Lecture notes on
Digital Electronics & Microprocessor

5th Semester

Diploma in Electrical and Electronics Engineering

Prepared By:

S Sanjay Kumar Patra

Department of Electrical and Electronics Engineering

Vedang Institute of Technology

Bhubaneswar, Khurda

BASICS OF DIGITAL ELECTRONICS

1.1 Binary, Octal, Hexadecimal number systems and compare with Decimal system.
1.2 Binary addition, subtraction, Multiplication and Division.
1.3 1's complement and 2's complement numbers for a binary number
1.4 Subtraction of binary numbers in 2's complement method.
1.5 Use of weighted and Un-weighted codes & write Binary equivalent number for a number in 8421, Excess-3 and Gray Code and vice-versa.
1.6 Importance of parity Bit.
1.7 Logic Gates: AND, OR, NOT, NAND, NOR and EX-OR gates with truth table.
1.8 Realize AND, OR, NOT operations using NAND, NOR gates.
1.9 Different postulates and De-Morgan's theorems in Boolean algebra.
1.10 Use Of Boolean Algebra For Simplification Of Logic Expression
1.11 Karnaugh Map For 2,3,4 Variable, Simplification Of SOP And POS Logic Expression Using K-Map.

COMBINATIONAL LOGIC CIRCUITS

2.1 Give the concept of combinational logic circuits.
2.2 Half adder circuit and verify its functionality using truth table.
2.3 Realize a Half-adder using NAND gates only and NOR gates only.
2.4 Full adder circuit and explain its operation with truth table.
2.5 Realize full-adder using two Half-adders and an OR – gate and write truth table
2.6 Full subtractor circuit and explain its operation with truth table.
2.7 Operation of 4 X 1 Multiplexers and 1 X 4 demultiplexer
2.8 Working of Binary-Decimal Encoder &3 X 8 Decoder.
2.9 Working of Two bit magnitude comparator.

SEQUENTIAL LOGIC CIRCUITS

3.1 Give the idea of Sequential logic circuits.
3.2 State the necessity of clock and give the concept of level clocking and edge triggering,
3.3 Clocked SR flip flop with preset and clear inputs.
3.5 Construct level clocked JK flip flop using S-R flip-flop and explain with truth table
3.6 Concept of race around condition and study of master slave JK flip flop.
3.7 Give the truth tables of edge triggered D and T flip flops and draw their symbols.
3.8 Applications of flip flops.
3.9 Define modulus of a counter
3.10 4-bit asynchronous counter and its timing diagram.
3.11 Asynchronous decade counter
3.12 4-bit synchronous counter.
3.13 Distinguish between synchronous and asynchronous counters.
3.14 State the need for a Register and list the four types of registers.
3.15 Working of SISO, SIPO, PISO, PIPO Register with truth table using flip flo

8085 MICROPROCESSOR

4.1 Introduction to Microprocessors, Microcomputers
4.2 Architecture of Intel 8085A Microprocessor and description of each block.
4.3 Pin diagram and description.
4.4 Stack, Stack pointer & stack top
4.5 Interrupts
4.6 Opcode& Operand,
4.7 Differentiate between one byte, two byte & three byte instruction with example.

4.8 Instruction set of 8085 example
4.9 Addressing mode
4 .10 Fetch Cycle, Machine Cycle, Instruction Cycle, T-State
4.11 Timing Diagram for memory read, memory write, I/O read, I/O write
4.12 Timing Diagram for 8085 instruction
4.13 Counter and time delay.
4.14 Simple assembly language programming of 8085

INTERFACING AND SUPPORT CHIPS

5.1 Basic Interfacing Concepts, Memory mapping & I/O mapping
5.2 Functional block diagram and description of each block of Programmable peripheral interface Intel 8255 ,
5.3 Application using 8255: Seven segment LED display, Square wave generator, Traffic light Controller

Unit-1

Number System

When we type some letters or words, the computer translates them in numbers as computers can understand only numbers. A computer can understand the positional number system where there are only a few symbols called digits and these symbols represent different values depending on the position they occupy in the number.

The value of each digit in a number can be determined using −

- The digit
- The position of the digit in the number
- The base of the number system (where the base is defined as the total number of digits available in the number system)

Decimal Number System

The number system that we use in our day-to-day life is the decimal number system. Decimal number system has base 10 as it uses 10 digits from 0 to 9. In decimal number system, the successive positions to the left of the decimal point represent units, tens, hundreds, thousands, and so on.

Each position represents a specific power of the base (10). For example, the decimal number 1234 consists of the digit 4 in the units position, 3 in the tens position, 2 in the hundreds position, and 1 in the thousands position. Its value can be written as

(1 x 1000)+ (2 x 100)+ (3 x 10)+ (4 x l)

$(1 \times 10^3)+ (2 \times 10^2)+ (3 \times 10^1)+ (4 \times l0^0)$

1000 + 200 + 30 + 4

1234

As a computer programmer or an IT professional, you should understand the following number systems which are frequently used in computers.

S.No.	Number System and Description
1	**Binary Number System** Base 2. Digits used : 0, 1
2	**Octal Number System** Base 8. Digits used : 0 to 7
3	**Hexa Decimal Number System** Base 16. Digits used: 0 to 9, Letters used : A- F

Binary Number System

Characteristics of the binary number system are as follows −

- Uses two digits, 0 and 1
- Also called as base 2 number system
- Each position in a binary number represents a **0** power of the base (2). Example 2^0
- Last position in a binary number represents a **x** power of the base (2). Example 2^x where **x** represents the last position - 1.

Example

Binary Number: 10101_2

Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	10101_2	$((1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	10101_2	$(16 + 0 + 4 + 0 + 1)_{10}$

Step 3	10101_2	21_{10}

Note − 10101_2 is normally written as 10101.

Octal Number System

Characteristics of the octal number system are as follows −

- Uses eight digits, 0,1,2,3,4,5,6,7
- Also called as base 8 number system
- Each position in an octal number represents a **0** power of the base (8). Example 8^0
- Last position in an octal number represents a **x** power of the base (8). Example 8^x where **x** represents the last position - 1

Example
Octal Number: 12570_8
Calculating Decimal Equivalent −

Step	Octal Number	Decimal Number
Step 1	12570_8	$((1 \times 8^4) + (2 \times 8^3) + (5 \times 8^2) + (7 \times 8^1) + (0 \times 8^0))_{10}$
Step 2	12570_8	$(4096 + 1024 + 320 + 56 + 0)_{10}$
Step 3	12570_8	5496_{10}

Note − 12570_8 is normally written as 12570.

Hexadecimal Number System

Characteristics of hexadecimal number system are as follows −

- Uses 10 digits and 6 letters, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Letters represent the numbers starting from 10. A = 10. B = 11, C = 12, D = 13, E = 14, F = 15
- Also called as base 16 number system
- Each position in a hexadecimal number represents a **0** power of the base (16). Example, 16^0
- Last position in a hexadecimal number represents a **x** power of the base (16). Example 16^x where **x** represents the last position - 1

Example
Hexadecimal Number: $19FDE_{16}$
Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (F \times 16^2) + (D \times 16^1) + (E \times 16^0))_{10}$
Step 2	$19FDE_{16}$	$((1 \times 16^4) + (9 \times 16^3) + (15 \times 16^2) + (13 \times 16^1) + (14 \times 16^0))_{10}$
Step 3	$19FDE_{16}$	$(65536+ 36864 + 3840 + 208 + 14)_{10}$
Step 4	$19FDE_{16}$	106462_{10}

Note − $19FDE_{16}$ is normally written as 19FDE.

Number Conversion

There are many methods or techniques which can be used to convert numbers from one base to another. In this chapter, we'll demonstrate the following −

- Decimal to Other Base System

- Other Base System to Decimal
- Other Base System to Non-Decimal
- Shortcut method - Binary to Octal
- Shortcut method - Octal to Binary
- Shortcut method - Binary to Hexadecimal
- Shortcut method - Hexadecimal to Binary

Decimal to Other Base System
Step 1 − Divide the decimal number to be converted by the value of the new base.
Step 2 − Get the remainder from Step 1 as the rightmost digit (least significant digit) of the new base number.
Step 3 − Divide the quotient of the previous divide by the new base.
Step 4 − Record the remainder from Step 3 as the next digit (to the left) of the new base number.
Repeat Steps 3 and 4, getting remainders from right to left, until the quotient becomes zero in Step 3.
The last remainder thus obtained will be the Most Significant Digit (MSD) of the new base number.
Example
Decimal Number: 29_{10}
Calculating Binary Equivalent −

Step	Operation	Result	Remainder
Step 1	29 / 2	14	1
Step 2	14 / 2	7	0
Step 3	7 / 2	3	1
Step 4	3 / 2	1	1
Step 5	1 / 2	0	1

As mentioned in Steps 2 and 4, the remainders have to be arranged in the reverse order so that the first remainder becomes the Least Significant Digit (LSD) and the last remainder becomes the Most Significant Digit (MSD).
Decimal Number : 29_{10} = Binary Number : 11101_2.

Other Base System to Decimal System
Step 1 − Determine the column (positional) value of each digit (this depends on the position of the digit and the base of the number system).
Step 2 − Multiply the obtained column values (in Step 1) by the digits in the corresponding columns.
Step 3 − Sum the products calculated in Step 2. The total is the equivalent value in decimal.
Example
Binary Number: 11101_2
Calculating Decimal Equivalent −

Step	Binary Number	Decimal Number
Step 1	11101_2	$((1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0))_{10}$
Step 2	11101_2	$(16 + 8 + 4 + 0 + 1)_{10}$
Step 3	11101_2	29_{10}

Binary Number : 11101_2 = Decimal Number : 29_{10}

Other Base System to Non-Decimal System
Step 1 − Convert the original number to a decimal number (base 10).
Step 2 − Convert the decimal number so obtained to the new base number.
Example
Octal Number : 25_8

Calculating Binary Equivalent −
Step 1 - Convert to Decimal

Step	Octal Number	Decimal Number
Step 1	25_8	$((2 \times 8^1) + (5 \times 8^0))_{10}$
Step 2	25_8	$(16 + 5)_{10}$
Step 3	25_8	21_{10}

Octal Number : 25_8 = Decimal Number : 21_{10}
Step 2 - Convert Decimal to Binary

Step	Operation	Result	Remainder
Step 1	21 / 2	10	1
Step 2	10 / 2	5	0
Step 3	5 / 2	2	1
Step 4	2 / 2	1	0
Step 5	1 / 2	0	1

Decimal Number : 21_{10} = Binary Number : 10101_2
Octal Number : 25_8 = Binary Number : 10101_2
Shortcut Method — Binary to Octal
Step 1 − Divide the binary digits into groups of three (starting from the right).
Step 2 − Convert each group of three binary digits to one octal digit.
Example
Binary Number : 10101_2
Calculating Octal Equivalent −

Step	Binary Number	Octal Number
Step 1	10101_2	010 101
Step 2	10101_2	2_8 5_8
Step 3	10101_2	25_8

Binary Number : 10101_2 = Octal Number : 25_8
Shortcut Method — Octal to Binary
Step 1 − Convert each octal digit to a 3-digit binary number (the octal digits may be treated as decimal for this conversion).
Step 2 − Combine all the resulting binary groups (of 3 digits each) into a single binary number.
Example
Octal Number : 25_8
Calculating Binary Equivalent −

Step	Octal Number	Binary Number
Step 1	25_8	2_{10} 5_{10}
Step 2	25_8	010_2 101_2
Step 3	25_8	010101_2

Octal Number : 25_8 = Binary Number : 10101_2
Shortcut Method — Binary to Hexadecimal

Step 1 − Divide the binary digits into groups of four (starting from the right).
Step 2 − Convert each group of four binary digits to one hexadecimal symbol.
Example
Binary Number : 10101_2
Calculating hexadecimal Equivalent −

Step	Binary Number	Hexadecimal Number
Step 1	10101_2	0001 0101
Step 2	10101_2	1_{10} 5_{10}
Step 3	10101_2	15_{16}

Binary Number : 10101_2 = Hexadecimal Number : 15_{16}
Shortcut Method - Hexadecimal to Binary
Step 1 − Convert each hexadecimal digit to a 4-digit binary number (the hexadecimal digits may be treated as decimal for this conversion).
Step 2 − Combine all the resulting binary groups (of 4 digits each) into a single binary number.
Example
Hexadecimal Number : 15_{16}
Calculating Binary Equivalent −

Step	Hexadecimal Number	Binary Number
Step 1	15_{16}	1_{10} 5_{10}
Step 2	15_{16}	0001_2 0101_2
Step 3	15_{16}	00010101_2

Hexadecimal Number : 15_{16} = Binary Number : 10101_2
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.
$108108_{10} = 11011001101100_2$
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.

$+108+108_{10} = 0110110001101100_2$

Therefore, the **signed binary representation** of positive decimal number +108 is **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.

$-108-108_{10} = 1110110011101100_2$

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.

$-108-108_{10} = 1001001110010011_2$

Therefore, the **1's complement of** 108108_{10} is 1001001110010011_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 **$(108)_{10}$** is **$(10010011)_2$**

2's compliment of 108108_{10} *= 1's compliment of* 108108_{10} *+ 1.*

= 10010011 + 1

= 10010100

Therefore, the **2's complement of** 108108_{10} is 1001010010010100_2.

BINARY ARITHMATICS

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 ignoret 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.
$+7+7_{10} = 0011100111_2$
$+4+4_{10} = 0010000100_2$
The addition of these two numbers is
$+7+7_{10} ++4+4_{10} = 0011100111_2+0010000100_2$
$\Rightarrow +7+7_{10} ++4+4_{10} = 0101101011_2$.
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.
$-7-7_{10} = 1100111001_2$
$-4-4_{10} = 1110011100_2$
The addition of these two numbers is
$-7-7_{10} + -4-4_{10} = 1100111001_2 + 1110011100_2$
$\Rightarrow -7-7_{10} + -4-4_{10} = 110101110101_2$.
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 $-7-7_{10} + -4-4_{10} = 1010110101_2$.
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′scomplementofB2′scomplementofB
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′scomplementofA2′scomplementofA
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
$+7+7_{10} - +4+4_{10} = +7+7_{10} + -4-4_{10}$.
The **2’s complement** representation of +7 and -4 with 5 bits each are shown below.
$+7+7_{10} = 0011100111_2$
$+4+4_{10} = 1110011100_2$
$\Rightarrow +7+7_{10} + +4+4_{10} = 0011100111_2 + 1110011100_2 = 0001100011_2$
Here, the carry obtained from sign bit. So, we can remove it. The resultant sum after removing carry is
$+7+7_{10} + +4+4_{10} = 0001100011_2$
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
$+4+4_{10} - +7+7_{10} = +4+4_{10} + -7-7_{10}$.
The **2’s complement** representation of +4 and -7 with 5 bits each are shown below.
$+4+4_{10} = 0010000100_2$
$-7-7_{10} = 1100111001_2$
$\Rightarrow +4+4_{10} + -7-7_{10} = 0010000100_2 + 1100111001_2 = 1110111101_2$
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.

CODES

n 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 84218421 codes of 7, 8 and 6 are 0111, 1000 and 0110 respectively.

∴ $786786_{10} = 011110000110011110000110_{BCD}$

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.

