SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SECA1201 - DIGITAL LOGIC CIRCUITS
UNIT I
NUMBER SYSTEMS, LOGIC FUNCTIONS AND BOOLEAN ALGEBRA

Number systems – Number systems conversions - Binary arithmetic – Binary codes – Logic functions- Universal gate functions - Boolean algebra – Functionally complete operation sets, Reduction of switching equations using Boolean algebra, Realization of switching function.

REVIEW OF NUMBER SYSTEMS

Many number systems are in use in digital technology. The most common are the decimal, binary, octal, and hexadecimal systems. The decimal system is clearly the most familiar to us because it is tools that we use every day.

Types of Number Systems are

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

Table: Types of Number Systems

DECIMAL	BINARY	OCTAL	HEXADECIMAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table: Number system and their Base value

Numbering Systems		
System	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Decimal system: Decimal system is composed of 10 numerals or symbols. These 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Using these symbols as digits of a number, we can express any quantity. The decimal system is also called the base-10 system because it has 10 digits. Even though the decimal system has only 10 symbols, any number of any magnitude can be expressed by using our system of positional weighting.

10^3	10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
=1000	=100	=10	=1	.	=0.1	=0.01	=0.001
Most Significant Digit				Decimal point			Least Significant Digit

Example: 3.14_{10} , 52_{10} ,1024_{10}

Binary System: In the binary system, there are only two symbols or possible digit values, 0 and 1. This base-2 system can be used to represent any quantity that can be represented in decimal or other base system.

2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}
=8	=4	=2	=1	.	=0.5	=0.25	=0.125
Most Significant Digit				Binary point			Least Significant Digit

In digital systems the information that is being processed is usually presented in binary form. Binary quantities can be represented by any device that has only two operating states or possible conditions. E.g.. A switch is only open or closed. We arbitrarily (as we define them) let an open switch represent binary 0 and a closed switch represent binary 1. Thus we can represent any binary number by using series of switches.
Binary 1: Any voltage between 2V to 5V Binary 0: Any voltage between 0V to 0.8V
Not used: Voltage between 0.8V to 2V in 5 Volt CMOS and TTL Logic, this may cause error in a digital circuit. Today's digital circuits works at 1.8 volts, so this statement may not hold true for all logic circuits.

Octal System: The octal number system has a base of eight, meaning that it has eight possible digits: 0,1,2,3,4,5,6,7.

8^3	8^2	8^1	8^0		8^{-1}	8^{-2}	8^{-3}
=512	=64	=8	=1	.	=1/8	=1/64	=1/512
Most Significant Digit				Octal point			Least Significant Digit

Hexadecimal System: The hexadecimal system uses base 16. Thus, it has 16 possible digit symbols. It uses the digits 0 through 9 plus the letters A, B, C, D, E, and F as the 16 digit symbols.

16^3	16^2	16^1	16^0		16^{-1}	16^{-2}	16^{-3}
=4096	=256	=16	=1	.	=1/16	=1/256	=1/4096
Most Significant Digit				Hexadecimal point			Least Significant Digit

Code Conversion

Converting from one code form to another code form is called code conversion, like converting from binary to decimal or converting from hexadecimal to decimal.

- **Binary-To-Decimal Conversion:** Any binary number can be converted to its decimal equivalent simply by summing together the weights of the various positions in the binary number which contain a 1.

Binary	Decimal
11011_2	
$= (1*2^4)+(1*2^3)+0+(1*2^1)+(1*2^0)$	=16+8+0+2+1
Result	27_{10}

- **Decimal to binary Conversion:**

There are 2 methods:

- Reverse of Binary-To-Decimal Method
- Repeat Division

Reverse of Binary-To-Decimal Method

Decimal	Binary
45_{10}	=32 + 0 + 8 + 4 +0 + 1
	$=2^5+0+2^3+2^2+0+2^0$
Result	$=101101_2$

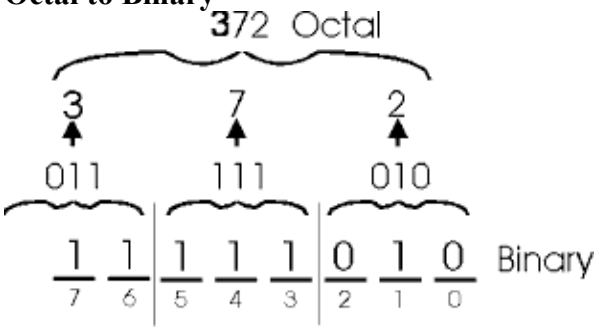
Repeat Division-Convert decimal to binary: This method uses repeated division by 2.

Division	Remainder	Binary
25/2	= 12+ remainder of 1	1 (Least Significant Bit)
12/2	= 6 + remainder of 0	0
6/2	= 3 + remainder of 0	0
3/2	= 1 + remainder of 1	1
1/2	= 0 + remainder of 1	1 (Most Significant Bit)
Result	2510	= 110012

- **Binary-To-Octal / Octal-To-Binary Conversion Binary to octal**

$100\ 111\ 010_2 = (100)\ (111)\ (010)2 = 4\ 7\ 2_8$

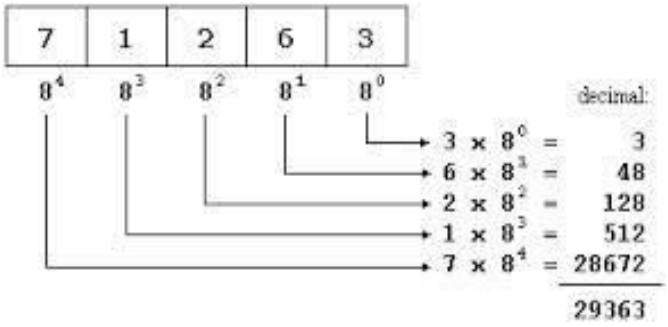
Octal to Binary



- **Decimal -To-Octal / Octal-To- Decimal Conversion Decimal to octal**

Division	Result	Binary
177/8	= 22+ remainder of 1	1 (Least Significant Bit)
22/ 8	= 2 + remainder of 6	6
2 / 8	= 0 + remainder of 2	2 (Most Significant Bit)
Result	17710	= 2618
Binary		= 0101100012

Octal to Decimal



- **Hexadecimal to Decimal/Decimal to Hexadecimal Conversion Decimal to**

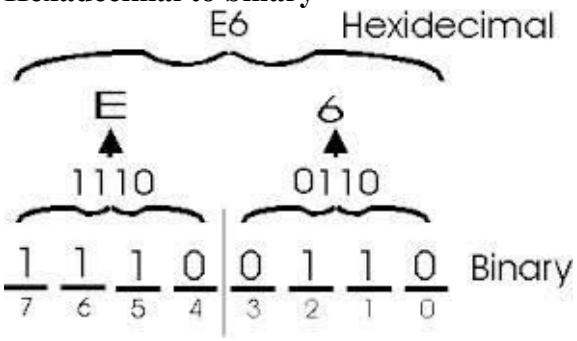
Hexadecimal

Division	Result	Hexadecimal
378/16	= 23+ remainder of 10	A (Least Significant Bit)23
23/16	= 1 + remainder of 7	7
1/16	= 0 + remainder of 1	1 (Most Significant Bit)
Result	378_{10}	$= 17A_{16}$

- **Binary-To-Hexadecimal /Hexadecimal-To-Binary Conversion**

Binary-To-Hexadecimal: $1011\ 0010\ 1111_2 = (1011)\ (0010)\ (1111)_2 = B\ 2\ F_{16}$

Hexadecimal to binary



- **Octal-To-Hexadecimal Hexadecimal-To-Octal Conversion**
  - Convert Octal (Hexadecimal) to Binary first.
  - Regroup the binary number by three bits per group starting from LSB if Octal is required.
  - Regroup the binary number by four bits per group starting from LSB if Hexadecimal is required.

Octal to Hexadecimal

Octal	Hexadecimal
= 2 6 5 0	
010 110 **101** 000	= **0101** 1010 **1000** (Binary)
Result	$=(5A8)_{16}$

Hexadecimal to octal

Hexadecimal	Octal
$(5A8)_{16}$	= **0101** 1010 **1000** (Binary)
	= **010** 110 **101** 000 (Binary)
Result	= 2 6 5 0 (Octal)

1's and 2's complement

Complements are used in digital computers to simplify the subtraction operation and for logical manipulation. There are TWO types of complements for each base-r system: the radix complement and the diminished radix complement. The first is referred to as the r's complement and the second as the (r - 1)'s complement, when the value of the base r is substituted in the name. The two types are referred to as

The 2's complement and 1's complement for binary numbers and the 10's complement and 9's
complement for decimal numbers.

