for few combination(s) of inputs. That's why the Ex-OR gate symbol is represented like that. The output of Ex-OR gate is '1', when odd number of ones present at the inputs. Hence, the output of Ex-OR gate is also called as an **odd function**.

Ex-NOR gate

The full form of Ex-NOR gate is **Exclusive-NOR** gate. Its function is same as that of NOR gate except for some cases, when the inputs having even number of ones.

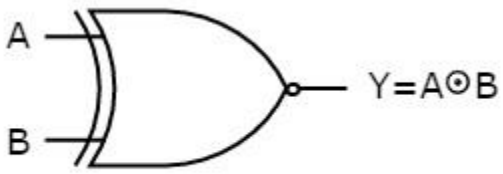
The following table shows the **truth table** of 2-input Ex-NOR gate.

A	B	Y = A⊙B
0	0	1
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output. The truth table of Ex-NOR gate is same as that of NOR gate for first three rows. The only modification is in the fourth row. That means, the output is one instead of zero, when both the inputs are one.

Therefore, the output of Ex-NOR gate is '1', when both inputs are same. And it is zero, when both the inputs are different.

The following figure shows the **symbol** of Ex-NOR gate, which is having two inputs A, B and one output, Y.



Ex-NOR gate operation is similar to that of NOR gate, except for few combination(s) of inputs. That's why the Ex-NOR gate symbol is represented like that. The output of Ex-NOR gate is '1', when even

number of ones present at the inputs. Hence, the output of Ex-NOR gate is also called as an **even function**.

From the above truth tables of Ex-OR & Ex-NOR logic gates, we can easily notice that the Ex-NOR operation is just the logical inversion of Ex-OR operation.

The maximum number of levels that are present between inputs and output is two in **two level logic**. That means, irrespective of total number of logic gates, the maximum number of Logic gates that are present (cascaded) between any input and output is two in two level logic. Here, the outputs of first level Logic gates are connected as inputs of second level Logic gate(s).

Consider the four Logic gates AND, OR, NAND & NOR. Since, there are 4 Logic gates, we will get 16 possible ways of realizing two level logic. Those are AND-AND, AND-OR, ANDNAND, AND-NOR, OR-AND, OR-OR, OR-NAND, OR-NOR, NAND-AND, NAND-OR, NANDNAND, NAND-NOR, NOR-AND, NOR-OR, NOR-NAND, NOR-NOR.

These two level logic realizations can be classified into the following two categories.

- Degenerative form
- Non-degenerative form

Degenerative Form

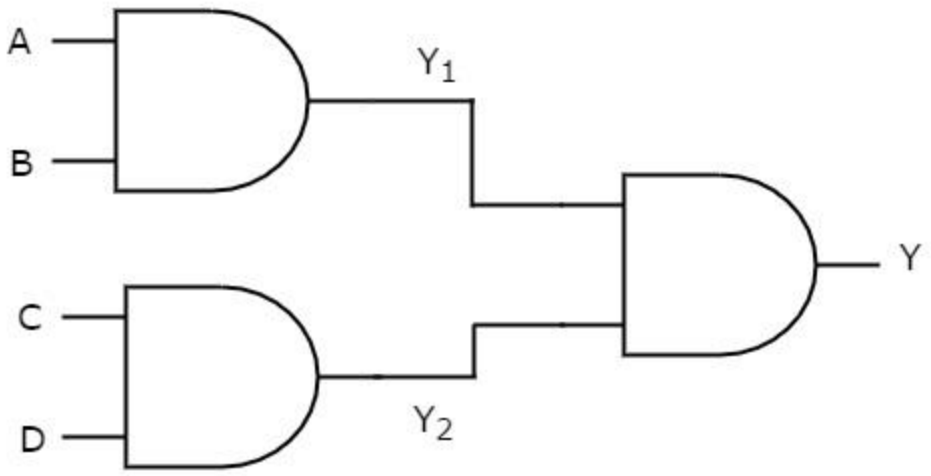
If the output of two level logic realization can be obtained by using single Logic gate, then it is called as **degenerative form**. Obviously, the number of inputs of single Logic gate increases. Due to this, the fan-in of Logic gate increases. This is an advantage of degenerative form.

Only **6 combinations** of two level logic realizations out of 16 combinations come under degenerative form. Those are AND-AND, AND-NAND, OR-OR, OR-NOR, NAND-NOR, NORNAND.

In this section, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-AND Logic

In this logic realization, AND gates are present in both levels. Below figure shows an example for **AND-AND logic** realization.



We will get the outputs of first level logic gates as Y1=ABY1=AB and Y2=CDY2=CD

These outputs, Y1Y1 and Y2Y2 are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

$$Y=Y1Y2Y=Y1Y2$$

Substitute Y1Y1 and Y2Y2 values in the above equation.

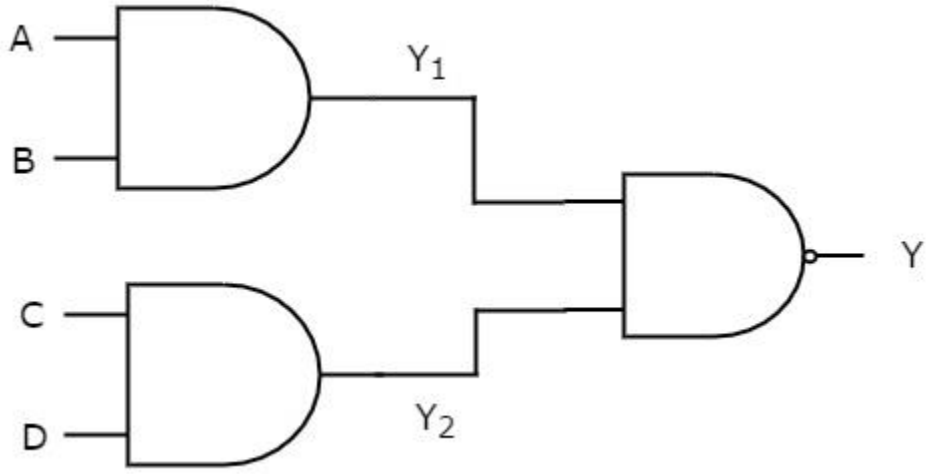
$$Y=(AB)(CD)Y=(AB)(CD)$$

$\Rightarrow Y=ABCD\Rightarrow Y=ABCD$

Therefore, the output of this AND-AND logic realization is **ABCD**. This Boolean function can be implemented by using a 4 input AND gate. Hence, it is **degenerative form**.

AND-NAND Logic

In this logic realization, AND gates are present in first level and NAND gate(s) are present in second level. The following figure shows an example for **AND-NAND logic** realization.



Previously, we got the outputs of first level logic gates as Y1=ABY1=AB and Y2=CDY2=CD

These outputs,Y1Y1 and Y2Y2 are applied as inputs of NAND gate that is present in second level. So, the output of this NAND gate is

$$Y=(Y1Y2)'Y=(Y1Y2)'$$

Substitute Y1Y1 and Y2Y2 values in the above equation.

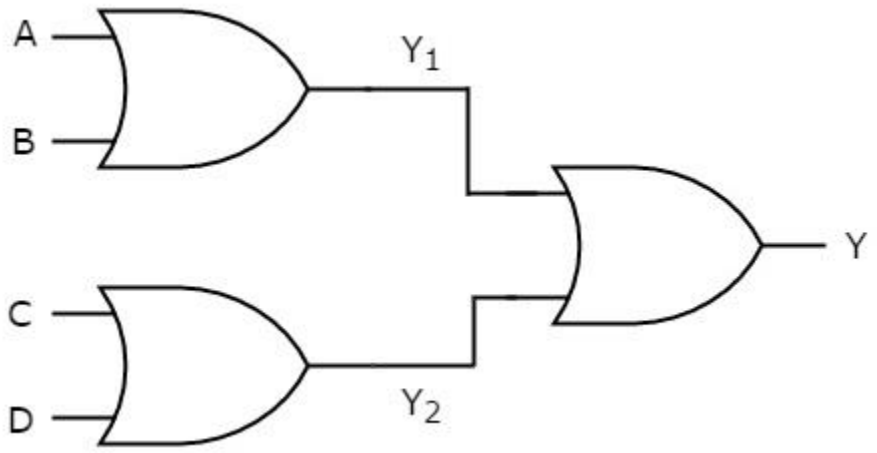
$$Y=((AB)(CD))'Y=((AB)(CD))'$$

$\Rightarrow Y=(ABCD)'\Rightarrow Y=(ABCD)'$

Therefore, the output of this AND-NAND logic realization is (ABCD)'(ABCD)'. This Boolean function can be implemented by using a 4 input NAND gate. Hence, it is **degenerative form**.

OR-OR Logic

In this logic realization, OR gates are present in both levels. The following figure shows an example for **OR-OR logic** realization.



We will get the outputs of first level logic gates as Y1=A+BY1=A+Band Y2=C+DY2=C+D.

These outputs, Y1Y1 and Y2Y2 are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

$$Y=Y1+Y2Y=Y1+Y2$$

Substitute Y1Y1 and Y2Y2 values in the above equation.

$$Y=(A+B)+(C+D)Y=(A+B)+(C+D)$$

$\Rightarrow Y=A+B+C+D\Rightarrow Y=A+B+C+D$

Therefore, the output of this OR-OR logic realization is **A+B+C+D**. This Boolean function can be implemented by using a 4 input OR gate. Hence, it is **degenerative form**.

Similarly, you can verify whether the remaining realizations belong to this category or not.

Non-degenerative Form

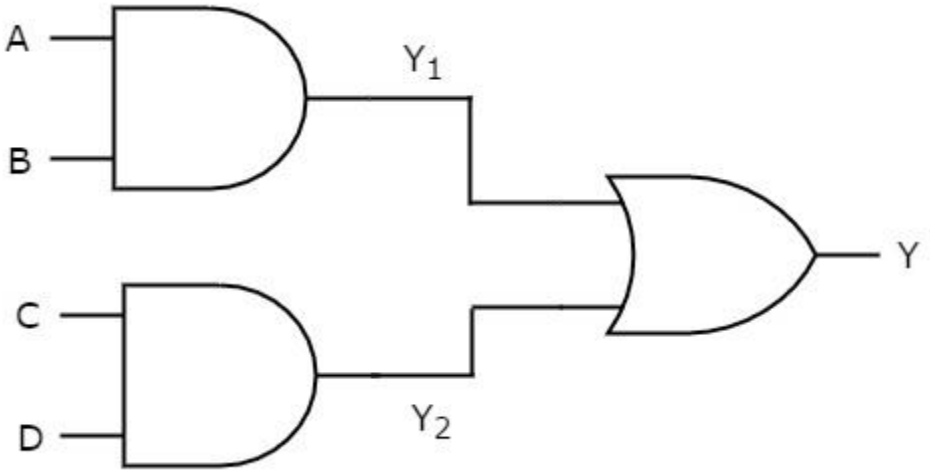
If the output of two level logic realization can't be obtained by using single logic gate, then it is called as **non-degenerative form**.

The remaining **10 combinations** of two level logic realizations come under nondegenerative form. Those are AND-OR, AND-NOR, OR-AND, OR-NAND, NAND-AND, NANDOR, NAND-NAND, NOR-AND, NOR-OR, NOR-NOR.

Now, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-OR Logic

In this logic realization, AND gates are present in first level and OR gate(s) are present in second level. Below figure shows an example for **AND-OR logic** realization.



Previously, we got the outputs of first level logic gates as Y1=ABY1=AB and Y2=CDY2=CD.
These outputs, Y1 and Y2 are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

$$Y=Y1+Y2Y=Y1+Y2$$

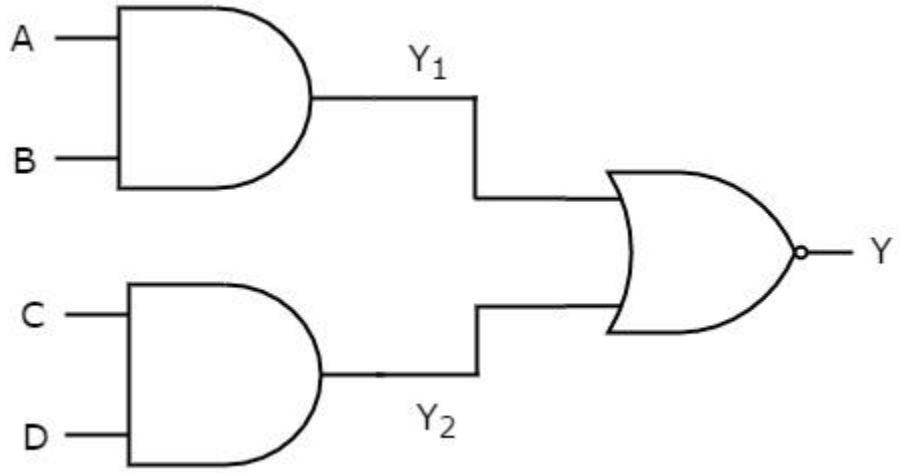
Substitute Y1Y1 and Y2Y2 values in the above equation

$$Y=AB+CDY=AB+CD$$

Therefore, the output of this AND-OR logic realization is **AB+CD**. This Boolean function is in **Sum of Products** form. Since, we can't implement it by using single logic gate, this AND-OR logic realization is a **non-degenerative form**.

AND-NOR Logic

In this logic realization, AND gates are present in first level and NOR gate(s) are present in second level. The following figure shows an example for **AND-NOR logic** realization.



We know the outputs of first level logic gates as Y1=ABY1=AB and Y2=CDY2=CD

These outputs, Y1 and Y2 are applied as inputs of NOR gate that is present in second level. So, the output of this NOR gate is

$$Y=(Y1+Y2)'Y=(Y1+Y2)'$$

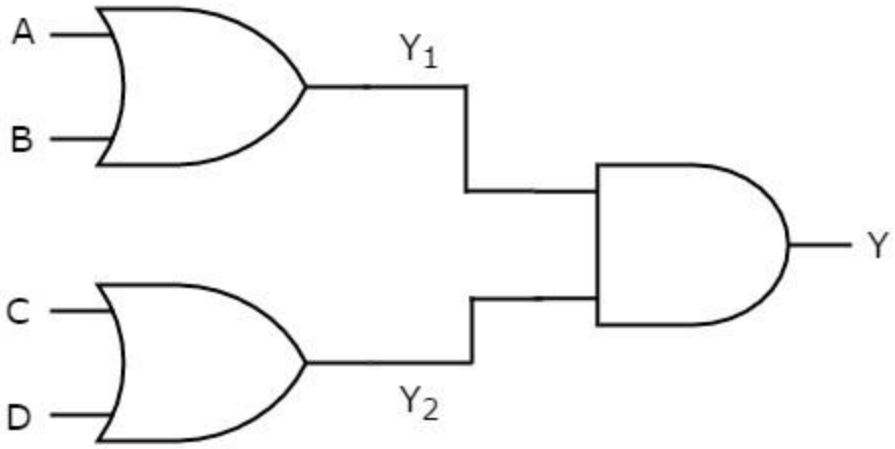
Substitute Y1Y1 and Y2Y2 values in the above equation.

$$Y=(AB+CD)'Y=(AB+CD)'$$

Therefore, the output of this AND-NOR logic realization is (AB+CD)'(AB+CD)'. This Boolean function is in **AND-OR-Invert** form. Since, we can't implement it by using single logic gate, this AND-NOR logic realization is a **non-degenerative form**

OR-AND Logic

In this logic realization, OR gates are present in first level & AND gate(s) are present in second level. The following figure shows an example for **OR-AND logic** realization.



Previously, we got the outputs of first level logic gates as Y1=A+BY1=A+B and Y2=C+DY2=C+D.
These outputs, Y1Y1 and Y2Y2 are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

$$Y=Y1Y2Y=Y1Y2$$

Substitute Y1Y1 and Y2Y2 values in the above equation.

$$Y=(A+B)(C+D)Y=(A+B)(C+D)$$

Therefore, the output of this OR-AND logic realization is **(A + B) (C + D)**. This Boolean function is in **Product of Sums** form. Since, we can't implement it by using single logic gate, this OR-AND logic realization is a **non-degenerative form**.

Similarly, you can verify whether the remaining realizations belong to this category or not.

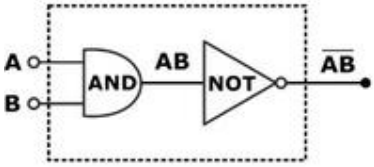
Universal Gates

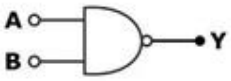
Universal gates are those gates that can perform the tasks of other gates with minor adjustments. Universal gates are widely used in formulating NAT-based questions in the **GATE exam**. There are two universal gates:

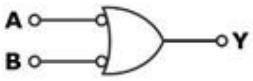
- NAND Gate
- NOR Gate

NAND Gate

The NAND gate is one of the universal gates. The NAND gate is a AND gate followed by a NOT gate. Thus, we can say it is a AND NOT operation. It may have two or more inputs but only one output. The logical symbols of a **NAND Gate** and the truth table are shown below.







A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

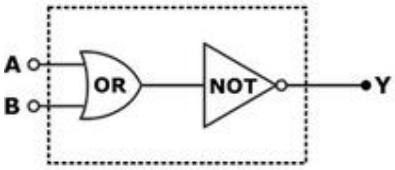
- Thus, the logical expression for the output is

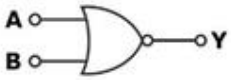
$$Y = \overline{A \cdot B} = \bar{A} + \bar{B}$$

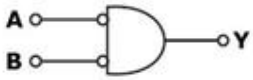
It is clear from the truth table of the two-input NAND gate that the output is 1 when either A or B or both the inputs are at logic '0'. We can say that if $\bar{A} = 1 = B$ are, both A and B are 1, and the output is 1. Therefore, the NAND gate can perform the OR function by inverting the inputs.

NOR Gate

The NOR gate is one of the universal gates. A NOR gate combines two **basic logic gates**: an OR gate and a NOT gate. So we can say it is an OR-NOT operation. It may have two or more inputs and an output. The logical symbols of the **NOR Gate** are shown:







A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

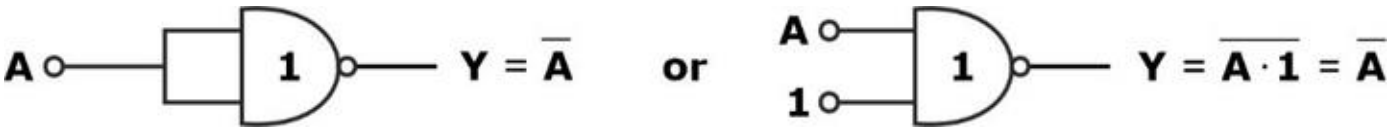
- It is clear from the truth table that the output is '1' only if all the inputs are at logic '0'. It can also say that if the inputs A' = B' = 1, the output Y is 1. Thus, the NOR gate is equivalent to the AND gate with inverted inputs, and it can be realized by a bubbled AND gate, as shown above.
- The logical expression for the output is

$$Y = \overline{A + B} = \overline{A}\,\overline{B}$$

NOT Gate Realization

Using NAND Gate

For the NOT gate realization, we require 1 NAND gate, as shown in the circuit diagram:



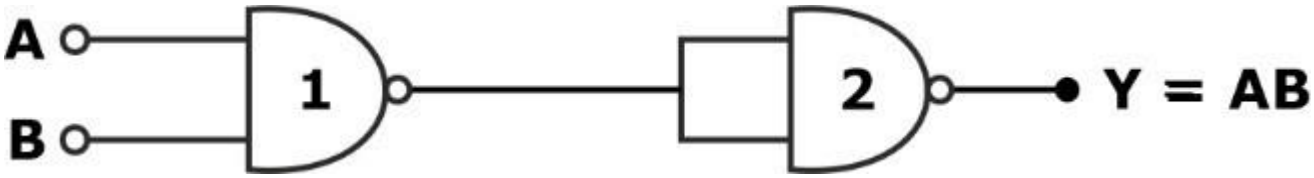
Using NOR Gate

For the NOT gate realization, we require 1 NOR gate, as shown in the circuit diagram:

AND Gate Realization

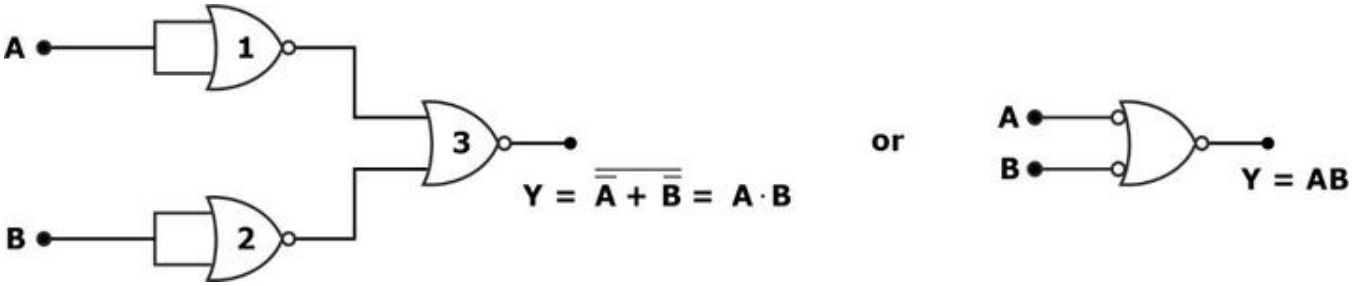
Using NAND gate

For the AND gate realization, we require 2 NAND gates, as shown in the circuit diagram:



Using NOR Gate

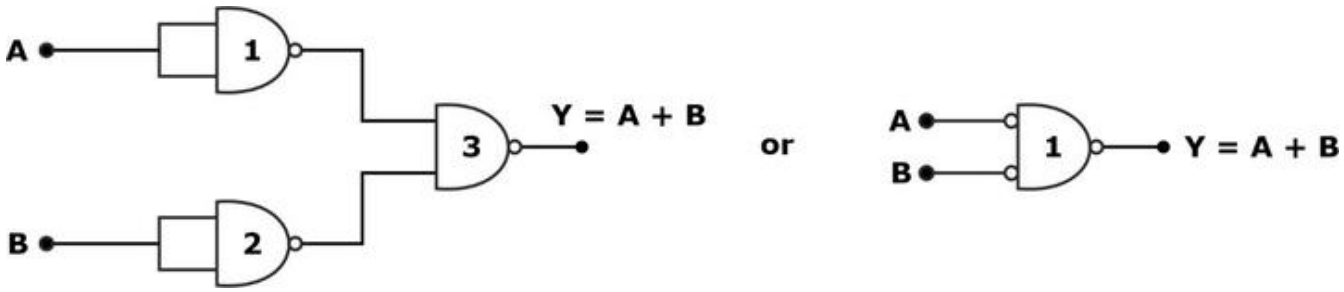
For the AND gate realization, we require 3 NOR gates if the inputs are not available in complement form, as shown in the circuit diagram:



OR Gate Realization

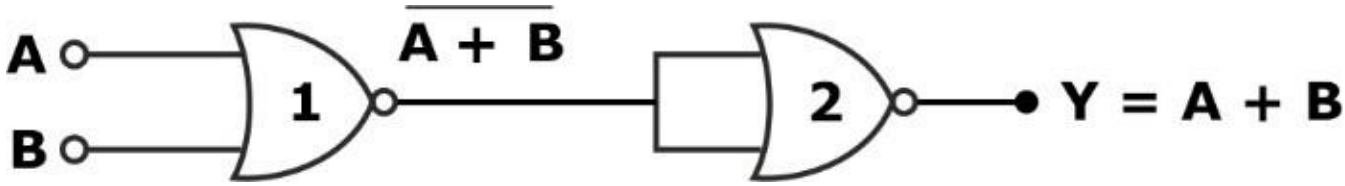
Using NAND Gate

For the OR gate realization, we require 3 NAND gates if the inputs are not available in complement form, as shown in the circuit diagram:



Using NOR Gate

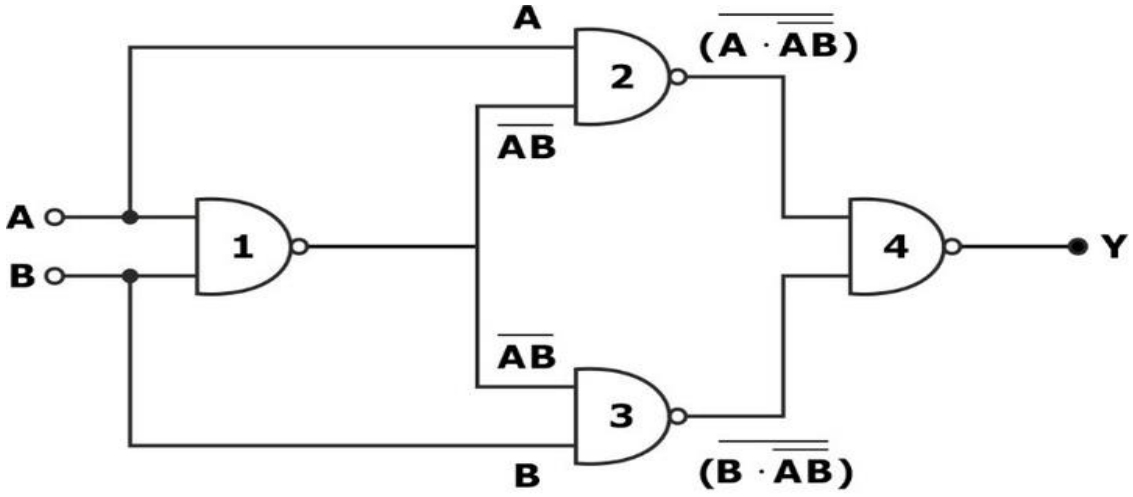
For the OR gate realization, we require 2 NOR gates, as shown in the circuit diagram:



EX-OR Gate Realization

Using NAND Gate

For the EX-OR gate realization, we require 4 NAND gates, as shown in the circuit diagram:

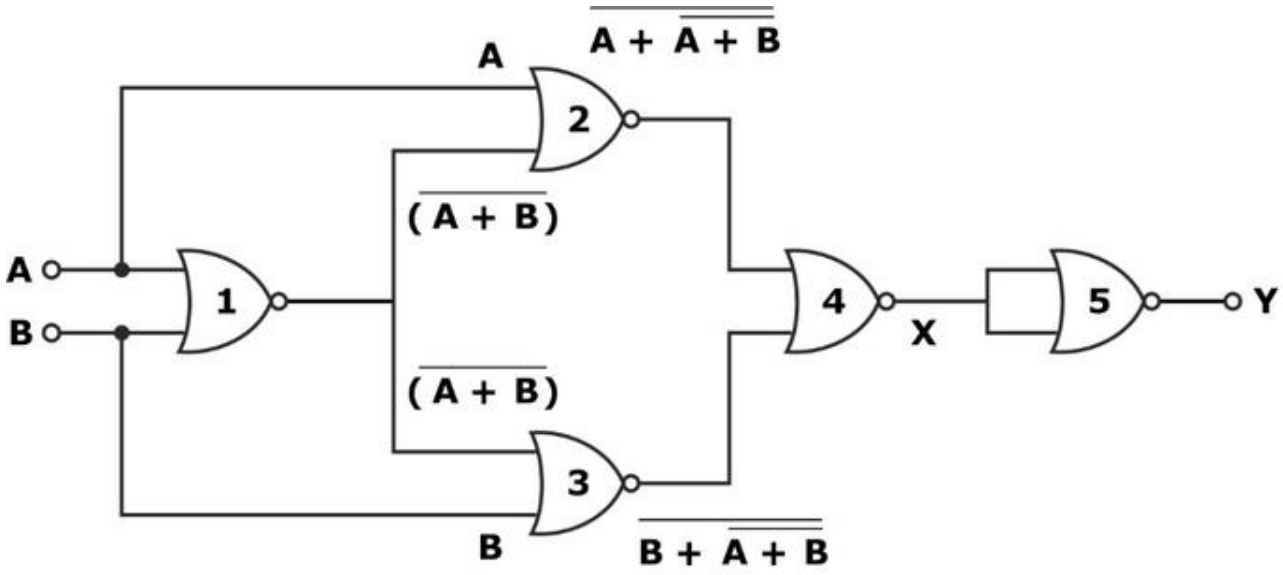


$$Y = \overline{\overline{(A \cdot \overline{AB})}\ \overline{(B \cdot \overline{AB})}} = A \cdot \overline{AB} + B \cdot \overline{AB}$$