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′′=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 relationsrelations 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**.
⇒ f = p′q+pq′r + pqr′+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**
⇒ f = p⊕qr + pq1
Step 3 − The first term can’t be simplified further. But, the second term can be simplified to pq using **Boolean postulate**.
⇒ f = p⊕qr + pq
Therefore, the simplified Boolean function is **f = p⊕qr + 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.
⇒ 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.
⇒ f = qrp′+p + prq′+q + pqr′+r
Step 3 − Use **Boolean postulate**, x + x’ = 1 for simplifying the terms present in each parenthesis.
⇒ f = qr1 + pr1 + pq1
Step 4 − Use **Boolean postulate**, x.1 = x for simplifying the above three terms.
⇒ f = qr + pr + pq
⇒ 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.

Canonical & Standard Forms

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 outputss. 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 functionfunction 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 ff.
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+m7
f=∑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 functionfunction 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 ff 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 ff.
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.M4
f=∏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+rp+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 outputss. Similarly, there are two standard forms of representing the Boolean outputss. 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.

⇒ 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.

⇒ f = qrp′+p + prq′+q + pqr′+r

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

⇒ f = qr1 + pr1 + pq1

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

⇒ f = qr + pr + pq

⇒ 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.

⇒ 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 6th parenthesis.

⇒ 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.

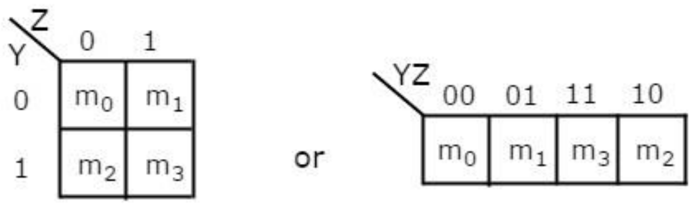
⇒ f = p+q+0.p+r+0.q+r+0

Step 4 − Use **Boolean postulate**, x + 0 = x for simplifying the terms present in each parenthesis
⇒ f = p+q.p+r.q+r
⇒ 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.

K-Map Method

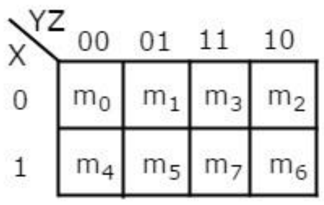
To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit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



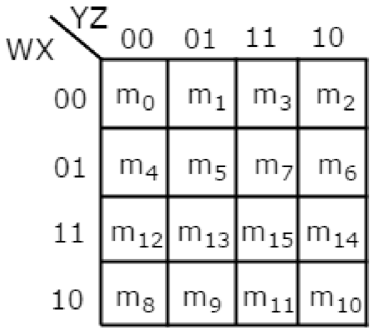
- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_2, m_3), (m_0, m_2) and (m_1, m_3)}.

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**.



- 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.

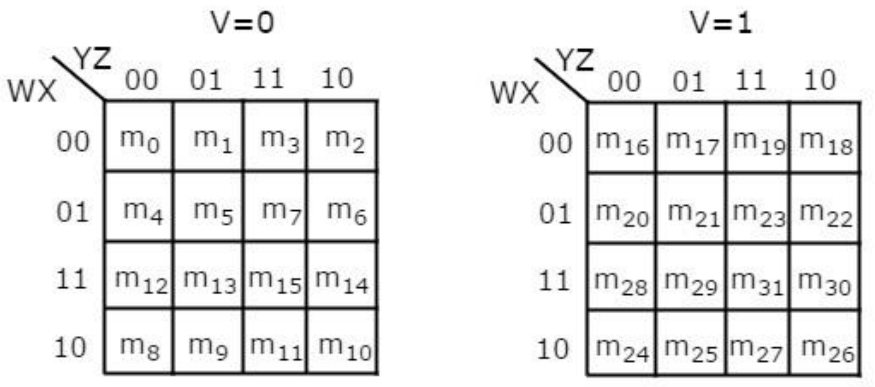
4 Variable K-Map
The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.



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

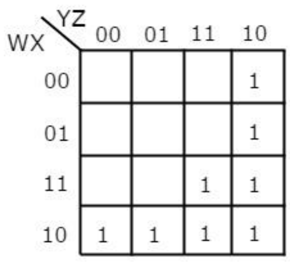
WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

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.

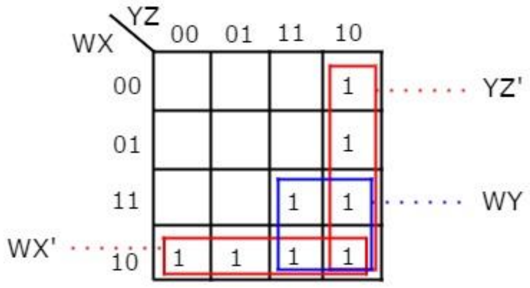
In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.
Minimization of Boolean Functions using K-Maps
If we consider the combination of inputs for which the Boolean function is '1', then we will get the Boolean function, which is 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 simplification.
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'

Logic Gates

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

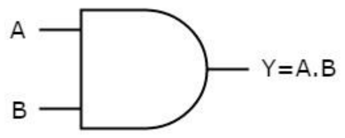
In earlier chapters, we learnt that the Boolean functions can be represented either in sum of products form or in product of sums form based on the requirement. So, we can implement these Boolean functions by using basic gates. The basic gates are AND, OR & NOT gates.

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.



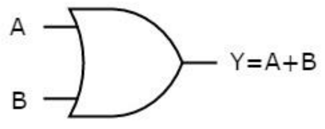
This AND gate produces an output Y, which is the **logical AND** of two inputs A, B. Similarly, if there are 'n' inputs, then the AND gate produces an output, which is the logical AND of all those inputs. That means, the output of AND gate will be '1', when all the inputs are '1'.

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.



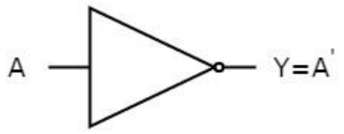
This OR gate produces an output Y, which is the **logical OR** of two inputs A, B. Similarly, if there are 'n' inputs, then the OR gate produces an output, which is the logical OR of all those inputs. That means, the output of an OR gate will be '1', when at least one of those inputs is '1'.

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

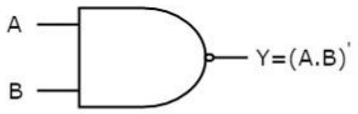
Here A and Y are the input and output of NOT gate respectively. If the input, A is '0', then the output, Y is '1'. Similarly, if the input, A is '1', then the output, Y is '0'.
The following figure shows the **symbol** of NOT gate, which is having one input, A and one output, Y.



This NOT gate produces an output YY, 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.BA.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.

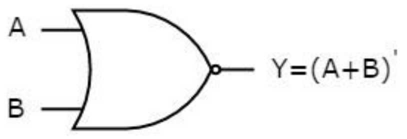


NAND gate operation is same as that of AND gate followed by an inverter. That's why the NAND gate symbol is represented like that.
NOR gate
NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.
The following table shows the **truth table** of 2-input NOR gate

A	**B**	**Y =** A+BA+B'
0	0	1
0	1	0

1	0	0
1	1	0

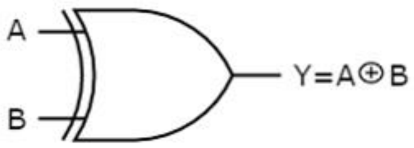
Here A, B are the inputs and Y is the output. If both inputs are ‘0’, then the output, Y is ‘1’. If at least one of the input is ‘1’, then the output, Y is ‘0’. This is just opposite to that of two input OR gate operation.
The following figure shows the **symbol** of NOR gate, which is having two inputs 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 YY 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.

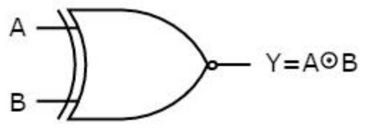


Ex-OR gate operation is similar to that of OR gate, except for few combinationss of inputs. That’s why the Ex-OR gate symbol is represented like that. The output of Ex-OR gate is ‘1’, when odd number of ones present at the inputs. Hence, the output of Ex-OR gate is also called as an **odd function**.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

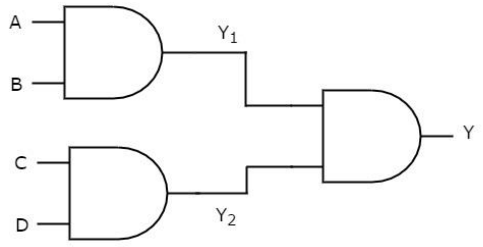
Here A, B are the inputs and Y is the output. The truth table of Ex-NOR gate is same as that of NOR gate for first three rows. The only modification is in the fourth row. That means, the output is one instead of zero, when both the inputs are one.
Therefore, the output of Ex-NOR gate is '1', when both 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 combinationss 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**.
Two-Level Logic Realization
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 cascadedcascaded 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 gatess.
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.

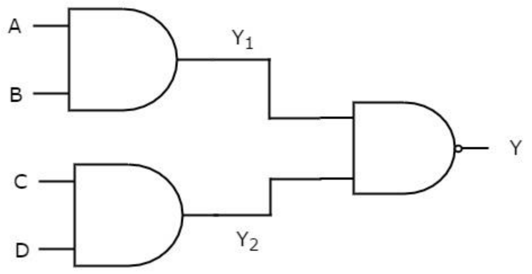
- Degenerative form
- Non-degenerative form

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=Y1Y2
Substitute Y1 and Y2 values in the above equation.
Y=(AB)(CD)
⇒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 gatess 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=AB and Y2=CD

These outputs,Y1 and Y2 are applied as inputs of NAND gate that is present in second level. So, the output of this NAND gate is

Y=(Y1Y2)′

Substitute Y1 and Y2 values in the above equation.

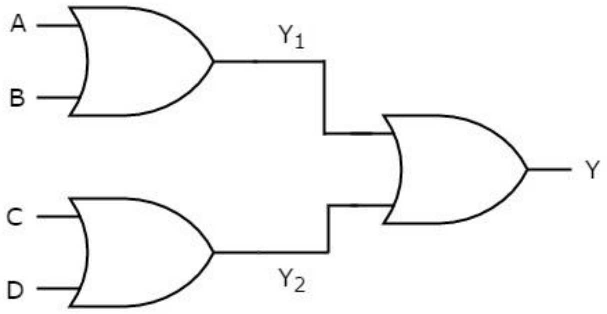
Y=((AB)(CD))′

⇒Y=(ABCD)′

Therefore, the output of this AND-NAND logic realization is (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+B and Y2=C+D.

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+Y2

Substitute Y1 and Y2 values in the above equation Y=(A+B)+(C+D)

⇒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

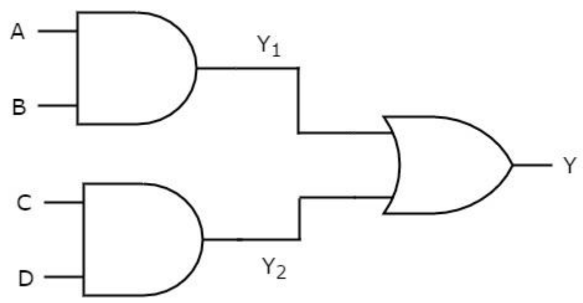
If the output of two level logic realization can’t be obtained by using single logic gate, then it is called as **non-degenerative form**.

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.

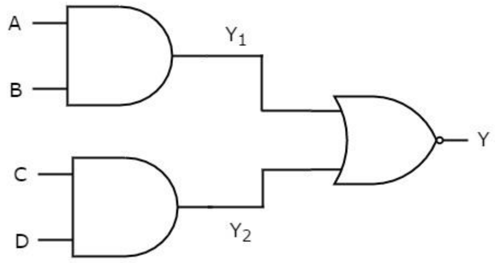
Now, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-OR Logic

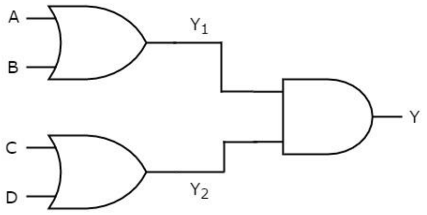
In this logic realization, AND gates are present in first level and OR gatess are present in second level. Below figure shows an example for **AND-OR logic** realization.



Previously, we got the outputs of first level logic gates as Y1=AB and Y2=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+Y2
Substitute Y1 and Y2 values in the above equation
Y=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 gatess 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=AB and Y2=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)′
Substitute Y1 and Y2 values in the above equation.
Y=(AB+CD)′
Therefore, the output of this AND-NOR logic realization is (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 gatess 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+B and Y2=C+D.
These outputs, Y1 and Y2 are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is
Y=Y1Y2
Substitute Y1 and Y2 values in the above equation.
Y=(A+B)(C+D)
Therefore, the output of this OR-AND logic realization is A+BA+B C+DC+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**
Universal Logic Gates
One of the main disdvantages of using the complete sets of AND, OR and NOT gates is that to produce any equivalent

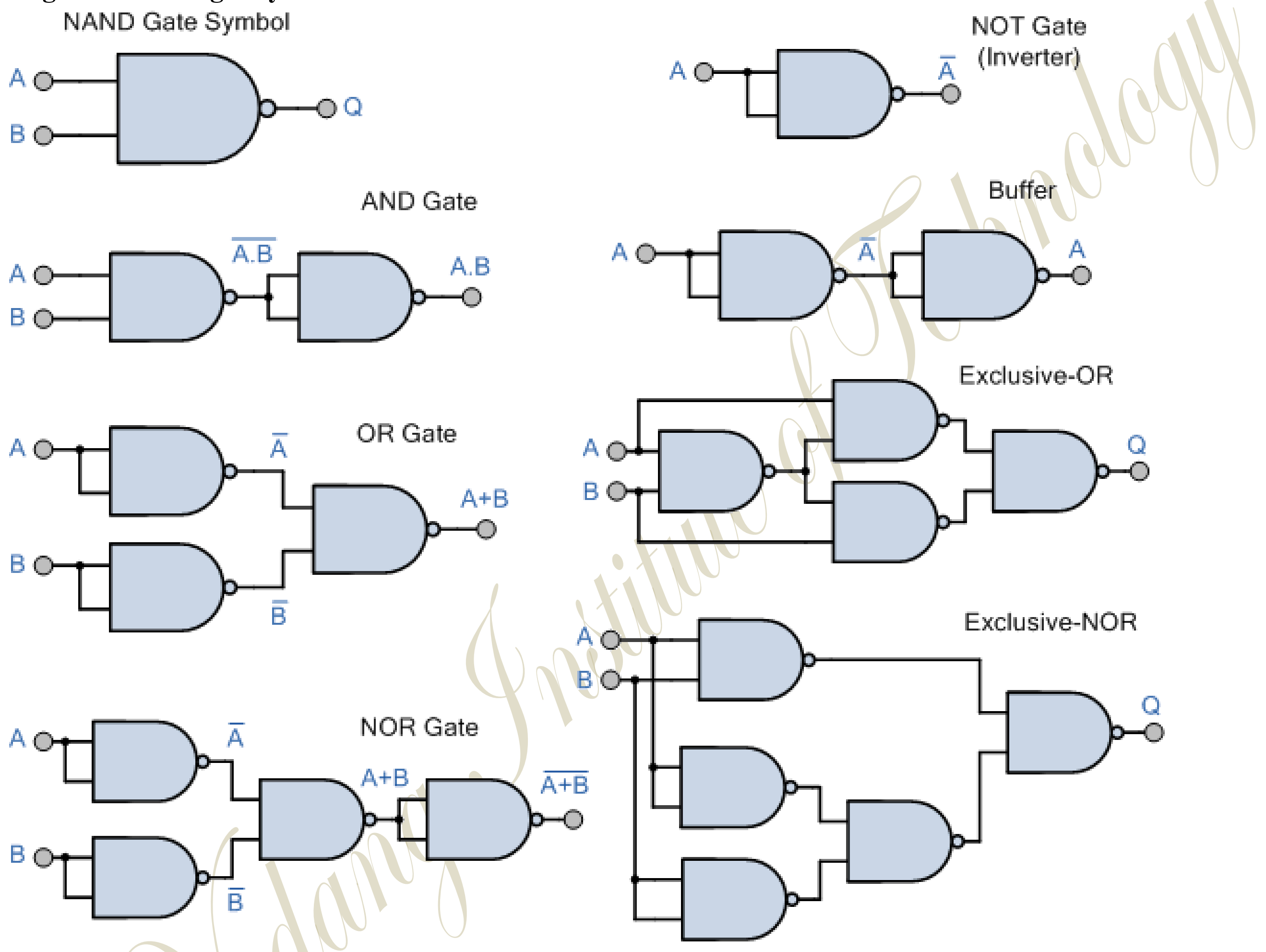
logic gate or function we require two (or more) different types of logic gate, AND and NOT, or OR and NOT, or all three as shown above. However, we can realise all of the other Boolean functions and gates by using just one single type of universal logic gate, the NAND (NOT AND) or the NOR (NOT OR) gate, thereby reducing the number of different types of logic gates required, and also the cost.