- **The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.**
- **The 2's complement is the binary number that results when we add 1 to the 1's complement. It is used to represent negative numbers.**

2's complement=1's complement+1

Example 1) : Find 1's complemnt of $(1101)_2$

1 1 0 1 number
0 0 1 0 1's complement

Example 2) : Find 1's complemnt of $(1001)_2$

1 0 0 1 number

0 1 1 0 1's complement
\+ 1
= 0 1 1 1

ARITHMETIC OPERATIONS

Binary Equivalents
1 Nybble (or nibble) = 4 bits 1 Byte = 2 nibbles = 8 bits
1 Kilobyte (KB) = 1024 bytes
1 Megabyte (MB) = 1024 kilobytes = 1,048,576 bytes
1 Gigabyte (GB) = 1024 megabytes = 1,073,741,824 bytes

Binary Addition

Rules of Binary Addition

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0, and carry 1 to the next more significant bit

Example : **00011010 + 00001100 = 00100110**

Binary Subtraction
Rules of Binary Subtraction

- 0 - 0 = 0
- 0 - 1 = 1, borrow 1 from the next bit
- 1 - 0 = 1
- 1 - 1 = 0

Example

00100101 - 00010001= 00010100	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
	+ 0	**0**	**0**	**1**	**0**	**0**	**0**	**1**
	0	**0**	**0**	**1**	**0**	**1**	**0**	**0**

Binary Multiplication
Rules of Binary Multiplication

- 0 x 0 = 0
- 0 x 1 = 0
- 1 x 0 = 0
- 1 x 1 = 1, and no carry or borrow bits

Example

	0	**0**	**1**	**0**	**1**	**0**	**0**	**1**
×	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**1**	**0**	**1**	**0**	**0**	**1**	+
0	**1**	**0**	**1**	**0**	**0**	**1**		+
0	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**

00101001 × 00000110 = 11110110

Binary Division

Binary division is the repeated process of subtraction, just as in decimal division.

Example 1: 00101010 ÷ 00000110 = 00000111

00101010÷00000110 1 1 1 = 7(base 10)

=00000111

1 1 0 0 0 1 1 0 1 0 = 42(base 10)

- 1 1 0 = 6(base 10)

1 borrows

1 0 1 0 1

- 1 1 0

1 1 0

- 1 1 0

0

Example 2: 10000111 ÷ 00000101 = 00011011

•

•

•

10000111÷00000101 1 1 0 1 1 = 27(base 10)

=00011011

1 0 1 1 0 0 1 0 0 1 1 1 = 135(base 10)

- 1 0 1 = 5(base 10)

1 1 1 0

- 1 0 1

1 1

- 0

1 1 1

- 1 0 1

1 0 1

- 1 0 1

0

BINARY CODES

Binary codes are codes which are represented in binary system with modification from the original ones. There are two types of binary codes: Weighted codes and Non-Weighted codes. BCD and the 2421 code are examples of weighted codes. In a weighted code, each bit position is assigned a weighting factor in such a way that each digit ca n be evaluated by adding the weight of all the 1's in the coded combination.

Weighted Code

- 8421 code , Most common, Default
- The corresponding decimal digit is determined by adding the weights associated with the 1s in the code group. 62310 = 0110 0010 0011

  2421, 5421,7536, etc… codes
- The weights associated with the bits in each code group are given by the name of the code

Nonweighted Codes

- **2421 code :** This is a weighted code; its weights are 2, 4, 2 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 2 + 4 + 2 + 1 = 9. Hence the 2421 code represents the decimal numbers from 0 to 9.
- **5211 code:** This is a weighted code; its weights are 5, 2, 1 and 1. A decimal number is represented in 4-bit form and the total four bits weight is 5 + 2 + 1 + 1 = 9. Hence the 5211 code represents the decimal numbers from 0 to 9.

Reflective code : A code is said to be reflective when code for 9 is complement for the code for 0, and so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential code : A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

- **Excess-3 code:** Excess-3 is a non weighted code used to express numbers. The code derives its corresponding 8421 code plus 0011(3).

Example: 1000 of 8421 = 1011 in Excess-3

- **Gray code :** The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. In digital Gray code has got a special place.

Decimal Number	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101

10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit-distance code.

Important when an analog quantity must be converted to a digital representation. Only one bit changes between two successive integers which are being coded.

Error Detecting and Correction Codes

- **Error detecting codes :** When data is transmitted from one point to another, like in wireless transmission, or it is just stored, like in hard disks and there are chances that data may get corrupted. To detect these data errors, we use special codes, which are error detection codes.

- **Error correcting code :** Error-correcting codes not only detect errors, but also correct them. This is used normally in Satellite communication, where turn-around delay is very high as is the probability of data getting corrupt.

- **Hamming codes :** Hamming code adds a minimum number of bits to the data transmitted in a noisy channel, to be able to correct every possible one-bit error. It can detect (not correct) two-bit errors and cannot distinguish between 1-bit and 2-bits inconsistencies. It can't - in general - detect 3(or more)-bits errors.

- **Parity codes :** A parity bit is an extra bit included with a message to make the total number of 1's either even or odd. In parity codes, every data byte, or nibble (according to how user wants to use it) is checked if they have even number of ones or even number of zeros. Based on this information an additional bit is appended to the original data. Thus if we consider 8-bit data, adding the parity bit will make it 9 bit long.
At the receiver side, once again parity is calculated and matched with the received parity (bit 9), and if they match, data is ok, otherwise data is corrupt.

Two types of parity

1) **Even parity:** Checks if there is an even number of ones; if so, parity bit is zero. When the number of one's is odd then parity bit is set to 1.

2) **Odd Parity:** Checks if there is an odd number of ones; if so, parity bit is zero. When the number of one's is even then parity bit is set to 1.

Alphanumeric codes

The binary codes that can be used to represent the letters of the alphabet, numbers and mathematical symbols, punctuation marks are known as alphanumeric codes or character codes. These codes enable us to interface the input-output devices like the keyboard, printers, video displays with the computer.

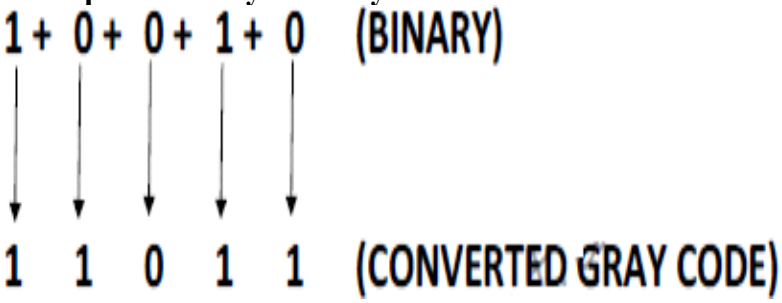
- **ASCII codes :** Codes to handle alphabetic and numeric information, special symbols, punctuation marks, and control characters. ASCII (American Standard Code for Information Interchange) is the best known. Unicode – a 16-bit coding system provides for foreign languages, mathematical symbols, geometrical shapes, dingbats, etc. It has become a world standard alphanumeric code for microcomputers and computers. It is a 7-bit code representing 128 different characters. These characters represe upper case letters (A to Z), 26 lowercase letters (a to z), 10 numbers (0 to 9), 33 special characters and symbols and 33 control characters.

- **EBCDIC codes :** EBCDIC stands for Extended Binary Coded Decimal Interchange. It is mainly used with large computer systems like mainframes. EBCDIC is an 8-bit code and thus accommodates up to 256 characters. An EBCDIC code is divided into two portions: 4 zone bits (on the left) and 4 numeric bits (on the right).

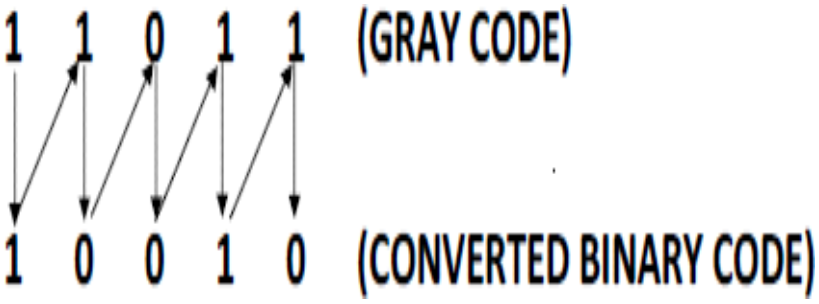
Example 1: Give the binary, BCD, Excess-3, gray code representations of numbers: 5,8,14.

Decimal Number	Binary code	BCD code	Excess-3 code	Gray code
5	0101	0101	1000	0111
8	1000	1000	1011	1100
14	1110	0001 0100	0100 0111	1001

Example 2: Binary To Gray Code Conversion



Example 3: Gray Code To Binary Code Conversion



BOOLEAN ALGEBRA

INTRODUCTION:

In 1854, George Boole, an English mathematician, proposed algebra for symbolically representing problems in logic so that they may be analyzed mathematically. The mathematical systems founded upon the work of Boole are called **Boolean algebra** in his honor.

Fundamental postulates of Boolean algebra:

The postulates of a mathematical system forms the basic assumption from which it is possible to deduce the theorems, laws and properties of the system.

The most common postulates used to formulate various structures are

Closure:

A set S is closed w.r.t. a binary operator, if for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S.

The result of each operation with operator (+) or (.) is either 1 or 0 and 1, 0 ЄB.

Identity element:

e* x = x * e = x

Eg: 0+ 0 = 0 0+ 1 = 1+ 0 = 1 a) **x+ 0= x**

1 . 1 = 1 1 . 0 = 0 . 1 = 1 b) **x. 1 = x**

Commutative law:

A binary operator * on a set S is said to be commutative if,

x * y = y * x

Eg: 0+ 1 = 1+ 0 = 1 a) **x+ y= y+ x**

0 . 1 = 1 . 0 = 0 b) **x. y= y. x**

Distributive law:

If * and • are two binary operation on a set S, • is said to be distributive over + whenever,

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

Similarly, + is said to be distributive over • whenever,
x + (y. z) = (x+ y). (x+ z)

Inverse:

a) x+ x' = 1, since 0 + 0' = 0+ 1 and 1+ 1' = 1+ 0 = 1

b) x. x' = 1, since 0 . 0' = 0. 1 and 1. 1' = 1. 0 = 0

Summary:

Postulates of Boolean algebra:

POSTULATES	**(a)**	**(b)**
Postulate 2 (Identity)	**x + 0 = x**	**x . 1 = x**
Postulate 3 (Commutative)	**x+ y = y+ x**	**x . y = y. x**
Postulate 4 (Distributive)	**x (y+ z) = xy+ xz**	**x+ yz = (x+ y). (x+ z)**
Postulate 5 (Inverse)	**x+x' = 1**	**x. x' = 0**

Basic theorem and properties of Boolean algebra:

Basic Theorems:

The theorems, like the postulates are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. The proofs of the theorems with one variable are presented below. At the right is listed the number of the postulate that justifies each step of the proof.