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

Using NOR Gate

For the EX-OR gate realization, we require 5 NOR gates, as shown in the circuit diagram:



$$X = \overline{\overline{A + \overline{A + B}} + \overline{B + \overline{A + B}}} = \left[A + \overline{(A + B)}\right] \cdot \left[B + \overline{(A + B)}\right]$$

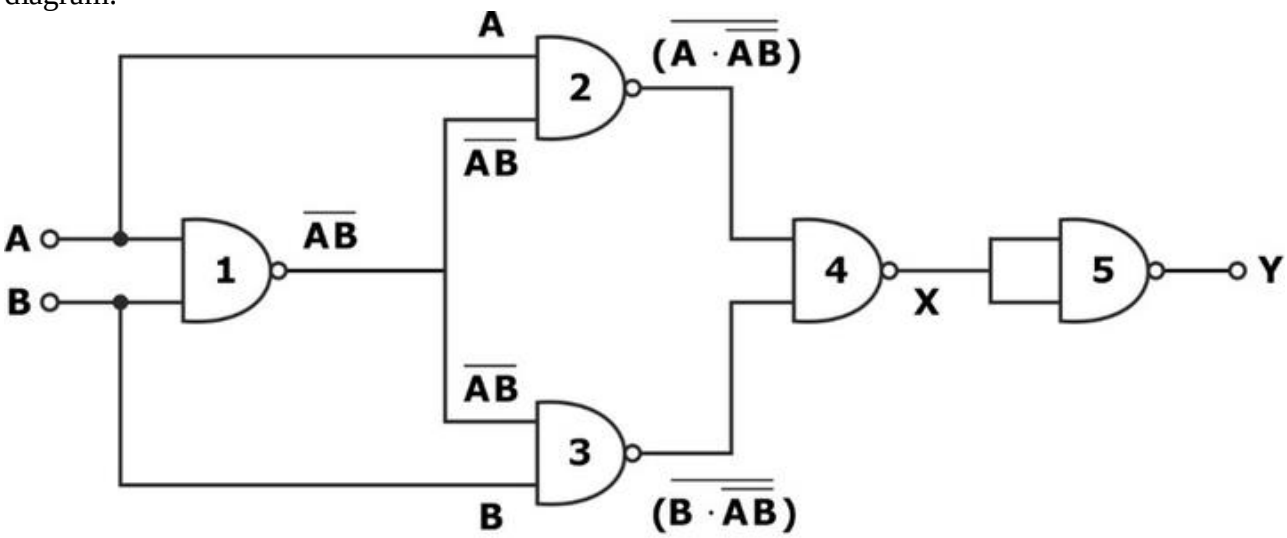
$$= \left[A + (\overline{A}\,\overline{B})\right]\left[B + \overline{A}\,\overline{B}\right] = AB + \overline{A}\,\overline{B}$$

$$Y = \overline{X} = \overline{(AB + \overline{A}\,\overline{B})} = A\overline{B} + \overline{A}B$$

EX-NOR Gate Realization

Using NAND Gate

For the EX-NOR gate realization, we require 5 NAND gates, as shown in the circuit diagram:



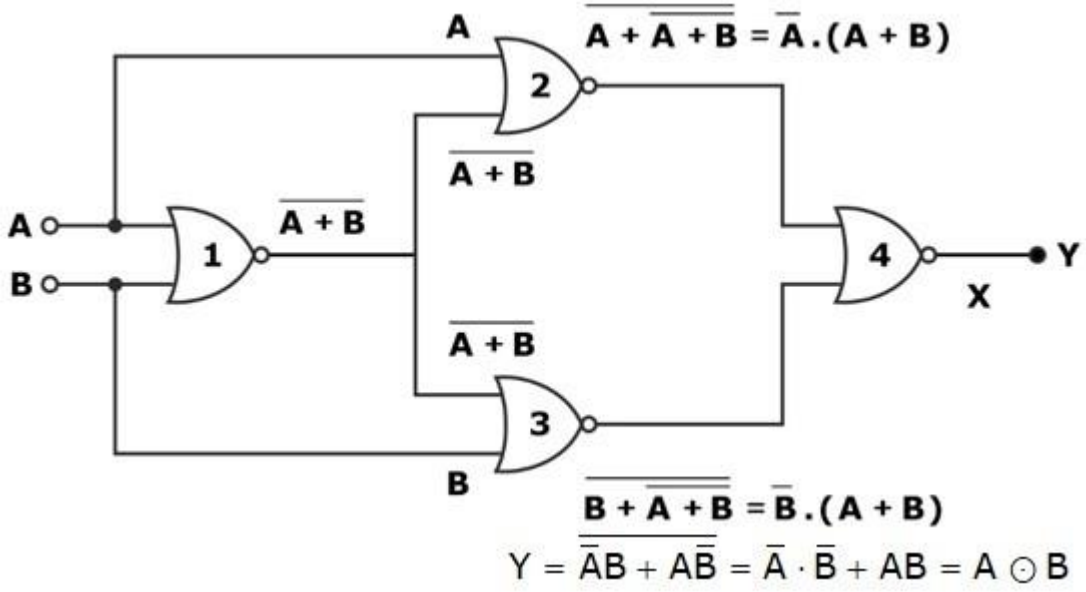
$$X = \overline{\overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}} = (A \cdot \overline{AB} + B \cdot \overline{AB})$$

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

$$Y = \overline{X} = \overline{A}\,\overline{B} + AB = A \odot B$$

Using NOR Gate

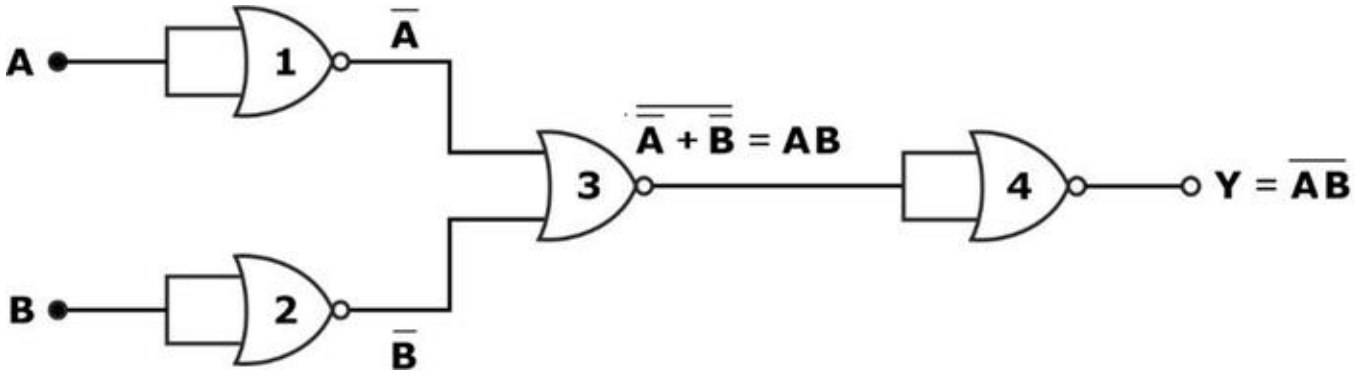
For the EX-NOR gate realization, we require 4 NOR gates, as shown in the circuit diagram:



$$Y = \overline{\overline{A}B + A\overline{B}} = \overline{A} \cdot \overline{B} + AB = A \odot B$$

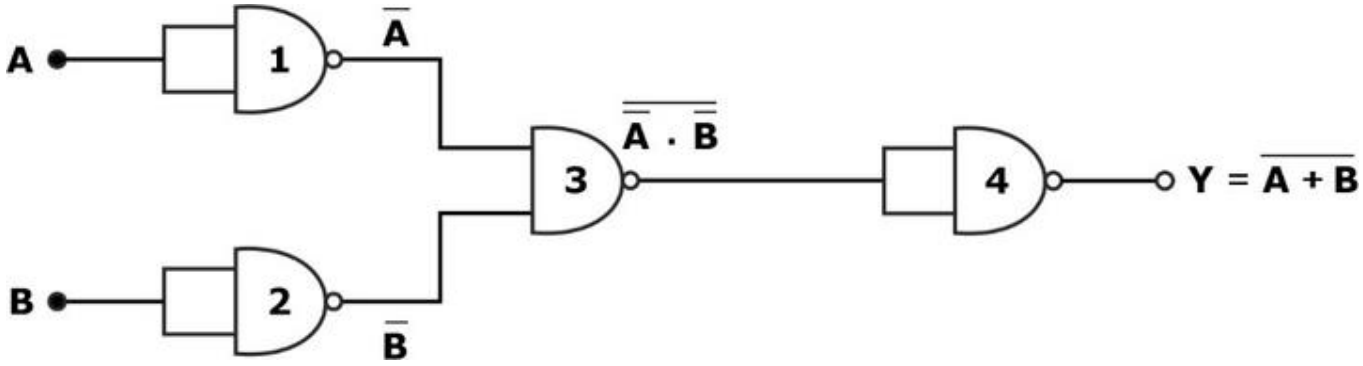
NOR Gate Realization Using NAND Gate

For the NOR gate realization using the NAND gate, we require 4 NAND gates, as shown in the circuit diagram:



NAND Gate Realization Using NOR Gate

For the NAND gate realization using the NOR gate, we require 4 NAND gates, as shown in the circuit diagram:



Properties of XOR Gate

Understanding the properties of XOR Gate is essential for understanding the working of XOR gates in various applications, including encryption, error detection, and arithmetic operations.

- Commutative Property: The XOR gate follows the commutative property, which means that the order of the inputs does not affect the output. In other words, K XOR L produces the same result as L XOR K.

Example: K XOR L = L XOR K

- Associative Property: The XOR gate also adheres to the associative property, implying that the grouping of inputs does not affect the final output. Thus, (K XOR L) XOR M is equivalent to K XOR (L XOR M).

Example: (K XOR L) XOR M = K XOR (L XOR M)

- Self-Inverse Property: When both inputs of the XOR gate are the same (either both 0 or both 1), the output is always 0. Conversely, when the inputs are different, the output is always 1. This self-inverse property makes the XOR gate its own complement.

Example: K XOR K = 0

- Exclusive Operation: The XOR gate performs an exclusive operation, producing a TRUE output (1) only when the inputs differ. If both inputs are the same, the output is FALSE (0).

Example: 0 XOR 0 = 0, 1 XOR 0 = 1, 0 XOR 1 = 1, 1 XOR 1 = 0

- Bit Flipping: The XOR gate is often used for bit flipping. XORing a bit with 1 toggles its value, while XORing it with 0 keeps the value unchanged.

Example: 1 XOR 1 = 0 (flips the bit), 0 XOR 1 = 1 (flips the bit), 1 XOR 0 = 1 (keeps the bit unchanged), 0 XOR 0 = 0 (keeps the bit unchanged)

- No Dependency on Input Order: The XOR gate does not depend on the order in which the inputs are given. The output remains the same regardless of whether A is the first input and B is the second input or vice versa.

Example: K XOR L produces the same output as L XOR K

- Non-Associative Property with More Than Two Inputs: While the XOR gate follows the associative property with two inputs, it does not hold the associative property when extended to more than two inputs. The grouping of inputs can affect the final output.

Example: (K XOR L) XOR M is not necessarily equal to K XOR (L XOR M)

The XOR gate, with its distinct behavior and logical operations, is a vital component in digital logic. Its ability to produce a HIGH output only when the inputs differ makes it indispensable in various applications, including data encryption, error detection, and arithmetic operations. Understanding the principles and applications of the XOR gate is essential for anyone venturing into the fascinating world of digital electronics and computer science. As technology continues to evolve, the XOR gate remains an integral part of our increasingly interconnected world.

UNIT-II

Minimization Techniques

In previous chapters, we have simplified the Boolean functions using Boolean postulates and theorems. It is a time consuming process and we have to re-write the simplified expressions after each step.

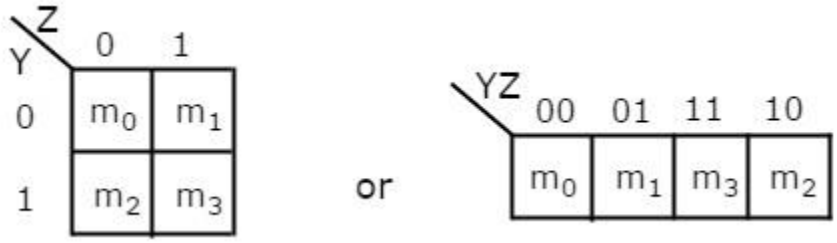
To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 5 Variables

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

2 Variable K-Map

The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms 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, since the number of variables is three. The following figure shows **3 variable K-Map**.

X \ YZ	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are {(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6) and (m_2, m_0, m_6, m_4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_1, m_3), (m_3, m_2), (m_2, m_0), (m_4, m_5), (m_5, m_7), (m_7, m_6), (m_6, m_4), (m_0, m_4), (m_1, m_5), (m_3, m_7) and (m_2, m_6)}.
- If x=0, then 3 variable K-map becomes 2 variable K-map.

4 Variable K-Map

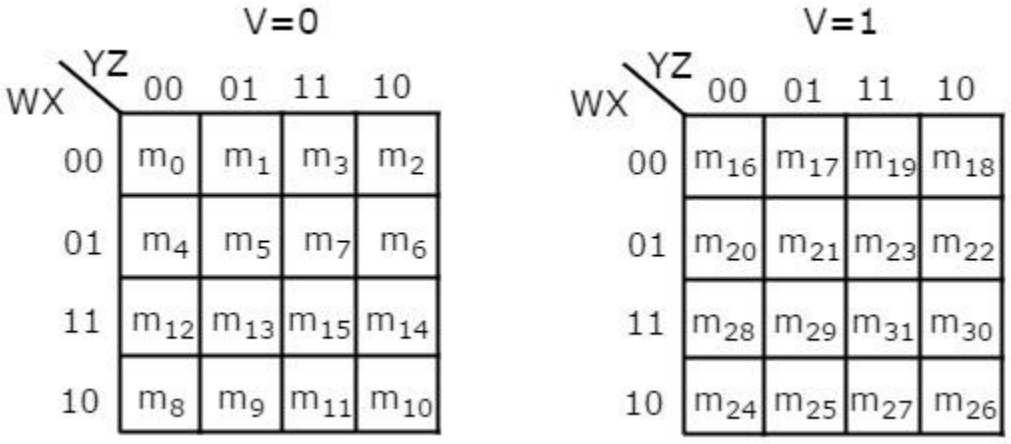
The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.

WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4 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 {(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)}.
- If w=0, then 4 variable K-map becomes 3 variable K-map.

5 Variable K-Map

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.



V=0

WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

V=1

WX \ YZ	00	01	11	10
00	m_{16}	m_{17}	m_{19}	m_{18}
01	m_{20}	m_{21}	m_{23}	m_{22}
11	m_{28}	m_{29}	m_{31}	m_{30}
10	m_{24}	m_{25}	m_{27}	m_{26}

- There is only one possibility of grouping 32 adjacent min terms.
- There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m_0 to m_{15} and m_{16} to m_{31}.
- If v=0, then 5 variable K-map becomes 4 variable K-map.

In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

Minimization of Boolean Functions using K-Maps

If we consider the combination of inputs for which the Boolean function is '1', then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is '0', then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.

- Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '1' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

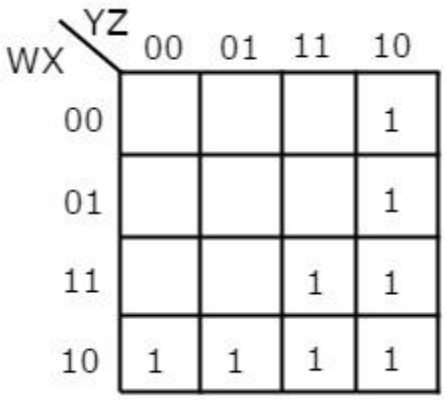
Note 1 − If outputs are not defined for some combination of inputs, then those output values will be represented with **don't care symbol 'x'**. That means, we can consider them as either '0' or '1'.

Note 2 − If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don't care value as '1'.

Example

Let us **simplify** the following Boolean function, **f(W, X, Y, Z)= WX'Y' + WY + W'YZ'** using K-map.

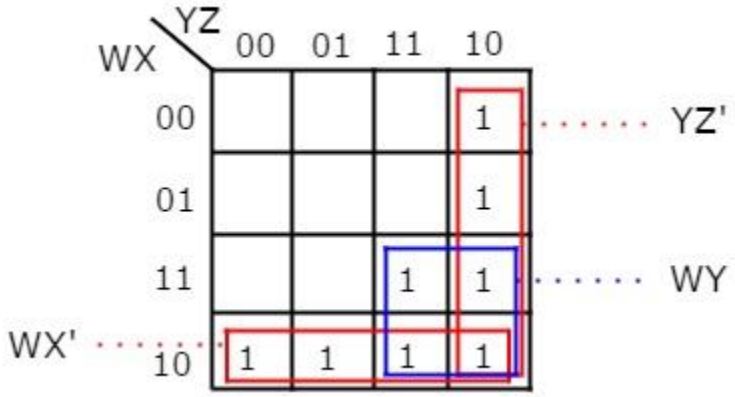
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.



Here, 1s are placed in the following cells of K-map.

- The cells, which are common to the intersection of Row 4 and columns 1 & 2 are corresponding to the product term, **WX'Y'**.
- The cells, which are common to the intersection of Rows 3 & 4 and columns 3 & 4 are corresponding to the product term, **WY**.
- The cells, which are common to the intersection of Rows 1 & 2 and column 4 are corresponding to the product term, **W'YZ'**.

There are no possibilities of grouping either 16 adjacent ones or 8 adjacent ones. There are three possibilities of grouping 4 adjacent ones. After these three groupings, there is no single one left as ungrouped. So, we no need to check for grouping of 2 adjacent ones. The **4 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants WX', WY & YZ'. All these prime implicants are **essential** because of following reasons.

- Two ones **(m_8 & m_9)** of fourth row grouping are not covered by any other groupings. Only fourth row grouping covers those two ones.
- Single one **(m_{15})** of square shape grouping is not covered by any other groupings. Only the square shape grouping covers that one.
- Two ones **(m_2 & m_6)** of fourth column grouping are not covered by any other groupings. Only fourth column grouping covers those two ones.

Therefore, the **simplified Boolean function** is

f = WX' + WY + YZ'

Follow these **rules for simplifying K-maps** in order to get standard product of sums form.

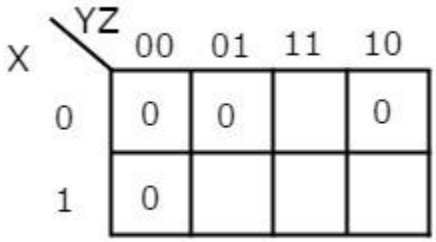
- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as product of Max terms form, then place the zeroes at respective Max term cells in the K-map. If the Boolean function is given as product of sums form, then place the zeroes in all possible cells of K-map for which the given sum terms are valid.
- Check for the possibilities of grouping maximum number of adjacent zeroes. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.

- Each grouping will give either a literal or one sum term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '0' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

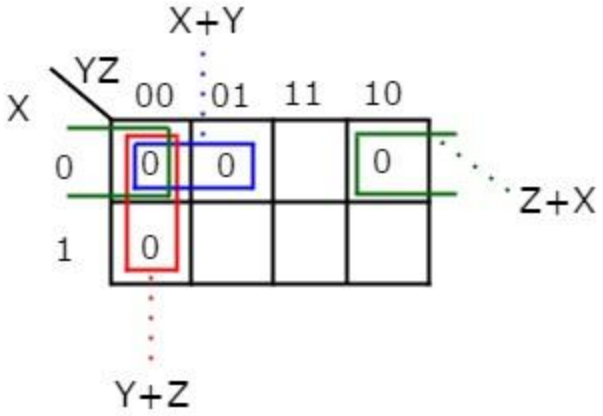
Note − If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent zeroes. In those cases, treat the don't care value as '0'.

Example

Let us **simplify** the following Boolean function, f(X,Y,Z)=∏M(0,1,2,4)f(X,Y,Z)=∏M(0,1,2,4) using K-map. The given Boolean function is in product of Max terms form. It is having 3 variables X, Y & Z. So, we require 3 variable K-map. The given Max terms are M_0, M_1, M_2 & M_4. The 3 **variable K-map** with zeroes corresponding to the given Max terms is shown in the following figure.



There are no possibilities of grouping either 8 adjacent zeroes or 4 adjacent zeroes. There are three possibilities of grouping 2 adjacent zeroes. After these three groupings, there is no single zero left as ungrouped. The **3 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants X + Y, Y + Z & Z + X. All these prime implicants are **essential** because one zero in each grouping is not covered by any other groupings except with their individual groupings.

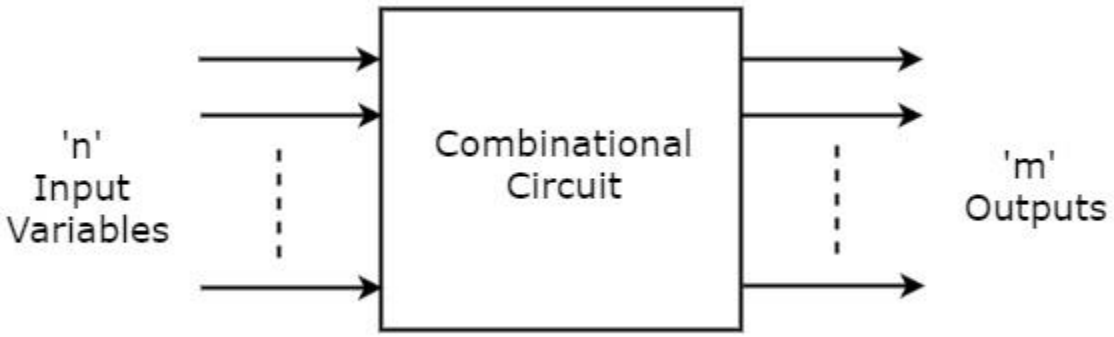
Therefore, the **simplified Boolean function** is

$$f = (X + Y).(Y + Z).(Z + X)$$

In this way, we can easily simplify the Boolean functions up to 5 variables using K-map method. For more than 5 variables, it is difficult to simplify the functions using K-Maps. Because, the number of **cells** in K-map gets **doubled** by including a new variable.

Due to this checking and grouping of adjacent ones (min terms) or adjacent zeros (Max terms) will be complicated. We will discuss **Tabular method** in next chapter to overcome the difficulties of K-map method.

Combinational circuits consist of Logic gates. These circuits operate with binary values. The output(s) of combinational circuit depends on the combination of present inputs. The following figure shows the **block diagram** of combinational circuit.



This combinational circuit has 'n' input variables and 'm' outputs. Each combination of input variables will affect the output(s).

Design procedure of Combinational circuits

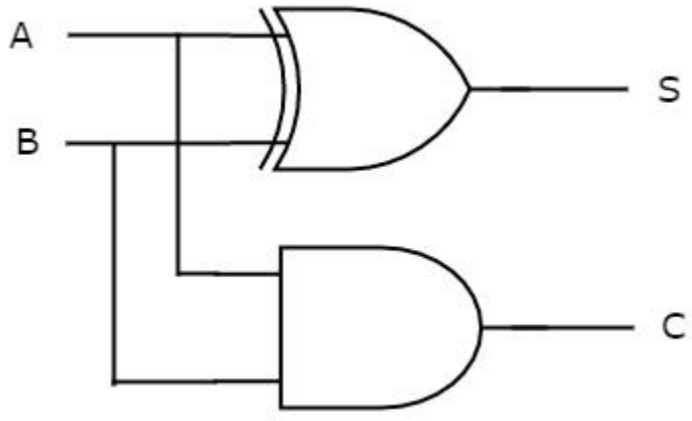
- Find the required number of input variables and outputs from given specifications.
- Formulate the **Truth table**. If there are 'n' input variables, then there will be 2n possible combinations. For each combination of input, find the output values.

- Find the **Boolean expressions** for each output. If necessary, simplify those expressions.
- Implement the above Boolean expressions corresponding to each output by using **Logic gates**.
- In this chapter, let us discuss about the basic arithmetic circuits like Binary adder and Binary subtractor. These circuits can be operated with binary values 0 and 1.
- Binary Adder
- The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.
- Half Adder
- Half adder is a combinational circuit, which performs the addition of two binary numbers A and B are of **single bit**. It produces two outputs sum, S & carry, C.
- The **Truth table** of Half adder is shown below.

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- When we do the addition of two bits, the resultant sum can have the values ranging from 0 to 2 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent decimal digit 2 with single bit in binary. So, we require two bits for representing it in binary.
- Let, sum, S is the Least significant bit and carry, C is the Most significant bit of the resultant sum. For first three combinations of inputs, carry, C is zero and the value of S will be either zero or one based on the **number of ones** present at the inputs. But, for last combination of inputs, carry, C is one and sum, S is zero, since the resultant sum is two.

- From Truth table, we can directly write the **Boolean functions**for each output as
  - S=A⊕BS=A⊕B
- C=ABC=AB
- We can implement the above functions with 2-input Ex-OR gate & 2-input AND gate. The **circuit diagram** of Half adder is shown in the following figure.

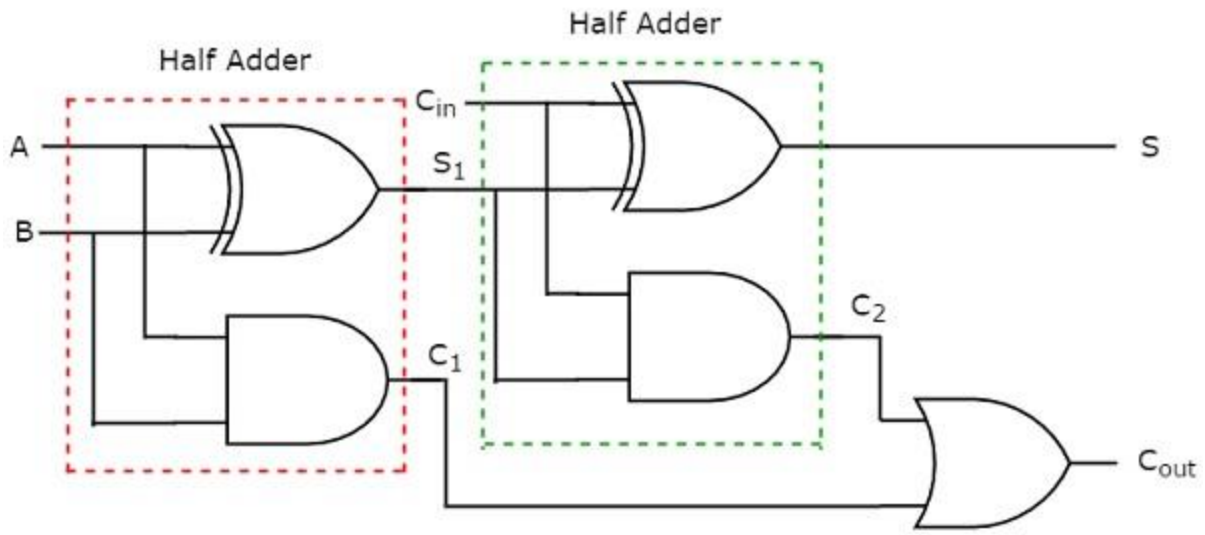


- In the above circuit, a two input Ex-OR gate & two input AND gate produces sum, S & carry, C respectively. Therefore, Half-adder performs the addition of two bits.
- Full Adder
- Full adder is a combinational circuit, which performs the **addition of three bits** A, B and C_{in}. Where, A & B are the two parallel significant bits and C_{in} is the carry bit, which is generated from previous stage. This Full adder also produces two outputs sum, S & carry, C_{out}, which are similar to Half adder.
- The **Truth table** of Full adder is shown below.