The NAND and NOR gates are the complements of the previous AND and OR functions respectively and are individually a complete set of logic as they can be used to implement any other Boolean function or gate. But as we can construct other logic switching functions using just these gates on their own, they are both called a minimal set of gates. Thus the NAND and the NOR gates are commonly referred to as **Universal Logic Gates**.

Implementation of Logic Functions Using Only NAND Gates

The 7400 (or the 74LS00 or 74HC00) quad 2-input NAND TTL chip has four individual NAND gates within a single IC package. Thus we can use a single 7400 TTL chip to produce all the Boolean functions from a NOT gate to a NOR gate as shown.

Logic Gates using only NAND Gates

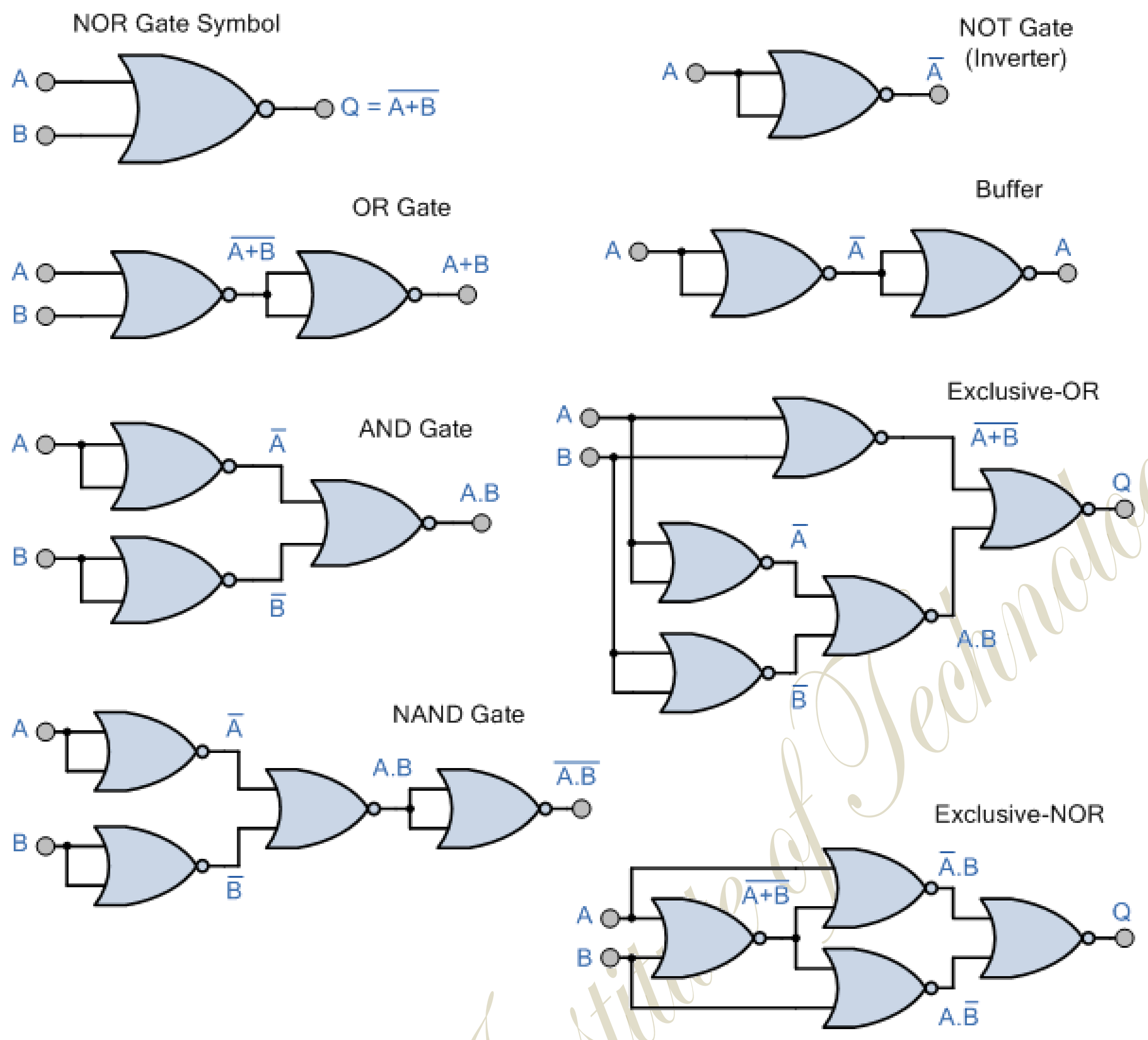


Thus ALL other logic gate functions can be created using only NAND gates making it a universal logic gate.

Implementation of Logic Functions Using Only NOR Gates

The 7402 (or the 74LS02 or 74HC02) quad 2-input NOR TTL chip has four individual NOR gates within a single IC package. Thus like the previous 7400 NAND IC we can use a single 7402 TTL chip to produce all the Boolean functions from a single NOT gate to a NAND gate as shown.

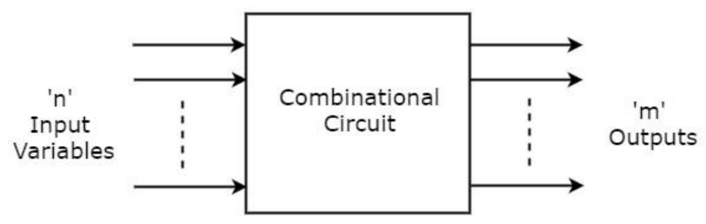
Logic Gates using only NOR Gates



Thus ALL other logic gate functions can be created using only NOR gates making it also a universal logic gate.

Unit-2
Combinational Circuits

Combinational circuits consist of Logic gates. These circuits operate with binary values. The outputss 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 outputss.

Design procedure of Combinational circuits

- Find the required number of input variables and outputs from given specifications.

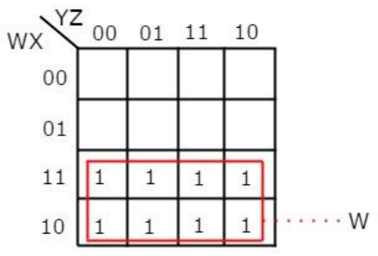
- Formulate the **Truth table**. If there are 'n' input variables, then there will be 2n possible combinations. For each combination of input, find the output values.
- Find the **Boolean expressions** for each output. If necessary, simplify those expressions.
- Implement the above Boolean expressions corresponding to each output by using **Logic gates**.

Code Converters

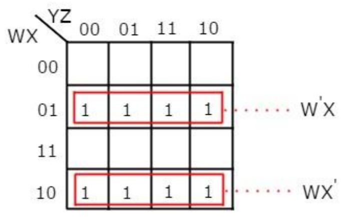
We have discussed various codes in the chapter named codes. The converters, which convert one code to other code are called as **code converters**. These code converters basically consist of Logic gates.
Example
Binary code to Gray code converter
Let us implement a converter, which converts a 4-bit binary code WXYZ into its equivalent Gray code ABCD.
The following table shows the **Truth table** of a 4-bit binary code to Gray code converter.

Binary code WXYZ	WXYZ Gray code ABCD
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

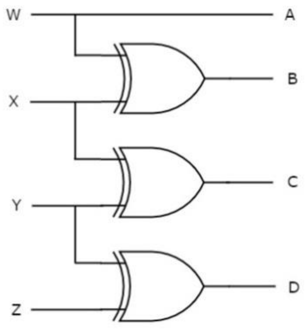
From Truth table, we can write the **Boolean functions** for each output bit of Gray code as below.
$A=\sum m(8,9,10,11,12,13,14,15)$
$B=\sum m(4,5,6,7,8,9,10,11)$
$C=\sum m(2,3,4,5,10,11,12,13)$
$D=\sum m(1,2,5,6,9,10,13,14)$
Let us simplify the above functions using 4 variable K-Maps.
The following figure shows the **4 variable K-Map** for simplifying **Boolean function, A**.



By grouping 8 adjacent ones, we got A=W.
The following figure shows the **4 variable K-Map** for simplifying **Boolean function, B**.



There are two groups of 4 adjacent ones. After grouping, we will get B as
$B=W'X+WX'=W\oplus X$
Similarly, we will get the following Boolean functions for C & D after simplifying.
$C=X'Y+XY'=X\oplus Y$
$D=Y'Z+YZ'=Y\oplus Z$
The following figure shows the **circuit diagram** of 4-bit binary code to Gray code converter.



Since the outputs depend only on the present inputs, this 4-bit Binary code to Gray code converter is a combinational circuit. Similarly, you can implement other code converters.

Parity Bit Generator

There are two types of parity bit generators based on the type of parity bit being generated. **Even parity generator** generates an even parity bit. Similarly, **odd parity generator** generates an odd parity bit.

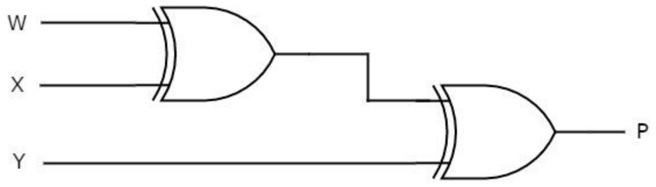
Even Parity Generator

Now, let us implement an even parity generator for a 3-bit binary input, WXY. It generates an even parity bit, P. If odd number of ones present in the input, then even parity bit, P should be '1' so that the resultant word contains even number of ones. For other combinations of input, even parity bit, P should be '0'. The following table shows the **Truth table** of even parity generator.

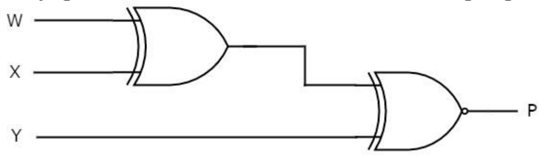
Binary Input WXY	Even Parity bit P
000	0
001	1
010	1
011	0
100	1
101	0

110	0
111	1

From the above Truth table, we can write the **Boolean function** for even parity bit as
$P=W'X'Y+W'XY'+WX'Y'+WXY$
$\Rightarrow P=W'(X'Y+XY')+W(X'Y'+XY)$
$\Rightarrow P=W'(X\oplus Y)+W(X\oplus Y)'=W\oplus X\oplus Y$
The following figure shows the **circuit diagram** of even parity generator.



This circuit consists of two **Exclusive-OR gates** having two inputs each. First ExclusiveOR gate having two inputs W & X and produces an output $W \oplus X$. This output is given as one input of second Exclusive-OR gate. The other input of this second Exclusive-OR gate is Y and produces an output of $W \oplus X \oplus Y$.
Odd Parity Generator
If even number of ones present in the input, then odd parity bit, P should be '1' so that the resultant word contains odd number of ones. For other combinations of input, odd parity bit, P should be '0'.
Follow the same procedure of even parity generator for implementing odd parity generator. The **circuit diagram** of odd parity generator is shown in the following figure.



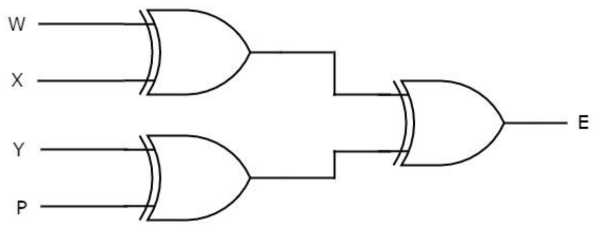
The above circuit diagram consists of Ex-OR gate in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity generator. In that case, the first and second levels contain an ExOR gate in each level and third level consist of an inverter.

Parity Checker
There are two types of parity checkers based on the type of parity has to be checked. **Even parity checker** checks error in the transmitted data, which contains message bits along with even parity. Similarly, **odd parity checker** checks error in the transmitted data, which contains message bits along with odd parity.
Even parity checker
Now, let us implement an even parity checker circuit. Assume a 3-bit binary input, WXY is transmitted along with an even parity bit, P. So, the resultant word datadata contains 4 bits, which will be received as the input of even parity checker.
It generates an **even parity check bit, E**. This bit will be zero, if the received data contains an even number of ones. That means, there is no error in the received data. This even parity check bit will be one, if the received data contains an odd number of ones. That means, there is an error in the received data.
The following table shows the **Truth table** of an even parity checker.

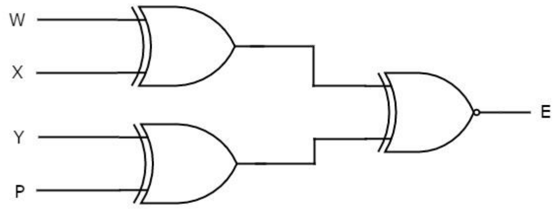
4-bit Received Data WXYP	**Even Parity Check bit E**
0000	0
0001	1
0010	1
0011	0
0100	1
0101	0

0110	0
0111	1
1000	1
1001	0
1010	0
1011	1
1100	0
1101	1
1110	1
1111	0

From the above Truth table, we can observe that the even parity check bit value is ‘1’, when odd number of ones present in the received data. That means the Boolean function of even parity check bit is an **odd function**. Exclusive-OR function satisfies this condition. Hence, we can directly write the **Boolean function** of even parity check bit as
$E=W\oplus X\oplus Y\oplus P$
The following figure shows the **circuit diagram** of even parity checker.



This circuit consists of three **Exclusive-OR gates** having two inputs each. The first level gates produce outputs of $W\oplus X$& $Y\oplus P$. The Exclusive-OR gate, which is in second level produces an output of $W\oplus X\oplus Y\oplus P$
Odd Parity Checker
Assume a 3-bit binary input, WXY is transmitted along with odd parity bit, P. So, the resultant word data contains 4 bits, which will be received as the input of odd parity checker.
It generates an **odd parity check bit, E**. This bit will be zero, if the received data contains an odd number of ones. That means, there is no error in the received data. This odd parity check bit will be one, if the received data contains even number of ones. That means, there is an error in the received data.
Follow the same procedure of an even parity checker for implementing an odd parity checker. The **circuit diagram** of odd parity checker is shown in the following figure.



The above circuit diagram consists of Ex-OR gates in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity checker. In that case, the first, second and third levels contain two Ex-OR gates, one Ex-OR gate and one inverter respectively.