1a) **x+ x = x**
1b) x. x = x
2) **x .0 = 0**
3) (x')' = x
Absorption Theorem:
x+ xy = x

x+ xy = x. 1 + xy	-------	by postulate 2(b) [x. 1 = x]
= x (1+ y)	-------	4(a) [x (y+z) = (xy)+ (xz)]
= x (1)	-------	by theorem 2(a [x+ 1 = x]
= x.	-------	by postulate 2(a)[x. 1 = x]

x. (x+ y) = x

x. (x+ y) = x. x+ x. y	---------	4(a) [x (y+z) = (xy)+ (xz)]

= x + x.y ---------- by theorem 1(b) [x. x = x]
= x. ---------- by theorem 4(a) [x+ xy = x]

x+ x'y = x+ y

x+ x'y = x+ xy+ x'y ---------- by theorem 4(a) [x+ xy = x]
= x+ y (x+ x') ----------by postulate 4(a) [x (y+z) = (xy)+ (xz)]

= x+ y (1) ---------- 5(a)[x+ x' = 1]
= x+ y ---------- 2(b)[x. 1= x]

x. (x'+y) = xy

x. (x'+y) = x.x'+ xy --------- by postulate 4(a) [x (y+z) = (xy)+ (xz)]
= 0+ xy ----------5(b) [x. x' = 0]
= xy. ----------2(a) [x+ 0= x]

Properties of Boolean algebra:

Commutative property:

Boolean addition is commutative, given by

x+ y = y+ x

According to this property, the order of the OR operation conducted on the variables makes no difference.

Boolean algebra is also commutative over multiplication given by,

x. y = y. x

This means that the order of the AND operation conducted on the variables makes no difference.

Associative property:

The associative property of addition is given by,

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

The OR operation of several variables results in the same, regardless of the grouping of the variables.

The associative law of multiplication is given by,

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

It makes no difference in what order the variables are grouped during the AND operation of several variables.

Distributive property:

The Boolean addition is distributive over Boolean multiplication, given by

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

The Boolean addition is distributive over Boolean addition, given by

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

Duality:

It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

x+ x' = 1 is **x. x' = 0**

Duality is a very important property of Boolean algebra.

Summary:

Theorems of Boolean algebra:

	THEOREMS	(a)	(b)
1	Idempotent	**x + x = x**	**x . x = x**
		x + 1 = 1	**x . 0 = 0**
2	Involution	**(x')' = x**	
3	Absorption	**x+ xy = x**	**x (x+ y) = x**
		x+ x'y = x+ y	**x. (x'+ y)= xy**
4	Associative	**x+(y+ z)= (x+ y)+ z**	**x (yz) = (xy) z**
5	DeMorgan's Theorem	**(x+ y)'= x'. y'**	**(x. y)'= x'+ y'**

DeMorgan's Theorems:

Two theorems that are an important part of Boolean algebra were proposed by DeMorgan.

The first theorem states that the complement of a product is equal to the sum of the complements.

(AB)' = A'+ B'

The second theorem states that the complement of a sum is equal to the product of the complements.

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

Consensus Theorem:

In simplification of Boolean expression, an expression of the form **AB+ A'C+ BC**, the term BC is redundant and can be eliminated to form the equivalent expression AB+ A'C. The theorem used for this simplification is known as consensus theorem and is stated as,

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

The dual form of consensus theorem is stated as,

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

BOOLEAN FUNCTIONS:

Minimization of Boolean Expressions:

The Boolean expressions can be simplified by applying properties, laws and theorems of Boolean algebra.

Simplify the following Boolean functions to a minimum number of literals:

1. x (x'+y)
= xx'+ xy **[x. x'= 0]**
= 0 + xy **[x+ 0 = x]**
= xy.

2. x+ x'y
= x + xy + x'y **[x+ xy= x]**
= x+ y (x+x')
= x+ y (1) **[x+ x' = 1]**
= x+ y.

3. (x+ y) (x'+ z) (y+ z)
= (x+ y) (x'+ z) **[** dual form of consensus theorem**,**
(A+ B) (A'+ C) (B+ C) = (A+ B) (A'+ C)]

4. x'y+ xy+ x'y'
= y (x'+ x) + x'y' **[x (y+ z) = xy+ xz]**
= y (1) + x'y' **[x+ x' = 1]**
= y+ x'y' **[x+ x'y' = x+ y']**
= y+ x'.

5. x+ xy'+ x'y
= x (1+ y')+ x'y
= x (1) + x'y **[1+ x = 1]**
= x+ x'y **[x+ x'y = x+ y]**
= x+ y.

6. AB + (AC)' + AB'C (AB + C)
= AB + (AC)' + AAB'BC + AB'CC
= AB + (AC)' + 0+ AB'CC **[B.B' = 0]**
= AB + (AC)' + AB'C **[C.C = 1]**
= AB + A' + C' +AB'C **[(AC)' = A' + C']**
= AB + A' + C' + AB' **[C' + AB'C = C' + AB']**
= A' + B+ C'+ AB' **[A' + AB = A' + B]**
Re- arranging,
= A' + AB'+ B+ C' **[A' + AB = A' + B]**

= A' + B'+ B+ C' **[B'+ B= 1]**
= A' +1+ C' **[A+ 1= 1]**
= 1

7. (x'+ y) (x+ y)
= x'.x+ x'y+ yx+ y.y
= 0+ x'y+ xy+ y **[x.x'= 0]; [x. x= x]**
= y (x'+ x+ 1)
= y(1) **[1+ x = 1]**
= y.

8. x'yz+ xy'z'+ x'y'z'+ xy'z+ xyz
= yz (x'+x) + xy'z'+ x'y'z'+ xy'z
= yz (1) + y'z' (x+ x') + xy'z **[x+ x'=1]**
= yz+ y'z' (1) + xy'z **[x+ x'=1]**
= yz+ y'z'+ xy'z
= yz+ y' (z'+ xz)
= yz+ y' (z'+ x) **[x'+ xy = x'+ y]**
= yz+ y'z'+ xy'

9. [(xy)'+ x'+ xy]'
= [x'+ y'+ x'+ xy]'
= [x'+ y'+ xy]' **[x+ x= x]**

= [x'+ y'+ x]' **[x'+ xy = x'+ y]**
= [y'+ 1]' **[x+ x'= 1]**
= [1]' **[1+ x = 1]**
= 0.

10. [xy+ xz]'+ x'y'z
= (xy)'. (xz)'+ x'y'z
= (x'+ y'). (x'+ z')+ x'y'z
= x'x'+ x'z'+ x'y'+ y'z'+ x'y'z
= x'+ x'z'+ x'y'+ y'z'+ x'y'z **[x+ x= x]**
= x'+ x'z'+ x'y'+ y' [z'+ x'z]
= x'+ x'z'+ x'y'+ y' [z'+ x'] **[x'+ xy = x'+ y]**
= x'+ x'y'+ y' [z'+ x'] **[x+ xy = x]**
= x'+ x'y'+ y'z'+ x'y'
= x'+ y'z'+ x'y' **[x+ xy = x]**
= x'+ y'z'. **[x+ xy = x]**

11. xy+ xy'(x'z')'
= xy+ xy' (x''+ z'')
= xy+ xy' (x+ z) **[x'' = x]**
= xy+ xy'x+ xy'z

= xy+ xy'+ xy'z **[x. x= x]**
= xy+ xy' [1+ z]
= xy+ xy' [1] **[1+ x = 1]**
= xy+ xy'
= x(y+ y')
= x [1] **[x+ x'= 1]**
= x.

12. [(xy'+ xyz)'+ x (y+ xy')]'
= [x(y'+yz)'+ x (y+ xy')]'
= [x(y'+z)'+ x (y+ x)]' **[x'+ xy = x'+ y]; [x+ x'y = x+ y]**
= [x(y'+z)'+ xy+ x.x)]'
= [(xy'+xz)'+ xy+ x)]' **[x. x= x]**
= [(xy'+xz)'+ x)]' **[x+ xy = x]**
= [(xy')'. (xz)'+ x]'
= [(x'+y''). (x'+z')+ x]'
= [(x'+y). (x'+z')+ x]' **[x'' = x]**
= [(x'+yz')+ x]' **[(x+ y) (x+ z)= x+ yz]**
= [x'+ yz'+ x]'
= [1+ yz']' **[x+ x'= 1]**
= [1]' **[1+ x = 1]**
= 0.

COMPLEMENT OF A FUNCTION:

The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F. The complement of a function may be derived algebraically through DeMorgan's theorem.

DeMorgan's theorems for any number of variables resemble in form the two- variable case and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as –

(A+ B+ C+ D+ … + F)' = A' B' C' D' … F'

(A B C D … F)' = A'+B'+ C'+ D'+ … +F'.