Inputs			Outputs	
A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1

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

- When we do the addition of three bits, the resultant sum can have the values ranging from 0 to 3 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent the decimal digits 2 and 3 with single bit in binary. So, we require two bits for representing those two decimal digits in binary.
- Let, sum, S is the Least significant bit and carry, C_{out} is the Most significant bit of resultant sum. It is easy to fill the values of outputs for all combinations of inputs in the truth table. Just count the **number of ones** present at the inputs and write the equivalent binary number at outputs. If C_{in} is equal to zero, then Full adder truth table is same as that of Half adder truth table.
- We will get the following **Boolean functions** for each output after simplification.
    - S=A⊕B⊕CinS=A⊕B⊕Cin
- cout=AB+(A⊕B)cincout=AB+(A⊕B)cin
- The sum, S is equal to one, when odd number of ones present at the inputs. We know that Ex-OR gate produces an output, which is an odd function. So, we can use either two 2input Ex-OR gates or one 3-input Ex-OR gate in order to produce sum, S. We can implement carry, C_{out} using two 2-input AND gates & one OR gate. The **circuit diagram** of Full adder is shown in the following figure.

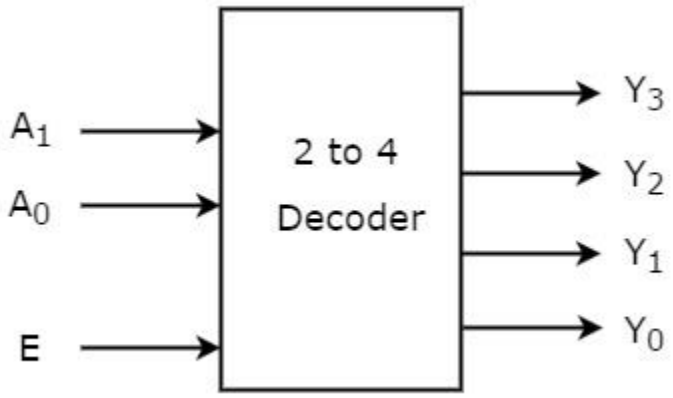


- This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If C_{in} is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.

Decoder is a combinational circuit that has 'n' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min terms**of 'n' input variables (lines), when it is enabled.

2 to 4 Decoder

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



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

Enable	Inputs		Outputs			
E	**A_1**	**A_0**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

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

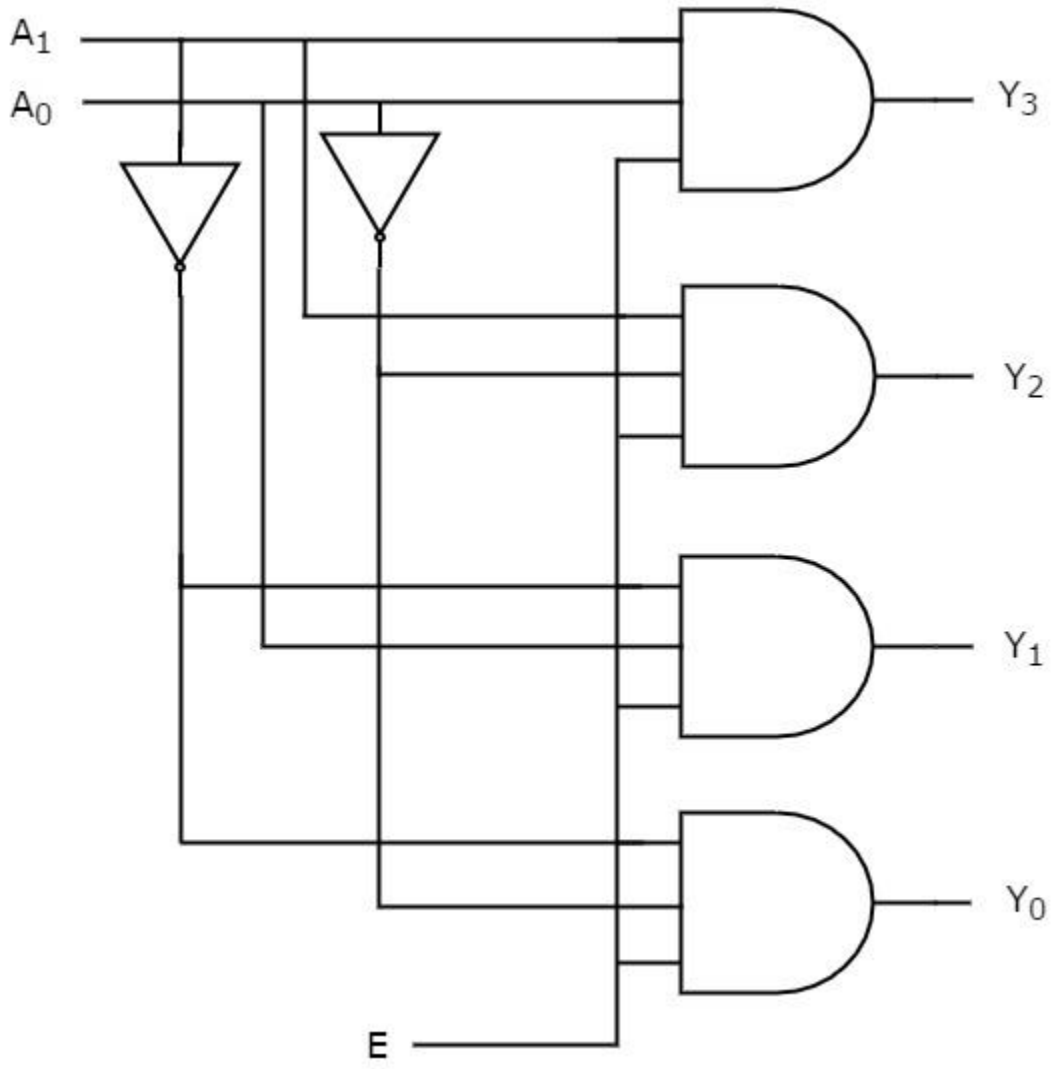
Y3=E.A1.A0Y3=E.A1.A0

Y2=E.A1.A0'Y2=E.A1.A0'

Y1=E.A1'.A0Y1=E.A1'.A0

Y0=E.A1'.A0'Y0=E.A1'.A0'

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each & two inverters. The **circuit diagram** of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A_1 & A_0, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A_2, A_1 & A_0 and 4 to 16 decoder produces sixteen min terms of four input variables A_3, A_2, A_1 & A_0.

Implementation of Higher-order Decoders

Now, let us implement the following two higher-order decoders using lower-order decoders.

- 3 to 8 decoder
- 4 to 16 decoder

3 to 8 Decoder

In this section, let us implement **3 to 8 decoder using 2 to 4 decoders**. We know that 2 to 4 Decoder has two inputs, A_1 & $A_0$and four outputs, Y_3 to Y_0. Whereas, 3 to 8 Decoder has three inputs A_2, A_1 & A_0 and eight outputs, Y_7 to Y_0.

We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

Requirednumberoflowerorderdecoders=m2m1Requirednumberoflowerorderdecoders=m2m1

Where,

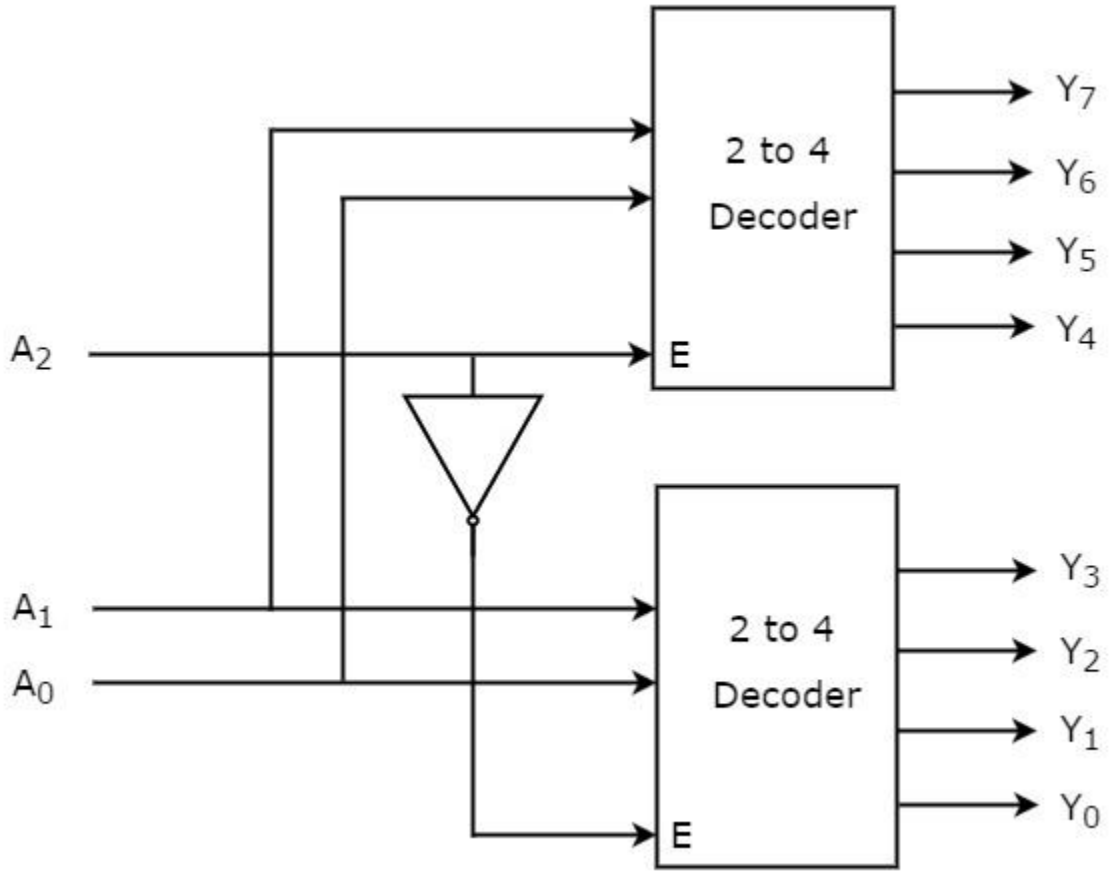
m1m1 is the number of outputs of lower order decoder.

m2m2 is the number of outputs of higher order decoder.

Here, m1m1 = 4 and m2m2 = 8. Substitute, these two values in the above formula.

Requirednumberof2to4decoders=84=2Requirednumberof2to4decoders=84=2

Therefore, we require two 2 to 4 decoders for implementing one 3 to 8 decoder. The **block diagram** of 3 to 8 decoder using 2 to 4 decoders is shown in the following figure.



The parallel inputs A_1 & A_0 are applied to each 2 to 4 decoder. The complement of input A_2 is connected to Enable, E of lower 2 to 4 decoder in order to get the outputs, Y_3 to Y_0. These are the **lower four min terms**. The input, A_2 is directly connected to Enable, E of upper 2 to 4 decoder in order to get the outputs, $Y_7$to Y_4. These are the **higher four min terms**.

4 to 16 Decoder

In this section, let us implement **4 to 16 decoder using 3 to 8 decoders**. We know that 3 to 8 Decoder has three inputs A_2, A_1& A_0 and eight outputs, Y_7 to Y_0. Whereas, 4 to 16 Decoder has four inputs A_3, A_2, A_1 & A_0 and sixteen outputs, Y_{15} to Y_0

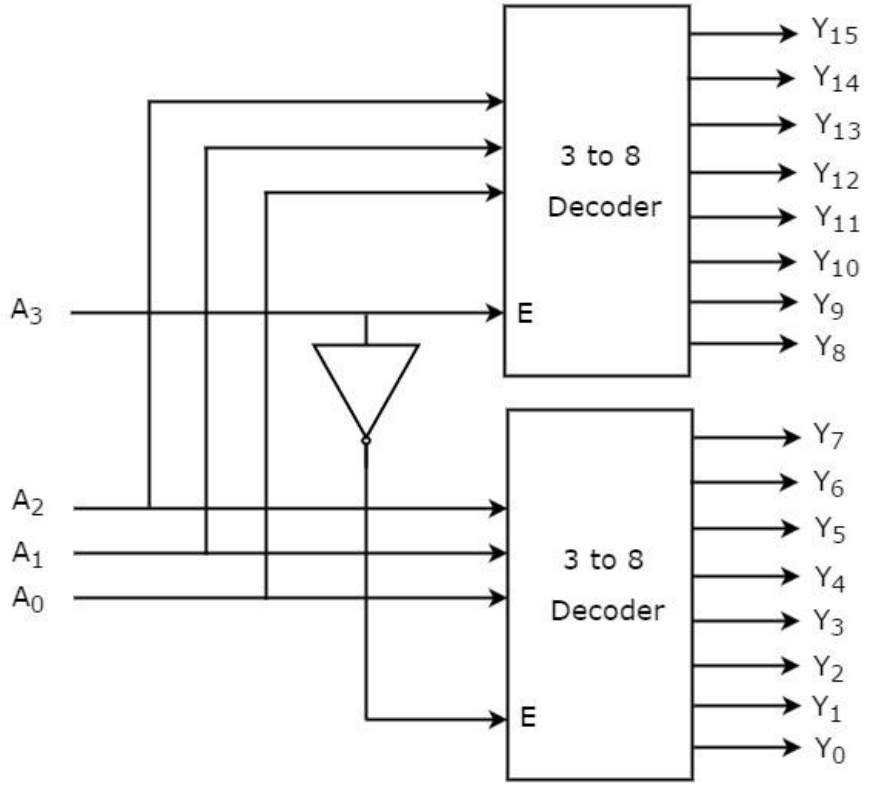
We know the following formula for finding the number of lower order decoders required.

Requirednumberoflowerorderdecoders=m2m1Requirednumberoflowerorderdecoders=m2m1

Substitute, m1m1 = 8 and m2m2 = 16 in the above formula.

Requirednumberof3to8decoders=168=2Requirednumberof3to8decoders=168=2

Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The **block diagram** of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.



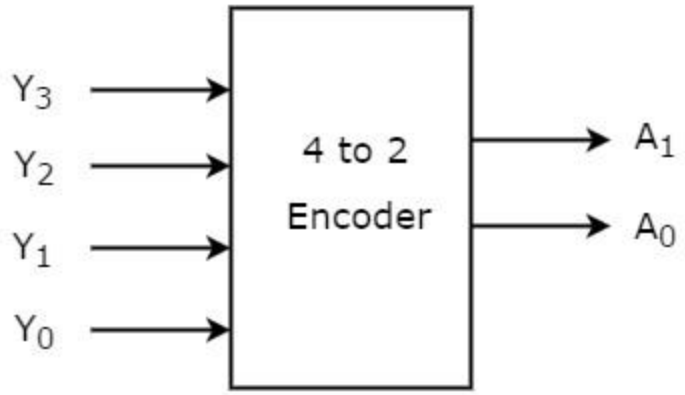
The parallel inputs A_2, A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0. These are

the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y_{15} to Y_8. These are the **higher eight min terms**.

An **Encoder** is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^ninput lines with 'n' bits. It is optional to represent the enable signal in encoders.

4 to 2 Encoder

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



At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The **Truth table** of 4 to 2 encoder is shown below.

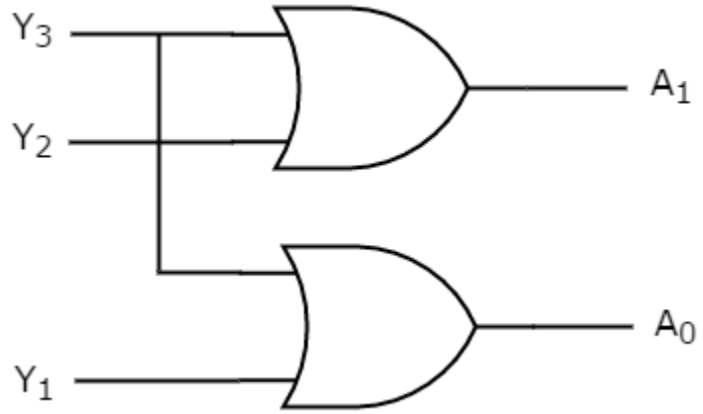
Inputs				**Outputs**	
Y_3	**Y_2**	**Y_1**	**Y_0**	**A_1**	**A_0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

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

$$A1=Y3+Y2A1=Y3+Y2$$

$$A0=Y3+Y1A0=Y3+Y1$$

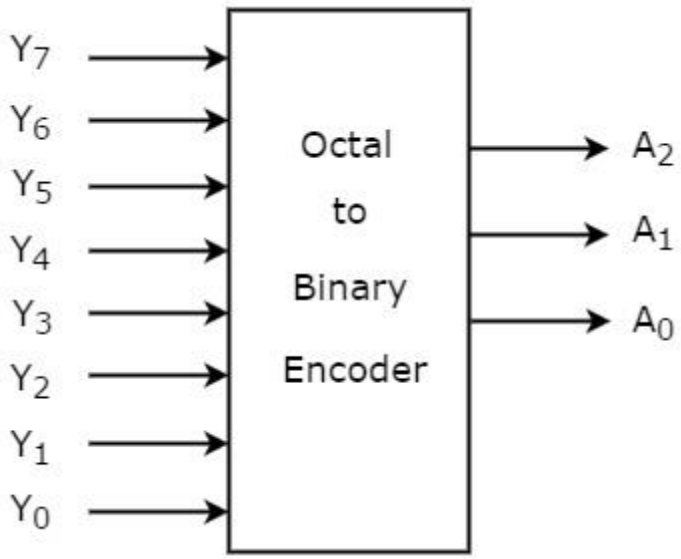
We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



The above circuit diagram contains two OR gates. These OR gates encode the four inputs with two bits

Octal to Binary Encoder

Octal to binary Encoder has eight inputs, Y_7 to Y_0 and three outputs A_2, A_1 & A_0. Octal to binary encoder is nothing but 8 to 3 encoder. The **block diagram** of octal to binary Encoder is shown in the following figure.



At any time, only one of these eight inputs can be '1' in order to get the respective binary code. The **Truth table** of octal to binary encoder is shown below.

Inputs								Outputs		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

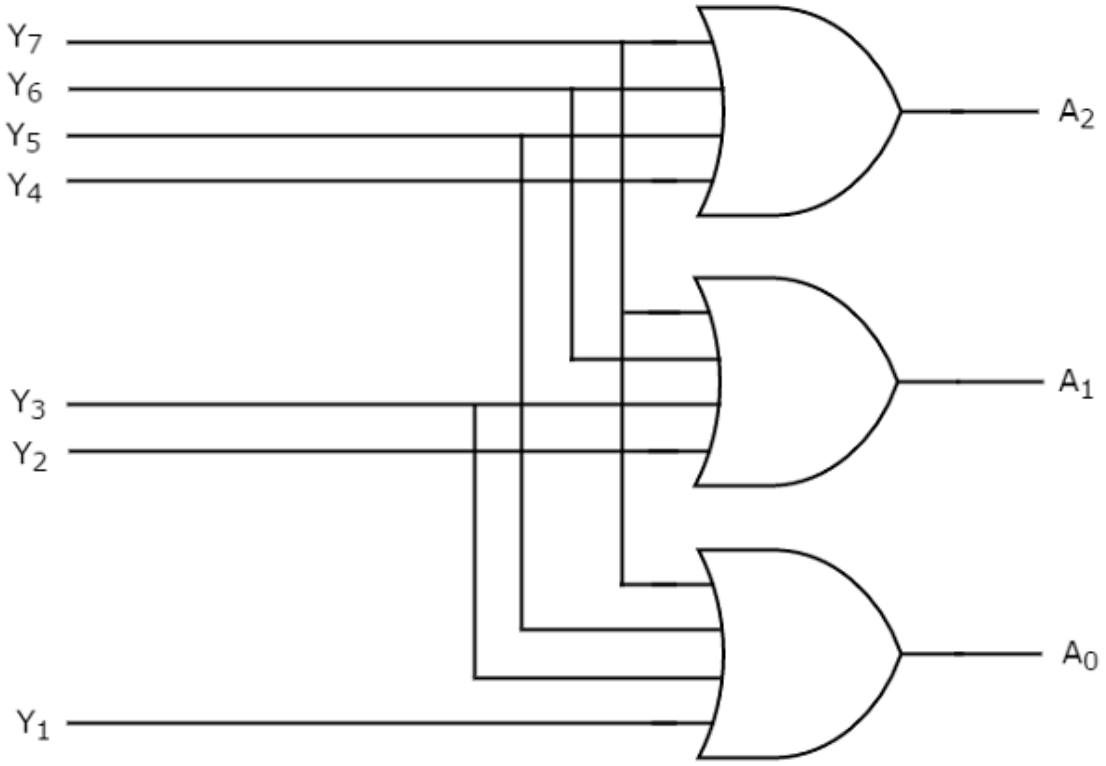
From Truth table, we can write the **Boolean functions** for each output as

$$A2=Y7+Y6+Y5+Y4A2=Y7+Y6+Y5+Y4$$

$$A1=Y7+Y6+Y3+Y2A1=Y7+Y6+Y3+Y2$$

$$A0=Y7+Y5+Y3+Y1A0=Y7+Y5+Y3+Y1$$

We can implement the above Boolean functions by using four input OR gates. The **circuit diagram** of octal to binary encoder is shown in the following figure.



The above circuit diagram contains three 4-input OR gates. These OR gates encode the eight inputs with three bits.

Drawbacks of Encoder

Following are the drawbacks of normal encoder.

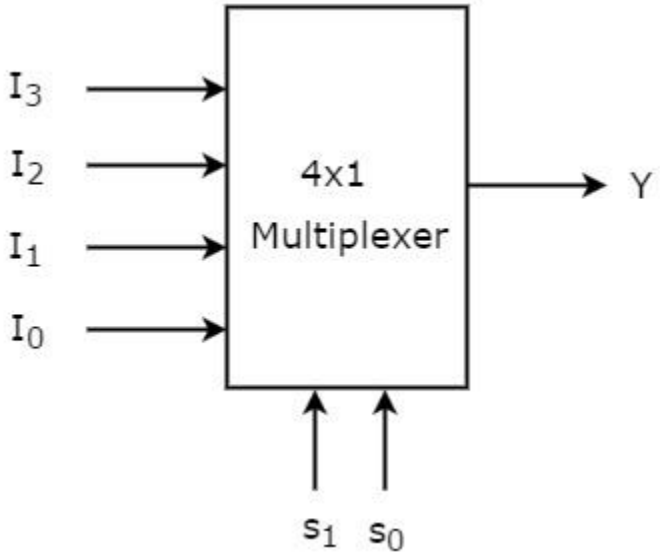
- There is an ambiguity, when all outputs of encoder are equal to zero. Because, it could be the code corresponding to the inputs, when only least significant input is one or when all inputs are zero.
- If more than one input is active High, then the encoder produces an output, which may not be the correct code. For **example**, if both Y_3 and Y_6 are '1', then the encoder produces 111 at the output. This is neither equivalent code corresponding to Y_3, when it is '1' nor the equivalent code corresponding to Y_6, when it is '1'.

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

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

4x1 Multiplexer

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



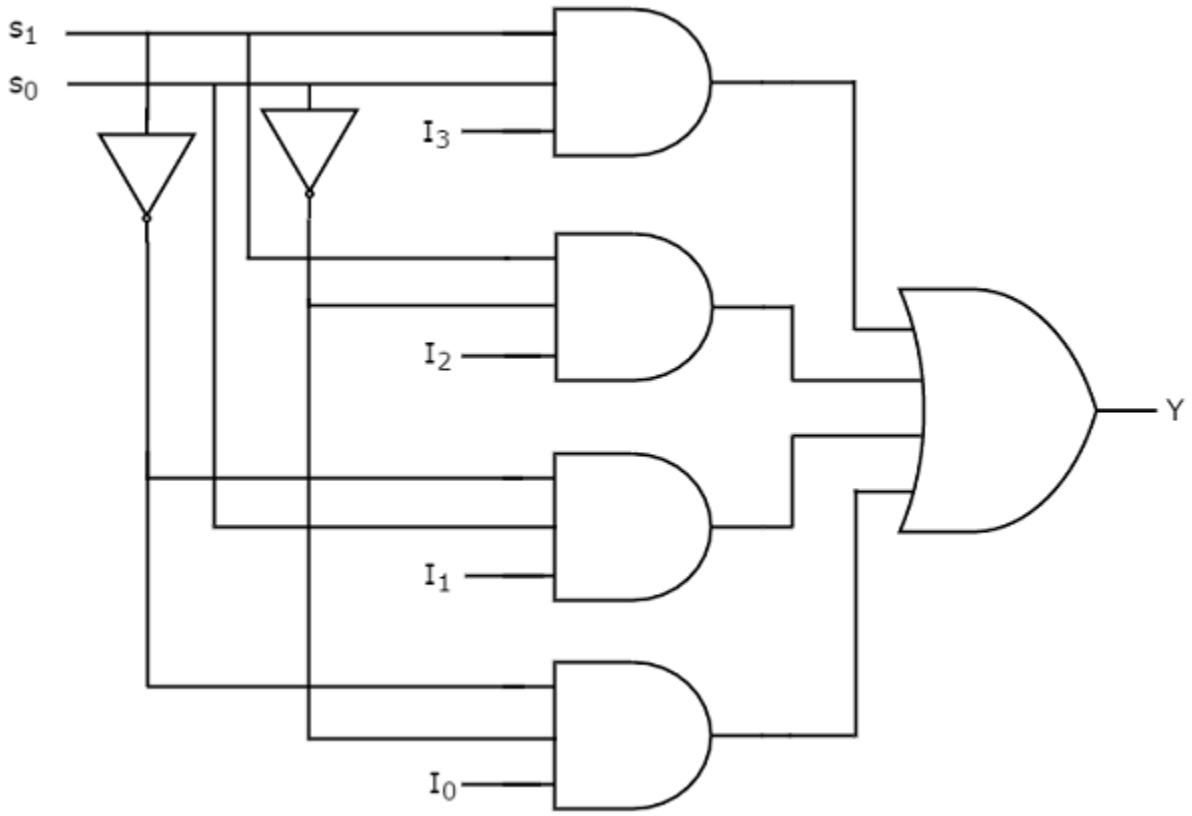
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

Selection Lines		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

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

$$Y=S1'S0'I0+S1'S0I1+S1S0'I2+S1S0I3Y=S1'S0'I0+S1'S0I1+S1S0'I2+S1S0I3$$

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



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer

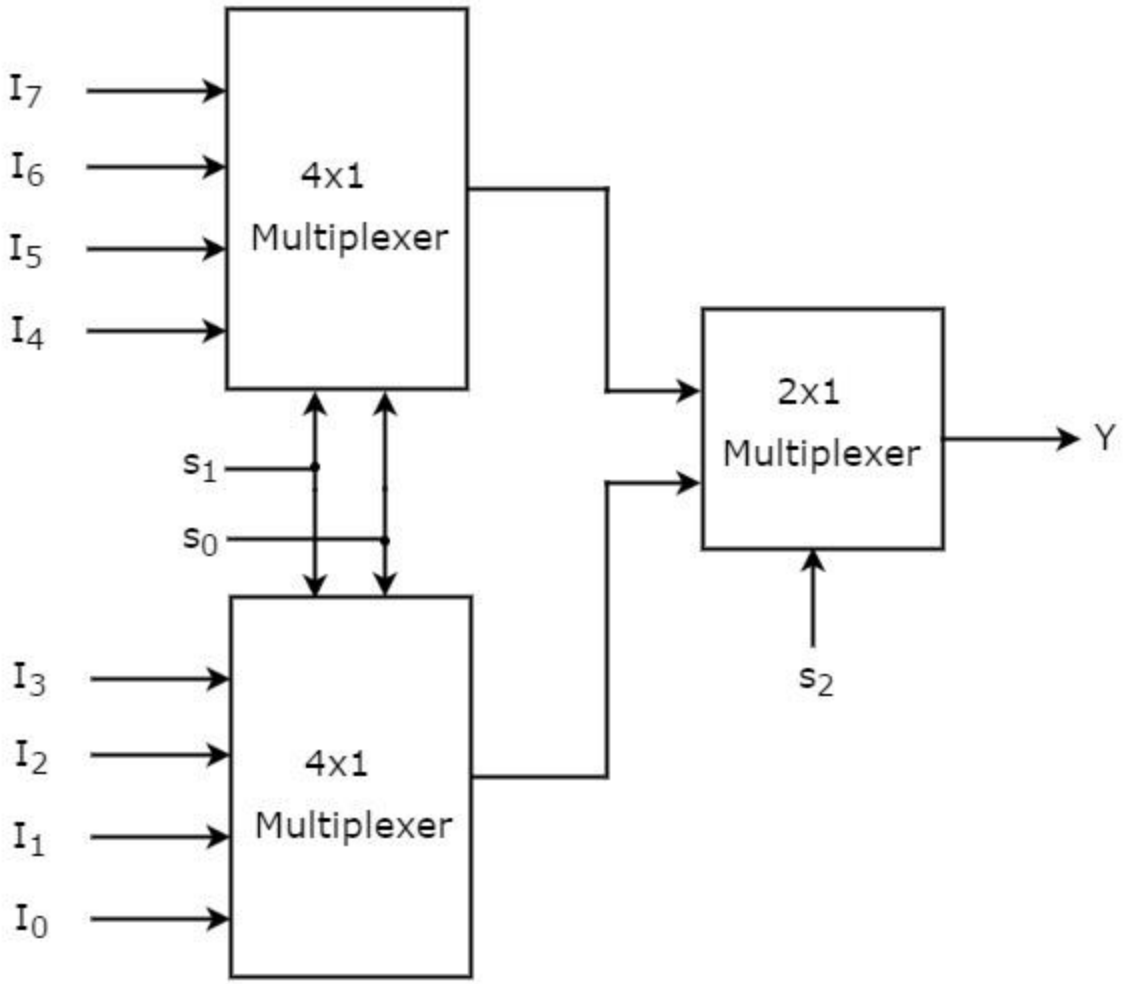
In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I_7 to I_0, three selection lines s_2, s_1 & s0 and one output Y. The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs			Output
S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram**of 8x1 Multiplexer is shown in the following figure.