Binary Adder

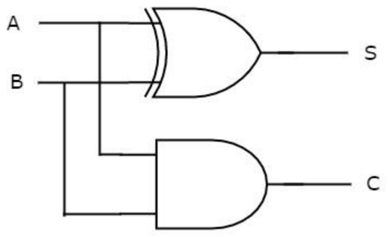
The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.

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
$S=A\oplus B$
$C=AB$
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.

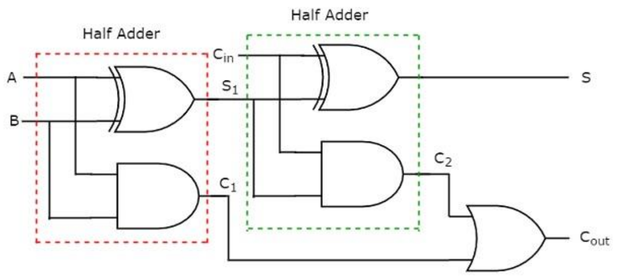
Full Adder

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.
$S=A\oplus B\oplus Cin$
$cout=AB+(A\oplus B)cin$
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.



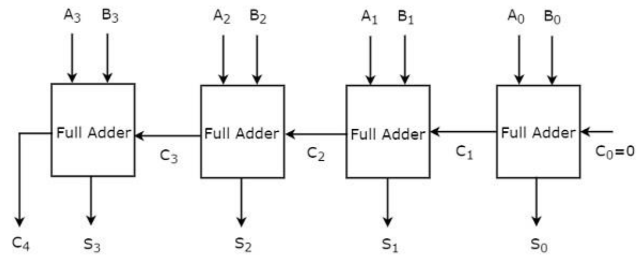
This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If C_{in} is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.

4-bit Binary Adder

The 4-bit binary adder performs the **addition of two 4-bit numbers**. Let the 4-bit binary numbers, A=A3A2A1A0 and B=B3B2B1B0. We can implement 4-bit binary adder in one of the two following ways.

- Use one Half adder for doing the addition of two Least significant bits and three Full adders for doing the addition of three higher significant bits.
- Use four Full adders for uniformity. Since, initial carry C_{in} is zero, the Full adder which is used for adding the least significant bits becomes Half adder.

For the time being, we considered second approach. The **block diagram** of 4-bit binary adder is shown in the following figure.



Here, the 4 Full adders are cascaded. Each Full adder is getting the respective bits of two parallel inputs A & B. The carry output of one Full adder will be the carry input of subsequent higher order Full adder. This 4-bit binary adder produces the resultant sum having at most 5 bits. So, carry out of last stage Full adder will be the MSB.
In this way, we can implement any higher order binary adder just by cascading the required number of Full adders. This binary adder is also called as **ripple carry** binary **adder** because the carry propagates ripples from one stage to the next stage.

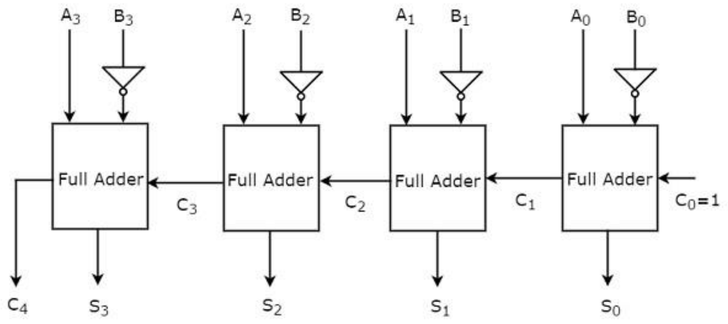
Binary Subtractor

The circuit, which performs the subtraction of two binary numbers is known as **Binary subtractor**. We can implement Binary subtractor in following two methods.

- Cascade Full subtractors
- 2's complement method

In first method, we will get an n-bit binary subtractor by cascading 'n' Full subtractors. So, first you can implement Halfsubtractor and Full subtractor, similar to Half adder & Full adder. Then, you can implement an n-bit binary

subtractor, by cascading 'n' Full subtractors. So, we will be having two separate circuits for binary addition and subtraction of two binary numbers.
In second method, we can use same binary adder for subtracting two binary numbers just by doing some modifications in the second input. So, internally binary addition operation takes place but, the output is resultant subtraction.
We know that the subtraction of two binary numbers A & B can be written as,
A−B=A+(2′scomplimentofB)
⇒A−B=A+(1′scomplimentofB)+1
4-bit Binary Subtractor
The 4-bit binary subtractor produces the **subtraction of two 4-bit numbers**. Let the 4bit binary numbers, A=A3A2A1A0 and B=B3B2B1B0. Internally, the operation of 4-bit Binary subtractor is similar to that of 4-bit Binary adder. If the normal bits of binary number A, complemented bits of binary number B and initial carry borrow, C_{in} as one are applied to 4-bit Binary adder, then it becomes 4-bit Binary subtractor. The **block diagram** of 4-bit binary subtractor is shown in the following figure.

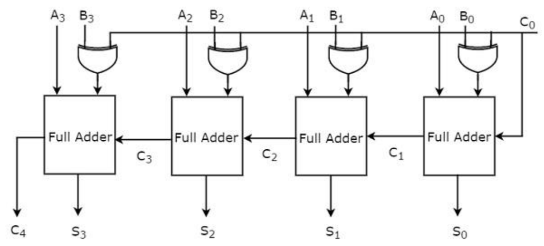


This 4-bit binary subtractor produces an output, which is having at most 5 bits. If Binary number A is greater than Binary number B, then MSB of the output is zero and the remaining bits hold the magnitude of A-B. If Binary number A is less than Binary number B, then MSB of the output is one. So, take the 2's complement of output in order to get the magnitude of A-B.
In this way, we can implement any higher order binary subtractor just by cascading the required number of Full adders with necessary modifications.
Binary Adder / Subtractor
The circuit, which can be used to perform either addition or subtraction of two binary numbers at any time is known as **Binary Adder / subtractor**. Both, Binary adder and Binary subtractor contain a set of Full adders, which are cascaded. The input bits of binary number A are directly applied in both Binary adder and Binary subtractor.
There are two differences in the inputs of Full adders that are present in Binary adder and Binary subtractor.

- The input bits of binary number B are directly applied to Full adders in Binary adder, whereas the complemented bits of binary number B are applied to Full adders in Binary subtractor.

- The initial carry, $C_0 = 0$ is applied in 4-bit Binary adder, whereas the initial carry borrowborrow, $C_0 = 1$ is applied in 4-bit Binary subtractor.

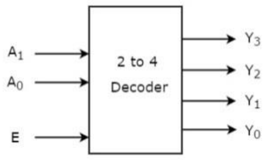
We know that a **2-input Ex-OR gate** produces an output, which is same as that of first input when other input is zero. Similarly, it produces an output, which is complement of first input when other input is one.
Therefore, we can apply the input bits of binary number B, to 2-input Ex-OR gates. The other input to all these Ex-OR gates is C_0. So, based on the value of C_0, the Ex-OR gates produce either the normal or complemented bits of binary number B.
4-bit Binary Adder / Subtractor
The 4-bit binary adder / subtractor produces either the addition or the subtraction of two 4-bit numbers based on the value of initial carry or borrow,C_0. Let the 4-bit binary numbers, A=A3A2A1A0A=A3A2A1A0 and B=B3B2B1B0B=B3B2B1B0. The operation of 4-bit Binary adder / subtractor is similar to that of 4-bit Binary adder and 4-bit Binary subtractor.
Apply the normal bits of binary numbers A and B & initial carry or borrow, C_0 from externally to a 4-bit binary adder. The **block diagram** of 4-bit binary adder / subtractor is shown in the following figure.



If initial carry, C_0 is zero, then each full adder gets the normal bits of binary numbers A & B. So, the 4-bit binary adder / subtractor produces an output, which is the **addition of two binary numbers** A & B.
If initial borrow, C_0 is one, then each full adder gets the normal bits of binary number A & complemented bits of binary number B. So, the 4-bit binary adder / subtractor produces an output, which is the **subtraction of two binary numbers** A & B.
Therefore, with the help of additional Ex-OR gates, the same circuit can be used for both addition and subtraction of two binary numbers.

Decoders

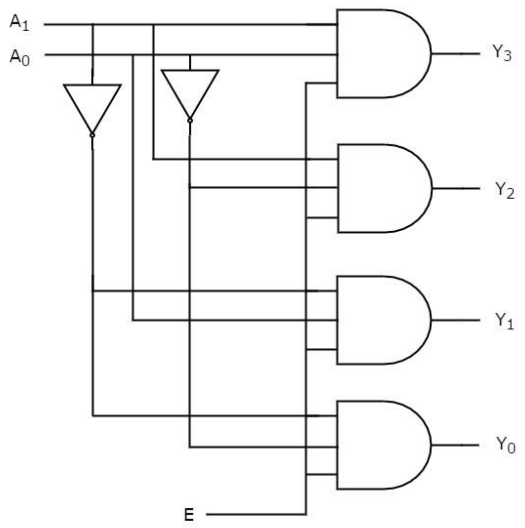
Decoder is a combinational circuit that has 'n' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min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

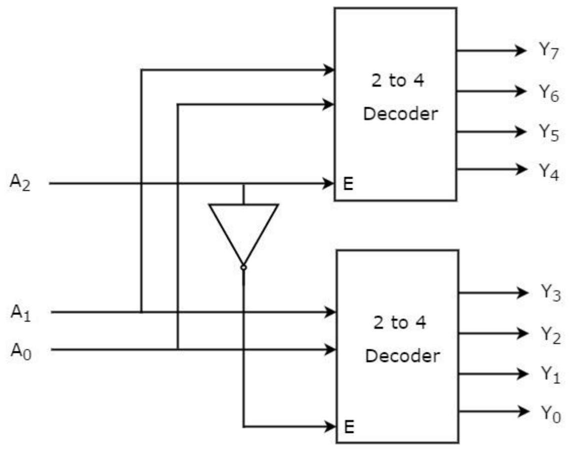
From Truth table, we can write the **Boolean functions** for each output as
Y3=E.A1.A0
Y2=E.A1.A0′
Y1=E.A1′.A0
Y0=E.A1′.A0′
Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each & two 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=m2/m1
Where,
m1 is the number of outputs of lower order decoder.
m2 is the number of outputs of higher order decoder.
Here, m1 = 4 and m2 = 8. Substitute, these two values in the above formula.
Requirednumberof2to4decoders=8/4=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

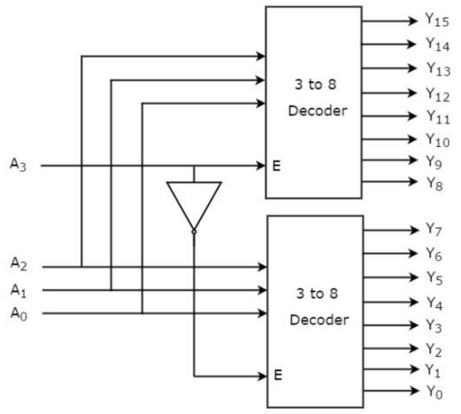
We know the following formula for finding the number of lower order decoders required.

Requirednumberoflowerorderdecoders=m2/m1

Substitute, m1 = 8 and m2 = 16 in the above formula.

Requirednumberof3to8decoders=16/8=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.



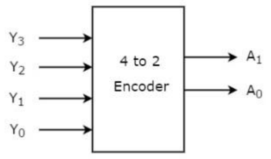
The parallel inputs A_2, A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0. These are the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y_{15} to Y_8. These are the **higher eight min terms**.

Encoders

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^n input 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.

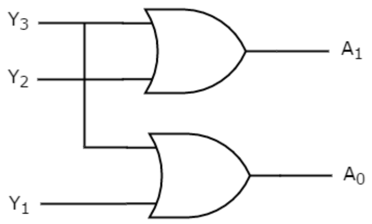
Inputs				**Outputs**	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

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

A1=Y3+Y2

A0=Y3+Y1

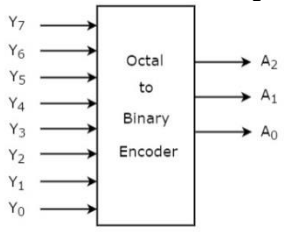
We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



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

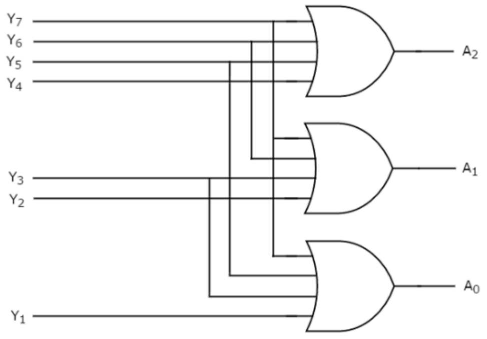
From Truth table, we can write the **Boolean functions** for each output as

$A2=Y7+Y6+Y5+Y4$

$A1=Y7+Y6+Y3+Y2$

$A0=Y7+Y5+Y3+Y1$

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'.

So, to overcome these difficulties, we should assign priorities to each input of encoder. Then, the output of encoder will be the binarybinary code corresponding to the active High inputss, which has higher priority. This encoder is called as **priority encoder**.

Priority Encoder

A 4 to 2 priority encoder has four inputs Y_3, Y_2, Y_1 & Y_0 and two outputs A_1 & A_0. Here, the input, Y_3 has the highest priority, whereas the input, Y_0 has the lowest priority. In this case, even if more than one input is '1' at the same time, the output will be the binary code corresponding to the input, which is having **higher priority**.

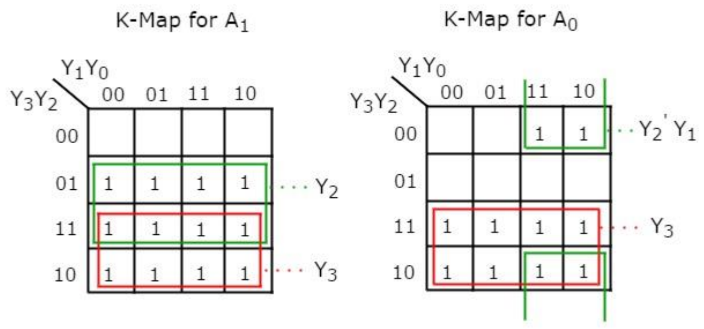
We considered one more **output, V** in order to know, whether the code available at outputs is valid or not.

- If at least one input of the encoder is '1', then the code available at outputs is a valid one. In this case, the output, V will be equal to 1.
- If all the inputs of encoder are '0', then the code available at outputs is not a valid one. In this case, the output, V will be equal to 0.

The **Truth table** of 4 to 2 priority encoder is shown below.

Inputs				**Outputs**		
Y_3	**Y_2**	**Y_1**	**Y_0**	**A_1**	**A_0**	**V**
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	x	X	1	0	1
1	X	x	X	1	1	1

Use **4 variable K-maps** for getting simplified expressions for each output.