Find the complement of the following functions,

1. F= x'yz'+ x'y'z

F'= (x'yz'+ x'y'z)'
= (x''+ y'+ z'') . (x''+ y''+z')
= (x+ y'+ z). (x+ y+ z').

2. F= (xy + y'z + xz) x.

F’ = [(xy + y’z + xz) x]’
= (xy + y’z + xz)’ + x’
= [(xy)’ . (y’z)’. (xz)’] + x’
= [(x’+y’). (y+z’). (x’+z’)] + x’
= [(x’y+ x’z’+ 0+ y’z’) (x’+z’)] + x’
= x’x’y+ x’x’z’+ x’y’z’+ x’yz’+ x’z’z’+ y’z’z’+ x’
= x’y+ x’z’+ x’y’z’+ x’yz’+ x’z’+ y’z’+ x’ [x+ x = x], [x. x =x]
= x’y+ x’z’+ x’z’ (y’+ y) + y’z’+ x’ [x+ x’= 1]
= x’y+ x’z’+ x’z’ (1) + y’z’+ x’
= x’y+ x’z’+ y’z’+ x’
= x’y+ x’+ x’z’+ y’z’
= x’(y+1) + x’z+ y’z’ [y+1= 1]
= x’ (1+z) + y’z’ [y+1= 1]
= x’+ y’z’

3. F= x (y’z’+ yz)

F’= [x (y’z’+yz)]’
= x’+ (y’z’+yz)’
= x’+ (y’z’)’. (yz)’
= x’+ (y”+ z”) . (y’+ z’)
= x’+ (y+ z) . (y’+ z’).

4. F= xy’+ x’y

F’= (xy’+ x’y)’
= (xy’)’. (x’y)’
= (x’+y) (x+y’)
= x’x+ x’y’+ yx+ yy’
= x’y’+ xy.

5. f = wx’y + xy’+ wxz

f’ = (wx’y + xy’+ wxz)’
= (wx’y)’ (xy’)’ (wxz)’
= (w’+x+ y’) (x’+ y) (w’+ x’+ z’)
= (w’x’+ w’y+ xx’+ xy+ x’y’+ yy’) (w’+ x’+ z’)
= (w’x’+ w’y+ xy+ x’y’) (w’+ x’+ z’)
= w’x’. w’+ w’y. w’+ xy. w’+ x’y’. w’+ w’x’. x’+w’y. x’+ xy. x’+ x’y’. x’+ w’x’. z’+ w’y. z’+ xy. z’+ x’y’.z’
= w’x’+ w’y+ w’xy+ w’x’y’+ w’x’+ w’x’y+ 0 + x’y’+ w’x’z’+ w’yz’+ xyz’+ x’y’z’
= w’x’+ w’y+ w’xy+ w’x’y’+ w’x’y+ x’y’+ w’x’z’+ w’yz’+ xyz’+ x’y’z’

= w'x'(1+ y'+ y+ z')+ w'y(1+ x+ z')+ x'y'(1+ z')+ xyz'

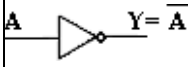
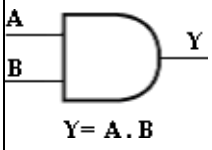
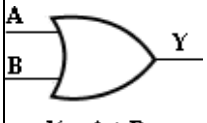
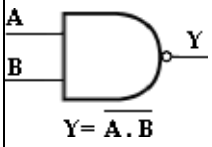
= w'x'(1)+ w'y(1)+ x'y'(1)+ xyz'

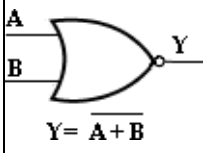
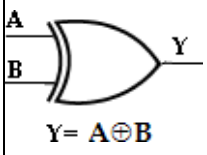
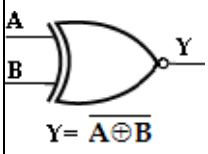
= w'x'+ w'y+ x'y'+ xyz'

LOGIC GATES BASIC LOGIC GATES:

Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinational logic.

There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates that are derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVE- OR gate and the EXCLUSIVE-NOR gate.

GATE	SYMBOL	OPERATION	TRUTH TABLE
NOT (7404)		NOT gate (Invertion), produces an inverted output pulse for a given input pulse.	<table><tr><th>A</th><th>Y= $\overline{A}$</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>
AND (7408)		AND gate performs logical **multiplication**. The output is HIGH only when all the inputs are HIGH. When any of the inputs are low, the output is LOW.	<table><tr><th>A</th><th>B</th><th>Y= A . B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>
OR (7432)		OR gate performs logical **addition**. It produces a HIGH on the output when any of the inputs are HIGH. The output is LOW only when all inputs are LOW.	<table><tr><th>A</th><th>B</th><th>Y= A+B</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>
NAND (7400)		It is a universal gate. When any of the inputs are LOW, the output will be HIGH. LOW output occurs only when all inputs are HIGH.	<table><tr><th>A</th><th>B</th><th>Y= $\overline{A . B}$</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>

NOR (7402)		It is a universal gate. LOW output occurs when any of its input is HIGH. When all its inputs are LOW, the output is HIGH.	A B $Y=\overline{A+B}$ 0 0 1 0 1 0 1 0 0 1 1 0
EX- OR (7486)		The output is HIGH only when odd number of inputs is HIGH.	A B Y= A⊕B 0 0 0 0 1 1 1 0 1 1 1 0
EX- NOR		The output is HIGH only when even number of inputs is HIGH. Or when all inputs are zeros.	A B Y= A⊙B 0 0 1 0 1 0 1 0 0 1 1 1

<u>UNIVERSAL GATES</u>:

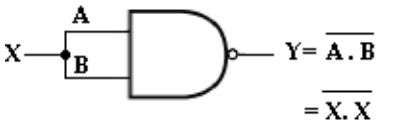
The NAND and NOR gates are known as universal gates, since any logic function can be implemented using NAND or NOR gates. This is illustrated in the following sections.

- **<u>NAND Gate</u>**:

The NAND gate can be used to generate the NOT function, the AND function,

the OR function and the NOR function.

- <u>NOT function</u>:



	A	B	$Y=\overline{A.B}$	
X= 0	0	0	1	Y= 1
	0	1	1	
	1	0	1	
X= 1	1	1	0	Y= 0

By connecting all the inputs together and creating a single common input.

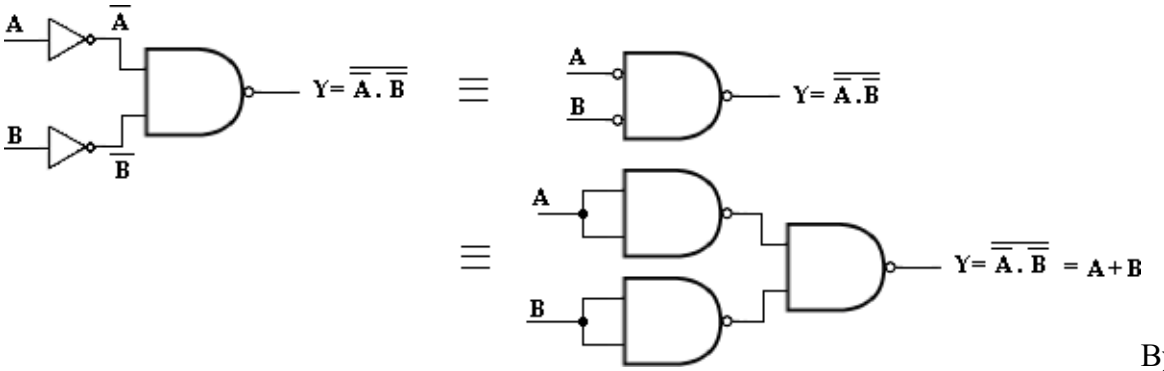
<u>NOT function using NAND gate</u>

- <u>AND function</u>:

By simply inverting output of the NAND gate. i.e.,

AND function using NAND gates

- OR function:



By simply inverting inputs of the NAND gate. i.e.,

OR function using NAND gates

Bubble at the input of NAND gate indicates inverted input.

A	B	Y= A+B
0	0	0
0	1	1
1	0	1
1	1	1

≡

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

- NOR function**:**

By inverting inputs and outputs of the NAND gate.

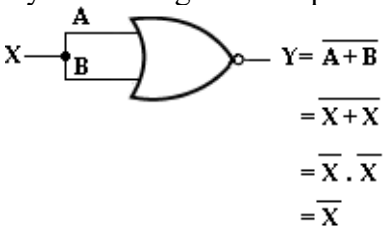
NOR function using NAND gates

- **NOR Gate**:

Similar to NAND gate, the NOR gate is also a universal gate, since it can be used to generate the NOT, AND, OR and NAND functions.

- NOT function:

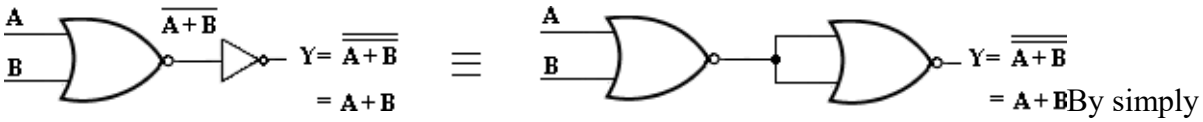
By connecting all the inputs together and creating a single common input.