The same **selection lines, s_1 & s_0** are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I_7 to I_4 and the data inputs of lower 4x1 Multiplexer are I_3 to I_0. Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s_1 & s_0.

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_2** is applied to 2x1 Multiplexer.

- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0.
- If s_2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I_7 to I_4 based on the values of selection lines s_1 & s_0.

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

16x1 Multiplexer

In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

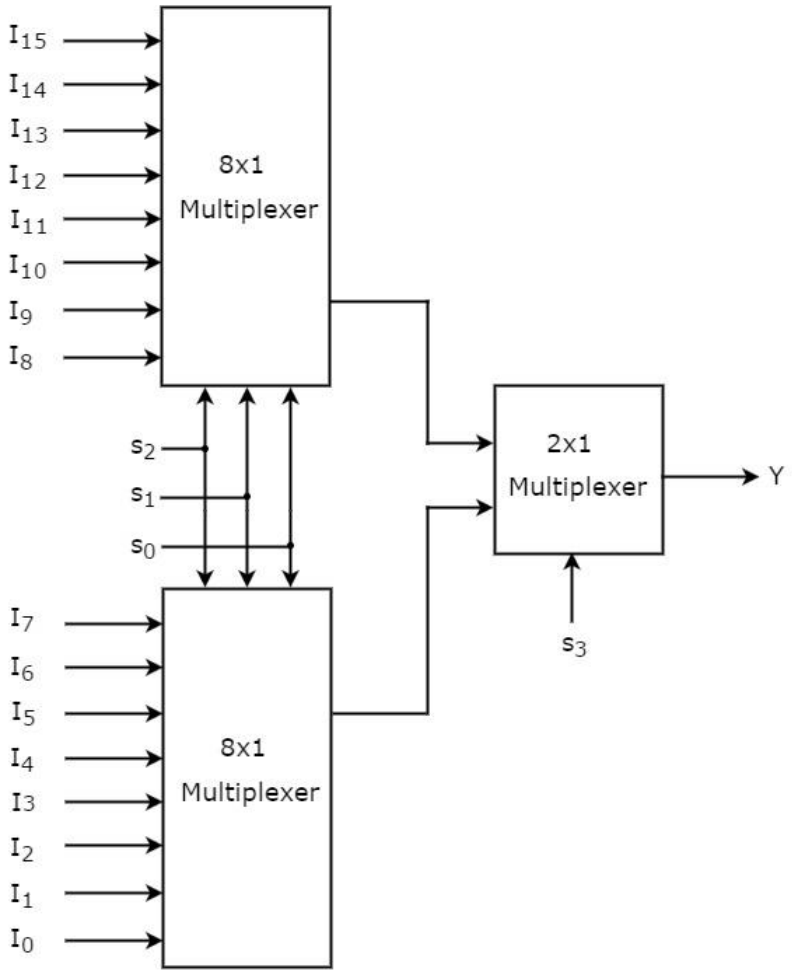
So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I_{15} to I_0, four selection lines s_3 to s_0 and one output Y. The **Truth table** of 16x1 Multiplexer is shown below.

Selection Inputs				**Output**
S_3	**S_2**	**S_1**	**S_0**	**Y**
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9

1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}
1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines, s_2, s_1 & s_0** are applied to both 8x1 Multiplexers. The data inputs of upper 8x1 Multiplexer are I_{15} to I_8 and the data inputs of lower 8x1 Multiplexer are I_7 to I_0. Therefore, each 8x1 Multiplexer produces an output based on the values of selection lines, s_2, s_1 & s_0.

The outputs of first stage 8x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_3** is applied to 2x1 Multiplexer.

- If s_3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs Is_7 to I_0 based on the values of selection lines s_2, s_1 & s_0.
- If s_3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I_{15} to I_8 based on the values of selection lines s_2, s_1 & s_0.

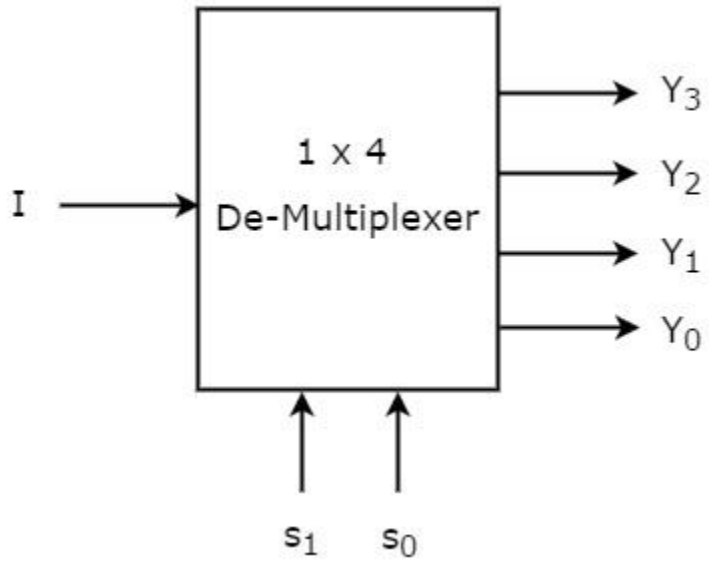
Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.

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

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s_1 & $s_0$and four outputs Y_3, Y_2, Y_1 &Y_0. The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



The single input 'I' will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s0. The **Truth table** of 1x4 De-Multiplexer is shown below.

Selection Inputs		**Outputs**			
S_1	**S_0**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	0	0	0	0	**I**
0	1	0	0	**I**	0
1	0	0	**I**	0	0
1	1	**I**	0	0	0

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

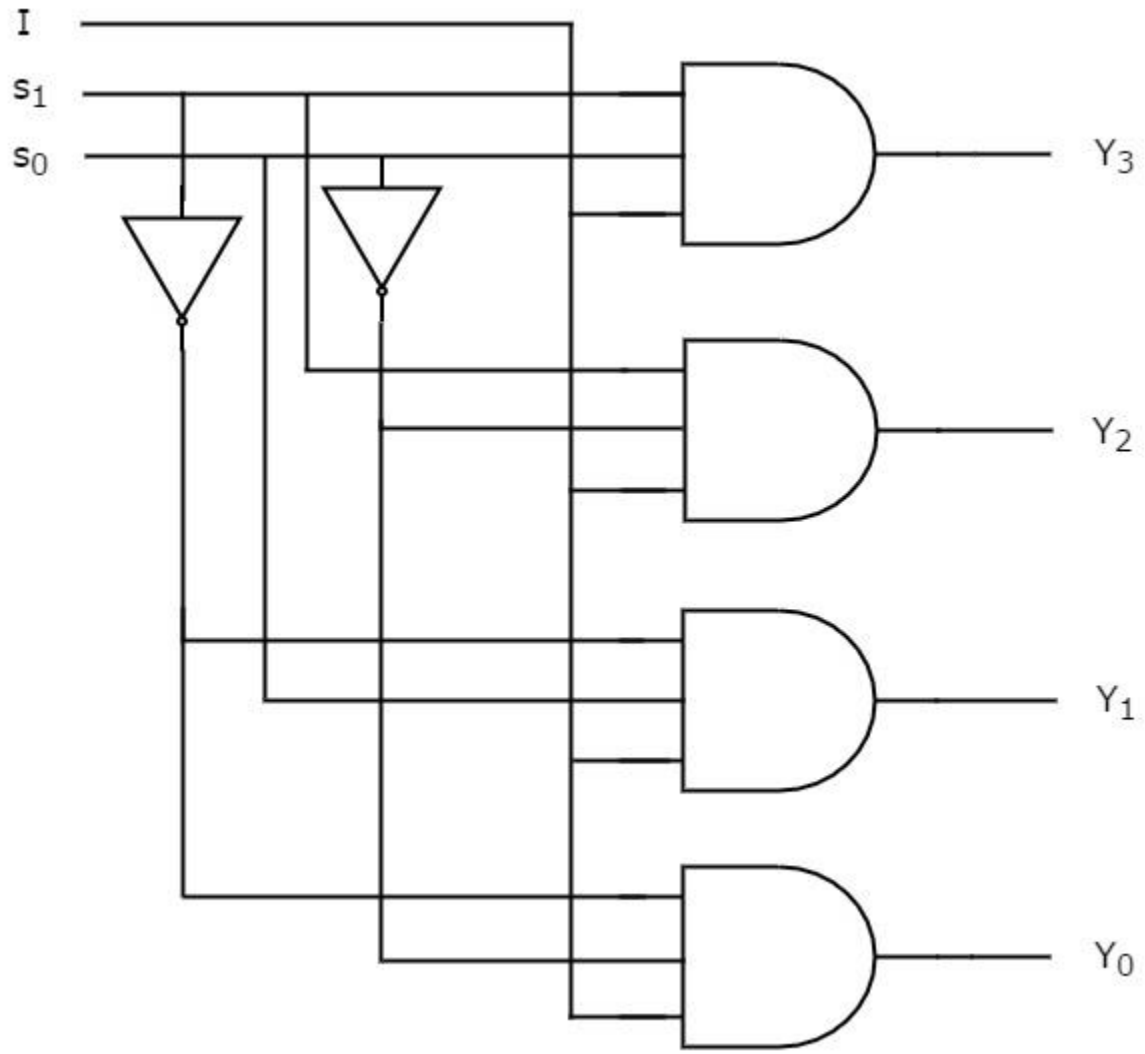
Y3=s1s0IY3=s1s0I

Y2=s1s0'IY2=s1s0'I

Y1=s1's0IY1=s1's0I

Y0=s1's0'IY0=s1's0'I

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



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

Implementation of Higher-order De-Multiplexers

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

1x8 De-Multiplexer

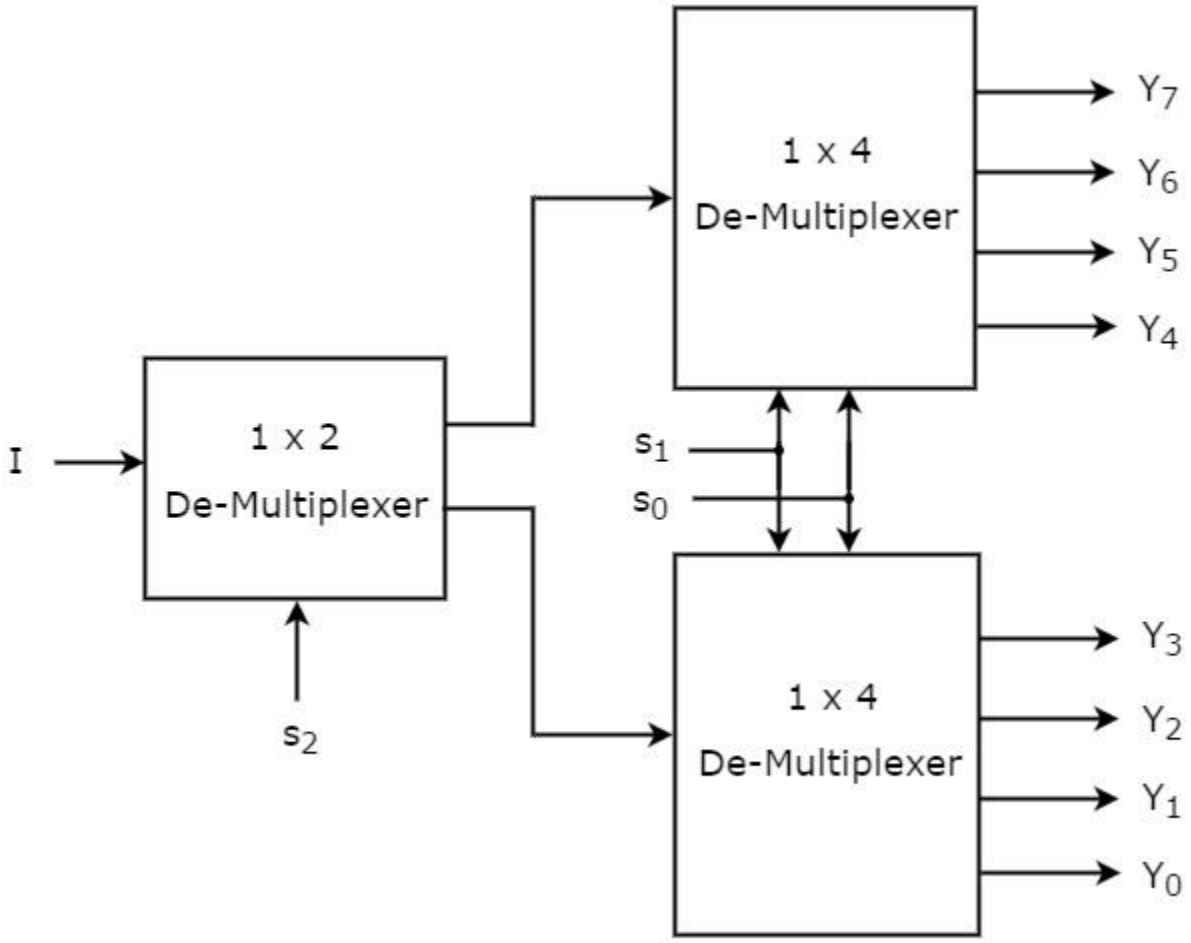
In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input I, three selection lines s_2, s_1 & s_0 and outputs Y_7 to Y_0. The **Truth table** of 1x8 De-Multiplexer is shown below.

Selection Inputs			**Outputs**							
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	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0
1	1	1	**I**	0	0	0	0	0	0	0

We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1x8 De-Multiplexer is shown in the following figure.



The common **selection lines, s_1 & s_0** are applied to both 1x4 De-Multiplexers. The outputs of upper 1x4 De-Multiplexer are $Y_7$to Y_4 and the outputs of lower 1x4 De-Multiplexer are Y_3 to Y_0.

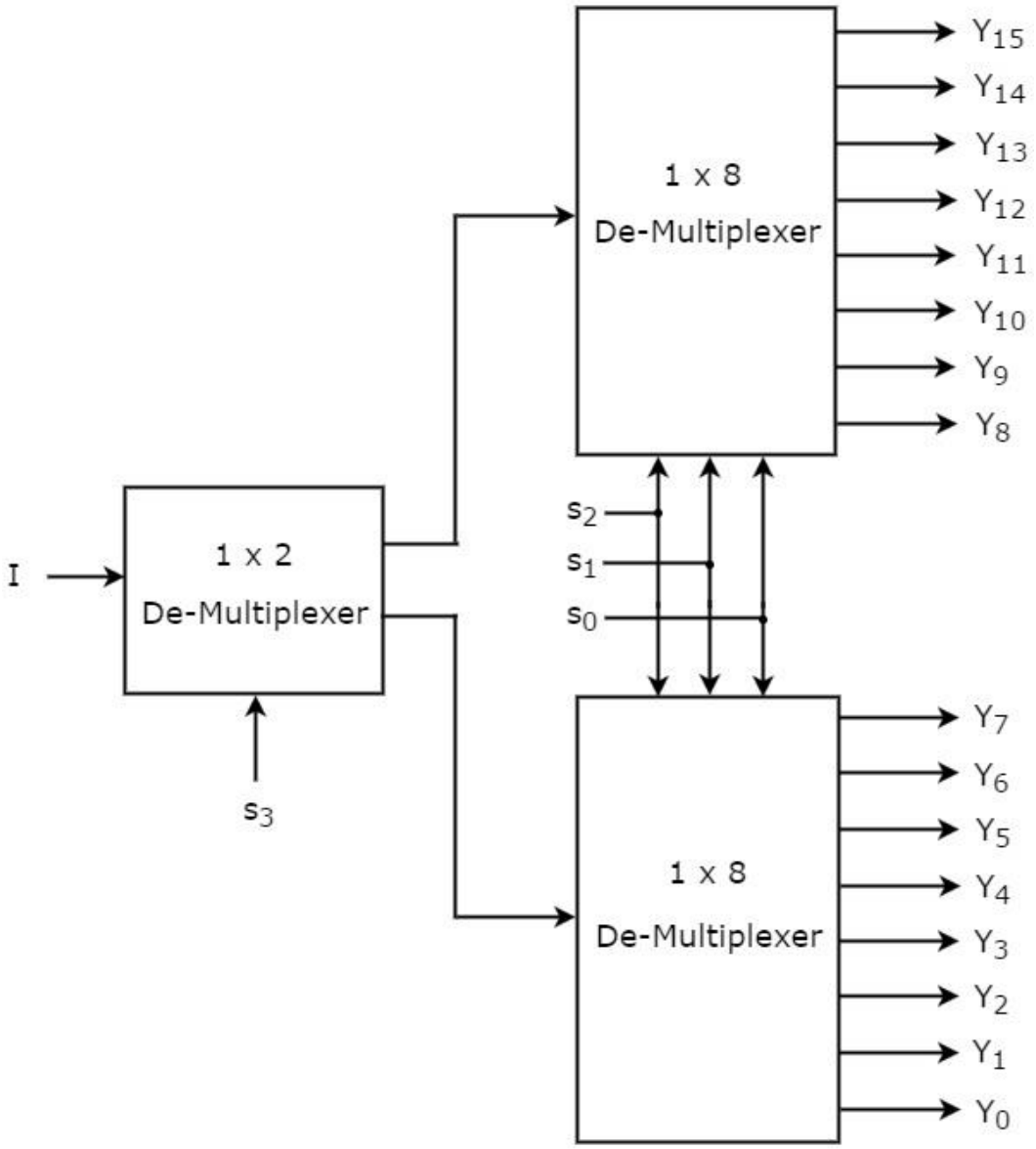
The other **selection line, s_2** is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0. Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will be equal to input, I based on the values of selection lines s_1 & s_0.

1x16 De-Multiplexer

In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1x8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x16 De-Multiplexer.

Let the 1x16 De-Multiplexer has one input I, four selection lines s_3, s_2, s_1 & s_0 and outputs Y_{15} to Y_0. The **block diagram** of 1x16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



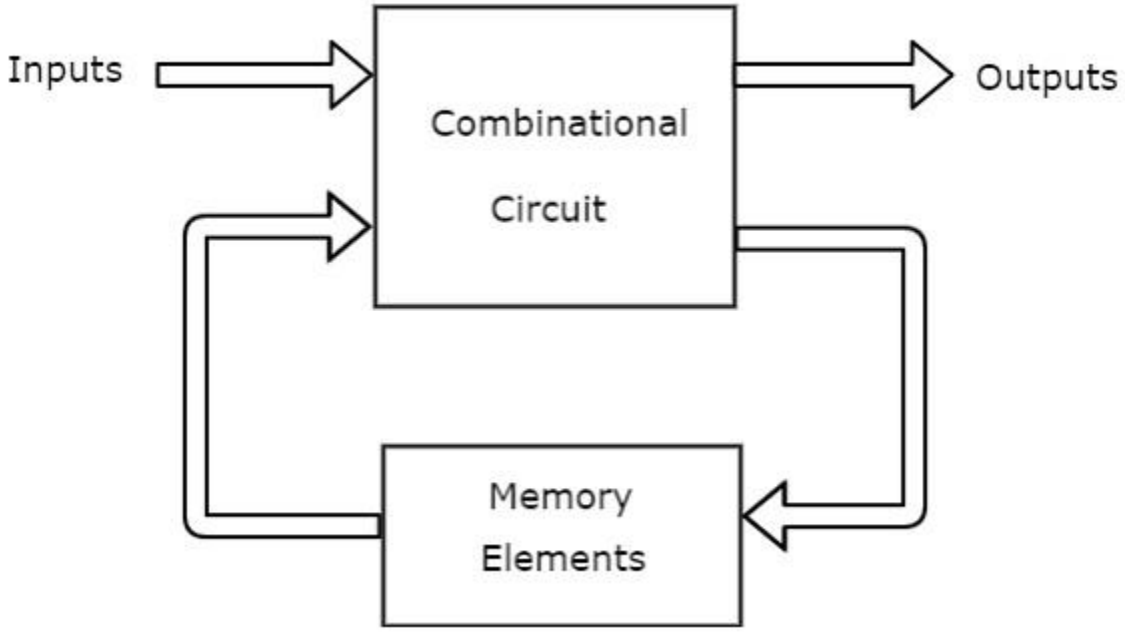
The common **selection lines s_2, s_1 & s_0** are applied to both 1x8 De-Multiplexers. The outputs of upper 1x8 De-Multiplexer are Y_{15} to Y_8 and the outputs of lower 1x8 DeMultiplexer are $Y_7$to Y_0.

The other **selection line, s_3** is applied to 1x2 De-Multiplexer. If s_3 is zero, then one of the eight outputs of lower 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0. Similarly, if s3 is one, then one of the 8 outputs of upper 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0.

UNIT III

Sequential Machines Fundamentals

We discussed various combinational circuits in earlier chapters. All these circuits have a set of output(s), which depends only on the combination of present inputs. The following figure shows the **block diagram** of sequential circuit.



This sequential circuit contains a set of inputs and output(s). The output(s) of sequential circuit depends not only on the combination of present inputs but also on the previous output(s). Previous output is nothing but the **present state**. Therefore, sequential circuits contain combinational circuits along with memory (storage) elements. Some sequential circuits may not contain combinational circuits, but only memory elements.

Following table shows the **differences** between combinational circuits and sequential circuits.

Combinational Circuits	Sequential Circuits
Outputs depend only on present inputs.	Outputs depend on both present inputs and present state.
Feedback path is not present.	Feedback path is present.

Memory elements are not required.	Memory elements are required.
Clock signal is not required.	Clock signal is required.
Easy to design.	Difficult to design.

Types of Sequential Circuits

Following are the two types of sequential circuits −

- Asynchronous sequential circuits
- Synchronous sequential circuits

Asynchronous sequential circuits

If some or all the outputs of a sequential circuit do not change (affect) with respect to active transition of clock signal, then that sequential circuit is called as **Asynchronous sequential circuit**. That means, all the outputs of asynchronous sequential circuits do not change (affect) at the same time. Therefore, most of the outputs of asynchronous sequential circuits are **not in synchronous** with either only positive edges or only negative edges of clock signal.

Synchronous sequential circuits

If all the outputs of a sequential circuit change (affect) with respect to active transition of clock signal, then that sequential circuit is called as **Synchronous sequential circuit**. That means, all the outputs of synchronous sequential circuits change (affect) at the same time. Therefore, the outputs of synchronous sequential circuits are in synchronous with either only positive edges or only negative edges of clock signal.

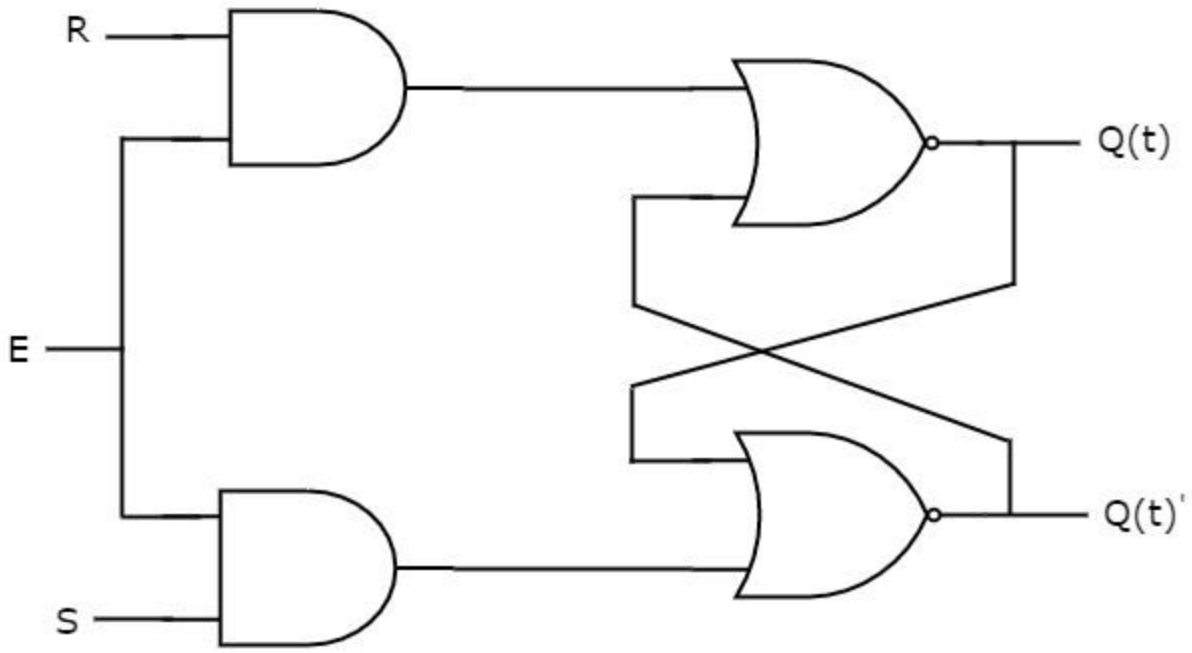
There are two types of memory elements based on the type of triggering that is suitable to operate it.

- Latches
- Flip-flops

Latches operate with enable signal, which is **level sensitive**. Whereas, flip-flops are edge sensitive. We will discuss about flip-flops in next chapter. Now, let us discuss about SR Latch & D Latch one by one.

SR Latch

SR Latch is also called as **Set Reset Latch**. This latch affects the outputs as long as the enable, E is maintained at '1'. The **circuit diagram** of SR Latch is shown in the following figure.



This circuit has two inputs S & R and two outputs Q(t) & Q(t)'. The **upper NOR gate** has two inputs R & complement of present state, Q(t)' and produces next state, Q(t+1) when enable, E is '1'.