The simplified **Boolean functions** are

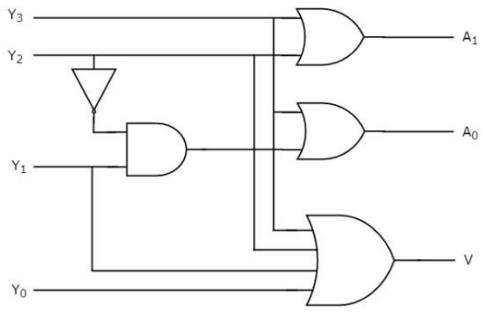
A1=Y3+Y2

A0=Y3+Y2′Y1

Similarly, we will get the Boolean function of output, V as

V=Y3+Y2+Y1+Y0

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



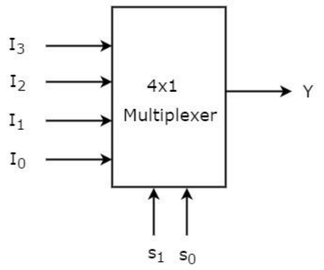
The above circuit diagram contains two 2-input OR gates, one 4-input OR gate, one 2input AND gate & an inverter. Here AND gate & inverter combination are used for producing a valid code at the outputs, even when multiple inputs are equal to '1' at the same time. Hence, this circuit encodes the four inputs with two bits based on the **priority** assigned to each input.

Multiplexers

Multiplexer is a combinational circuit that has maximum of 2^n data inputs, 'n' selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.
Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

4x1 Multiplexer

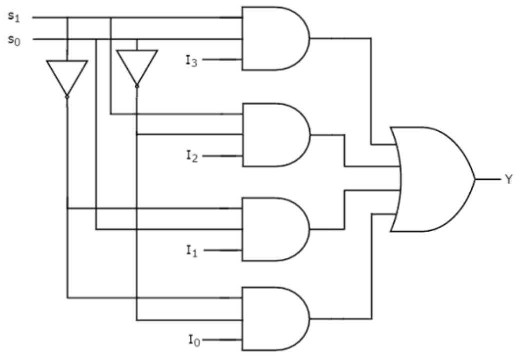
4x1 Multiplexer has four data inputs I_3, I_2, I_1 & I_0, two selection lines s_1 & s_0 and one output Y. The **block diagram** of 4x1 Multiplexer is shown in the following figure.



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

Selection Lines		**Output**
S_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 Y=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

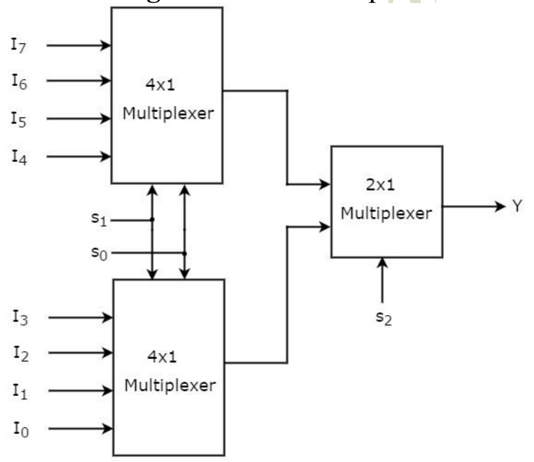
multiplexer by following the same procedure.
Implementation of Higher-order Multiplexers.
Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 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.

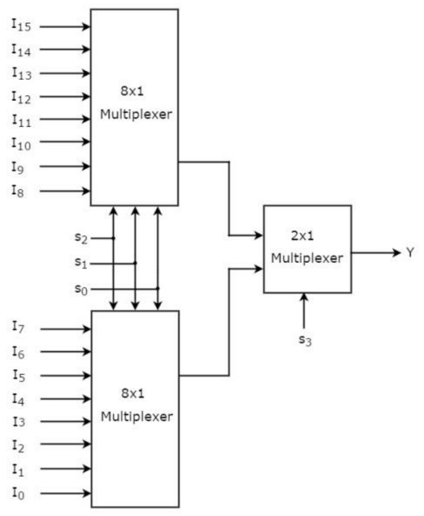
- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0.

- 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.

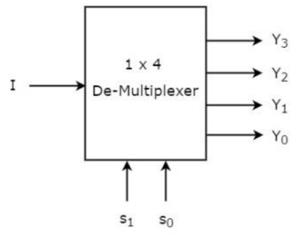
- If s_3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs Is_7 to I_0 based on the values of selection lines s_2, s_1 & s_0.

- 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-Multiplexers

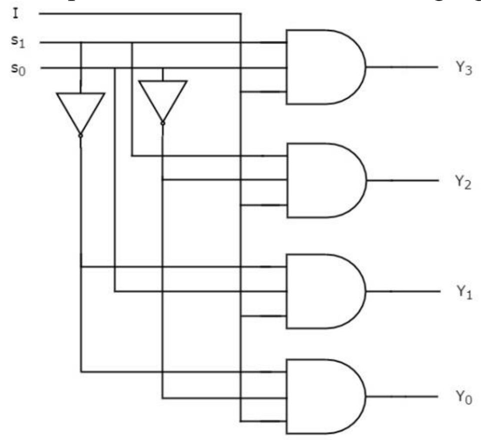
De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.
Since there are 'n' selection lines, there will 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=s1s0I
Y2=s1s0′I
Y1=s1′s0I
Y0=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.

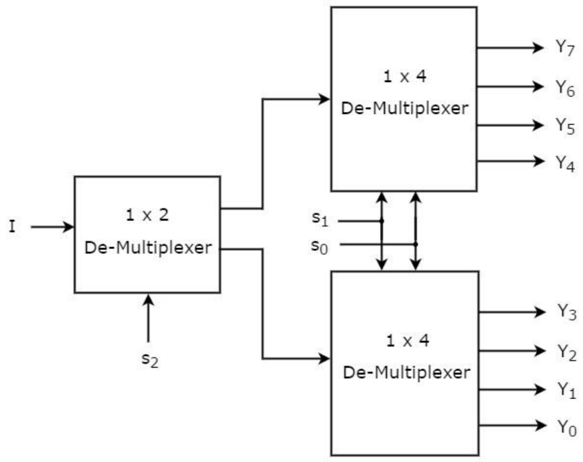
- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

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

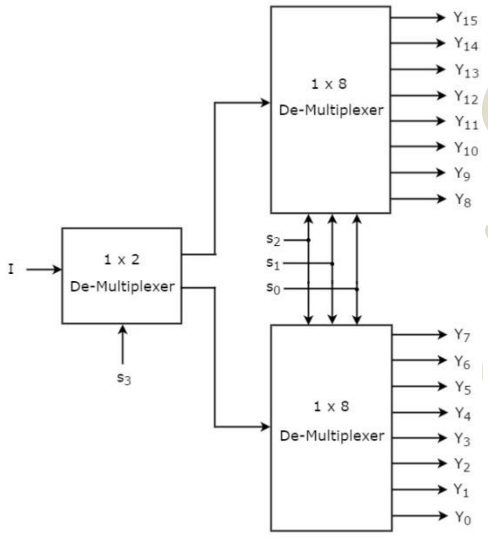
We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table.

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.

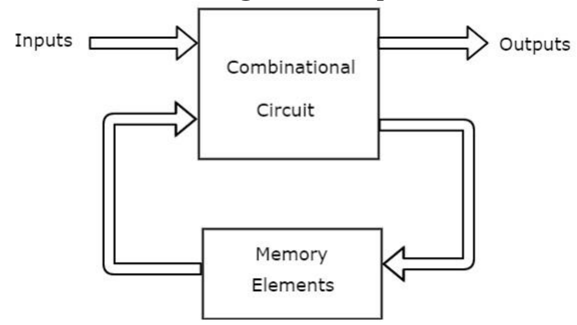
1x16 De-Multiplexer
In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.
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-3
Sequential Circuits

All these circuits have a set of outputss, which depends only on the combination of present inputs. The following figure shows the **block diagram** of sequential circuit.



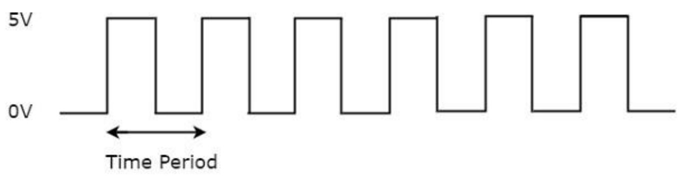
This sequential circuit contains a set of inputs and outputss. The outputss of sequential circuit depends not only on the combination of present inputs but also on the previous outputss. Previous output is nothing but the **present state**. Therefore, sequential circuits contain combinational circuits 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.

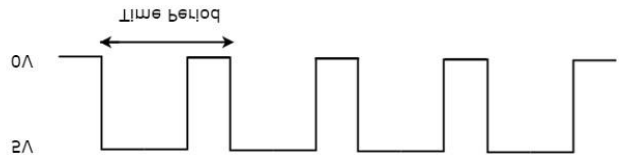
Types of Sequential Circuits
Following are the two types of sequential circuits −

- 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.
Clock Signal and Triggering
Clock signal
Clock signal is a periodic signal and its ON time and OFF time need not be the same. We can represent the clock signal as a **square wave**, when both its ON time and OFF time are same. This clock signal is shown in the following figure.



n the above figure, square wave is considered as clock signal. This signal stays at logic High 5V5V for some time and stays at logic Low 0V0V for equal amount of time. This pattern repeats with some time period. In this case, the **time period** will be equal to either twice of ON time or twice of OFF time.
We can represent the clock signal as **train of pulses**, when ON time and OFF time are not same. This clock signal is shown in the following figure.



In the above figure, train of pulses is considered as clock signal. This signal stays at logic High 5V5V for some time and stays at logic Low 0V0V for some other time. This pattern repeats with some time period. In this case, the **time period** will be equal to sum of ON time and OFF time.
The reciprocal of the time period of clock signal is known as the **frequency** of the clock signal. All sequential circuits are operated with clock signal. So, the frequency at which the sequential circuits can be operated accordingly the clock signal frequency has to be chosen.

Types of Triggering

Following are the two possible types of triggering that are used in sequential circuits.

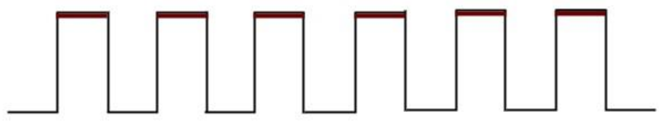
- Level triggering
- Edge triggering

Level triggering

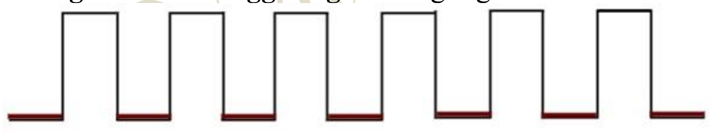
There are two levels, namely logic High and logic Low in clock signal. Following are the two **types of level triggering**.

- Positive level triggering
- Negative level triggering

If the sequential circuit is operated with the clock signal when it is in **Logic High**, then that type of triggering is known as **Positive level triggering**. It is highlighted in below figure.



If the sequential circuit is operated with the clock signal when it is in **Logic Low**, then that type of triggering is known as **Negative level triggering**. It is highlighted in the following figure.

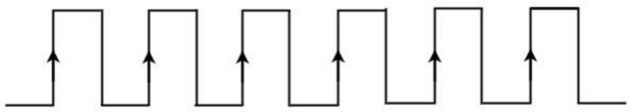


Edge triggering

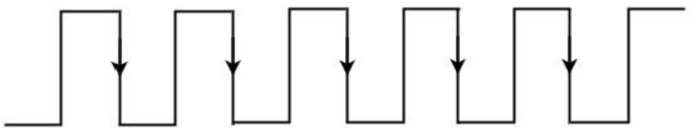
There are two types of transitions that occur in clock signal. That means, the clock signal transitions either from Logic Low to Logic High or Logic High to Logic Low.
Following are the two **types of edge triggering** based on the transitions of clock signal.

- Positive edge triggering
- Negative edge triggering

If the sequential circuit is operated with the clock signal that is transitioning from Logic Low to Logic High, then that type of triggering is known as **Positive edge triggering**. It is also called as rising edge triggering. It is shown in the following figure.



If the sequential circuit is operated with the clock signal that is transitioning from Logic High to Logic Low, then that type of triggering is known as **Negative edge triggering**. It is also called as falling edge triggering. It is shown in the following figure.



Latches

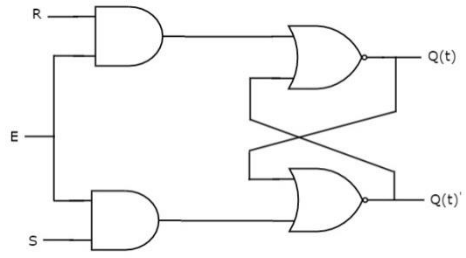
There are two types of memory elements based on the type of triggering that is suitable to operate it.

- 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 Qt & Qt'. The **upper NOR gate** has two inputs R &complement of present state, Qt' and produces next state, Qt+1 when enable, E is '1'.
Similarly, the **lower NOR gate** has two inputs S & present state, Qt and produces complement of next state, Qt+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 Qt+1 will be equal to '1' irrespective of present state, Qt values.
- If R = 1, then next state Qt+1 will be equal to '0' irrespective of present state, Qt values.

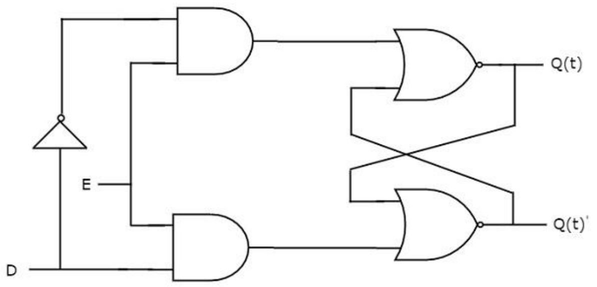
At any time, only of those two inputs should be '1'. If both inputs are '1', then the next state Qt+1 value is undefined. The following table shows the **state table** of SR latch.

S	**R**	**Q**t+1
0	0	Qt
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 Qt & Qt'. 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 Qt+1 will be equal to '0' irrespective of present state, Qt values. This is corresponding to the second row of SR Latch state table.

- If D = 1 → S = 1 & R = 0, then next state Qt+1 will be equal to '1' irrespective of present state, Qt 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

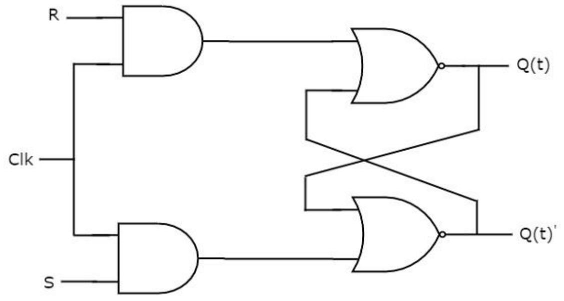
Flip-Flops

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 Qt & Qt'. The operation of SR flip 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.

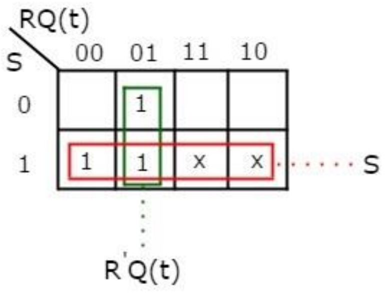
S	**R**	**Q**t+1

0	0	Qt
0	1	0
1	0	1
1	1	-

Here, Qtt & Qt+1t+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**	**Qt**	**Qt+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, Qt+1. The **three variable K-Map** for next state, Qt+1 is shown in the following figure.