	A	B	$Y=\overline{A+B}$	
X= 0	0	0	1	Y= 1
	0	1	0	
	1	0	0	
X= 1	1	1	0	Y= 0

NOT function using NOR gates

- OR function:



By simply inverting output of the NOR gate. i.e.,

OR function using NOR gates

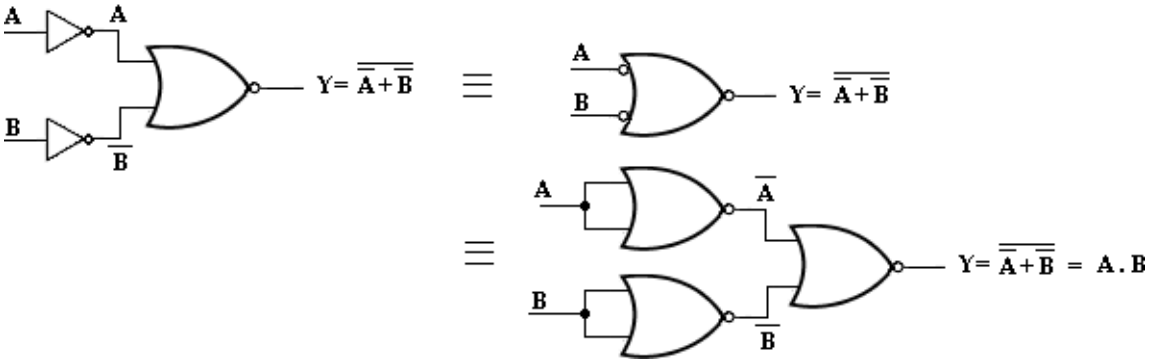
A	B	Y= A+B
0	0	0
0	1	1
1	0	1
1	1	1

≡

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

- AND function:

By simply inverting inputs of the NOR gate. i.e.,



AND function using NOR gates

Bubble at the input of NOR gate indicates inverted input.

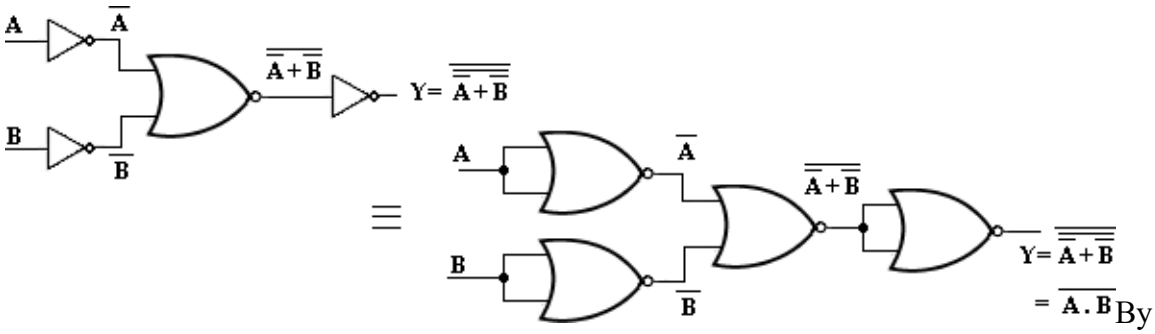
A	B	Y= A.B
0	0	0
0	1	0
1	0	0
1	1	1

≡

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

Truth table

- NAND Function:



By inverting inputs and outputs of the NOR gate.

NAND function using NOR gates

A	B	$Y=\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

≡

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

Conversion of AND/OR/NOT to NAND/NOR:

- Draw AND/OR logic.
- If NAND hardware has been chosen, add bubbles on the output of each AND gate and bubbles on input side to all OR gates.

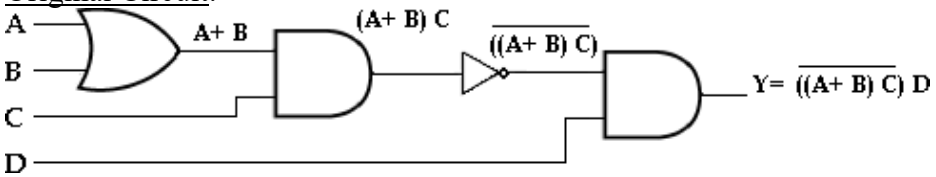
If NOR hardware has been chosen, add bubbles on the output of each OR gate and bubbles on input side to all AND gates.

- Add or subtract an inverter on each line that received a bubble in step 2.
- Replace bubbled OR by NAND and bubbled AND by NOR.
- Eliminate double inversions.

Implement Boolean expression using NAND gates:

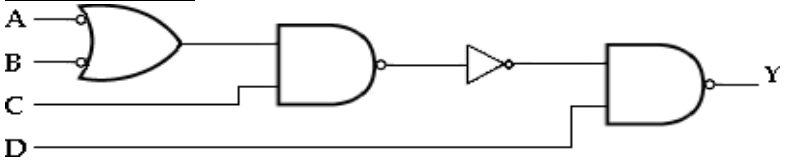
$\overline{((A+B)C)}\,D$

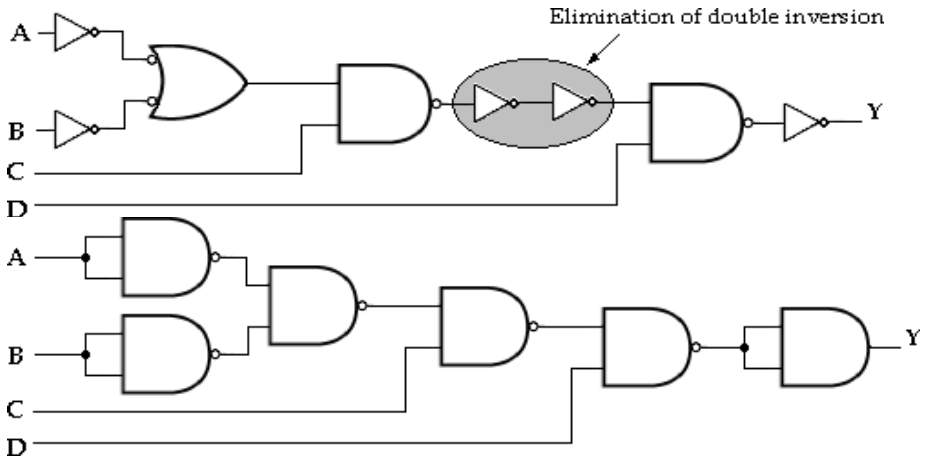
Original Circuit:



Soln: _____

NAND Circuit:




Elimination of double inversion
A
B
C
D
Y
A
B
C
D
Y

TEXT BOOKS:

1. Morris Mano, "Digital design", 3rd Edition, Prentice Hall of India, 2008.

REFERENCE BOOKS:

1. Milos Ercegovac, Jomas Lang, "Introduction to Digital Systems", Wiley publications, 1998.
2. John M. Yarbrough, "Digital logic: Applications and Design", Thomas – Vikas Publishing House, 2002.
3. R.P.Jain, "Modern digital Electronics",3rd Edition, TMH, 2003.
4. William H. Gothmann, "Digital Electronics", Prentice Hall, 2001.

QUESTION BANK

PART-A

1.Practice Subtraction of $(0101)_2$ from$(1110)_2$ using 2's complement method.
2. Describe BCD code and its advantages.
3. Illustrate the Logic Diagram for AND, OR and NOT using only NAND.
4. Convert $(345.54)_8$ to decimal.
5. Convert $(ABC.EF)_{16}$ to Octal.
6. State DeMargan's Theorems.
7. Relate the equivalent Gray code for the binary number 110110.
8. Define the meaning of self-complementing code, Give an example.
9. Recall 2 input XOR gate using only NOR gate.
10. Illustrate 2 input XNOR gate using only NAND gate.

PART-B

1. Implement AND, OR, NOT, XOR, XNOR using universal gates.
2. Practice the following conversion
   (a) $(11101)_2$= $(?)_{10}$
   (b) $(1110101)_2$ = $(?)_{16}$
   (c) $(24.32)_{10}$ = $(?)_2$
   (d) $(5E7)_{16}$= $(?)_2$
   (e) $(157)_8$= $(?)_2$
3.Examine and simplify the following boolean expressions
   a) AB C+A'B+ABC'
   b) X'YZ+XZ
   c) (X+Y)' (X'+Y')
4.Explain briefly about the binary codes.
5.Apply Boolean theorems to simplify the following expression to a minimum number of literals.
   i. ABC+A'B+ABC'
   ii. (X+Y)' (X'+Y')
   iii. (BC'+A'D)(AB'+CD')
   iv. a+ab'+ab'c'+ab'c'd'

UNIT II
DESIGN OF COMBINATIONAL LOGIC

Design procedure of Combinational Logic – Design of two level gate networks -Sum of Products (SOP) - Product of Sums(POS) - Canonical SOP - Canonical POS - Karnaugh Map - Simplifications of Boolean functions using Karnaugh Map and implementation using Logic function – Advantages and limitations of K-Map - Tabulation method - Simplifications of Boolean functions using Tabulation method.

CANONICAL AND STANDARD FORMS:

Minterms and Maxterms:

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now either two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations:

x'y', x'y, xy' and xy Each of these four AND terms is called a '**minterm**'.

In a similar fashion, when two binary variables x and y combined with an OR operation, there are four possible combinations: x'+ y', x'+ y, x+ y' and x+ y

Each of these four OR terms is called a '**maxterm**'.

The minterms and maxterms of a 3- variable function can be represented as in table below.