Similarly, the **lower NOR gate** has two inputs S & present state, Q(t) and produces complement of next state, Q(t+1)' when enable, E is '1'.

We know that a **2-input NOR gate** produces an output, which is the complement of another input when one of the input is '0'. Similarly, it produces '0' output, when one of the input is '1'.

- If S = 1, then next state Q(t + 1) will be equal to '1' irrespective of present state, Q(t) values.
- If R = 1, then next state Q(t + 1) will be equal to '0' irrespective of present state, Q(t) values.

At any time, only of those two inputs should be '1'. If both inputs are '1', then the next state Q(t + 1) value is undefined.

The following table shows the **state table** of SR latch.

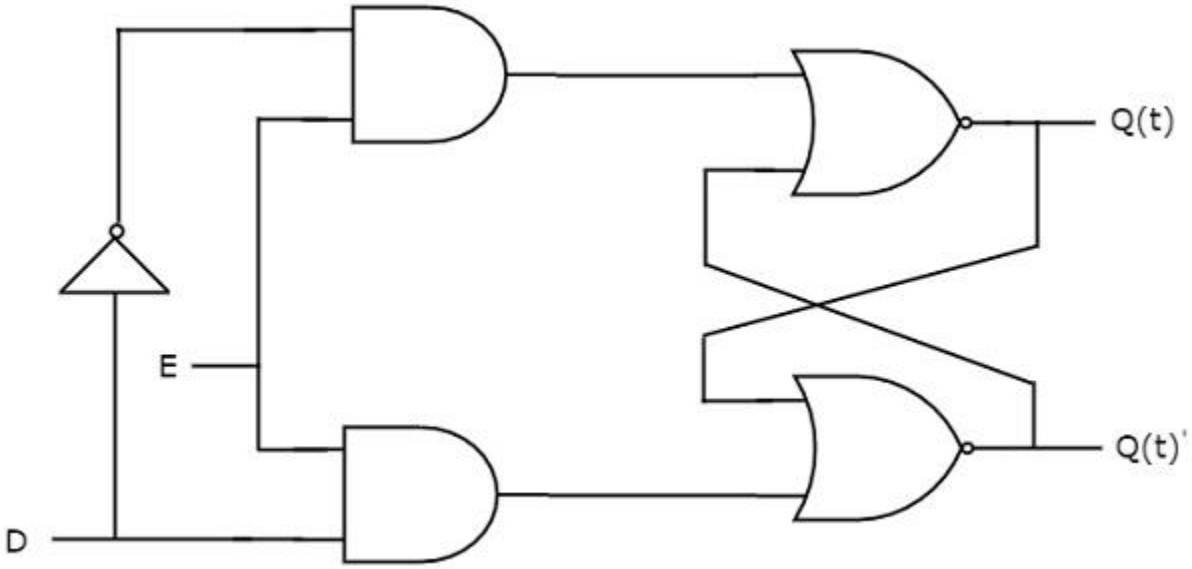
S	R	Q(t + 1)
0	0	Q(t)
0	1	0

1	0	1
1	1	-

Therefore, SR Latch performs three types of functions such as Hold, Set & Reset based on the input conditions.

D Latch

There is one drawback of SR Latch. That is the next state value can't be predicted when both the inputs S & R are one. So, we can overcome this difficulty by D Latch. It is also called as Data Latch. The **circuit diagram** of D Latch is shown in the following figure.



This circuit has single input D and two outputs Q(t) & Q(t)'. D Latch is obtained from SR Latch by placing an inverter between S amp;& R inputs and connect D input to S. That means we eliminated the combinations of S & R are of same value.

- If D = 0 → S = 0 & R = 1, then next state Q(t + 1) will be equal to '0' irrespective of present state, Q(t) values. This is corresponding to the second row of SR Latch state table.
- If D = 1 → S = 1 & R = 0, then next state Q(t + 1) will be equal to '1' irrespective of present state, Q(t) values. This is corresponding to the third row of SR Latch state table.

The following table shows the **state table** of D latch.

D	**Q(t + 1)**

0	0
1	1

Therefore, D Latch Hold the information that is available on data input, D. That means the output of D Latch is sensitive to the changes in the input, D as long as the enable is High.

In this chapter, we implemented various Latches by providing the cross coupling between NOR gates. Similarly, you can implement these Latches using NAND gates.

In previous chapter, we discussed about Latches. Those are the basic building blocks of flip-flops. We can implement flip-flops in two methods.

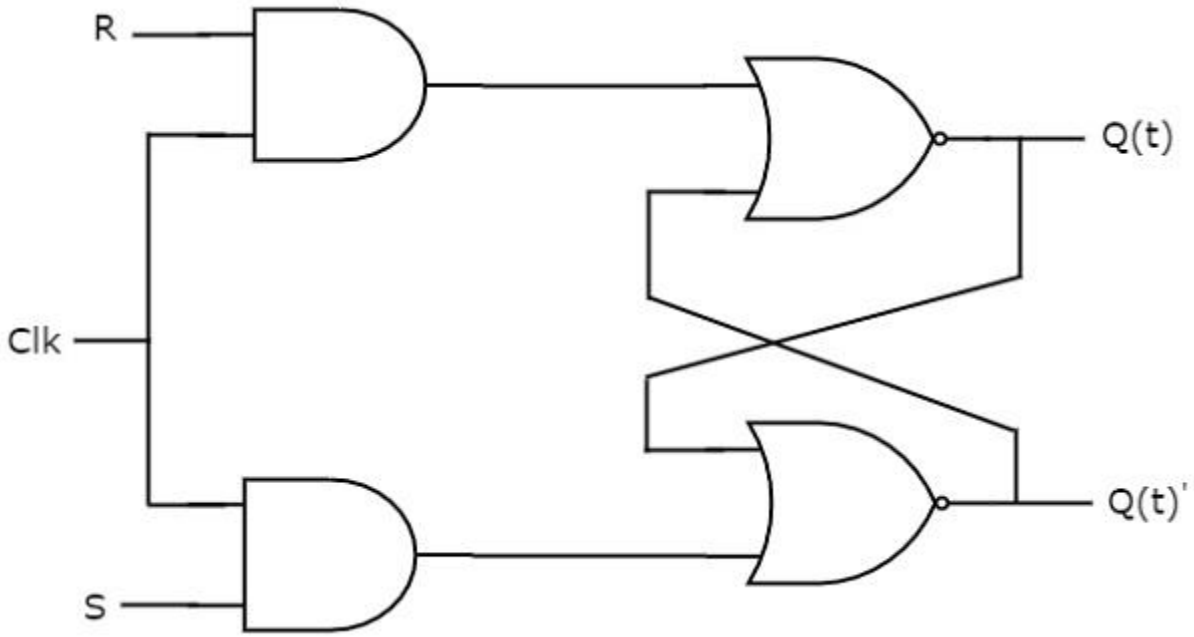
In first method, **cascade two latches** in such a way that the first latch is enabled for every positive clock pulse and second latch is enabled for every negative clock pulse. So that the combination of these two latches become a flip-flop.

In second method, we can directly implement the flip-flop, which is edge sensitive. In this chapter, let us discuss the following **flip-flops** using second method.

- SR Flip-Flop
- D Flip-Flop
- JK Flip-Flop
- T Flip-Flop

SR Flip-Flop

SR flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, SR latch operates with enable signal. The **circuit diagram** of SR flip-flop is shown in the following figure.



This circuit has two inputs S & R and two outputs Q(t) & Q(t)'. The operation of SR flipflop is similar to SR Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of SR flip-flop.

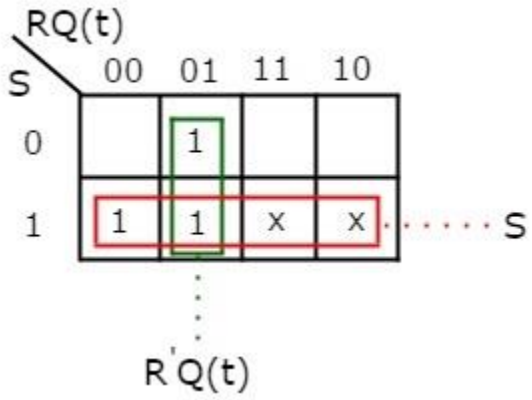
S	R	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	-

Here, Q(t) & Q(t + 1) are present state & next state respectively. So, SR flip-flop can be used for one of these three functions such as Hold, Reset & Set based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of SR flip-flop.

Present Inputs	Present State	Next State

S	R	Q(t)	Q(t + 1)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

By using three variable K-Map, we can get the simplified expression for next state, Q(t + 1). The **three variable K-Map**for next state, Q(t + 1) is shown in the following figure.

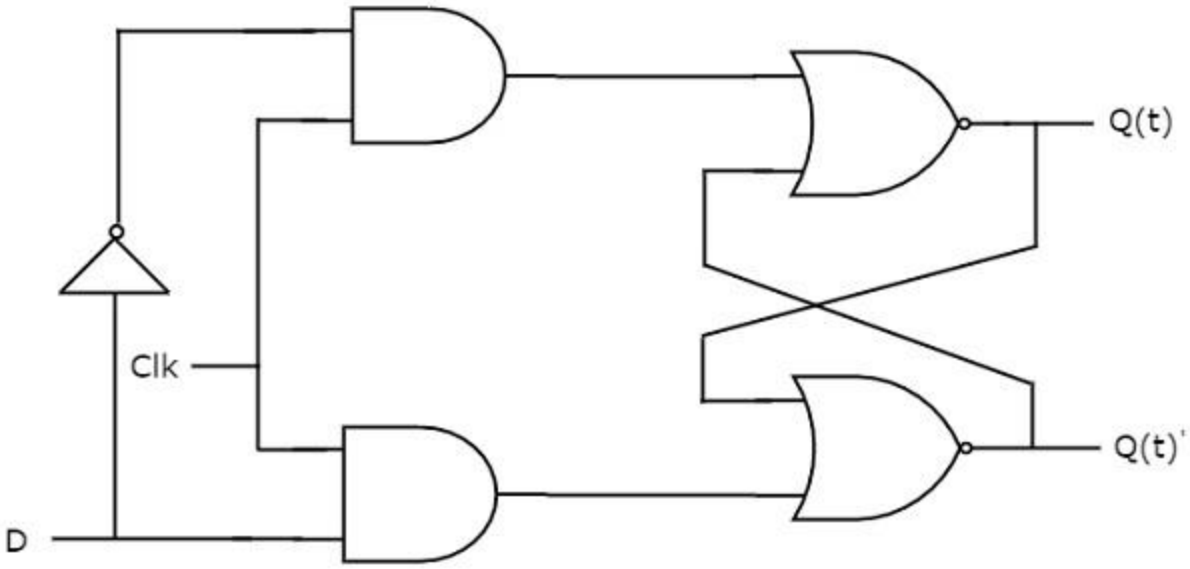


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Q(t + 1) is

$$Q(t+1)=S+R'Q(t)Q(t+1)=S+R'Q(t)$$

D Flip-Flop

D flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, D latch operates with enable signal. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit has single input D and two outputs Q(t) & Q(t)'. The operation of D flip-flop is similar to D Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of D flip-flop.

D	Q(t + 1)
0	0
0	1

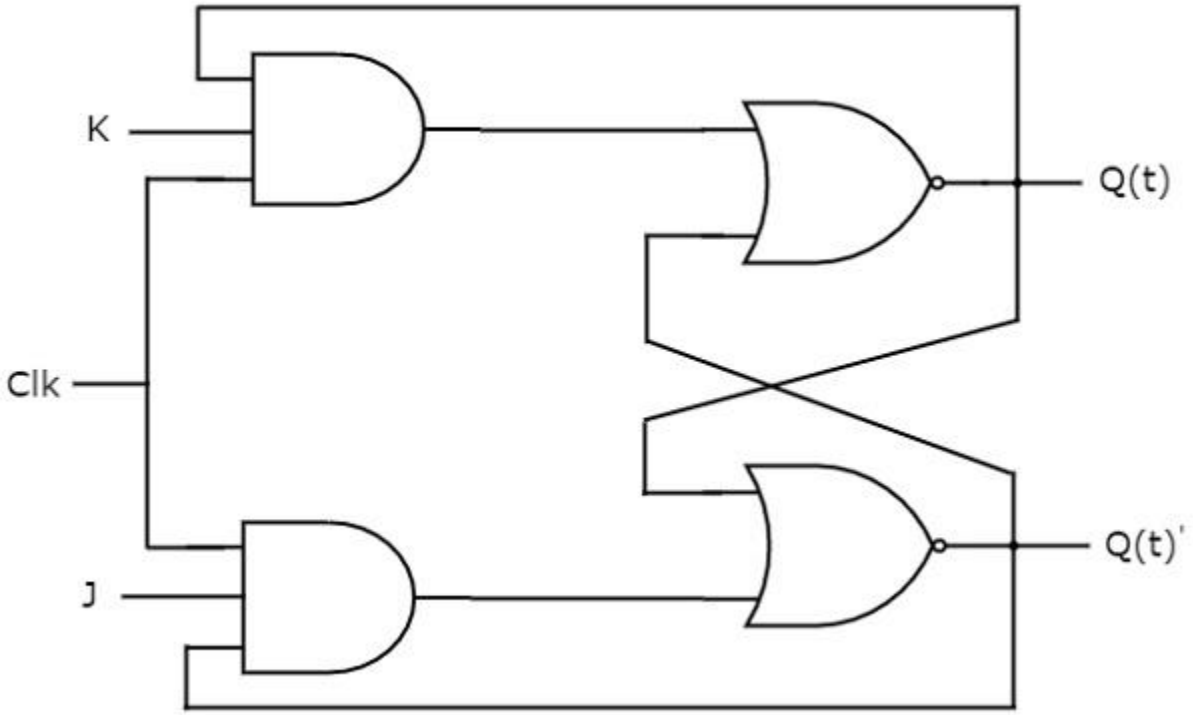
Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

$$Q(t + 1) = D$$

Next state of D flip-flop is always equal to data input, D for every positive transition of the clock signal. Hence, D flip-flops can be used in registers, **shift registers** and some of the counters.

JK Flip-Flop

JK flip-flop is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions. The **circuit diagram** of JK flip-flop is shown in the following figure.



This circuit has two inputs J & K and two outputs Q(t) & Q(t)'. The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as **S = J Q(t)'** and **R = KQ(t)** in order to utilize the modified SR flip-flop for 4 combinations of inputs.

The following table shows the **state table** of JK flip-flop.

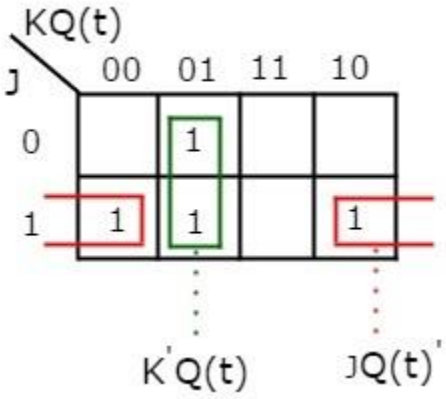
J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q(t)'

Here, Q(t) & Q(t + 1) are present state & next state respectively. So, JK flip-flop can be used for one of these four functions such as Hold, Reset, Set & Complement of present state based on the input

conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of JK flip-flop.

Present Inputs		Present State	Next State
J	**K**	**Q(t)**	**Q(t+1)**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

By using three variable K-Map, we can get the simplified expression for next state, Q(t + 1). **Three variable K-Map** for next state, Q(t + 1) is shown in the following figure.

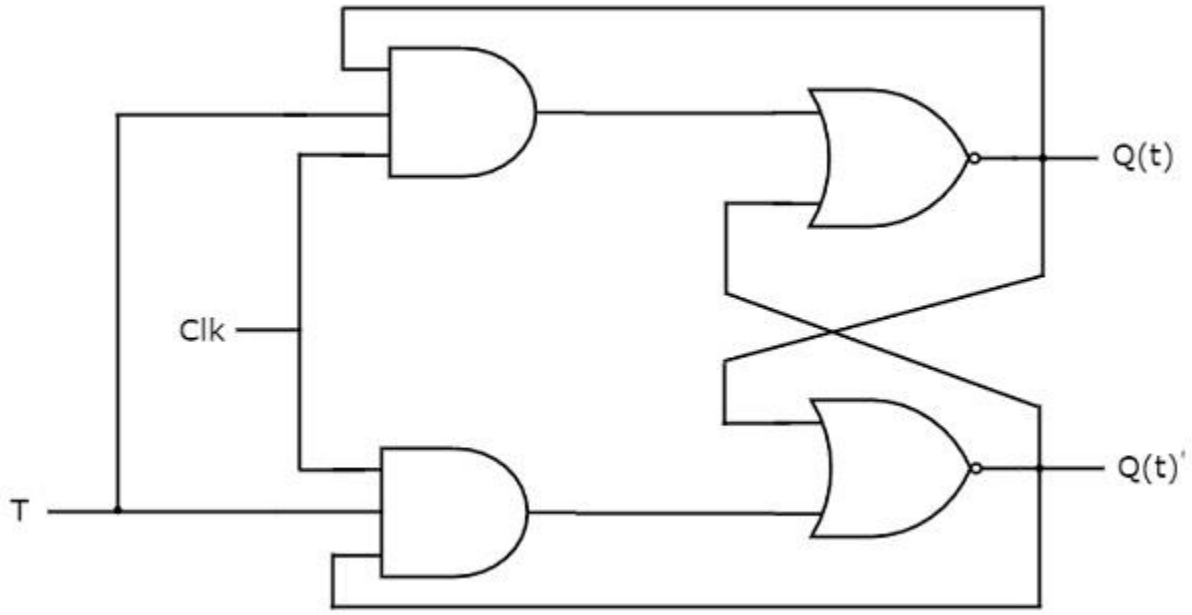


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Q(t+1) is

$$Q(t+1)=JQ(t)'+K'Q(t)Q(t+1)=JQ(t)'+K'Q(t)$$

T Flip-Flop

T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions. The **circuit diagram** of T flip-flop is shown in the following figure.



This circuit has single input T and two outputs Q(t) & Q(t)'. The operation of T flip-flop is same as that of JK flip-flop. Here, we considered the inputs of JK flip-flop as **J = T** and **K = T** in order to utilize the modified JK flip-flop for 2 combinations of inputs. So, we eliminated the other two combinations of J & K, for which those two values are complement to each other in T flip-flop.

The following table shows the **state table** of T flip-flop.

D	Q(t + 1)
0	Q(t)
1	Q(t)'

Here, Q(t) & Q(t + 1) are present state & next state respectively. So, T flip-flop can be used for one of these two functions such as Hold, & Complement of present state based on the input conditions,

when positive transition of clock signal is applied. The following table shows the **characteristic table** of T flip-flop.

Inputs	Present State	Next State
T	**Q(t)**	**Q(t + 1)**
0	0	0
0	1	1
1	0	1
1	1	0

From the above characteristic table, we can directly write the **next state equation** as

$$Q(t+1)=T'Q(t)+TQ(t)' \Rightarrow Q(t+1)=T'Q(t)+TQ(t)'$$

$$\Rightarrow Q(t+1)=T \oplus Q(t) \Rightarrow Q(t+1)=T \oplus Q(t)$$

The output of T flip-flop always toggles for every positive transition of the clock signal, when input T remains at logic High (1). Hence, T flip-flop can be used in **counters**.

In this chapter, we implemented various flip-flops by providing the cross coupling between NOR gates. Similarly, you can implement these flip-flops by using NAND gates.

In previous chapter, we discussed the four flip-flops, namely SR flip-flop, D flip-flop, JK flip-flop & T flip-flop. We can convert one flip-flop into the remaining three flip-flops by including some additional logic. So, there will be total of twelve **flip-flop conversions**.

Follow these **steps** for converting one flip-flop to the other.

- Consider the **characteristic table** of desired flip-flop.
- Fill the excitation values (inputs) of given flip-flop for each combination of present state and next state. The **excitation table** for all flip-flops is shown below.

Present State	Next State	SR flip-flop inputs		D flip-flop input	JK flip-flop inputs		T flip-flop input
Q(t)	Q(t+1)	S	R	D	J	K	T
0	0	0	x	0	0	x	0
0	1	1	0	1	1	x	1
1	0	0	1	0	x	1	1
1	1	x	0	1	x	0	0

- Get the **simplified expressions** for each excitation input. If necessary, use Kmaps for simplifying.
- Draw the **circuit diagram** of desired flip-flop according to the simplified expressions using given flip-flop and necessary logic gates.

Now, let us convert few flip-flops into other. Follow the same process for remaining flipflop conversions.

SR Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of SR flip-flop to other flip-flops.

- SR flip-flop to D flip-flop
- SR flip-flop to JK flip-flop
- SR flip-flop to T flip-flop

SR flip-flop to D flip-flop conversion

Here, the given flip-flop is SR flip-flop and the desired flip-flop is D flip-flop. Therefore, consider the following **characteristic table** of D flip-flop.

D flip-flop input	Present State	Next State
D	**Q(t)**	**Q(t + 1)**
0	0	0
0	1	0
1	0	1
1	1	1

We know that SR flip-flop has two inputs S & R. So, write down the excitation values of SR flip-flop for each combination of present state and next state values. The following table shows the characteristic table of D flip-flop along with the **excitation inputs** of SR flip-flop.

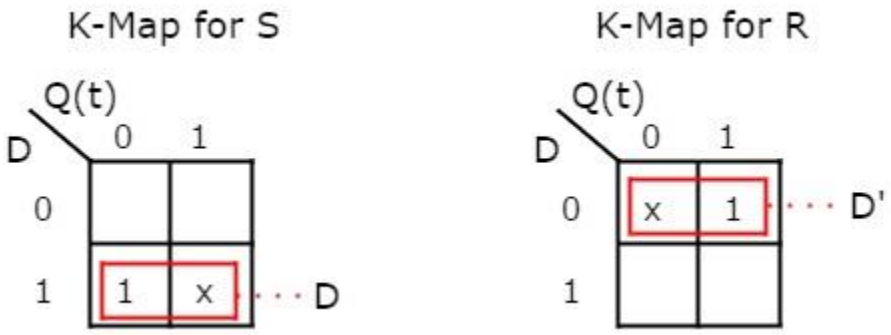
D flip-flop input	Present State	Next State	SR flip-flop inputs	
D	**Q(t)**	**Q(t + 1)**	**S**	**R**
0	0	0	0	x
0	1	0	0	1
1	0	1	1	0
1	1	1	x	0

From the above table, we can write the **Boolean functions** for each input as below.

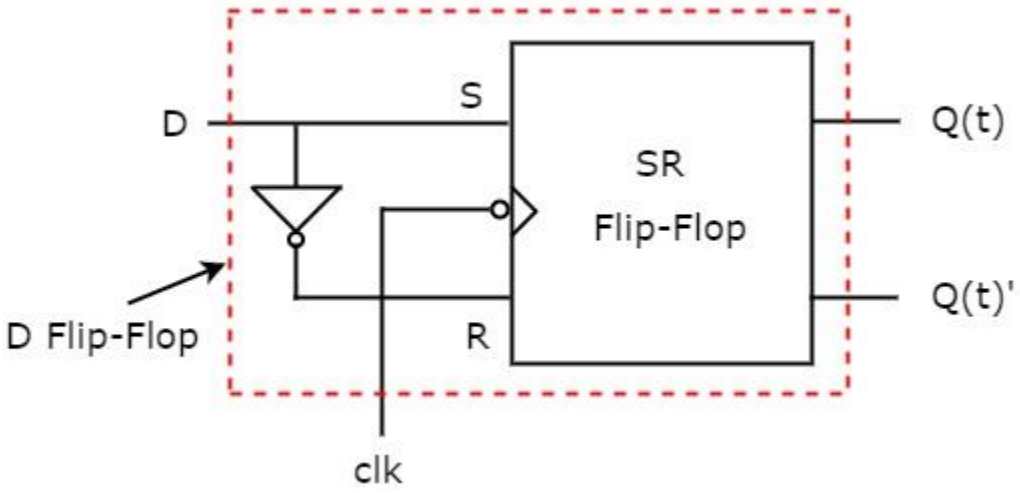
S=m2+d3S=m2+d3

R=m1+d0R=m1+d0

We can use 2 variable K-Maps for getting simplified expressions for these inputs. The **k-Maps** for S & R are shown below.



So, we got S = D & R = D' after simplifying. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit consists of SR flip-flop and an inverter. This inverter produces an output, which is complement of input, D. So, the overall circuit has single input, D and two outputs Q(t) & Q(t)'. Hence, it is a **D flip-flop**. Similarly, you can do other two conversions.

D Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of D flip-flop to other flip-flops.

- D flip-flop to T flip-flop
- D flip-flop to SR flip-flop
- D flip-flop to JK flip-flop

D flip-flop to T flip-flop conversion

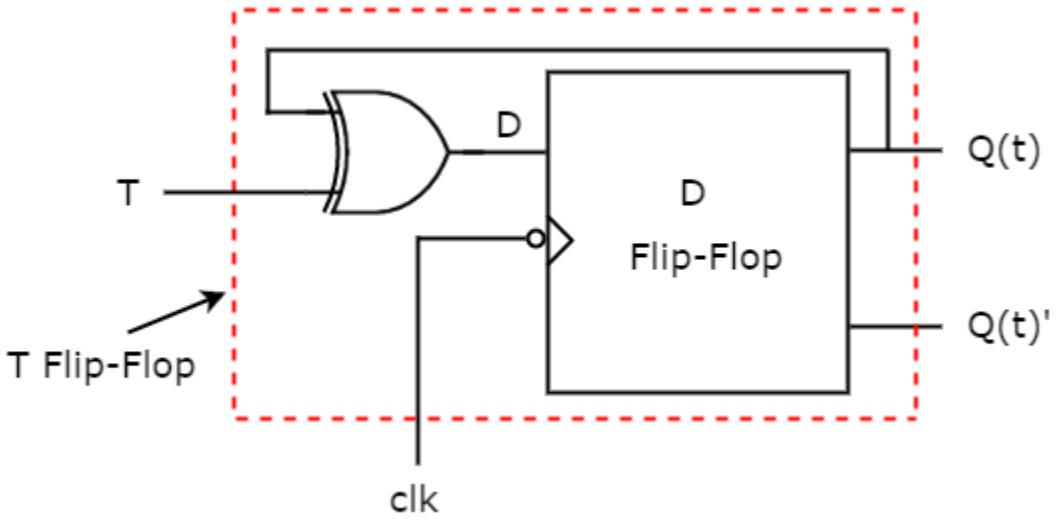
Here, the given flip-flop is D flip-flop and the desired flip-flop is T flip-flop. Therefore, consider the following **characteristic table**of T flip-flop.

T flip-flop input	Present State	Next State
T	**Q(t)**	**Q(t + 1)**
0	0	0
0	1	1
1	0	1
1	1	0

We know that D flip-flop has single input D. So, write down the excitation values of D flip-flop for each combination of present state and next state values. The following table shows the characteristic table of T flip-flop along with the **excitation input**of D flip-flop.

T flip-flop input	Present State	Next State	D flip-flop input
T	**Q(t)**	**Q(t + 1)**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

So, we require a two input Exclusive-OR gate along with D flip-flop. The **circuit diagram** of T flip-flop is shown in the following figure.



This circuit consists of D flip-flop and an Exclusive-OR gate. This Exclusive-OR gate produces an output, which is Ex-OR of T and Q(t). So, the overall circuit has single input, T and two outputs Q(t) & Q(t)'. Hence, it is a **T flip-flop**. Similarly, you can do other two conversions.

JK Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of JK flip-flop to other flip-flops.

- JK flip-flop to T flip-flop
- JK flip-flop to D flip-flop
- JK flip-flop to SR flip-flop

JK flip-flop to T flip-flop conversion

Here, the given flip-flop is JK flip-flop and the desired flip-flop is T flip-flop. Therefore, consider the following **characteristic table** of T flip-flop.