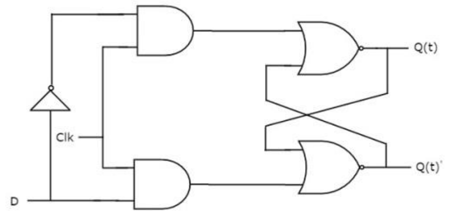


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Qt+1 is

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 Qt & Qt'. 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	**Qt + 1**
0	0
1	1

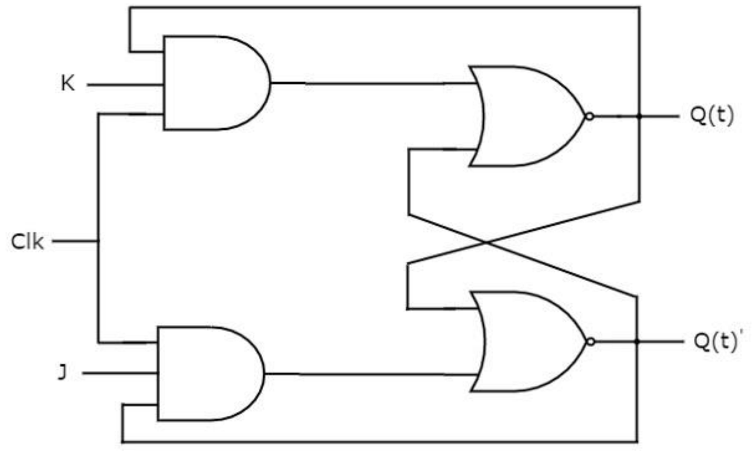
Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

Qt+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 Qtt &Qtt'. The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as **S = J Qt'** 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.

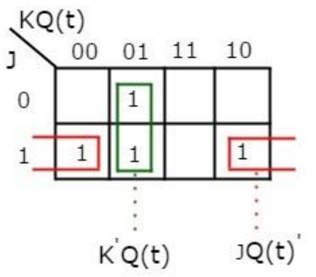
J	**K**	**Q**t+1
0	0	Qt
0	1	0
1	0	1
1	1	Qt'

Here, Qt & Qt+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, Qt+1. **Three variable K-Map** for next state, Qt+1 is shown in the following figure.

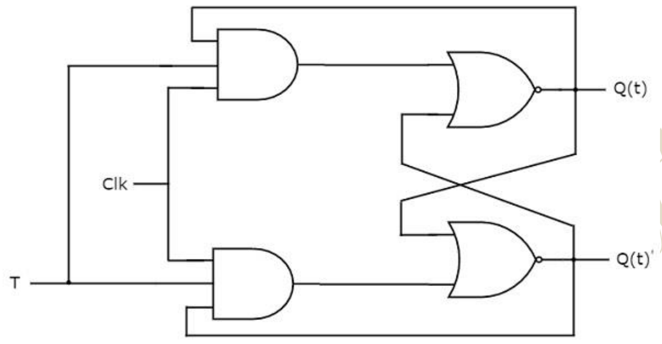


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Qt+1 is

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

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 Qt & Qt'. 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	Qt
1	Qt'

Here, Qt & Qt+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.

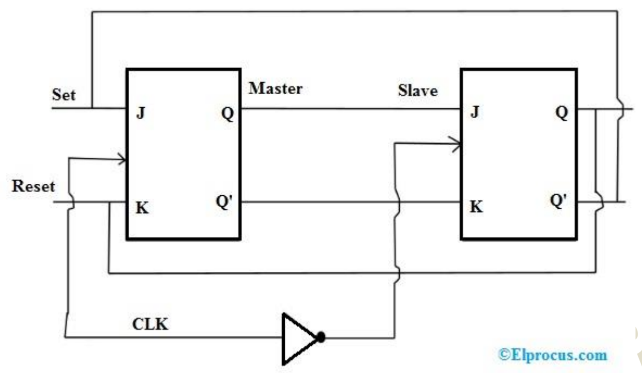
Inputs	**Present State**	**Next State**
T	**Q**t	**Q**t+1
0	0	0
0	1	1

1	0	1
1	1	0

From the above characteristic table, we can directly write the **next state equation** as
$Q(t+1)=T'Q(t)+TQ(t)'$
$\Rightarrow Q(t+1)=T\oplus Q(t)$
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.

Master-Slave Flip Flop

Basically, this type of flip flop can be designed with two JK FFs by connecting in series. One of these FFs, one FF works as the master as well as other FF works as a slave. The connection of these FFs can be done like this, the master FF output can be connected to the inputs of the slave FF. Here slave FF's outputs can be connected to the inputs of the master FF.
In this type of FF, an inverter is also used addition to two FFs. The inverter connection can be done in such a way that where the inverted CLK pulse can be connected to the slave FF. In other terms, if CLK pulse is 0 for a master FF, then CLK pulse will be 1 for a slave FF. Similarly, when CLK pulse is 1 for master FF, then CLK pulse will be 0 for slave FF.



master-slave-flip-flop-circuit

Master-Slave FF Working
Whenever the CLK pulse goes to high which means 1, then the slave can be separated; the inputs like J & K may change the condition of the system.
The slave FF can be is detached until the CLK pulse goes to low which means to 0. Whenever the CLK pulse goes back to low-state, then the data can be transmitted from the master FF to the slave FF and finally, the o/p can be obtained.

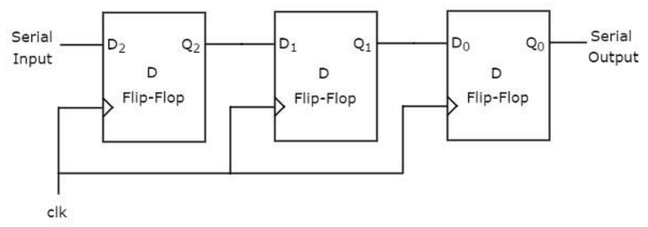
Shift Registers

one flip-flop can store one-bit of information. In order to store multiple bits of information, we require multiple flip-flops. The group of flip-flops, which are used to hold storestore the binary data is known as **register**.
If the register is capable of shifting bits either towards right hand side or towards left hand side is known as **shift register**. An 'N' bit shift register contains 'N' flip-flops. Following are the four types of shift registers based on applying inputs and accessing of outputs.

- Serial In − Serial Out shift register
- Serial In − Parallel Out shift register
- Parallel In − Serial Out shift register
- Parallel In − Parallel Out shift register

Serial In − Serial Out SISO Shift Register

The shift register, which allows serial input and produces serial output is known as Serial In – Serial Out SISO shift register. The **block diagram** of 3-bit SISO shift register is shown in the following figure.



This block diagram consists of three D flip-flops, which are **cascaded**. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we can receive the bits serially from the output of right most D flip-flop. Hence, this output is also called as **serial output**.

Example

Let us see the working of 3-bit SISO shift register by sending the binary information **"011"** from LSB to MSB serially at the input.

Assume, initial status of the D flip-flops from leftmost to rightmost is Q2Q1Q0=000Q2Q1Q0=000. We can understand the **working of 3-bit SISO shift register** from the following table.

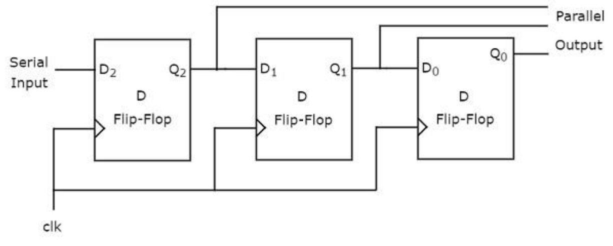
No of positive edge of Clock	**Serial Input**	**Q_2**	**Q_1**	**Q_0**
0	-	0	0	0
1	1LSB	1	0	0
2	1	1	1	0
3	0MSB	0	1	1LSB
4	-	-	0	1
5	-	-	-	0MSB

The initial status of the D flip-flops in the absence of clock signal is Q2Q1Q0=000. Here, the serial output is coming from Q0. So, the LSB 11 is received at 3rd positive edge of clock and the MSB 00 is received at 5th positive edge of clock.

Therefore, the 3-bit SISO shift register requires five clock pulses in order to produce the valid output. Similarly, the **N-bit SISO shift register** requires **2N-1** clock pulses in order to shift 'N' bit information.

Serial In - Parallel Out SIPO Shift Register

The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out SIPOSIPO shift register. The **block diagram** of 3-bit SIPO shift register is shown in the following figure.



This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. In this case, we can access the outputs of each D flip-flop in parallel. So, we will get **parallel outputs** from this shift register.

Example

Let us see the working of 3-bit SIPO shift register by sending the binary information **"011"** from LSB to MSB serially at the input.

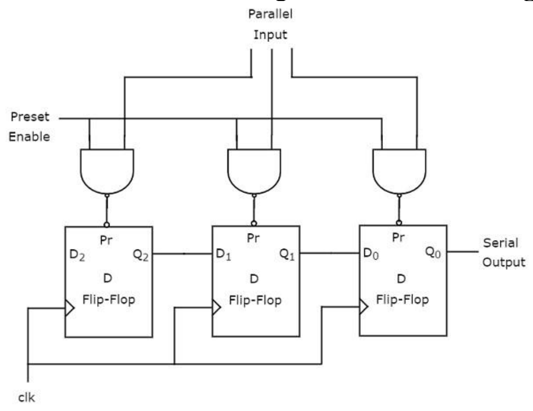
Assume, initial status of the D flip-flops from leftmost to rightmost is Q2Q1Q0=000. Here, Q2Q2 & Q0Q0 are MSB & LSB respectively. We can understand the **working of 3-bit SIPO shift register** from the following table.

No of positive edge of Clock	Serial Input	Q_2MSB	Q_1	Q_0LSB
0	-	0	0	0
1	1LSB	1	0	0
2	1	1	1	0
3	0MSB	0	1	1

The initial status of the D flip-flops in the absence of clock signal is Q2Q1Q0=000. The binary information **"011"** is obtained in parallel at the outputs of D flip-flops for third positive edge of clock.
So, the 3-bit SIPO shift register requires three clock pulses in order to produce the valid output. Similarly, the **N-bit SIPO shift register** requires **N** clock pulses in order to shift 'N' bit information.

Parallel In − Serial Out PISO Shift Register

The shift register, which allows parallel input and produces serial output is known as Parallel In − Serial Out PISOPISO shift register. The **block diagram** of 3-bit PISO shift register is shown in the following figure.



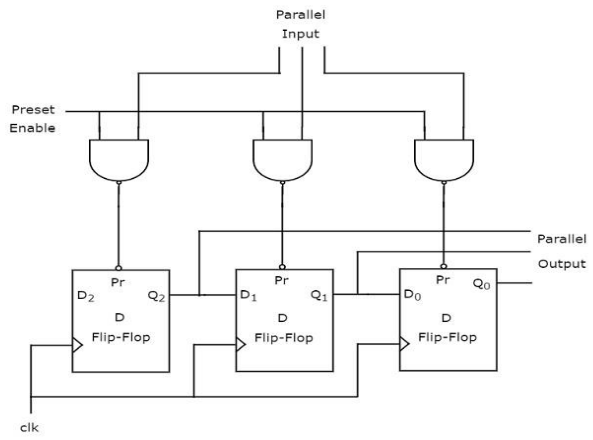
This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.
In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we will get the **serial output** from the right most D flip-flop.
Example
Let us see the working of 3-bit PISO shift register by applying the binary information **"011"** in parallel through preset inputs.
Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will bQ2Q1Q0=011. We can understand the **working of 3-bit PISO shift register** from the following table.

No of positive edge of Clock	Q_2	Q_1	Q_0
0	0	1	1LSB
1	-	0	1
2	-	-	0LSB

Here, the serial output is coming from Q0. So, the LSB 11 is received before applying positive edge of clock and the MSB 00 is received at 2nd positive edge of clock.
Therefore, the 3-bit PISO shift register requires two clock pulses in order to produce the valid output. Similarly, the **N-bit PISO shift register** requires **N-1** clock pulses in order to shift 'N' bit information.

Parallel In - Parallel Out PIPO Shift Register

The shift register, which allows parallel input and produces parallel output is known as Parallel In − Parallel Out PIPOPIPO shift register. The **block diagram** of 3-bit PIPO shift register is shown in the following figure.



This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. We can apply the parallel inputs through preset or clear. These two are asynchronous inputs. That means, the flip-flops produce the corresponding outputs, based on the values of asynchronous inputs. In this case, the effect of outputs is independent of clock transition. So, we will get the **parallel outputs** from each D flip-flop.

Example

Let us see the working of 3-bit PIPO shift register by applying the binary information **"011"** in parallel through preset inputs.

Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be Q2Q1Q0=011. So, the binary information **"011"** is obtained in parallel at the outputs of D flip-flops before applying positive edge of clock.

Therefore, the 3-bit PIPO shift register requires zero clock pulses in order to produce the valid output. Similarly, the **N-bit PIPO shift register** doesn't require any clock pulse in order to shift 'N' bit information.

Microprocessor - 8085 Architecture

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

It has the following configuration −

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

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

Unit-4

8085 Microprocessor – Functional Units

8085 consists of the following functional units −

Accumulator

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

Arithmetic and logic unit

As the name suggests, it performs arithmetic and logical operations like Addition, Subtraction, AND, OR, etc. on 8-bit data.

General purpose register

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

Program counter

It is a 16-bit register used to store the memory address location of the next instruction to be executed. Microprocessor increments the program whenever an instruction is being executed, so that the program counter points to the memory address of the next instruction that is going to be executed.

Stack pointer

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

Temporary register

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

Flag register

It is an 8-bit register having five 1-bit flip-flops, which holds either 0 or 1 depending upon the result stored in the accumulator.
These are the set of 5 flip-flops −

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

Its bit position is shown in the following table −

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

Instruction register and decoder

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

Timing and control unit

It provides timing and control signal to the microprocessor to perform operations. Following are the timing and control signals, which control external and internal circuits −

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

- RESET Signals: RESET IN, RESET OUT

Interrupt control

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

There are 5 interrupt signals in 8085 microprocessor: INTR, RST 7.5, RST 6.5, RST 5.5, TRAP.