Variables			**Minterms**	**Maxterms**
x	**y**	**Z**	**m_i**	**M_i**
0	0	0	x'y'z' = m0	x+ y+ z= M_0
0	0	1	x'y'z = m1	x+ y+ z'= M1
0	1	0	x'yz' = m2	x+ y'+ z= M2
0	1	1	x'yz = m3	x+ y'+ z'= M3
1	0	0	xy'z' = m4	x'+ y+ z= M_4
1	0	1	xy'z = m5	x'+ y+ z'= M5
1	1	0	xyz' = m6	x'+ y'+ z= M6
1	1	1	xyz = m_7	x'+ y'+ z'= M7

Sum of Minterm: (Sum of Products)

1. Y= AB+ BC+ AC

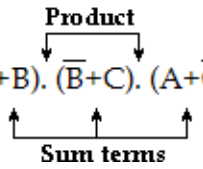
2. Y= $AB+ \bar{B}C+ A\bar{C}$ (Sum; Product terms)

The logical sum of two or more logical product terms is called sum of products expression. It is logically an OR operation of AND operated variables such as:

Sum of Maxterm: (Product of Sums)

1. $Y= (A+B).(B+C).(A+C)$

2. $Y= (A+B).(\overline{B}+C).(A+\overline{C})$



A product of sums expression is a logical product of two or more logical sum terms. It is basically an AND operation of OR operated variables such as,

Canonical Sum of product expression:

If each term in SOP form contains all the literals then the SOP is known as standard (or) canonical SOP form. Each individual term in standard SOP form is called minterm canonical form.

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

Steps to convert general SOP to standard SOP form:

1) Find the missing literals in each product term if any.
2) AND each product term having missing literals by ORing the literal and its complement.
3) Expand the term by applying distributive law and reorder the literals in the product term.
4) Reduce the expression by omitting repeated product terms if any.

Obtain the canonical SOP form of the function:

1) Y(A, B) = A+ B

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

= AB+ AB'+ AB+ A'B

= AB+ AB'+ A'B.

2) Y (A, B, C) = A+ ABC

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

= (AB+ AB'). (C+ C')+ ABC

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

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

= $m_7+ m_6+ m_5+ m_4$

= $\sum m (4, 5, 6, 7)$.

3) Y (A, B, C) = A+ BC

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

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

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

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

= m_7+ m_6+ m_5+ m_4+ m_3

= ∑m (3, 4, 5, 6, 7).

4) Y (A, B, C) = AC+ AB+ BC

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

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

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

= ∑m (3, 5, 6, 7).

5) Y (A, B, C, D) = AB+ ACD

= AB (C+ C') (D+ D') + ACD (B+ B')

= (ABC+ ABC') (D+ D') + ABCD+ AB'CD

= ABCD+ ABCD'+ ABC'D+ ABC'D'+ ABCD+ AB'CD

= ABCD+ ABCD'+ ABC'D+ ABC'D'+ AB'CD.

Canonical Product of sum expression:

If each term in POS form contains all literals then the POS is known as standard (or) Canonical POS form. Each individual term in standard POS form is called Maxterm canonical form.

- F (A, B, C) = (A+ B+ C). (A+ B'+ C). (A+ B+ C')
- F (x, y, z) = (x+ y'+ z'). (x'+ y+ z). (x+ y+ z)

Steps to convert general POS to standard POS form:

1) Find the missing literals in each sum term if any.
2) OR each sum term having missing literals by ANDing the literal and its complement.
3) Expand the term by applying distributive law and reorder the literals in the sum term.
4) Reduce the expression by omitting repeated sum terms if any.

Obtain the canonical POS expression of the functions:

1. Y= A+ B'C

= (A+ B') (A+ C) [**A+ BC = (A+B) (A+C)**]

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

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

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

= M_2. M_3. M_0

= ∏M (0, 2, 3)

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

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

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

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

= M_0. M_1. M_4. M_2

= ∏M (0, 1, 2, 4)

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

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

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

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

= M_0. M_1. M_2. M_3

= ∏M (0, 1, 2, 3)

4. Y= (A+B') (B+C) (A+C')

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

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

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

= M_2. M_3. M_0. M_4. M_1

= ∏M (0, 1, 2, 3, 4)

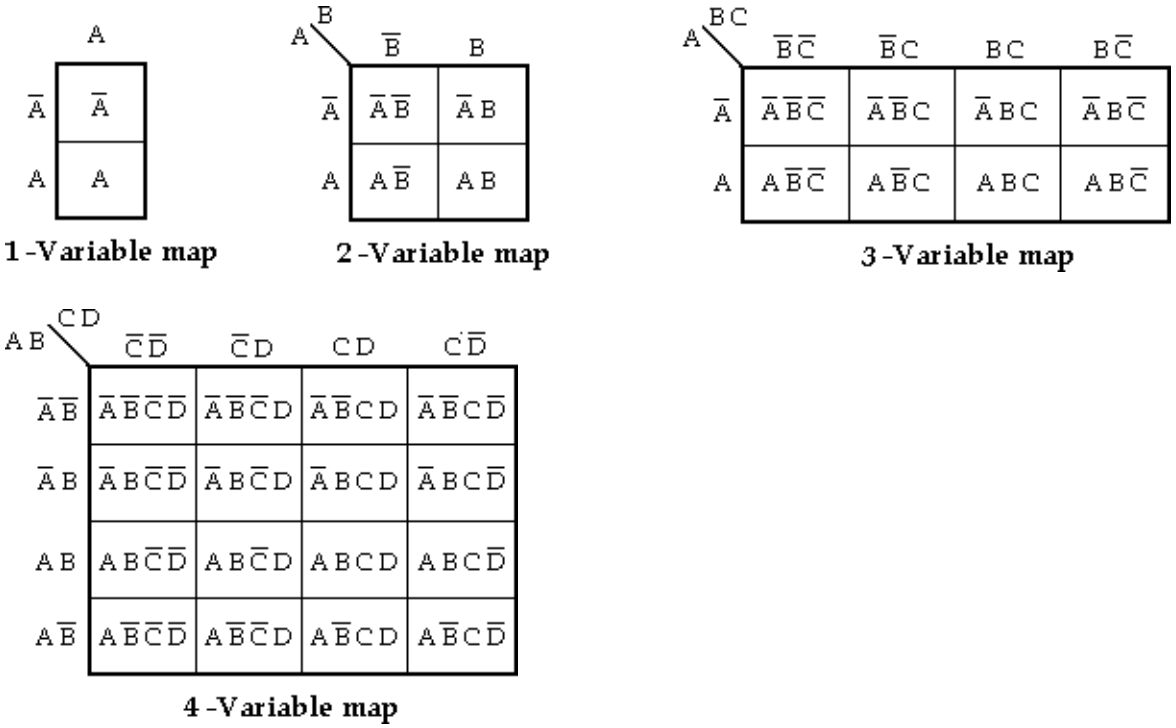
KARNAUGH MAP MINIMIZATION:

The simplification of the functions using Boolean laws and theorems becomes complex with the increase in the number of variables and terms. The map method, first proposed by Veitch and slightly improvised by Karnaugh, provides a simple, straightforward procedure for the simplification of Boolean functions. The method is called **Veitch diagram** or **Karnaugh map**, which may be regarded as a pictorial representation of a truth table.

The Karnaugh map technique provides a systematic method for simplifying and manipulation of Boolean expressions. **A** K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. For n variables on a Karnaugh map there are 2^n numbers of squares. Each square or cell represents one of the minterms. It can be drawn directly from either minterm (sum-of-products) or maxterm (product-of-sums) Boolean expressions.

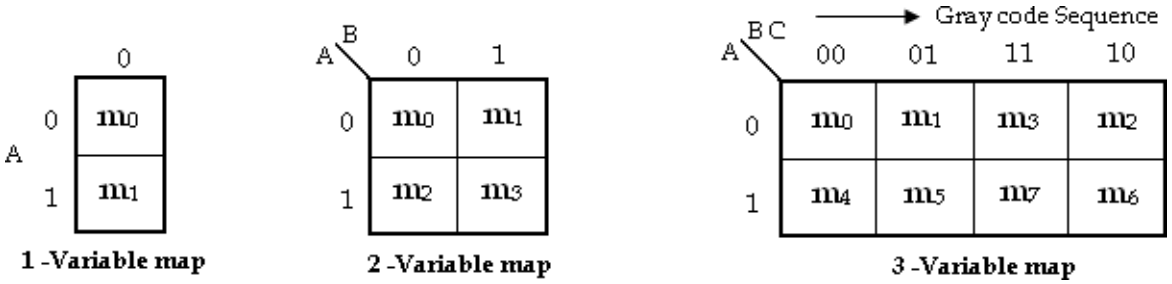
Two- Variable, Three Variable and Four Variable Maps

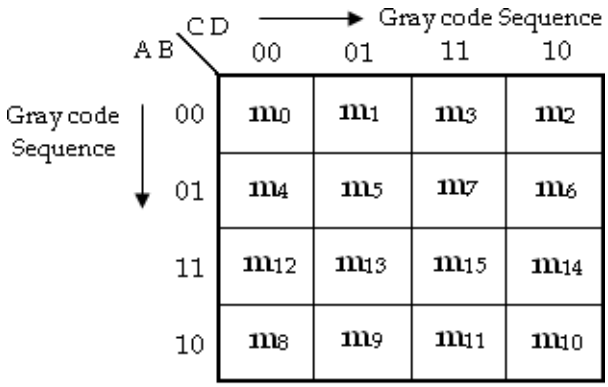
Karnaugh maps can be used for expressions with two, three, four and five variables. The number of cells in a Karnaugh map is equal to the total number of possible input variable combinations as is the number of rows in a truth table. For three variables, the number of cells is $2^3 = 8$. For four variables, the number of cells is $24 = 16$.