T flip-flop input	Present State	Next State
T	**Q(t)**	**Q(t + 1)**
0	0	0
0	1	1
1	0	1

1	1	0

We know that JK flip-flop has two inputs J & K. So, write down the excitation values of JK flip-flop for each combination of present state and next state values. The following table shows the characteristic table of T flip-flop along with the **excitation inputs** of JK flipflop.

T flip-flop input	**Present State**	**Next State**	**JK flip-flop inputs**	
T	**Q(t)**	**Q(t + 1)**	**J**	**K**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

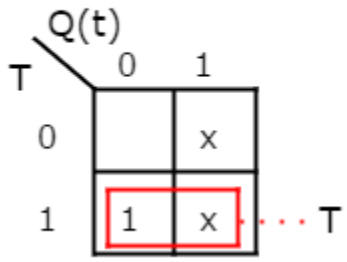
From the above table, we can write the **Boolean functions** for each input as below.

$$J=m2+d1+d3J=m2+d1+d3$$

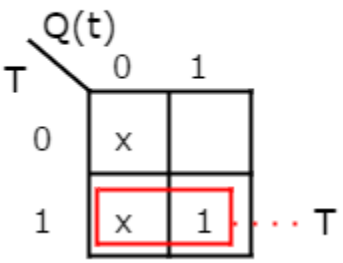
$$K=m3+d0+d2K=m3+d0+d2$$

We can use 2 variable K-Maps for getting simplified expressions for these two inputs. The **k-Maps** for J & K are shown below.

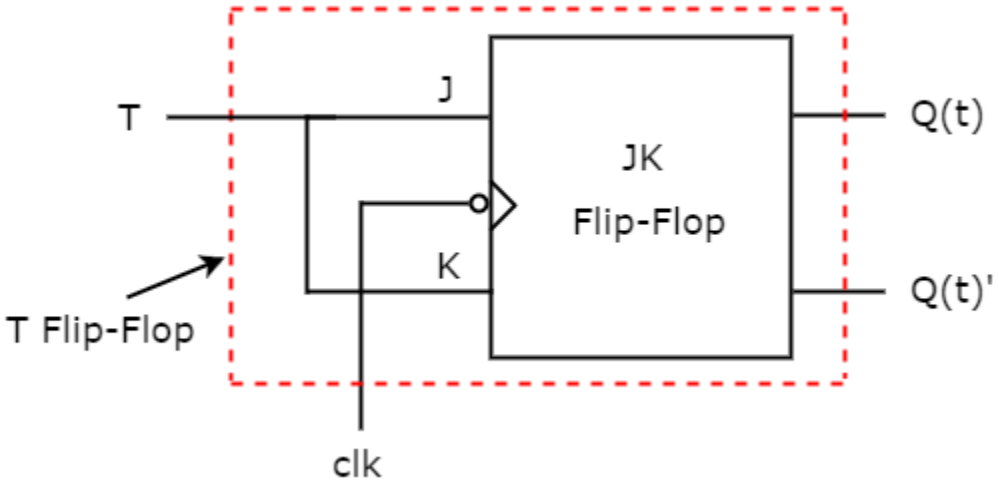
K-Map for J



K-Map for K



So, we got, J = T & K = T after simplifying. The **circuit diagram**of T flip-flop is shown in the following figure.



This circuit consists of JK flip-flop only. It doesn't require any other gates. Just connect the same input T to both J & K. So, the overall circuit has single input, T and two outputs Q(t) & Q(t)'. Hence, it is a **T flip-flop**. Similarly, you can do other two conversions.

T Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of T flip-flop to other flip-flops.

- T flip-flop to D flip-flop
- T flip-flop to SR flip-flop
- T flip-flop to JK flip-flop

T flip-flop to D flip-flop conversion

Here, the given flip-flop is T flip-flop and the desired flip-flop is D flip-flop. Therefore, consider the characteristic table of D flip-flop and write down the excitation values of T flip-flop for each combination of present state and next state values. The following table shows the **characteristic table** of D flip-flop along with the **excitation input** of T flip-flop.

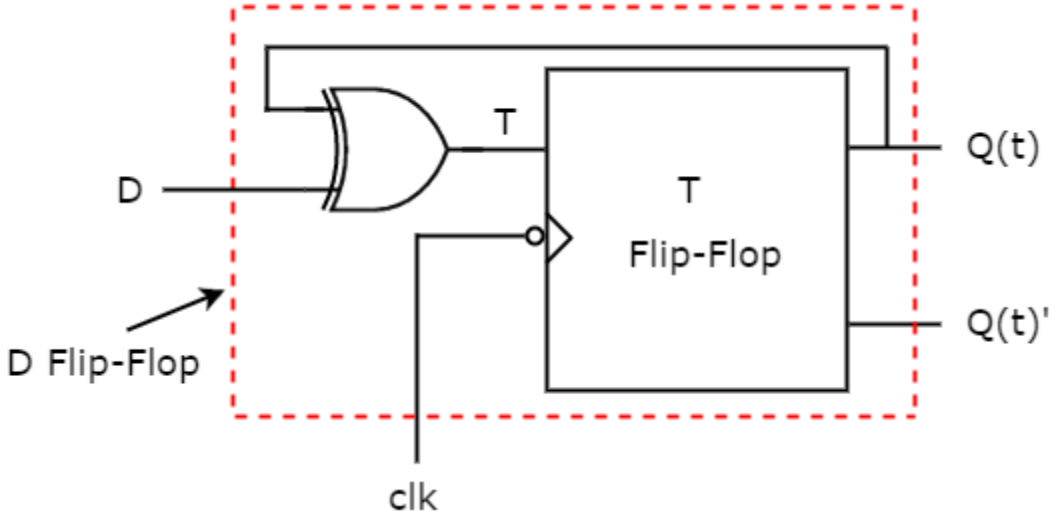
D flip-flop input	Present State	Next State	T flip-flop input
D	**Q(t)**	**Q(t + 1)**	**T**
0	0	0	0

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

From the above table, we can directly write the Boolean function of T as below.

$$T=D\oplus Q(t) T=D\oplus Q(t)$$

So, we require a two input Exclusive-OR gate along with T flip-flop. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit consists of T flip-flop and an Exclusive-OR gate. This Exclusive-OR gate produces an output, which is Ex-OR of D and Q(t). So, the overall circuit has single input, D and two outputs Q(t) & Q(t)'. Hence, it is a **D flip-flop**. Similarly, you can do other two conversions.

Unit-IV
Introduction to Verilog HDL

Verilog as HDL

Verilog HDL (Verilog Hardware Description Language) consists of various constructs. All are aimed at providing a functionally tested and verified design description for the targeted FPGA (Field Programmable Gate Array) or ASIC(Application Specific Integrated Circuit).

This language has dual functions –

1. Fulfilling the need for a design description.
2. Fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, setup, and hold times.

Levels of Design Description

Using Verilog HDL design constructs, a module can be defined using various levels of abstraction. There are four levels of abstraction in Verilog HDL.
They are: **1. Circuit Level 2. Gate Level 3. Data Flow Level 4. Behavioral Level**

1. Circuit Level

- At the circuit level, a switch is the basic element with which digital circuits are built. Switches can be combined to form inverters and other gates at the next higher level of abstraction.
- Verilog has the basic MOS switches built into its constructs, which can be used to build basic circuits like inverters, basic logic gates, simple 1-bit dynamic and static memories.

The below Figure1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

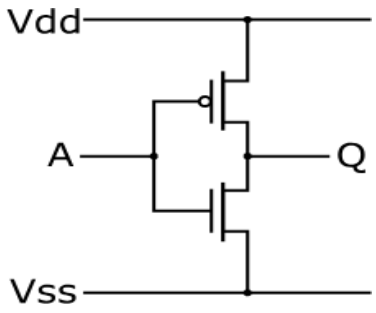


Figure 1: CMOS inverter

2. Gate Level

- It is the next higher level of abstraction, Here the design is carried out in terms of basic gates.
- All the basic gates are available as ready modules called “Primitives”. Each such primitive is defined in terms of its inputs and outputs. Primitives can be incorporated into design descriptions directly.
- Just as full physical hardware can be built using gates, the primitives can be used repeatedly to build larger systems.
- In gate level modeling (or structural modeling) we should know the structure of the design to build the model. However, beyond 20 to 30 of such gate primitives in a circuit, the design description becomes difficult; testing and debugging becomes more complex.

Figure 2 shows an AND gate suitable for description using the gate primitive of Verilog.

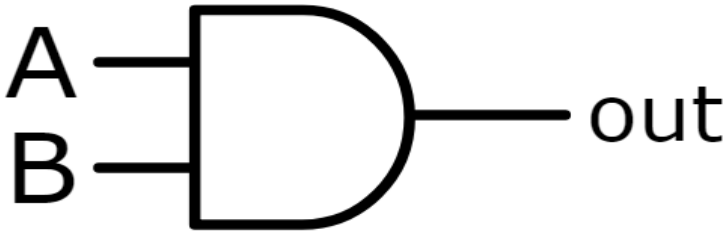


Figure 2 AND gate symbol

3. Data Flow

- Data flow is the next higher level of abstraction. All possible operations on signals and variables are represented here in terms of assignments.
- All logic and algebraic operations are accommodated. The assignments define the continuous functioning of the concerned block.
- At the data flow level, signals are assigned through the data manipulating equations. All such assignments are concurrent in nature. The design descriptions are more compact than those at the gate level.

Figure 3 shows an A-O-I relationship suitable for description with the Verilog HDL constructs at the data flow level.

$$e = \overline{a.b + c.d}$$

Figure 3 An A-O-I gate represented as a data flow type of relationship.

4. Behavioral Level

- Behavioral level constitutes the highest level of design description; it is essentially at the system level. (With the assignment possibilities, looping constructs and conditional operators, the design description looks like a "C" program.)
- A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.
- The statements involved are "dense" in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient.
- Figure 4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of Verilog.

If (a, b, c or d changes)
Compute e as
$$e = \overline{a.b + c.d}$$

Figure 4 An A-O-I gate in pseudo code at behavioral level.

The Overall Design module in Verilog HDL

Verilog HDL facilitates the mixing of the above-mentioned levels of design. A design built at data flow level can be instantiated to form a structural mode design. Data flow assignments can be incorporated in designs which are basically at behavioral level.

Concurrency

In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change. Simulation of such a circuit in an HDL calls for concurrency of operation.

A number of activities – may be spread over different modules – are to be run concurrently here. Verilog simulators are built to simulate concurrency. (This is in contrast to programs in the normal languages like C where execution is sequential.)

Concurrency is achieved by proceeding with simulation in equal time steps. The time step is kept small enough to be negligible compared with the propagation delay values. All the activities

scheduled at one time step are completed and then the simulator advances to the next time step and so on. The time step values refer to simulation time and not real time.

In some cases, the circuit itself may demand sequential operation as with data transfer and memory-based operations. Only in such cases sequential operation is ensured by the sequential constructs from Verilog HDL.

Simulation and Synthesis

- Testing the functionality of developed design (module) is called "**simulation**". The design that is specified and entered as described earlier is simulated for functionality and fully debugged.

- Translation of the debugged design into the corresponding hardware circuit is called "**synthesis.**"

- The tools available for synthesis relate more easily with the gate level and data flow level modules. In contrast many of the behavioral level constructs are not directly synthesizable. The way out is to take the behavioral level modules and re-do each of them at lower levels (either in gate level or in Dataflow level).

- The process is carried out successively with each of the behavioral level modules until practically the full design is available as a pack of modules at gate and data flow levels (more commonly called the "RTL level").

Programming Language Interface (PLI)

PLI provides an active interface to a compiled Verilog module. The interface adds a new dimension to working with Verilog routines from a C platform.

The key functions of the interface are as follows:

- One can read data from a file and pass it to a Verilog module as input. Such data can be test vectors or other input data to the module. Similarly, variables in Verilog modules can be accessed and their values written to output devices.
- Delay values, logic values, etc., within a module can be accessed and altered.
- Blocks written in C language can be linked to Verilog modules.

MODULE

Any Verilog HDL program begins with a keyword – called a "module".
A module is the name given to any system considering it as a black box with input and output terminals as shown in below figure.
The terminals of the module are referred to as "ports".

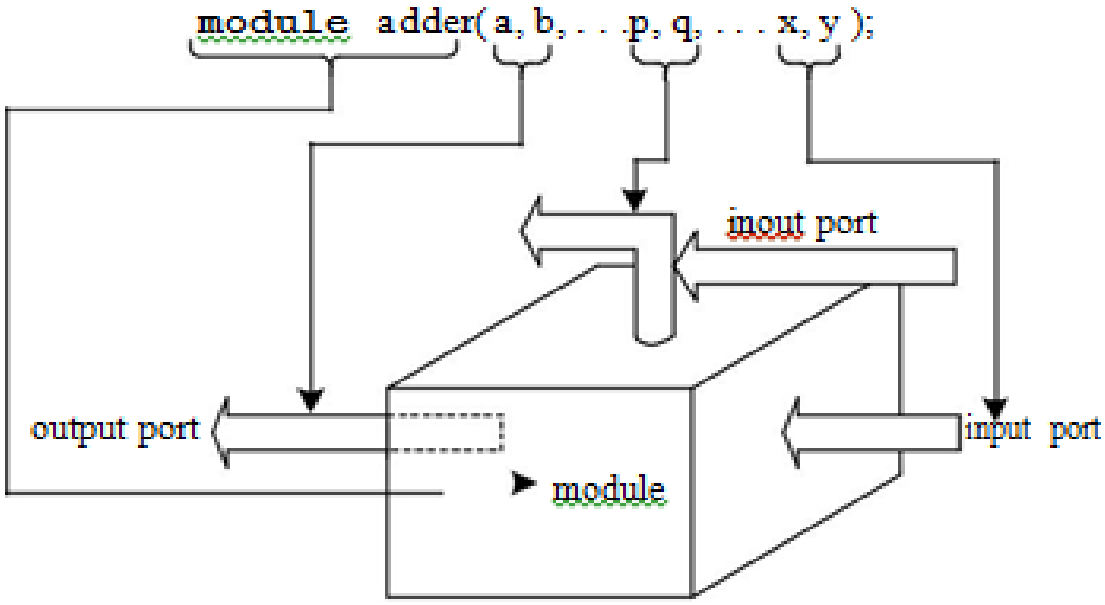


Figure 1 Representation of a module as black box with its ports.

The ports attached to a module can be of three types:

- input ports: Through which one gets entry into the module; they signify the input signal terminals of the module.
- output ports: Through which one exits the module; these signifythe output signal terminals of the module.
- inout ports: These represent ports through which one gets entry into the module or exits the module; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

Whether a module has any of the above ports and how many of each type are present depends on the functional nature of the module.

The structure of modules and the mode of invoking them in a design are discussed here.

- In Verilog HDL any program which forms a design description is a "**module**".

- Any program written to test a design description is also a "module". The latter are often called as "stimulus modules" or "**test benches**".
- A test bench module used to do simulation has the form shown in Figure 2. Whenever a module is invoked for testing or for incorporation into a bigger design module, the name of the module ("test" here) is used to identify it for the purpose.

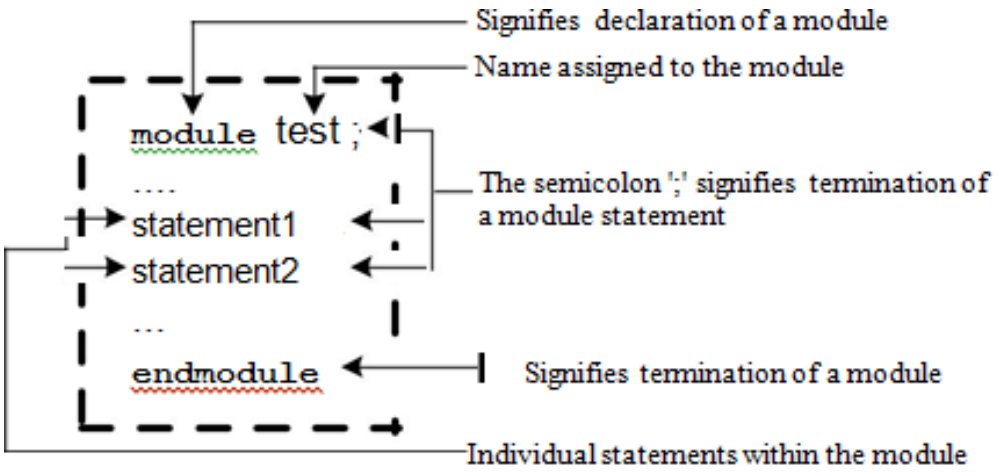


Figure 2 Structure of a typical simulation module.

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

Introduction:

The constructs and conventions implement any language. Verilog HDL has its own constructs and conventions [IEEE, Sutherland].

Any source file in Verilog (as with any file in any other programming language) is made up of a number of ASCII characters. The characters are grouped into sets — referred to as "lexical tokens".

A lexical token in Verilog HDL can be a single character or a group of characters. Verilog has 7 types of lexical tokens- keywords, identifiers, white spaces, comments, numbers, strings and operators.

***Verilog HDL is a Case Sensitive Language*.**

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,… etc., are all different entities in Verilog HDL.

*A module comprises a number of "**lexical tokens**" arranged according to some predefined order. The possible tokens are of seven categories:

1. **Keywords**
2. **Identifiers**
3. **White spaces characters**
4. **Comments**
5. **Numbers**
6. **Strings**
7. **Operators**

1. Keywords

- The keywords define the language constructs.
- A keyword signifies an activity to be carried out, initiated, or terminated.
- A programmer cannot use a keyword for any purpose other than that it is intended for.
- All keywords in Verilog are in small letters.

Examples:

module -- signifies the beginning of a module definition.
endmodule -- signifies the end of a module definition.
begin -- signifies the beginning of a block of statements.
end -- signifies the end of a block of statements.
if -- signifies a conditional activity to be checked.
while -- signifies a conditional activity to be carried out.

2. Identifiers

Identifiers are used to define language constructs.

- Identifiers refer objects to be referenced in the design.
- Identifiers are made of alphabets (both cases), numbers, the underscore '_' and the dollar sign '$'.
- They start with an alphabetic character or underscore.
- They cannot start with a number or with '$' which is reserved for system tasks.
- Identifiers are case sensitive i.e., identifiers differing in their case are distinct.
- An identifier say count is different from COUNT, count and cOuNT.

Example: clock, enable, gate_1, . . .

Example:

name, _name, Name, name1, name_$, . . . -- all these are allowed as identifiers

- name aa -- not allowed as an identifier because of the blank ("name" and "aa" are interpreted as two different identifiers)
- $name -- not allowed as an identifier because of the presence of "$" as the first character.
- 1_name -- not allowed as an identifier, since the numeral "1" is the first character
- @name -- not allowed as an identifier because of the presence of the character "@".
- A+b --not allowed as an identifier because of the presence of the character "+".

3. White Space Characters

- In any design description the white space characters are included to improve readability.
- White space characters have significance only when they appear inside strings.

Blank spaces (\b),

Tabs (\t),

Newlines (\n), are the white space characters in Verilog.

Example:

$monitor("The value of a=%b, b=%b, y=%b \n", a,b,y);

4. Comments

Comments can be inserted in the code to improve readability and helps good documentation. There are two ways to write comments.

- A one-line comment starts with "//". Verilog skips from that point to the end of line.
- A multiple-line comment starts with "/*" and ends with "*/".

Note: Multiple-line comments cannot be nested.

Example1:

```
module d_ff(Q, dp, clk);     //This is the design description of a D flip-flop.
                             //Here Q is the output.
                             // dpis the input and clkis the clock.
```

Example2:

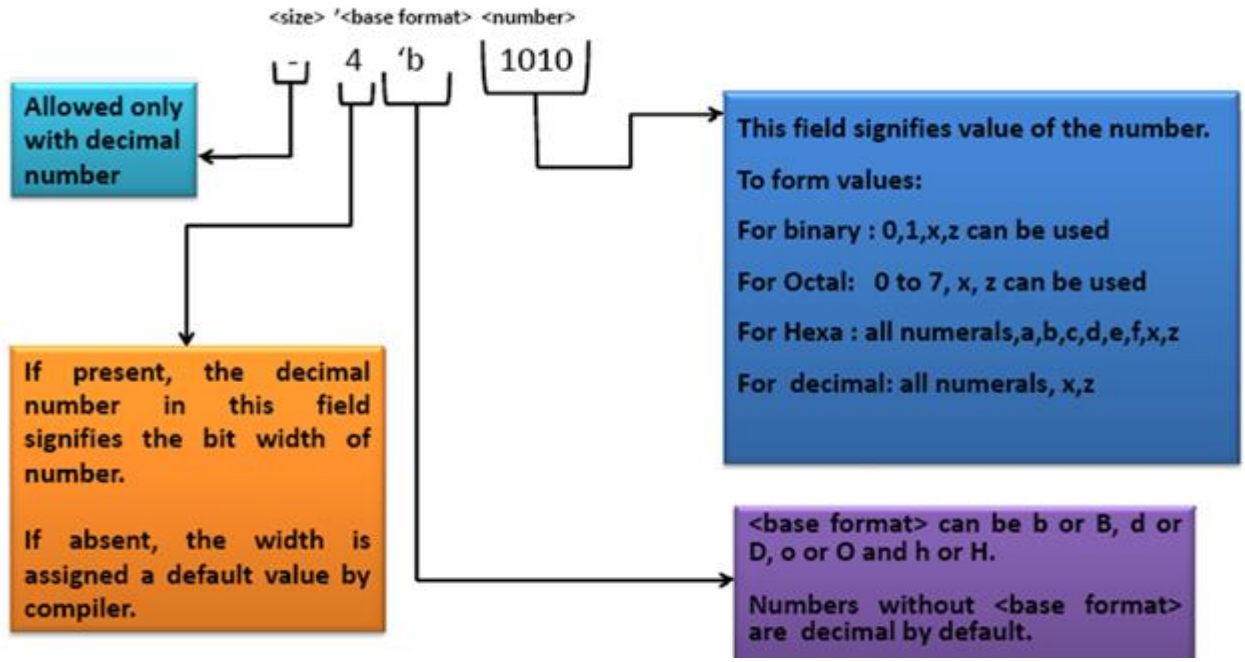
```
a = b && c; // This is a one-line comment
            /* This is a multiple
            line comment */

            /* This is /* an illegal */ comment */
```

5. Number Specification

There are two types of number specification in Verilog:

1. Sized numbers 2. Unsized numbers.



1. Sized numbers

- Sized numbers are represented as <size> '<base format> <number>.

- <size> is written only in decimal and specifies the number of bits in the number.
- Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).

- The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f.

- Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

Example:

```
4'b1111     // This is a 4-bit binary number.
12'habc     // This is a 12-bit hexadecimal number.
16'd255     // This is a 16-bit decimal number.
```

2. Unsized numbers

- Numbers that are specified without a <base format> specification are decimal numbers by default.

- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

Example:

23456 // This is a 32-bit.

'hc3 // This is a 32-bit .

'o21 // This is a 32-bit decimal number by default hexadecimal number octal number.

X or Z values:

- Verilog has two symbols for unknown and high impedance values.
- These values are very important for modeling real circuits. An unknown value is denoted by an x.
- A high impedance value is denoted by z.

Example:

12'h13x // This is a 12-bit hex number; 4 least significant bits areunknown.

6'hx // This is a 6-bit hex number.

32'bz // This is a 32-bit high impedance number.

- An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.
- This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers:

- Negative numbers can be specified by putting a minus sign before the size for a constant number.

- Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>.
- An optional signed specifier can be added for signed arithmetic.

  Example:

-8'd3 // 8-bit negative number stored as 2's complement of 3.

4'd-2 // Illegal specification.

Underscore characters"_" and question marks"?":

- An underscore character "_" is allowed anywhere in a number except the first character.

- Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.

- A question mark "?" is the Verilog HDL alternative for z in the context of numbers.

Exapmle:

12'b1111_0000_1010 // Use of underline characters for readability

4'b10?? // Equivalent of a 4'b10zz

6. Strings

- A string is a sequence of characters that are enclosed by double quotes.
- The restriction on a string is that it must be contained on a single line, that is, without a carriage return.
- It cannot be on multiple lines.

Example:

"Hello Verilog World" // is a string

"a / b" // is a string

7. Operators

Operators are of three types: unary, binary, and ternary.

- Unary operators precede the operand.
- Binary operators appear between two operands.
- Ternary operators have two separate operators that separate three operands.

Example:

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

Logic Values:

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table below.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
X	Unknown logic value
Z	High impedance, floating state

Data Types:

The data handled in Verilog fall into two categories:

(i) Net data type

(ii) Variable data type

The two types differ in the way they are used as well as with regard to their respective hardware structures. Data type of each variable or signal has to be declared prior to its use. The same is valid within the concerned block or module.

(i) Net Data Type

A net signifies a connection from one circuit unit to another. Such a net carries the value of the signal it is connected to and transmits to the circuit blocks connected to it. If the driving end of a net is left floating, the net goes to the high impedance state. A net can be specified in different ways.

wire: It represents a simple wire doing an interconnection. Only one output is connected to a wire and is driven by that.

tri: It represents a simple signal line as a wire. Unlike the wire, a tri can be driven by more than one signal outputs.
Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably.

(ii) Variable Data Type

A variable is an abstraction for a storage device. It can be declared through the keyword "reg" and stores the value of a logic level: 0, 1, x, or z.
A net or wire connected to a reg takes on the value stored in the reg and can be used as input to other circuit elements. But the output of a circuit cannot be connected to a reg. The value stored in a reg is changed through a fresh assignment in the program.
Example:
time, integer, real, and realtime are the other variable types of data.

Scalars and Vectors:

Entities representing single bits — whether the bit is stored, changed, or transferred — are called "scalars".
Similarly, a group of regs stores a value, which may be assigned, changed, and handled together. The collection here is treated as a "vector".

Figure below illustrates the difference between a scalar and a vector.

- wr and rd are two scalar nets connecting two circuit blocks circuit1 and circuit2.
- b is a 4-bit-wide vector net connecting the same two blocks.
- b[0], b[1], b[2], and b[3] are the individual bits of vector b. They are "part vectors."

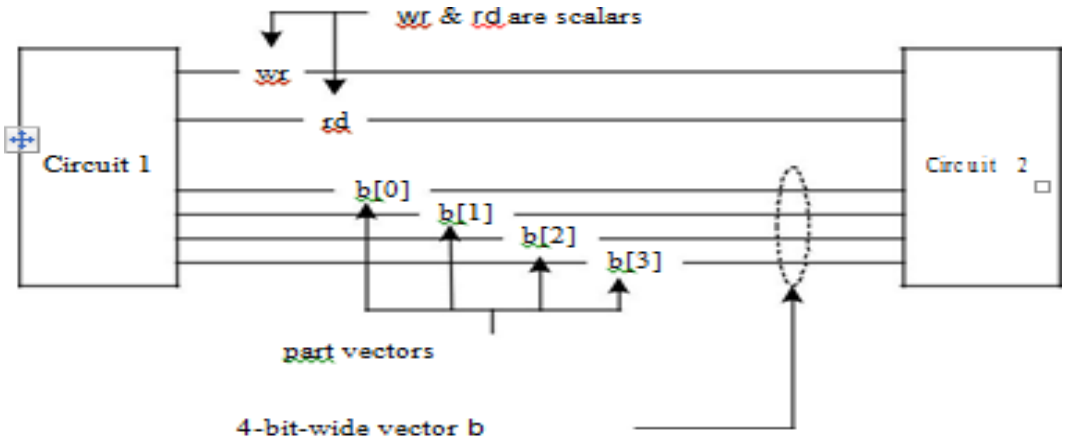


Figure Illustration of scalars and vectors.