Serial Input/output control

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

Address buffer and address-data buffer

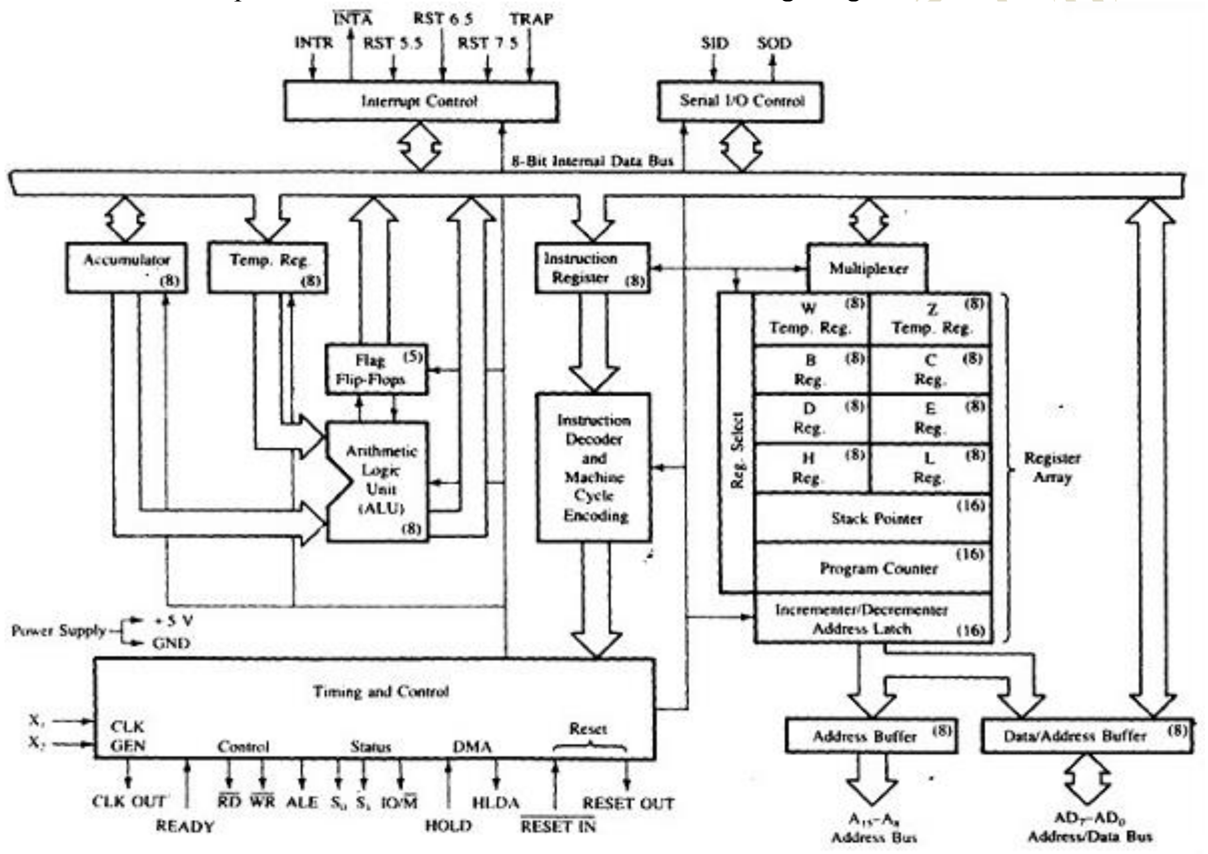
The content stored in the stack pointer and program counter is loaded into the address buffer and address-data buffer to communicate with the CPU. The memory and I/O chips are connected to these buses; the CPU can exchange the desired data with the memory and I/O chips.

Address bus and data bus

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

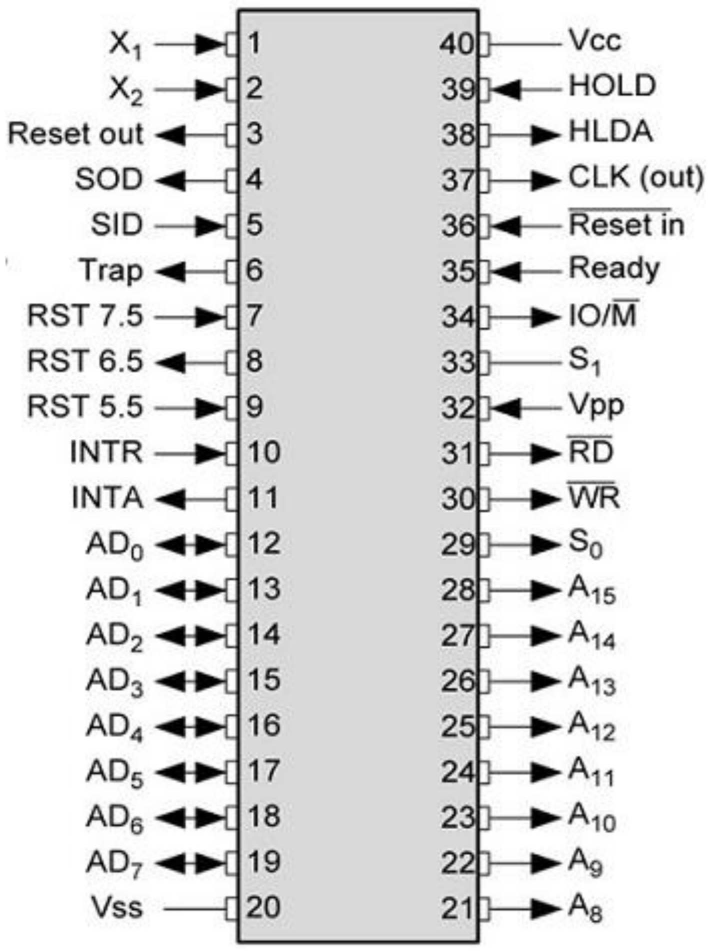
8085 Architecture

We have tried to depict the architecture of 8085 with this following image −



Microprocessor - 8085 Pin Configuration

The following image depicts the pin diagram of 8085 Microprocessor −



The pins of a 8085 microprocessor can be classified into seven groups −
Address bus
A15-A8, it carries the most significant 8-bits of memory/IO address.
Data bus
AD7-AD0, it carries the least significant 8-bit address and data bus.
Control and status signals
These signals are used to identify the nature of operation. There are 3 control signal and 3 status signals.
Three control signals are RD, WR & ALE.

- **RD** − This signal indicates that the selected IO or memory device is to be read and is ready for accepting data available on the data bus.
- **WR** − This signal indicates that the data on the data bus is to be written into a selected memory or IO location.
- **ALE** − It is a positive going pulse generated when a new operation is started by the microprocessor. When the pulse goes high, it indicates address. When the pulse goes down it indicates data.

Three status signals are IO/M, S0 & S1.
IO/M
This signal is used to differentiate between IO and Memory operations, i.e. when it is high indicates IO operation and when it is low then it indicates memory operation.
S1 & S0
These signals are used to identify the type of current operation.
Power supply
There are 2 power supply signals − VCC & VSS. VCC indicates +5v power supply and VSS indicates ground signal.
Clock signals
There are 3 clock signals, i.e. X1, X2, CLK OUT.

- **X1, X2** − A crystal (RC, LC N/W) is connected at these two pins and is used to set frequency of the internal clock generator. This frequency is internally divided by 2.
- **CLK OUT** − This signal is used as the system clock for devices connected with the microprocessor.

Interrupts & externally initiated signals
Interrupts are the signals generated by external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR. We will discuss interrupts in detail in interrupts section.

- **INTA** − It is an interrupt acknowledgment signal.
- **RESET IN** − This signal is used to reset the microprocessor by setting the program counter to zero.
- **RESET OUT** − This signal is used to reset all the connected devices when the microprocessor is reset.
- **READY** − This signal indicates that the device is ready to send or receive data. If READY is low, then the CPU has to wait for READY to go high.
- **HOLD** − This signal indicates that another master is requesting the use of the address and data buses.
- **HLDA (HOLD Acknowledge)** − It indicates that the CPU has received the HOLD request and it will relinquish the bus in the next clock cycle. HLDA is set to low after the HOLD signal is removed.

Serial I/O signals
There are 2 serial signals, i.e. SID and SOD and these signals are used for serial communication.

- **SOD** (Serial output data line) − The output SOD is set/reset as specified by the SIM instruction.
- **SID** (Serial input data line) − The data on this line is loaded into accumulator whenever a RIM instruction is executed.

Addressing Modes in 8085

These are the instructions used to transfer the data from one register to another register, from the memory to the register, and from the register to the memory without any alteration in the content. Addressing modes in 8085 is classified into 5 groups −
Immediate addressing mode
In this mode, the 8/16-bit data is specified in the instruction itself as one of its operand. **For example:** MVI K, 20F: means 20F is copied into register K.
Register addressing mode
In this mode, the data is copied from one register to another. **For example:** MOV K, B: means data in register B is copied to register K.
Direct addressing mode
In this mode, the data is directly copied from the given address to the register. **For example:** LDB 5000K: means the data at address 5000K is copied to register B.
Indirect addressing mode
In this mode, the data is transferred from one register to another by using the address pointed by the register. **For example:** MOV K, B: means data is transferred from the memory address pointed by the register to the register K.
Implied addressing mode
This mode doesn't require any operand; the data is specified by the opcode itself. **For example:** CMP.

Interrupts in 8085

Interrupts are the signals generated by the external devices to request the microprocessor to perform a task. There are 5 interrupt signals, i.e. TRAP, RST 7.5, RST 6.5, RST 5.5, and INTR.
Interrupt are classified into following groups based on their parameter −

- **Vector interrupt** − In this type of interrupt, the interrupt address is known to the processor. **For example:** RST7.5, RST6.5, RST5.5, TRAP.
- **Non-Vector interrupt** − In this type of interrupt, the interrupt address is not known to the processor so, the interrupt address needs to be sent externally by the device to perform interrupts. **For example:** INTR.
- **Maskable interrupt** − In this type of interrupt, we can disable the interrupt by writing some instructions into the program. **For example:** RST7.5, RST6.5, RST5.5.

- **Non-Maskable interrupt** − In this type of interrupt, we cannot disable the interrupt by writing some instructions into the program. **For example:** TRAP.
- **Software interrupt** − In this type of interrupt, the programmer has to add the instructions into the program to execute the interrupt. There are 8 software interrupts in 8085, i.e. RST0, RST1, RST2, RST3, RST4, RST5, RST6, and RST7.
- **Hardware interrupt** − There are 5 interrupt pins in 8085 used as hardware interrupts, i.e. TRAP, RST7.5, RST6.5, RST5.5, INTA.

Note − NTA is not an interrupt, it is used by the microprocessor for sending acknowledgement. TRAP has the highest priority, then RST7.5 and so on.
Interrupt Service Routine (ISR)
A small program or a routine that when executed, services the corresponding interrupting source is called an ISR.
TRAP
It is a non-maskable interrupt, having the highest priority among all interrupts. Bydefault, it is enabled until it gets acknowledged. In case of failure, it executes as ISR and sends the data to backup memory. This interrupt transfers the control to the location 0024H.
RST7.5
It is a maskable interrupt, having the second highest priority among all interrupts. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 003CH address.
RST 6.5
It is a maskable interrupt, having the third highest priority among all interrupts. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 0034H address.
RST 5.5
It is a maskable interrupt. When this interrupt is executed, the processor saves the content of the PC register into the stack and branches to 002CH address.
INTR
It is a maskable interrupt, having the lowest priority among all interrupts. It can be disabled by resetting the microprocessor.
When **INTR signal goes high**, the following events can occur −

- The microprocessor checks the status of INTR signal during the execution of each instruction.
- When the INTR signal is high, then the microprocessor completes its current instruction and sends active low interrupt acknowledge signal.
- When instructions are received, then the microprocessor saves the address of the next instruction on stack and executes the received instruction.

Adding Two 8-bit Numbers

Write a program to add data at 3005H & 3006H memory location and store the result at 3007H memory location.

(3005H) = 14H
(3006H) = 89H

Result −

14H + 89H = 9DH

The program code can be written like this −

```
LXI H 3005H:"HL points 3005H"
MOV A, M      :"Getting first operand"
```

```
INX H        :"HL points 3006H"
ADD M          :"Add second operand"
INX H        :"HL points 3007H"
MOV M, A      :"Store result at 3007H"
HLT          :"Exit program"
```

Exchanging the Memory Locations

Write a program to exchange the data at 5000M& 6000M memory location.

```
LDA 5000M:"Getting the contents at5000M location into accumulator"
MOV B, A    :"Save the contents into B register"
LDA 6000M:"Getting the contents at 6000M location into accumulator"
STA 5000M:"Store the contents of accumulator at address 5000M"
MOV A, B    :"Get the saved contents back into A register"
STA 6000M:"Store the contents of accumulator at address 6000M"
```

Arrange Numbers in an Ascending Order

Write a program to arrange first 10 numbers from memory address 3000H in an ascending order.

```
MVI B,09:"Initialize counter"
START            :"LXI H, 3000H: Initialize memory pointer"
MVI C,09H:"Initialize counter 2"
BACK: MOV A, M    :"Get the number"
INX H            :"Increment memory pointer"
CMP M             :"Compare number with next number"
JC SKIP          :"If less, don't interchange"
JZ SKIP          :"If equal, don't interchange"
MOV D, M
MOV M, A
DCX H
MOV M, D
INX H            :"Interchange two numbers"
SKIP:DCR C        :"Decrement counter 2"
JNZ BACK         :"If not zero, repeat"
DCR B            :"Decrement counter 1"
JNZ START
HLT              :"Terminate program execution
```

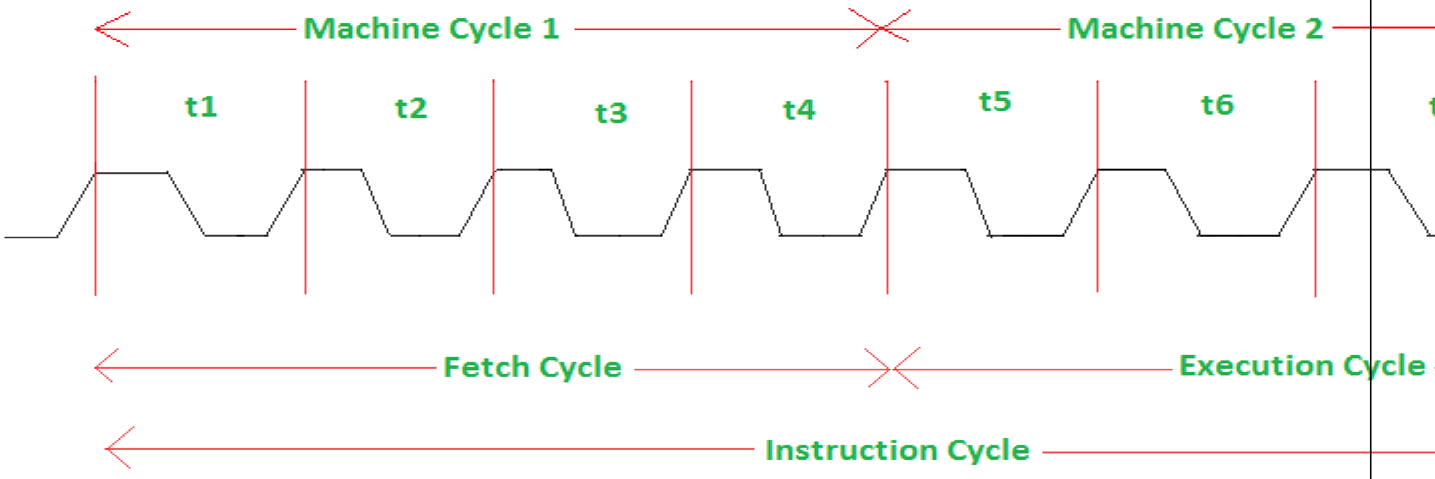
<u>Instruction cycle in 8085 microprocessor</u>