Product terms are assigned to the cells of a K-map by labeling each row and each column of a map with a variable, with its complement or with a combination of variables & complements. The below figure shows the way to label the rows & columns of a 1, 2, 3 and 4- variable maps and the product terms corresponding to each cell.

It is important to note that when we move from one cell to the next along any row or from one cell to the next along any column, one and only one variable in the product term changes (to a complement or to an uncomplemented form). Irrespective of number of variables the labels along each row and column must conform to a single change. Hence gray code is used to label the rows and columns of K-map as shown ow.





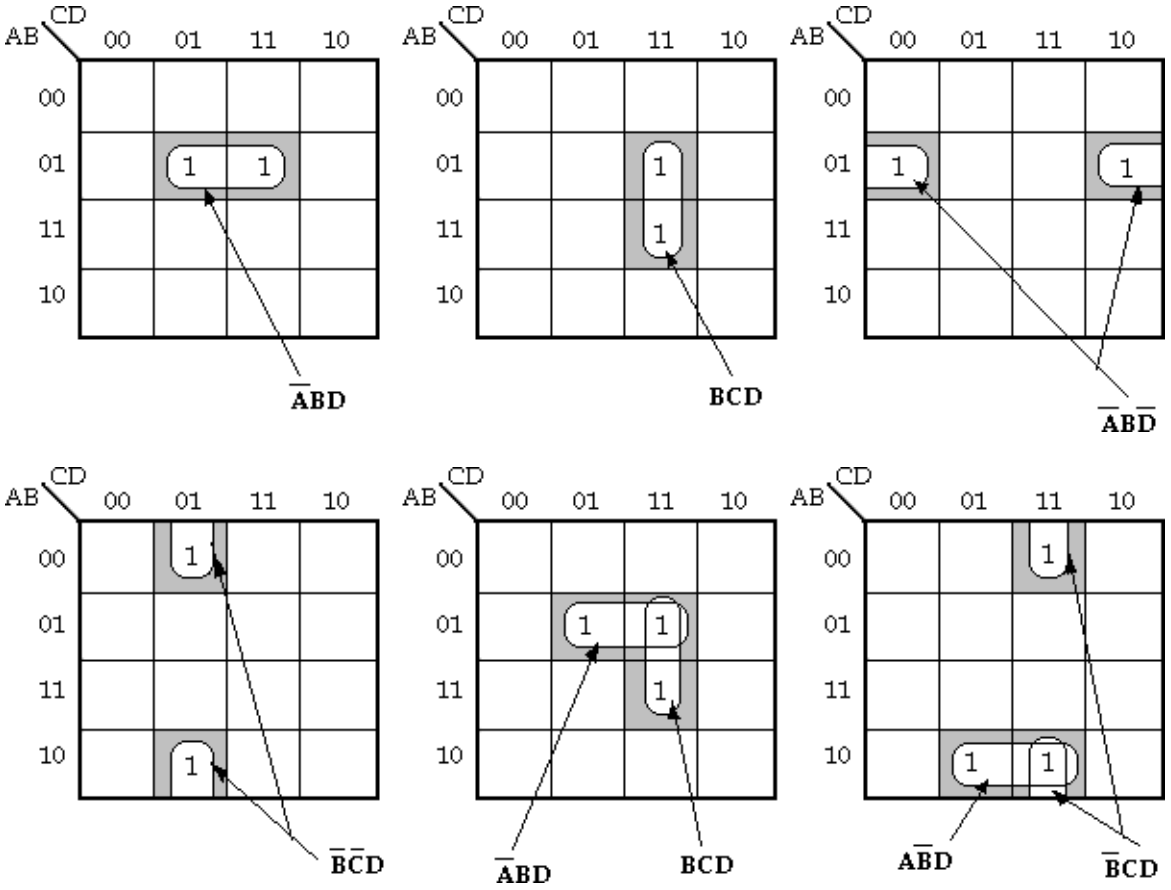
4- Variable map

Grouping cells for Simplification:

The grouping is nothing but combining terms in adjacent cells. The simplification is achieved by grouping adjacent 1's or 0's in groups of 2i, where i = 1, 2, …, n and n is the number of variables. When adjacent 1's are grouped then we get result in the sum of product form; otherwise we get result in the product of sum form.

Grouping Two Adjacent 1's: (Pair)

In a Karnaugh map we can group two adjacent 1's. The resultant group is called Pair.



Examples of Pairs

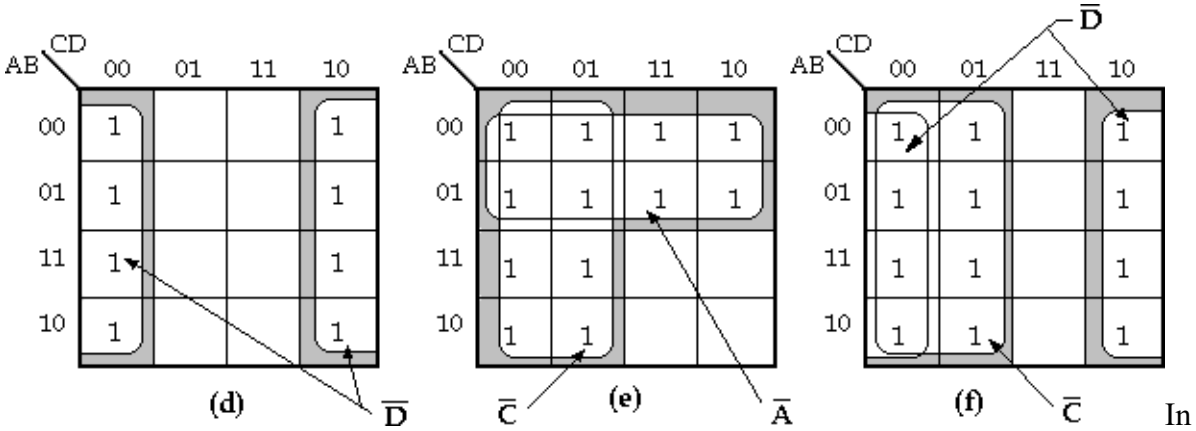
Grouping Four Adjacent 1's: (Quad)

In a Karnaugh map we can group four adjacent 1's. The resultant group is called Quad. Fig (a) shows the four 1's are horizontally adjacent and Fig (b) shows they are vertically adjacent. Fig (c) contains four 1's in a square, and they are considered adjacent to each other.

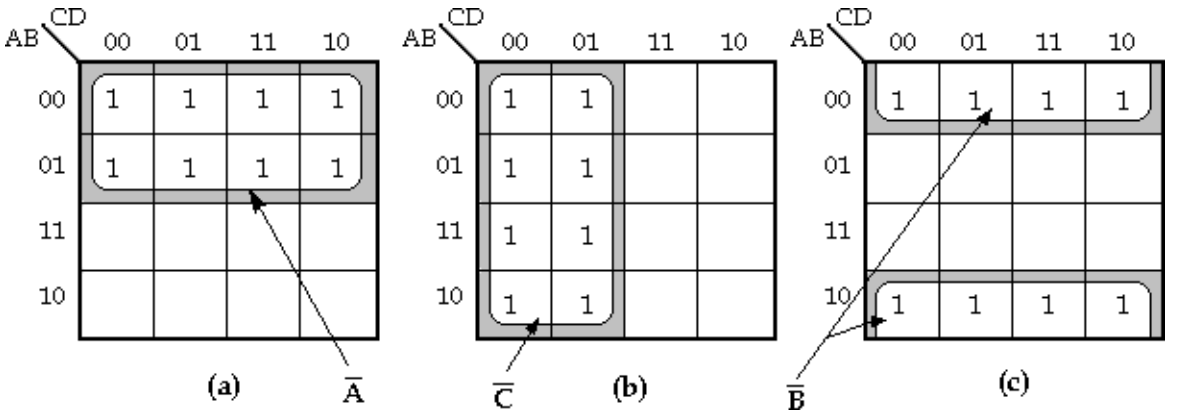
Examples of Quads

The four 1's in fig (d) and fig (e) are also adjacent, as are those in fig (f) because, the top and bottom rows are considered to be adjacent to each other and the leftmost and rightmost columns are also adjacent to each other.

Grouping Eight Adjacent 1's: (Octet)



In a Karnaugh map we can group eight adjacent 1's. The resultant group is called Octet.



Simplification of Sum of Products Expressions: (Minimal Sums)

The generalized procedure to simplify Boolean expressions as follows:

1) Plot the K-map and place 1's in those cells corresponding to the 1's in the sum of product expression. Place 0's in the other cells.
2) Check the K-map for adjacent 1's and encircle those 1's which are not adjacent to any other 1's. These are called **isolated 1's**.

3) Check for those 1's which are adjacent to only one other 1 and encircle such **pairs**.

4) Check for **quads** and **octets** of adjacent 1's even if it contains some 1's that have already been encircled. While doing this make sure that there are minimum number of groups.

5) Combine any pairs necessary to include any 1's that have not yet been grouped.