Examples:

```
wire[3:0] a;      /* a is a four bit vector of net type; the bits are designated as a[3], a[2], a[1] and
a[0]. */

reg[2:0] b;       /* b is a three bit vector of reg type; the bits are designated as b[2], b[1] and b[0]. */

reg[4:2] c;       /* c is a three bit vector of reg type; the bits are designated as c[4], c[3] and c[2].
*/

wire[-2:2] d ; /* d is a 5 bit vector with individual bits designated as d[-2], d[-1], d[0], d[1] and d[2]. */
```

- Whenever a range is not specified for a net or a reg, the same is treated as a scalar – a single bit quantity.
- In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values.
- These can also be expressions evaluating to integer constants – positive or negative.

Normally vectors – nets or regs – are treated as unsigned quantities. They have to be specifically declared as “signed” if so desired.

Examples:

```
wire signed[4:0] num;      // num is a vector in the range -16 to +15.

reg signed [3:0] num_1;         // num_1 is a vector in the range -8 to +7.
```

Unit-V

Gate Level Modeling

Introduction:

Digital designers are normally familiar with all the common logic gates, their symbols, and their functionality. Flip-flops are built from the logic gates. All other functionally complex circuits can also be built using the basic gates.

All the basic gates are available as "primitives" in Verilog HDL. Primitives are generalized modules that already exist in Verilog HDL. They can be instantiated directly in module descriptions.

and Gate Primitive:

The AND gate primitive in Verilog HDL is instantiated with the following

and **g1 (O, I1, I2, . . ., In);**

Here "and" is the keyword signifying an AND gate. "g1" is the name assigned to the specific instantiation. O is the gate output; I1, I2, etc., are the gate inputs.

- The AND module has only one output. The first port in the argument list is the output port.
- An AND gate instantiation can take any number of inputs.
- A name need not be necessarily assigned to the AND gate instantiation. It is applicable for all the gate primitives available in Verilog HDL.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table below It can be directly extended to AND gate instantiations with multiple inputs.

Truth table of AND gate primitive

		Input 1			
		0	1	X	Z
Input 2	0	**0**	**0**	**0**	**0**
	1	**0**	**1**	**X**	**X**
	X	**0**	**X**	**X**	**X**
	Z	**0**	**X**	**X**	**X**

- If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, X or Z state.

- The output is at 1 state if and only if every one of the inputs is at 1 state.

- For all other cases the output is at the x state.

- Note that the output is never at the high impedance (Z) state. This is true of all other gate primitives as well.

Module Structure:

In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

- The first statement of a module starts with the keyword module; it may be followed by the name of the module and the port list if any.

- All the variables in the ports-list are to be identified as inputs, outputs, or inouts. The corresponding declarations have the form shown below:
  module <name identifier> (a1,a2,b1,b2,c1,c2);
  input a1, a2;
  output b1, b2;
  inout c1, c2;

The port-type declarations here follow the module declaration mentioned above.

The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case.

Examples:

```
wire a1, a2, c;
reg b1, b2;
```

The type declaration must necessarily precede the first use of any variable or signal in the module.

- The executable body of the module follows the declaration indicated above.

- The last statement in any module definition is the keyword “endmodule”.

- Comments can appear anywhere in the module definition.

Other Gate Primitives:

All basic gates are available as primitives in Verilog HDL. Details of each gate instantiations are given in Table below.

Table for Basic gate primitives in Verilog HDL

Gate Primitive	Mode of instantiation	Output port(s)	Input port(s)
and	**and** n1 (o, i1, i2, . . . i8);	o	i1, i2, . .
or	**or** n2 (o, i1, i2, . . . i8);	o	i1, i2, . .
nand	**nand** n3 (o, i1, i2, . . . i8);	o	i1, i2, . .
nor	**nor** n4 (o, i1, i2, . . . i8);	o	i1, i2, . .
xor	**xor** n5 (o, i1, i2, . . . i8);	o	i1, i2, . .
xnor	**xnor** n6 (o, i1, i2, . . . i8);	o	i1, i2, . .
buf	**buf** n7 (o1, o2, …. i);	o1, o2, o3, . .	i
not	**not** n8 (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

- Assign a name to the gate instantiation is an optional. It provides more clarity on circuit description.
- In all the cases the output ports are declared first and the input ports are declared subsequently.
- The buffer and the inverter have only one input, and any number of outputs. All other gates have one output, and any number of inputs.

Example for a typical A-O-I gate circuit

The commonly used A-O-I gate is shown in below Figure. The module and the test bench for the same are in Verilog HDL given below. The circuit has been realized here by instantiating the AND & NOR gate primitives.

- The module named as "aoi_gate" in the figure has input and output ports.
- The module "aoi_tb" is a testbench module. It generates inputs to the aoi_gate module and gets its output. It has no input or output ports.

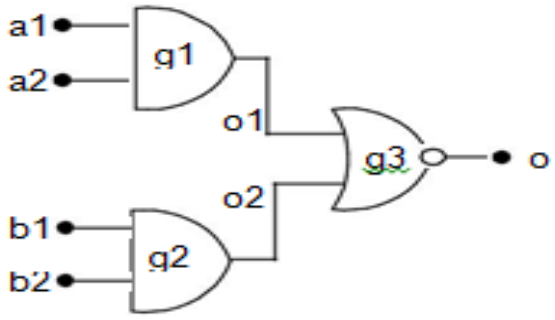


Figure for a typical A-O-I gate circuit.

```
/*module for the aoi-gate instantiating the gate primitives – above figure*/
module aoi_gate (o,a1,a2,b1,b2);
input a1,a2,b1,b2;   // a1,a2,b1,b2 form the input ports of the module.
output o;            //o is the single output port of the module.
wire o1,o2;          //o1 and o2 are intermediate signals within the module.
and g1(o1,a1,a2);    /*The AND gate primitive has two. In data flow this
expression written as “assign o1=a1 && a2;” */
and g2(o2,b1,b2);    /* instantiations with assigned names g1 & g2. In data
flow this expression written as “assign o2=b1 && b2;” */
nor g3(o,o1,o2);     /*The nor gate has one instantiation with assigned name
g3. In data flow this expression written as “assign o = !(o1 || o2);” */
endmodule
//Test-bench for the above module as follows
 module aoi_tb;
reg a1,a2,b1,b2;     /*specific values will be assigned to a1,a2,b1,and b2 and these connected
to input ports of the gate instantiations. Hence these variables are declared as reg.*/
wire o;
aoi_gate gg(o,a1,a2,b1,b2);
initial
begin
a1 = 0; a2 = 0; b1 = 0; b2 = 0;
#3 a1 = 1;
#3 a2 = 1;
#3 b1 = 1;
#3 b2 = 0;
#3 a1 = 1;
#3 a2 = 0;
#3 b1 = 0;
#100 $stop; //the simulation ends after running for 100 time units.
end
initial
$monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
endmodule
```

Tri-State Gates:

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a **control** signal.

The buffer is instantiated as

```
bufif1 n1 (out, in, control);
```

The symbol of the buffer is shown in Figure1.

- out as the output variable.
- in as the input variable.
- control as the control signal variable.

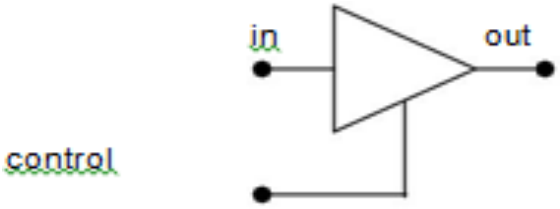
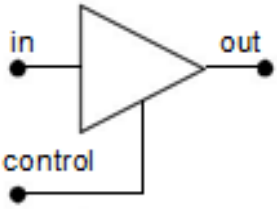
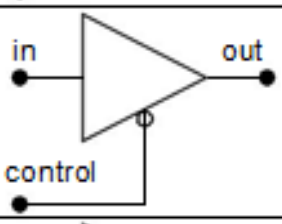
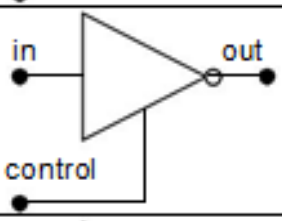
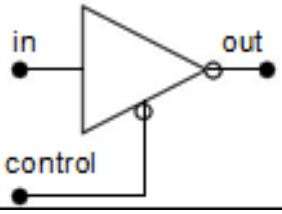


Figure 1 A tri-state buffer.

When control = 1, out = in.
When control = 0, out = tri-stated(Z). out is cut off from the input and tri-stated.

The output, input and control signals should appear in the instantiation in the same order as mentioned. Details of tri-state primitives are shown in Table 1.

Table 1 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
`bufif1 (out, in, control);`		Out = in if control = 1; else out = z
`bufif0 (out, in, control);`		Out = in if control = 0; else out = z
`notif1 (out, in, control);`		Out = complement of in if control = 1; else out = z
`notif0 (out, in, control);`		Out = complement of in if control = 0; else out = z

Gate Delays

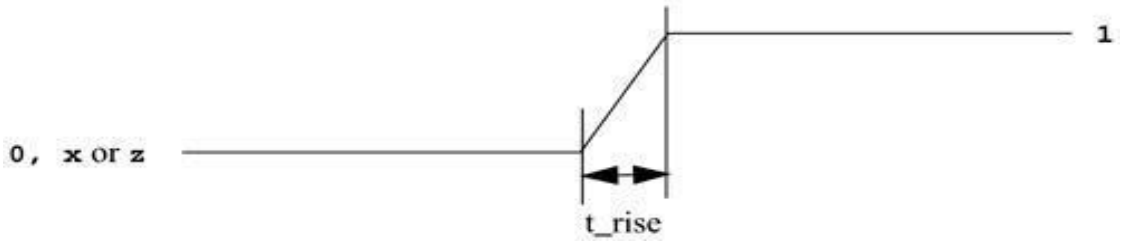
Until now, we described circuits without any delays (i.e., zero delay). In real time circuits, the logic gates have delays associated with them. Gate delays allow the Verilog HDL user to specify delays through the logic circuits.

There are three types of delays from the inputs to the output of a primitive gate as

1. Rise Delay.
2. Fall Delay.
3. Turn-off Delay.

1. Rise delay

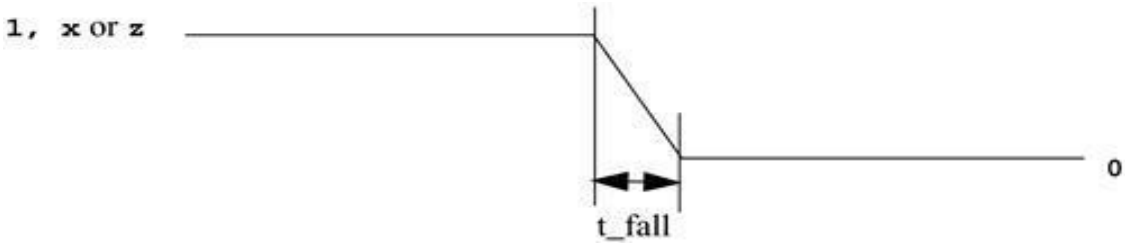
The rise delay is associated with a gate output transition to a 1 from another value (0 or x or z).



2. Fall delay

The fall delay is associated with a gate output transition to a 0 from another value

(1 or x or z).



3. Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (Z) from another value (0 or 1 or X).

NOTE: If the value changes to X, the minimum of the three delays is considered.

Three types of delay specifications are allowed.

- If only one delay is specified, this value is used for all transitions.

- If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.
- If all three delays are specified, they refer to rise, fall, and turn-off delay values.
- If no delays are specified, the default delay value is zero.

Example: Types of gate Delay Specifications

- ```
  and #(delay_time) g1(out, i1, i2);      //Delay of delay_time for all transitions.
  and #(5) a1(out, i1, i2);               //Delay of 5 for all transitions.
  ```

- ```
  and #(rise_val, fall_val) g2(out, i1, i2);   // Rise and Fall Delay Specification.
  and #(4,6) a2(out, i1, i2);                  // Rise= 4, Fall = 6, then Turn-off Delay is 4.
  ```

- ```
  bufif0 #(rise_val, fall_val, turnoff_val) g3 (out, in, control); /*Rise, Fall, and Turn-off Delay Specification*/
  bufif0 #(3,4,5) b1 (out, in, control);       // Rise = 3, Fall = 4, Turn-off = 5.
  ```
  ```

Modeling at DATA FLOW Level

Introduction

Gate-level modeling is very easy to a designer with a basic knowledge of digital logic design.

For small circuits, the gate-level modeling approach works very well, because the number of gates is limited and the designer can instantiate and connect each gate very easily. However, in complex designs a large number of gates are required. Thus, designers can design more effectively if they implement the function at a higher level of abstraction than gate level.

Dataflow modeling provides an efficient way to implement such complex designs. Verilog HDL allows a module to be designed in terms of the data flow between registers.

With gate densities on ICs increasing rapidly, dataflow modeling has assumed great importance. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog HDL description style that combines the concepts of gate-level, data flow, and behavioral design.

In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous Assignment structure

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net.

This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword "assign".

The assignment syntax starts with the keyword assign followed by the signal name which can be either by single signal or cocatenation of different signal nets.

Syntax:

assign <net expression> = [#delay] <expression of different signals or constant values>;

The delay value is an optional and can be used to specify delay on the assign statement.

Continuous assignments have the following rules:

1. The lefthand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.

3. The operands on the right-hand side can be registers or nets or functions.

4. Delay values can be specified for assignments in terms of time units. This feature is similar to specifying delays for gates.

Examples of Continuous Assignment

Continuous assign:

```
assign out = i1 & i2;        //out is a net. i1 and i2 are nets.
```

Continuous assign for vector nets:

```
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
    //addr is a 16-bit vector net addr1 and addr2 are 16-bit vector registers.
```

Concatenation:

Left-hand side is a concatenation of a scalar net and a vector net.

```
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit Continuous Assignment

Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because a net is declared only once.

In the example below, an implicit continuous assignment is contrasted with a regular continuous assignment.

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;

//Same effect is achieved by an implicit continuous assignment
wire out = in1 & in2;
```

Implicit Net Declaration

If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name. If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
wire i1, i2;
assign out = i1 & i2;          /*Note that out was not declared as a wire but an implicit wire
```

declaration for out is done by the simulator*/

Delays

Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side. Three ways of specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay, and net declaration delay.

Regular Assignment Delay:

The first method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword assign. Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out. If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

```
assign #10 out = in1 & in2;          // Delay in a continuous assign
```

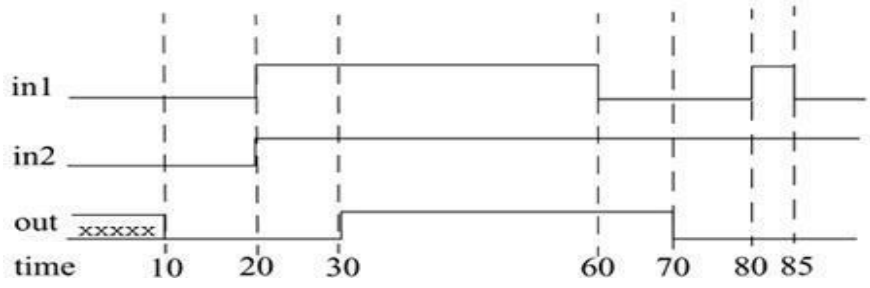


Figure: Delay instantiation in data flow modeling

The above waveform is generated by simulating the above assign statement. It shows the delay on signal out.

Note the following change:

When signals **in1 and in2** go high at time 20, out goes to a high 10 time units later (time = 30).
When **in1** goes low at 60, out changes to low at 70.

However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed. Hence, at the time of recomputation, 10 units after time 80, **in1** is 0. Thus, out gets the value 0.

Implicit Continuous Assignment Delay:

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay
wire #10 out = in1 & in2;
//The above statement has the same effect as the following.

wire out;
assign #10 out = in1 & in2;
```

Net Declaration Delay:

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly. Net declaration delays can also be used in gate-level modeling.

```
//Net Delays
wire # 10 out;
assign out = in1 & in2;
//The above statement has the same effect as the following.
wire out;
assign #10 out = in1 & in2;
```

Expressions, Operands and Operators:

Dataflow modeling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modeling.

Expressions:

Expressions are constructs that combine operators and operands to produce a result.

Eg: Expressions combines operands and operators as follows

```
a ^ b

addr1[20:17] + addr2[20:17]

in1 | in2
```

Operands:

Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories.

```
integer count, final_count;
final_count = count + 1;   //count is an integer operand

real a, b, c;
c = a - b; //a and b are real operands

reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];  //reg1[3:0] and reg2[3:0] are part-select register operands.
```

Operators:

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```
d1 && d2        // && is an operator on operands d1 and d2
!a              // ! is an operator on operand a
B >> 1          // >> is an operator on operands B and 1
```

Operator Types:

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The following table shows the complete listing of operator symbols used in Verilog HDL.

Table: Operator Types and Symbols

Operator Type Performed	Operator Symbol	Operation	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two

Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Let us now discuss each operator type in detail.

Arithmetic Operators:

There are two types of arithmetic operators: **Binary** and **Unary**.

1.Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2;  // D, E, F are integers
A * B               // Multiply A and B. Evaluates to 4'b1100
D / E               // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B               // Add A and B. Evaluates to 4'b0111
B - A               // Subtract A from B. Evaluates to 4'b0001
E ** F              //E to the power F, yields 16
```

If any operand bit has a value x, then the result of the entire expression is x (unknown).

```
in1 = 4'b101x; in2 = 4'b1010;
sum = in1 + in2;    // sum will be evaluated to the value 4'bx
```

Modulus operators (%) produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
13  % 3 // Evaluates to 1
16  % 4 // Evaluates to 0
-7  % 2 // Evaluates      to   -1, takes sign of the first operand
 7 % -2 // Evaluates      to   +1, takes sign of the first operand
```

2.Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand.

Unary + or - operators have higher precedence than the binary + or - operators.

```
-4 // Negative 4
+5 // Positive 5
```

- Negative numbers are represented as 2's complement internally in Verilog HDL.
- It is advisable to use negative numbers only of the type integer or real in expressions.
- Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

Advisable to use integer or real numbers -10 / 5 Evaluates to -2.

Logical Operators:

Logical operators are logical and (**&&**), logical or (**||**) and logical not (**!**).

Operators && and || are binary operators.

Operator ! is a unary operator.

Logical operators follow these conditions:

- Logical operators always evaluate to a single bit value as 0 (false), 1 (true), or x (ambiguous).
- If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
- Logical operators take variables or expressions as operands. Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Logical operations A = 3; B = 0;

```
A && B      // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B      // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A          // Evaluates to 0. Equivalent to not(logical-1)
!B          // Evaluates to 1. Equivalent to not(logical-0)
```

Unknowns:

```
A = 2'b0x; B = 2'b10;
A && B              // Evaluates to x. Equivalent to (x && logical 1)
```

Expressions:

```
(a == 2) && (b == 3)        // Evaluates to 1 if both a == 2 and b == 3 are true.
                            // Evaluates to 0 if either is false.
```

Relational Operators:

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).

- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true, and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.

Eg:

A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

```
A <= B      // Evaluates to a logical 0
A > B       // Evaluates to a logical 1
Y >= X      // Evaluates to a logical 1
Y < Z       // Evaluates to an x
```

Equality Operators:

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==) .

- When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table below lists the operators.
- It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==). The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits.
- However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z.
- The result is 1 if the operands match exactly, including x and z bits.
- The result is 0 if the operands do not match exactly.
- Case equality operators never result in an x.

Table: Equality Operators

Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if x or z in a or b	0, 1, x
a != b	a not equal to b, result unknown if x or z in a or b	0, 1, x
a === b	a equal to b, including x and z	0, 1
a !== b	a not equal to b, including x and z	0, 1

A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx A == B // Results in logical 0

X != Y // Results in logical

1.

X == Z // Results in x.

Z === M // Results in logical 1 (all bits match, including x and z).

Z === N // Results in logical 0 (least significant bit does not match) .

M !== N // Results in logical 1.

Bitwise Operators:

Bitwise operators are negation (~), and (&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand.

Logic tables for the bit-by-bit computation are shown in below Table.

A z is treated as an x in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Table: Truth Tables for Bitwise Operators

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.

X = 4'b1010, Y = 4'b1101 , Z = 4'b10x1

```
~X          // Negation Result is 4'b0101
X & Y       // Bitwise and Result is 4'b1000
X | Y       // Bitwise or Result is 4'b1111
X ^ Y       // Bitwise xor. Result is 4'b0111
X ^~ Y      // Bitwise xnor. Result is 4'b1000
X & Z       // Result is 4'b10x0
```

It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||. Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value.

X = 4'b1010, Y = 4'b0000

```
X | Y       // bitwise operation Result is 4'b1010
X || Y      // logical operation Equivalent to 1 || 0 Result is 1.
```

Reduction Operators:

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).

- Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.
- The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand.
- Reduction operators work bit by bit from right to left.

X = 4'b1010

```
&X          //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X          //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X          //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
    // A reduction xor or xnor can be used for even or odd parity generation of a vector.
```

The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially.

The difference lies in the number of operands each operator takes and also the value of results computed.

Shift Operators:

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<).

- Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift.
- When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

```
// X = 4'b1100
Y = X >> 1;          //Y is 4'b0110. Shift right by one bit. 0 filled in MSB position.
Y = X << 1;          //Y is 4'b1000. Shift left by one bit. 0 filled in LSB position.
Y = X << 2;          //Y is 4'b0000. Shift left two bits.

integer a, b, c;     //Signed data types a = 0;
b = -10;             // 00111...10110 binary
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

Concatenation Operator:

The concatenation operator ({ , }) provides a mechanism to append multiple operands.

The operands must be sized.

- Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
- Concatenations are expressed as operands within braces, with commas separating the operands.
- Operands can be scalar nets or registers, vector nets or registers, bit- select, part-select, or sized constants.

A =1'b1, B= 2'b00, C = 2'b10, D =3'b110

Y = {B, C} // Result Y is 4'b0010

Y = {A, B , C, D , 3'b001} // Result Y is 11'b10010110001

Y = {A, B[0], C[1]} // Result Y is 3'b101

Replication Operator:

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

reg A;

reg [1:0] B, C;

reg [2:0] D;

A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;

Y = { 4{A} } // Result Y is 4'b1111

Y = { 4{A} , 2{B} } // Result Y is 8'b11110000

Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010

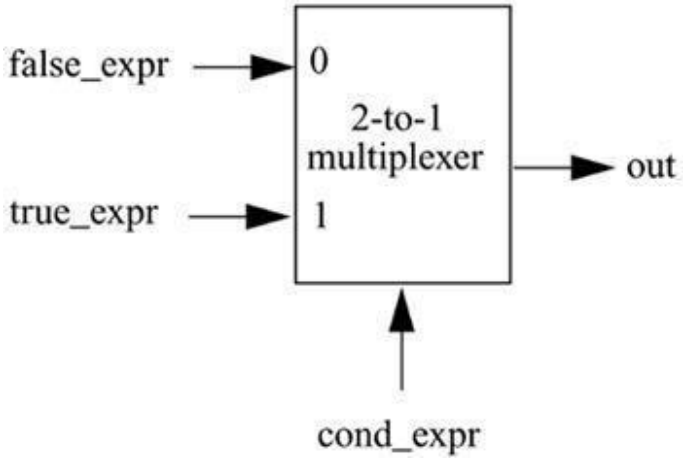
Conditional Operator:

The conditional operator(?:) takes three operands.

Syntax: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated. If the result is x (ambiguous), then both true_expr and false_ expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

Conditional operations can be nested. Each true_expr or false_expr can itself be a conditional operation.
In the example that follows, convince yourself that (A==3) and control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and out as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Behavioral Modeling

Introduction:

Behavioral modeling is the highest level of abstraction in the Verilog HDL.

The abstraction in this modeling is as simple as writing the logic in C language. The designer need is the algorithm of the design, which is the basic information for any design.

The difference between gate level (structural), data flow and behavioural modelling styles is based on the type of concurrent statements used:

- A gate level (structural) design uses only component (gate primitives) instantiation statements.
- A dataflow level design uses only concurrent signal assignment statements.
- A behavioral level design uses only process statements under procedural constructs (Eg: initial and always blocks).

The "initial" Construct:

The initial block is executed only once in the simulation, at time 0. If there is more than one initial block, then all the initial blocks are executed concurrently. The initial construct is used as follows:

Eg1:

```
initial
begin
reset=1'b0;
clk=1'b1;
end
```

Eg2:

```
initial
clk = 1'b1;
```

In Eg1 initial block, there are more than one statements hence they are written between begin and end.

If there is only one statement (as mentioned in Eg2) then there is no need to put begin and end.

The "always" construct:

The always block starts at time 0, and keeps on executing all the simulation times. It works like a infinite loop. It is generally used to model a functionality that is continuously repeated.

Eg3:

```
always
 #5 clk = ~clk;
 initial
clk = 1'b0;
```

The above code generates a clock signal clk, with a time period of 10 units. The initial blocks initiates the clk value to 0 at time 0. Then after every 5 units of time it toggled, hence we get a time period of 10 units. This is the way in general used to generate a clock signal for use in test benches.

Eg4:

```
always@(posedge clk, negedge reset) begin
a = b + c;
d = 1'b1;
end
```

In the above example, the always block will be executed whenever there is a positive edge in the clk signal, or there is negative edge in the reset signal. This type of always is generally used in implement a Finite State machine(FSM), which has a reset signal.

Eg5:

```
always @(b,c,d)
begin
a = ( b + c )*d;
e = b | c;
end
```

In the above example, whenever there is a change in b, c, or d the always block will be executed. Here the list b, c, and d is called the sensitivity list.

Note: In the Verilog 2000, we can replace always @(b,c,d) with always @(*), it is equivalent to include all input signals, used in the always block. This is very useful when always blocks are used for implementing the combination logic.

OPERATIONS AND ASSIGNMENTS:

The design description at the behavioral level is done through a sequence of assignments. These are called 'procedural assignments' – in contrast to the continuous assignments at the data flow level. Though it appears similar to the assignments at the data flow level discussed in the last chapter, the two are different.

The procedure assignment is characterized by the following:

- The assignment is done through the "=" symbol (or the "<=" symbol) as was the case with the continuous assignment earlier.
- An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "=" symbol.

  Example: N = ~N;
- Here the content of reg N is complemented and assigned to the reg N itself.
- The operation on the right can involve operands and operators. The operands can be of different types. logical variables, numbers(real or integer) and so on.

//Verilog code:

```
module ctr_wt(a,clk,N);
input clk;
input[3:0]N;
output[3:0]a;
```

```
reg[3:0]a;
initial
a=4'b1111;
always
begin @(negedge clk)
a=(a==N)?4'b0000:(a+1'b1);
end
endmodule
//Test Bench
module tst_ctr_wt;
reg clk;
reg[3:0]N;
wire[3:0]a;
ctr_wt c1(a,clk,N);
initial
begin
clk=0; N=4'b1111;
end
always
#2 clk=~clk;
initial #35 $stop;
initial $monitor($time,"clk=%h, N=%b, a=%b", clk, N, a,);
endmodule
```

Conditional (if-else) Statement:

The condition (if-else) statement is used to make a decision whether a statement is executed or not. The keywords if and else are used to make conditional statement. The conditional statement can appear in the following forms.

```
if ( condition_1 ) statement_1;
-------------------------------
if ( condition_2 ) statement_2;
else
statement_3;
-----------------------------------
if ( condition_3 ) statement_4;
else if ( condition_4 ) statement_5;
else
statement_6;
-------------------------------------
if ( condition_5 )
begin
statement_7;
statement_8;
end
else
begin
statement_9;
statement_10;
end
```

Conditional (if-else) statement usage is similar to that if-else statement of C programming language, except that parenthesis are replaced by begin and end.

Case Statement:

The case statement is a multi-way decision statement that tests whether an expression matches one of the expressions and branches accordingly. Keywords case and endcase are used to make a case statement. The case statement syntax is as follows.