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

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

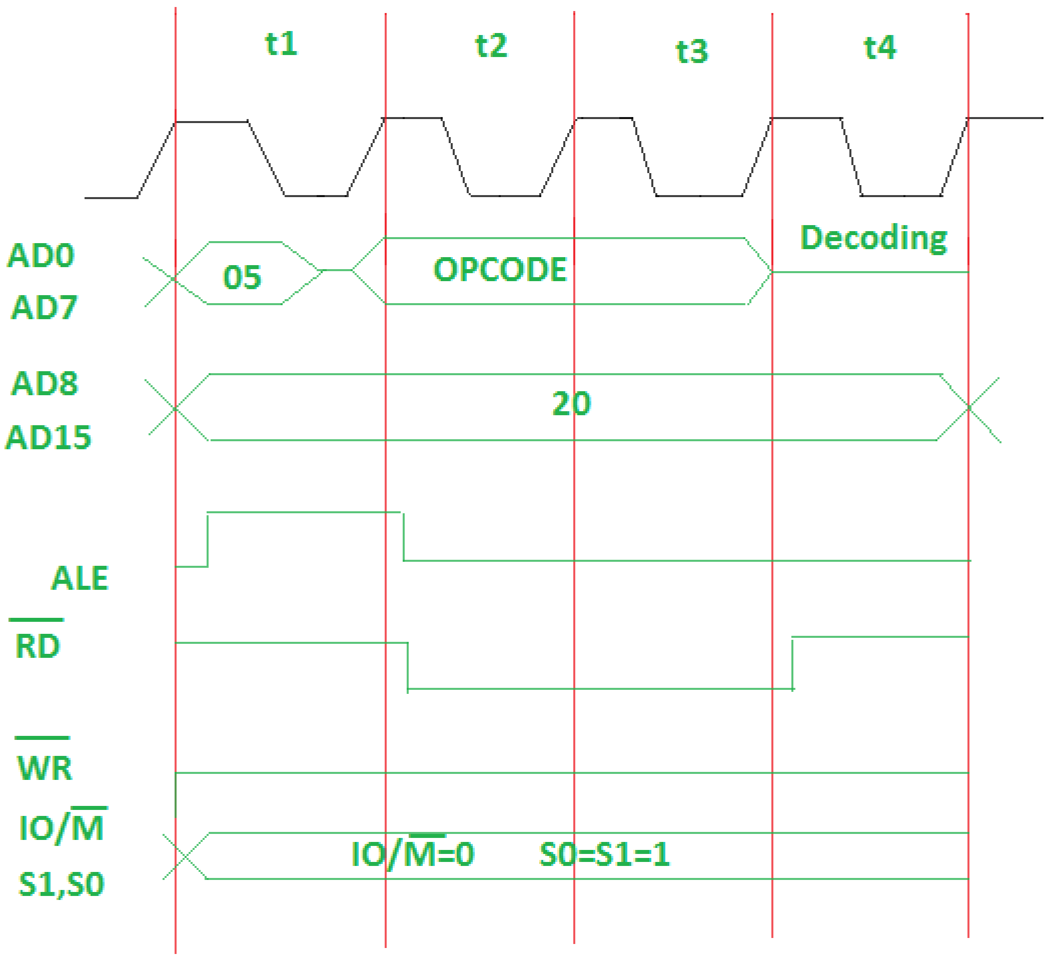
The time required by the microprocessor to complete an operation of accessing memory or input/output devices is called *machine cycle*. One time period of frequency of microprocessor is called *t-state*. A t-state is measured from the falling edge of one clock pulse to the falling edge of the next clock pulse.

Fetch cycle takes four t-states and execution cycle takes three t-states.



Instruction cycle in 8085 microprocessor

Timing diagram for fetch cycle or opcode fetch:



Timing diagram for opcode fetch

Above diagram represents:

- **05** – lower bit of address where opcode is stored. Multiplexed address and data bus AD0-AD7 are used.
- **20** – higher bit of address where opcode is stored. Multiplexed address and data bus AD8-AD15 are used.

- **ALE –** Provides signal for multiplexed address and data bus. If signal is high or 1, multiplexed address and data bus will be used as address bus. To fetch lower bit of address, signal is 1 so that multiplexed bus can act as address bus. If signal is low or 0, multiplexed bus will be used as data bus. When lower bit of address is fetched then it will act as data bus as the signal is low.
- **RD (low active) –** If signal is high or 1, no data is read by microprocessor. If signal is low or 0, data is read by microprocessor.
- **WR (low active) –** If signal is high or 1, no data is written by microprocessor. If signal is low or 0, data is written by microprocessor.
- **IO/M (low active) and S1, S0 –** If signal is high or 1, operation is performing on input output. If signal is low or 0, operation is performing on memory.

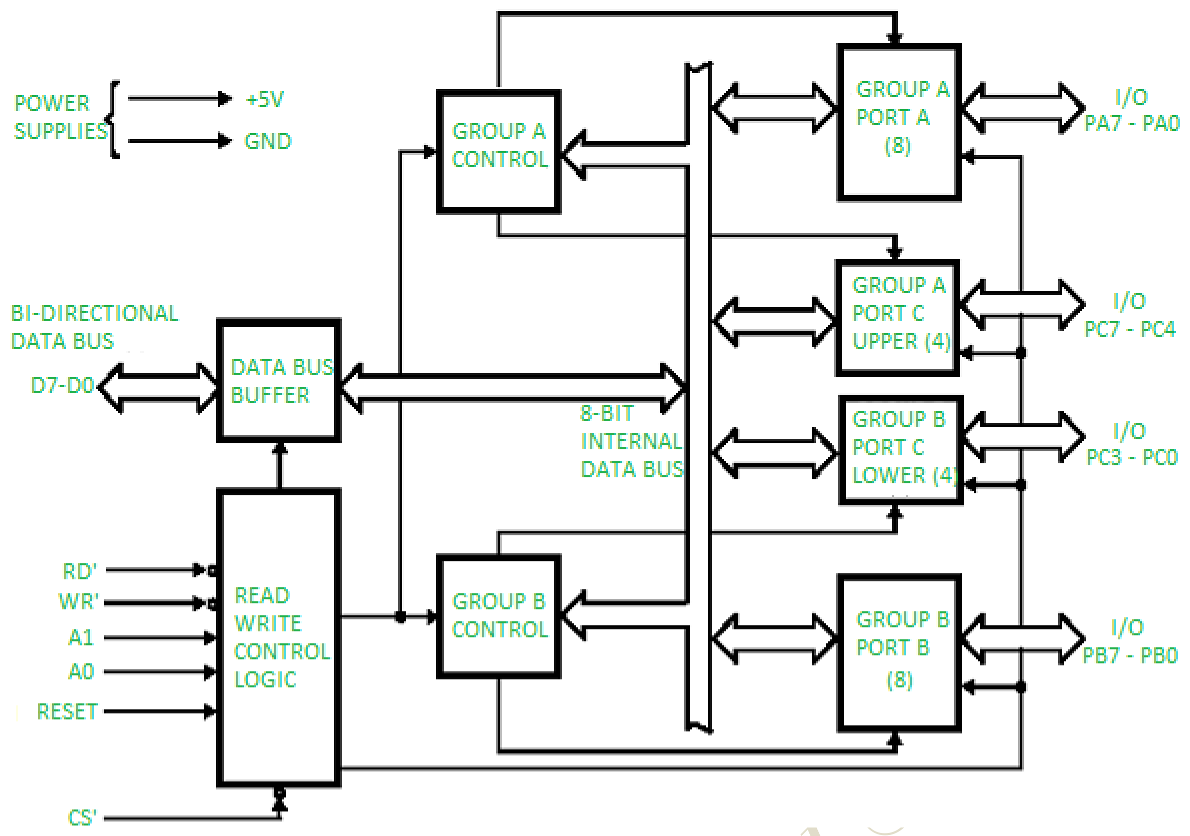
Unit-5

Programmable peripheral interface 8255

PPI 8255 is a general purpose programmable I/O device designed to interface the CPU with its outside world such as ADC, DAC, keyboard etc. We can program it according to the given condition. It can be used with almost any microprocessor.

It consists of three 8-bit bidirectional I/O ports i.e. PORT A, PORT B and PORT C. We can assign different ports as input or output functions.

Block diagram –



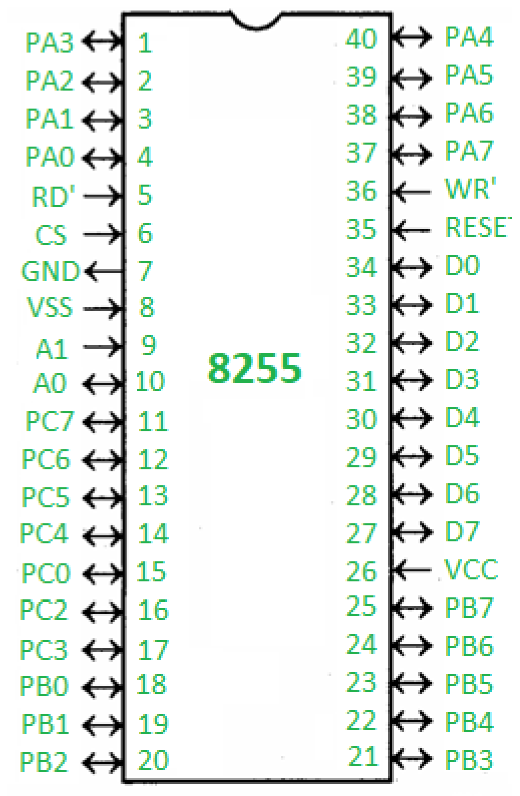
It consists of 40 pins and operates in +5V regulated power supply. Port C is further divided into two 4-bit ports i.e. port C lower and port C upper and port C can work in either BSR (bit set rest) mode or in mode 0 of input-output mode of 8255. Port B can work in either mode 0 or in mode 1 of input-output mode. Port A can work either in mode 0, mode 1 or mode 2 of input-output mode.

It has two control groups, control group A and control group B. Control group A consist of port A and port C upper. Control group B consists of port C lower and port B.

Depending upon the value if CS', A1 and A0 we can select different ports in different modes as input-output function or BSR. This is done by writing a suitable word in control register (control word D0-D7).

CS'	A1	A0	Selection	Address
0	0	0	PORT A	80 H
0	0	1	PORT B	81 H
0	1	0	PORT C	82 H
0	1	1	Control Register	83 H
1	X	X	No Selection	X

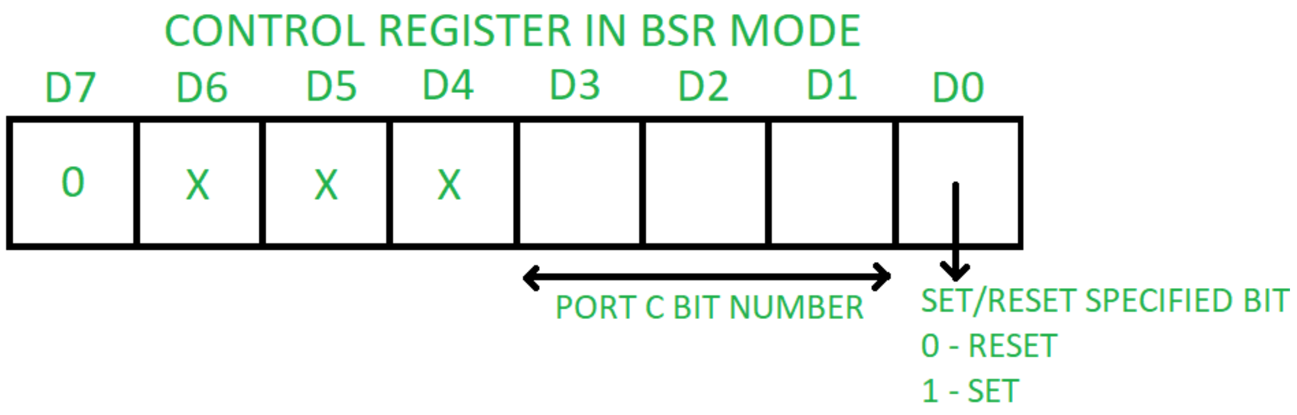
Pin diagram –



- **PA0 – PA7** – Pins of port A
- **PB0 – PB7** – Pins of port B
- **PC0 – PC7** – Pins of port C
- **D0 – D7** – Data pins for the transfer of data
- **RESET** – Reset input
- **RD’** – Read input
- **WR’** – Write input
- **CS’** – Chip select
- **A1 and A0** – Address pins

Operating modes –

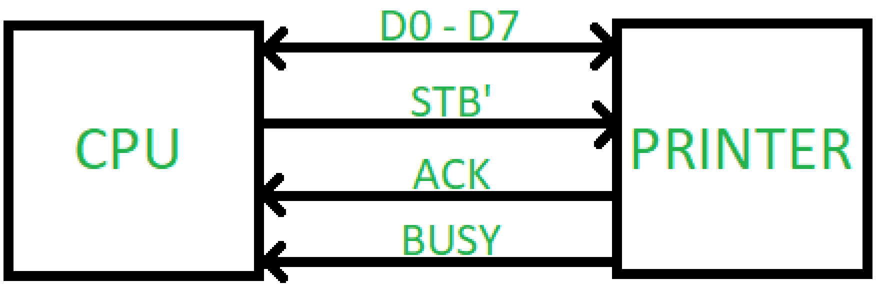
1. **Bit set reset (BSR) mode** –
   If MSB of control word (D7) is 0, PPI works in BSR mode. In this mode only port C bits are used for set or reset.



2. **Input-Output mode** –
   If MSB of control word (D7) is 1, PPI works in input-output mode. This is further divided into three modes:

- **Mode 0** –In this mode all the three ports (port A, B, C) can work as simple input function or simple output function. In this mode there is no interrupt handling capacity.
- **Mode 1** – Handshake I/O mode or strobbed I/O mode. In this mode either port A or port B can work as simple input port or simple output port, and port C bits are used for handshake signals before actual data transmission. It has interrupt handling capacity and input and output are latched.
  Example: A CPU wants to transfer data to a printer. In this case since speed of processor is very fast as compared to relatively slow printer, so before actual data

  transfer it will send handshake signals to the printer for synchronization of the speed of the CPU and the peripherals.



- **Mode 2** – Bi-directional data bus mode. In this mode only port A works, and port B can work either in mode 0 or mode 1. 6 bits port C are used as handshake signals. It also has interrupt handling capacity.

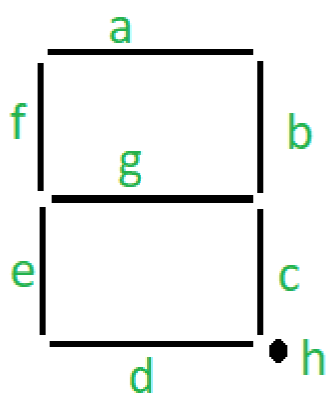
<u>Example of Interfacing Seven Segments LED Display with 8085</u>

Seven segments LED display :

A seven-segment LED is a kind of LED(Light Emitting Diode) consisting of 7 small LEDs it usually comes with the microprocessor's as we commonly need to interface them with microprocessors like 8085.

Structure of Seven Segments LED :

- The LED in Seven Segment display are arranged as below



- It can be used to represent numbers from 0 to 8 with a decimal point.
- We have eight segments in a Seven Segment LED display consisting of 7 segments which include '.'.
- The seven segments are denoted as "a, b, c, d, e, f, g, h" respectively, and '.' is represented by "h"

Interfacing Seven Segment Display with 8085 :

We will see a program to Interfacing Seven Segment Display with 8085 assuming address decoders with an address of AE H.

Note logic needed for activation –

- Common Anode – 0 will make an LED glow.
- Common Cathode – 1 will make an LED glow.

Common Anode Method :

Here we are using a common anode display therefore 0 logic is needed to activate the segment. Suppose to display number 9 at the seven-segment display, therefore the segments F, G, B, A, C, and D have to be activated.

The instructions to execute it is given as,

MVI A,99

OUT AE

- First, we are storing the 99H in the accumulator i.e. 10010000 by using MVI instruction.
- By OUT instruction we are sending the data stored in the accumulator to the port AFH

Common Cathode Method :

Here we are using common cathode 1 logic is needed to activate the signal. Suppose to display number 9 at the seven-segment display, therefore the segments F, G, B, A, C, and D have to be activated.

The instructions to execute it is given as,

MVI A,6F

OUT AE

- First, we are storing the 6FH in the accumulator i.e.01101111 by using MVI instruction.
- By OUT instruction we are sending the data stored in the accumulator to the port AFH