```
case (expression)
case_item_1: statement_1;
case_item_2: statement_2;
case_item_3: statement_3;
...
...
default: default_statement;
endcase
```

If there are multiple statements under a single match, then they are grouped using begin, and end keywords. The default item is optional.

Case statement with don't cares: casez and casex:

casez treats high-impedance values (z) as don't cares. casex treats both high-impedance (z) and unknown (x) values as don't cares. Don't-care values (z values for casez, z and x values for casex) in any bit of either the case expression or the case items shall be treated as don't-care conditions during the comparison, and that bit position shall not be considered. The don't cares are represented using the ? mark.

For Loop:

The For loop is defined using the keyword for. The execution of for loop block is controlled by a three step process, as follows:

- Executes an assignment, normally used to initialize a variable that controls the number of times the for block is executed.
- Evaluates an expression, if the result is false or z or x, the for-loop shall terminate, and if it is true, the for-loop shall execute its block.
- Executes an assignment normally used to modify the value of the loop-control variable and then repeats with second step.

Note that the first step is executed only once.

```
initial
begin
a = 20;
for (i = 0; i < a; i = i + 1, a = a - 1)
end
```

//The above example produces the same result as the example used to illustrate the functionality of the while loop.

Examples:

```
//Implementation of a multiplexer 4X1.
module mux4_1 (out, in0, in1, in2, in3, s0, s1);
output out;
// out is declared as reg
reg out;
// out is declared as reg, because we will do a procedural assignment to it.
input in0, in1, in2, in3, s0, s1;
// always @(*) is equivalent to always @( in0, in1, in2, in3, s0, s1 )
always @(*) begin
case ({s1,s0})
```

```
2'b00: out = in0;
2'b01: out = in1;
2'b10: out = in2;
2'b11: out = in3;
default: out = 1'bx;
endcase
end
endmodule
--------------------------------------------------------------------------
//Implementation of a full adder.
module full_adder (sum, c_out, in0, in1, c_in);
output sum, c_out;
reg sum, c_out;
input in0, in1, c_in;
always @( in0, in1, c_in)
{c_out, sum} = in0 + in1 + c_in;
endmodule
--------------------------------------------------------------------------
//Implementation of a 8-bit upcounter. It is a sequential circuit.
module upcounter (count, reset, clk );
output [7:0] count;
reg [7:0] count;
input reset, clk;
// consider reset as active low signal
always @( posedge clk, negedge reset)
begin
if(reset == 1'b 0)
count <= 8'b 00000000;
else
```

```
count <= count + 8'b 00000001;
end
endmodule
```

Implementation of a 8-bit upcounter is a very good example, which explains the advantage of behavioral modeling. Just imagine how difficult it will be implementing a 8-bit upcounter using gate-level modeling.

In the above example the incrementation occurs on every positive edge of the clock. When count becomes 8'b11111111, the next increment will make it 8'b00000000.

Verilog code for SR flip-flop – All Modeling styles

- Describe the SR-flip flop using the three levels of abstraction – Gate leve, Dataflow and Behavioral Modeling.
- Generate the RTL schematic for the SR flip flop.
- Write the testbench.
- Generate simulated waveforms.

Contents

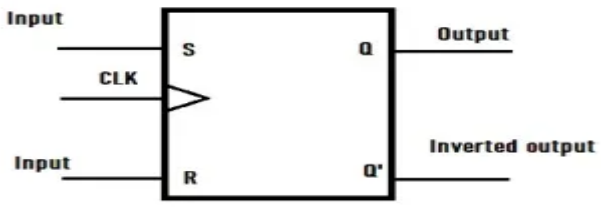
1. What is an SR flip flop?

Flip Flops are the basic building blocks of sequential circuits. They are memory elements made by connecting logic gates. They can shift between two states (0 and 1) and hence, formally called bi-stable multivibrator.

Did you know calculators and computers use flip flops to store data?

Each flip flop can store one bit of data. Thus, a combination of flip flops makes it possible to store a large amount of data.

An SR Flip Flop is short for Set-Reset Flip Flop. It has two inputs S(Set) and R(Reset) and two outputs Q(normal output) and Q'(inverted output).



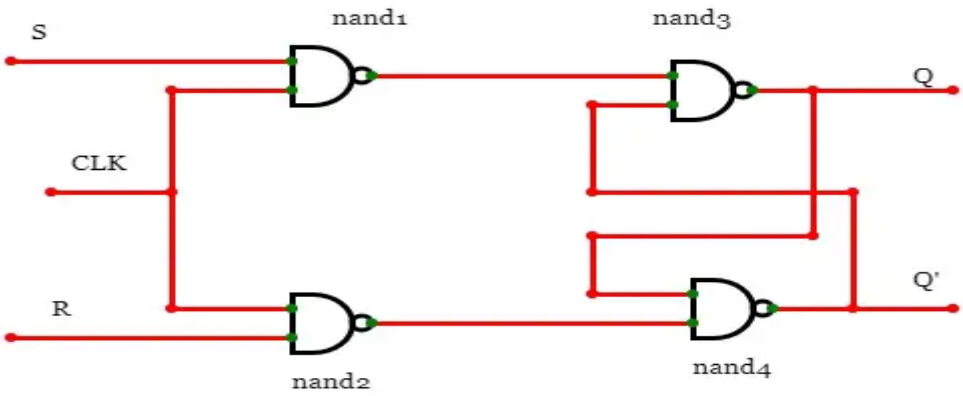
SR flip flop logic symbol

As we proceed, we will see how to write Verilog code for SR Flip Flop using different levels of abstraction.

2.Gate Level Modeling

With the assistance of a logic diagram, we will be able to know the essential logic gates needed to build a circuit. Verilog provides us with gate primitives, which help us create a circuit by connecting basic logic gates. Gate level Modeling enables us to describe thecircuit using these gate primitives.

Given below is the logic diagram of an SR Flip Flop.



SR flip flop logic circuit

From the above circuit, it is clear we need to interconnect four NAND gates in a specific fashion to obtain an SR flip flop. Let's see how we can do that using the gate-level modeling style.

2.1 Gate level Modeling of SR flip flop using Verilog HDL

```
module srff_gate(q, qbar, s, r, clk);
input s,r,clk;
output q, qbar;
wire nand1_out;    // output of nand1
wire nand2_out;    // output of nand2
nand (nand1_out,clk,s);
nand (nand2_out,clk,r);
nand (q,nand1_out,qbar);
nand (qbar,nand2_out,q);
endmodule
```

3.Dataflow Modeling

Gate level Modeling works for circuits having less number of logic gates. But, when the number of logic gates increases, the circuit complexity increases. Hence, it becomes challenging to instantiate a large number of gates and interconnections.

Instead of knowing the logic circuit, we can describe the circuit using its logic expression. For that, we use a higher level of abstraction, which is Dataflow Modeling. This style describes how data flows from input to output using logic equations.

Before moving on to the coding part, let's see the characteristic equation of the SR flip flop:

Q(next) = S +R'Q(previous)

Let's see how we code in this equation using dataflow Modeling.

3.1 Dataflow Modeling of SR Flip Flop:

```
module srff_dataflow(q,qbar,s,r,clk);
input s,r,clk;
output q, qbar;
assign q = clk? (s + ((~r) & q)) : q;        //using conditional operator
assign qbar = ~q;
endmodule
```

But…there's a problem.

Flip flops are edge-triggered circuits. When we use a conditional operator, the statement is not executed at the clock edges(HIGH to LOW or LOW to HIGH) but the clock level(HIGH and LOW). Hence, the dataflow model of SR flip flop will work only as a latch. And not as an authentic flip-flop that triggers on clock edges.

Therefore, we prefer the highest level of abstraction (Behavioral Modeling) for describing sequential circuits.

4.Behavioral Modeling

Behavioral Modeling is the highest level of abstraction in Verilog HDL. We can describe the circuit by just knowing how it works.

Moreover, there's additional good news! We do not need to know the logic circuit or logic equation. We just need a simple truth table.

We need to know how SR flip flop behaves. Hence let's go through the truth table.

CLOCK EDGE	S	R	Q	Q'	State
Positive/Negative	0	0	Q	Q'	No change
Positive/Negative	0	1	0	1	Reset State
Positive/Negative	1	0	1	0	Set State
Positive/Negative	1	1	X	X	Invalid

Using the described behaviour, we can now start coding.

4.1 Behavioral Modeling of SR Flip Flip

```
module srff_behave(q,qbar,s,r,clk);
input s,r,clk;
output reg q, qbar;
always@(posedgeclk)
begin
if(s == 1)
begin
q = 1;
qbar = 0;
end
else if(r == 1)
begin
q = 0;
qbar =1;
end
else if(s == 0 & r == 0)
begin
q <= q;
qbar<= qbar
end
end
endmodule
```

5.Testbench

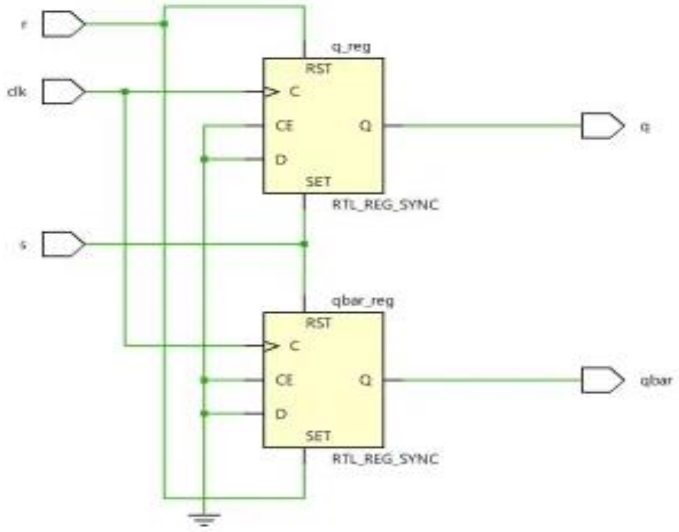
A **testbench** is an HDL module that is used to test another module, called the *device under test* (*DUT*). The test bench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Ler's see how we can write the testbench for SR flip flop.

```
//test bench for srff_behave flip flop
//1. Declare module and ports
module srff_test;
reg S,R, CLK;
wire Q, QBAR;
//2. Instantiate the module we want to test. We have instantiated the srff_behavior.
srff_behavie DUT(.q(Q), .qbar(QBAR), .s(S), .r(R), .clk(CLK)); // instantiation by port name.
//3. apply test vectors
initial clk=0;
always@(clk) #5 clk = ~clk;
initial
begin
 S= 1; R= 0;
 #10 S= 0; R= 1;
 #10 S= 0; R= 0;
 #10 S= 1; R= 1;
 #20 $finish;
end
//4. Monitor testbench ports
initial $monitor($time,"CLK = %b, S = %b, R = %b, Q = %b, QBAR = %b", CLK, S, R, Q, QBAR);
endmodule
```

6.RTL Schematic:

Here's how the RTL schematic will look if we peek into the elaborate design of the behavioral model of SR flip flop.



SR flip flop RTL Schematic

7.Simulated Waveform:

We can verify the functional correctness of described SR flip-flop by simulation. The simulated waveform of SR flip flop is given below:

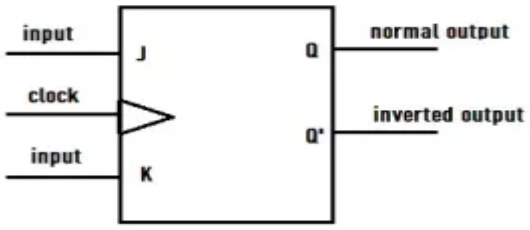
The simulated waveform of SR flip flop

Verilog code for JK flip-flop – All modeling styles

Contents

What is a JK flip flop?

- Flip-flops are fundamental building blocks of sequential circuits. A flip flop can store one bit of data. Hence, it is known as a memory cell. Since they work on the application of a clock signal, they come under the category of synchronous circuits.
- The J-K flip-flop is the most versatile of the basic flip flops.
- The JK flip flop is a gated SR flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic 1.
- Due to this additional clocked input, a JK flip-flop has four possible input combinations, “logic 1”, “logic 0”, “no change” and “toggle”.



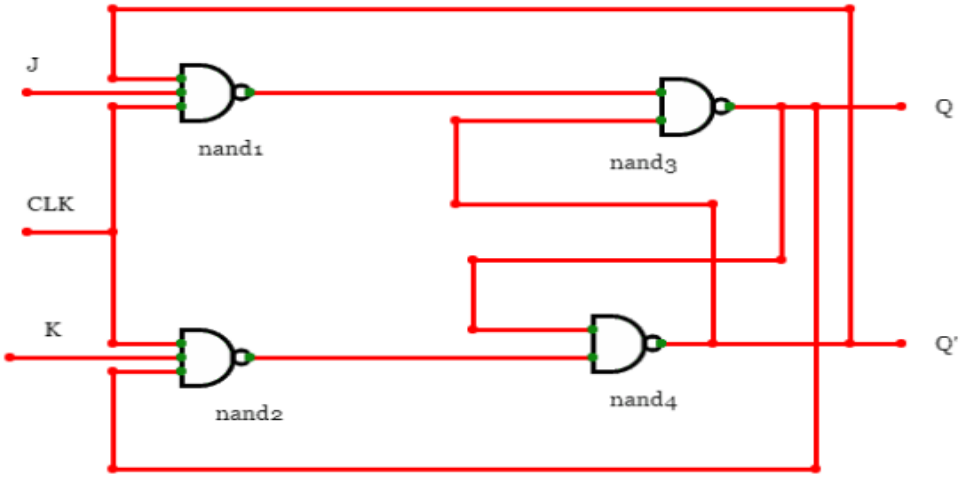
JK flip flop logic symbol

As we proceed, we will see how to write Verilog code for SR Flip Flop using different levels of abstraction.

2.Gate-Level Modeling

With the help of a logic diagram, we will be able to know the essential logic gates needed to build a circuit. Verilog provides us with gate primitives, which help us create a circuit by connecting basic logic gates.

Gate level modeling enables us to describe the circuit using these gate primitives.



JK flip flop logic circuit

2.1 Gate Level Modeling of JK Flip Flop:

```
module jkff_gate(q,qbar,clk,j,k);
input j,k,clk;
output q,qbar;
wire nand1_out;    // output from nand1
wire nand2_out;    // output from nand2
nand(nand1_out, j,clk,qbar);
nand(nand2_out, k,clk,q);
nand(q,qbar,nand1_out);
nand(qbar,q,nand2_out);
endmodule
```

3.Dataflow Modeling

Describing a flip flop using dataflow modeling is not applicable. Flip flops are supposed to work on edge-triggered clocks. In dataflow modeling, it is not possible to construct an edge-

triggered flip flop. It works more like a latch. Also, when modeling sequential circuits with dataflow, it can sometimes result in an unpredictable output during a simulation.

Hence, we prefer the highest level of abstraction (behavioral modeling) to describe sequential circuits like flip flops.

4.Behavioral Modeling

Behavioral modeling is the highest level of abstraction. Unlike gate and dataflow modeling, behavior modeling does not demand knowing logic circuits or logic equations.

As a designer, we just need to know the algorithm (behavior) of how we want the system to work. This type of modeling is simple since it does not involve using complex circuitry. A simple truth table will help us describe the design.

CLOCK	J	K	Q	Q'	State
Positive edge	0	0	Q	Q'	No change
Positive edge	0	1	0	1	Reset
Positive edge	1	0	1	0	Set
Positive edge	1	1	$(Q_{prev})'$	Q_{prev}	Toggle

4.1 Behavioral Modeling of JK Flip Flop

```
module jkff_behave(clk,j,knq,qbar);
input clk,j,k;
output reg q,qbar;
always@(posedge clk)
begin
if(k = 0)
begin
q <= 0;
qbar <= 1;
end
always@(posedge clk)
begin
```

```
 if(k = 0)
  begin
   q <= 0;
   qbar <= 1;
  end
 else if(j = 1)
  begin
   q <= 0;
   qbar <= 0;
  end
 else if(j = 0 & k = 0)
  begin
   q <= q;
   qbar <= qbar;
  end
 else if(j = 1 & k = 1)
  begin
   q <= ~q;
   qbar <= ~qbar;
  end
end
endmodule
```

5.Testbench

A testbench is an HDL module that is used to test another module, called the *device under test* (*DUT*). The test bench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

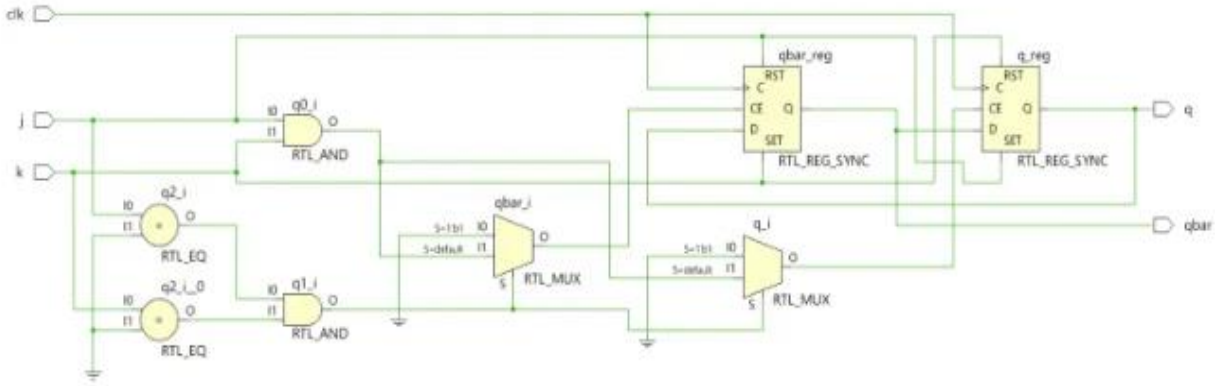
Ler's see how we can write the testbench for JK flip flop.

```
//test bench for JK flip flop
//1. Declare module and ports
module jkff_test;
```

```
reg J,K, CLK;
wire Q, QBAR;
//2. Instantiate the module we want to test. We have instantiated the jkff_behavior.
jkff_behave dut(.q(Q), .qbar(QBAR), .j(J), .k(K), .clk(CLK)); // instantiation by port name.
//3. apply test vectors
initial
  clk=0;
always@(clk)
 #5 clk = ~clk;
initial
begin
 J= 1; K= 0;
 #10; J= 0; K= 1;
 #10; J= 0; K= 0;
 #10; J= 1; K=1;
#20 $finish;
end
//4. Monitor TB ports
initial $monitor($time, "CLK = %b, J = %b, K = %b, Q = %b, QBAR = %b", CLK, J, K, Q, QBAR);
endmodule
```

6.RTL Schematic

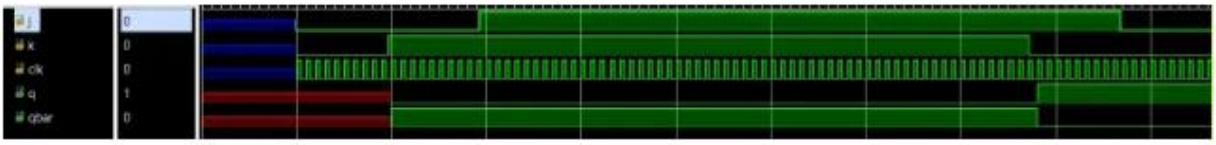
Here's how the RTL Schematic will look if we peek into the elaborate design of the behavioral model of the JK-flip flop.



RTL schematic of JK flip flop

7.Simulated Waveform

We can verify the functional correctness of described JK flip-flop by simulation. The simulated waveform of JK flip flop is given below:



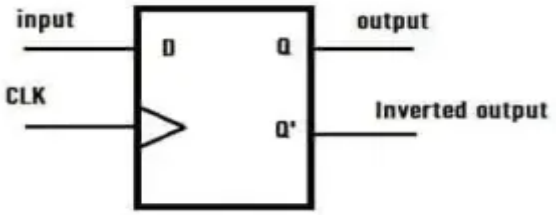
Simulated wave of JK flip flop

Verilog code for D flip-flop – All modeling styles

What is D flip flop?

A flip flop can store one bit of data. Hence, it is known as a memory cell. Flip-flops are synchronous circuits since they use a clock signal. Using flip flops, we build complex circuits such as RAMs, Shift Registers, etc.

A D flip-flop stands for data or delay flip-flop. The outputs of this flip-flop are equal to the inputs.

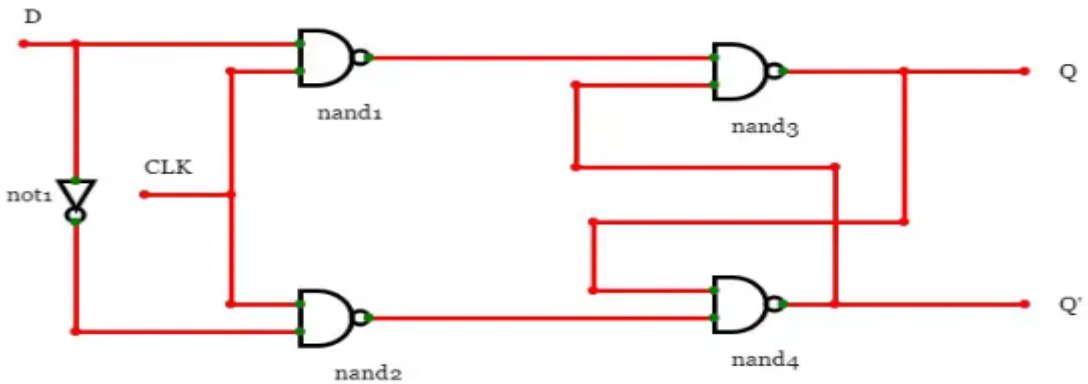


D flip flop Symbol

As we proceed, we will see how we can design a D flip flop using different levels of abstraction

1.Gate level modeling

Gate level modeling uses primitive gates available in Verilog to build circuits. Hence, we need to know the logic diagram of the circuit we want to design.



The logic circuit of D Flip Flop

From the above circuit, we can see that we need four NAND gates and one NOT gate to construct a D-flip flop in gate-level modeling.

1.1 Gate level Modeling of D flip flop

```
module d_ff_gate(q,qbar,d,clk);
input d,clk;
output q, qbar;
wire dbar,x,y;
not not1(dbar,d);
nand nand1(x,clk,d);
nand nand2(y,clk,dbar);
nand nand3(q,qbar,y);
nand nand4(qbar,q,x);
endmodule
```

2.Behavioral Modeling

2.1 Behavioral Modeling of D flip flop with Synchronous Clear:

Clear Input in Flip flop:

All hardware systems should have a pin to clear everything and have a fresh start. It applies to flip flops too. Hence, we will include a clear pin that forces the flip flop to a state where Q = 0 and Q' = 1 despite whatever input we provide at the D input. This clear input becomes handy when we tie up multiple flip flops to build counters, shift registers, etc.

For synchronous clear, the output will reset at the triggered edge(positive edge in this case) of the clock after the clear input is activated.

Here's the code:

```
module dff_behavioral(d,clk,clear,q,qbar);
input d, clk, clear;
output reg q, qbar;
always@(posedge clk)
begin
if(clear== 1)
q <= 0;
```

```
qbar <= 1;
else
q <= d;
qbar = !d;
end
endmodule
```

2.2 Behavioral Modeling of D flip flop with Asynchronous Clear

For asynchronous clear, the clear signal is independent of the clock. Here, as soon as clear input is activated, the output reset.

This can be achieved by adding a clear signal to the sensitivity list. Hence we write our code as:

```
module dff_behavioral(d,clk,clear,q,qbar);
input d, clk, clear;
output reg q, qbar;
always@(posedge clk or posedge clear)
begin
if(clear== 1)
q <= 0;
qbar <= 1;
else
q <= d;
qbar = !d;
end
endmodule
```

3.Testbench

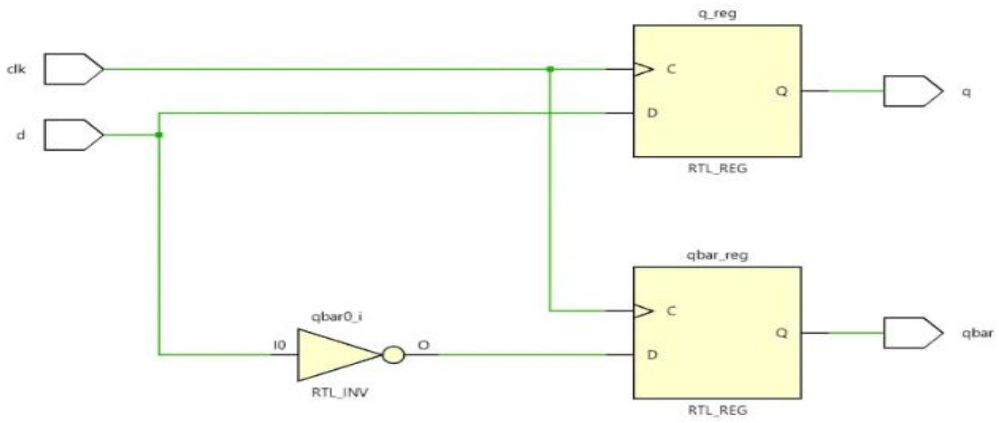
A **testbench** is an HDL module that is used to test another module, called the *device under test* (*DUT*). The test bench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced. The input and desired output patterns are called *test vectors*.

Let’s see how we can write a test bench for D-flip flop by following step by step instruction

```
//test bench for d flip flop
//1. Declare module and ports
module dff_test;
reg D, CLK,reset;
wire Q, QBAR;
//2. Instantiate the module we want to test. We have instantiated the dff_behavior.
dff_behavior dut(.q(Q), .qbar(QBAR), .clear(reset), .d(D), .clk(CLK));
//3. apply test vectors
initial   clk=0;
always@(clk)  #5 clk = ~clk;
initial begin
 reset=1; D <= 0;
 #10; reset=0; D <= 1;
 #10; D <= 0;
 #10; D <= 1;
#20 $finish;
end
//4. Monitor TB ports
initial $monitor($time, "CLK = %b, D = %b, reset = %b, Q = %b, QBAR = %b", CLK, D, reset, Q,
QBAR);
endmodule
```

4. RTL Schematic

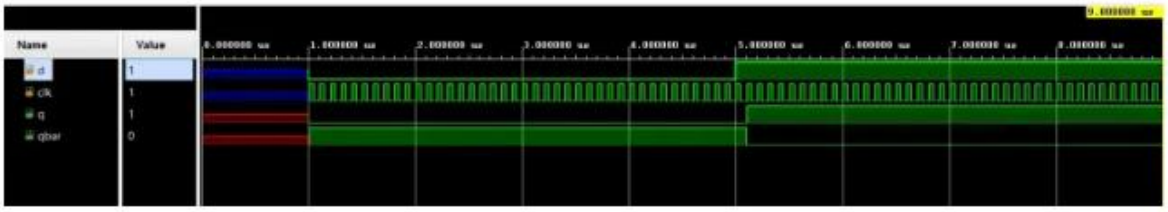
Here’s how the RTL Schematic will look if we peek into the elaborate design of the behavioral model of the D-flip flop without clear input.



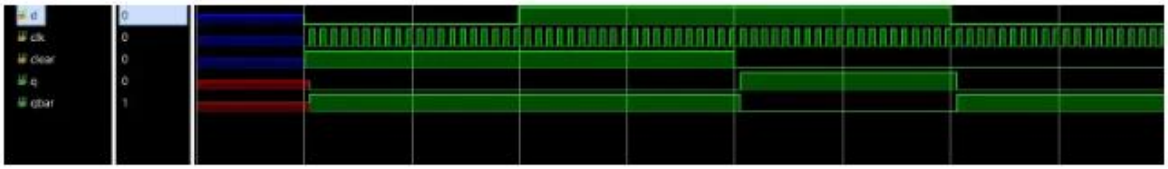
RTL schematic of D flip flop

5.Simulated Waveforms

D flip flop Without Reset



Simulated waveform of D flip flop without clear



Simulated waveform of D flip flop with synchronous clear

In this waveform, we can see that the **Q** and **Q'** will be reset state at the positive cycle after **clear** is activated

Simulated waveform of D flip flop with asynchronous clear

In this waveform, we can see that the **Q** and **Q'** will be in the reset state as soon as **clear** is activated.