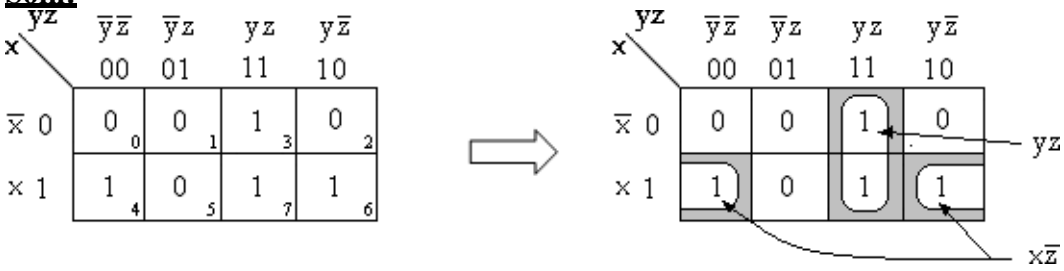
6) Form the simplified expression by summing product terms of all the groups.

Three- Variable Map:

1. Simplify the Boolean expression,

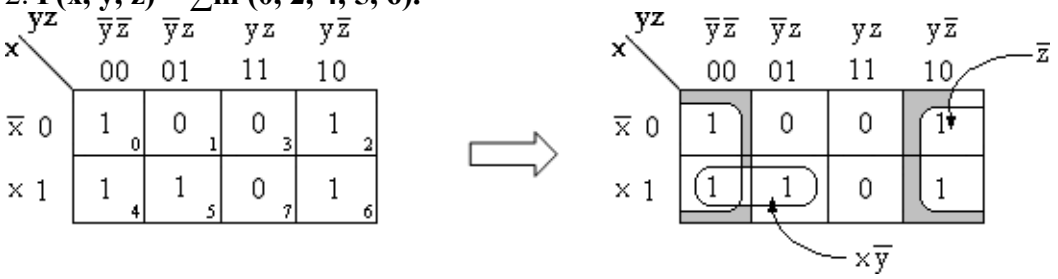
F(x, y, z) = ∑m (3, 4, 6, 7).

Soln:



F = yz+ xz'

2. **F(x, y, z) = ∑m (0, 2, 4, 5, 6).**



F = z'+ xy'

3. F = A'C + A'B + AB'C + BC

Soln:

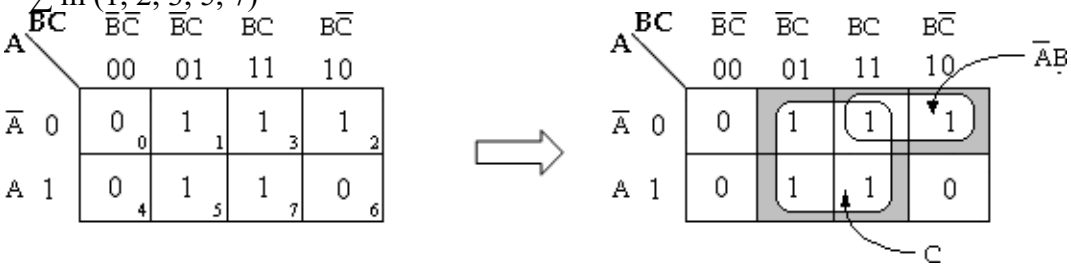
= A'C (B+ B') + A'B (C+ C') + AB'C + BC (A+ A')

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

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

= $m_3 + m_1 + m_2 + m_5 + m_7$

$= \sum m\ (1, 2, 3, 5, 7)$



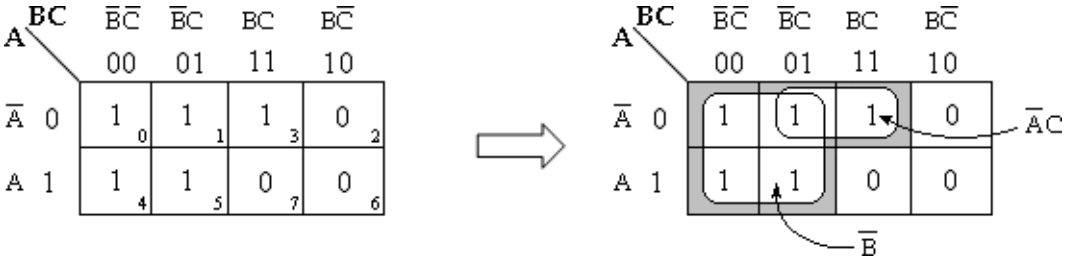
F = C + A'B

4. AB'C + A'B'C + A'BC + AB'C' + A'B'C'

Soln:

= m5 + m1 + m3 + m4 + m0

$= \sum m\ (0, 1, 3, 4, 5)$

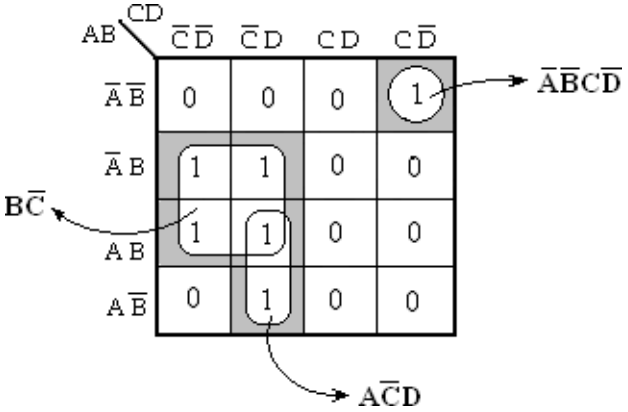


F = A'C + B'

Four - Variable Map:

1. Simplify the Boolean expression,

Y = A'BC'D' + A'BC'D + ABC'D' + ABC'D + AB'C'D + A'B'CD'

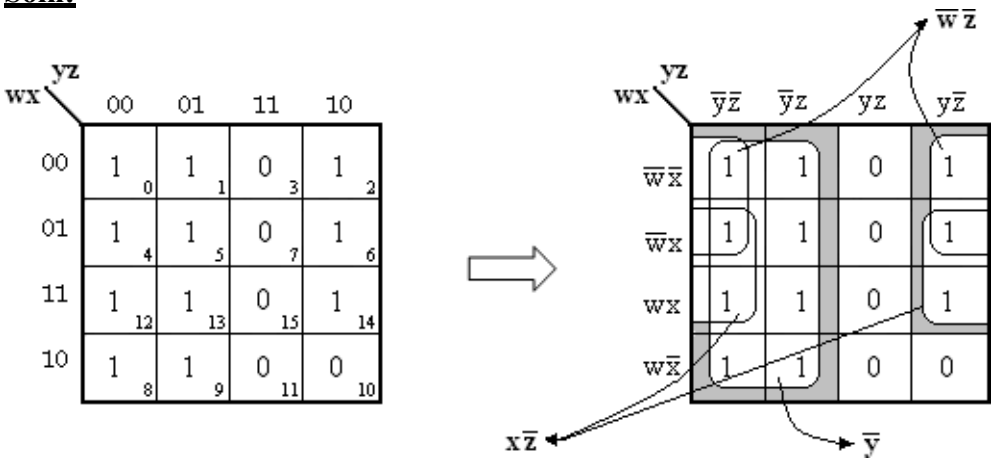


Soln:

Therefore, Y= A'B'CD'+ AC'D+ BC'

2. **F (w, x, y, z) = ∑ m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)**

Soln:



Therefore,

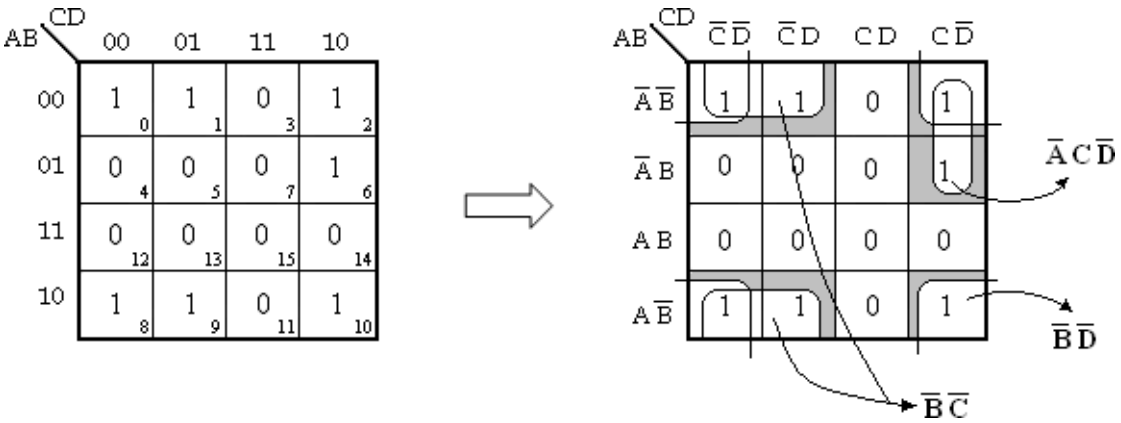
F= y'+ w'z'+ xz'

3.F= A'B'C'+ B'CD'+ A'BCD'+ AB'C'

= A'B'C' (D+ D') + B'CD' (A+ A') + A'BCD'+ AB'C' (D+ D')

= A'B'C'D+ A'B'C'D'+ AB'CD'+ A'B'CD'+ A'BCD'+ AB'C'D+ AB'C'D'

= m_1+ m_0+ m_{10}+ m_2+ m_6+ m_9+ m_8



= ∑ **m (0, 1, 2, 6, 8, 9, 10)**

Therefore,

F= B'D'+ B'C'+ A'CD'.

4.Y= ABCD+ AB'C'D'+ AB'C+ AB

= ABCD+ AB'C'D'+ AB'C (D+D')+ AB (C+C') (D+D')

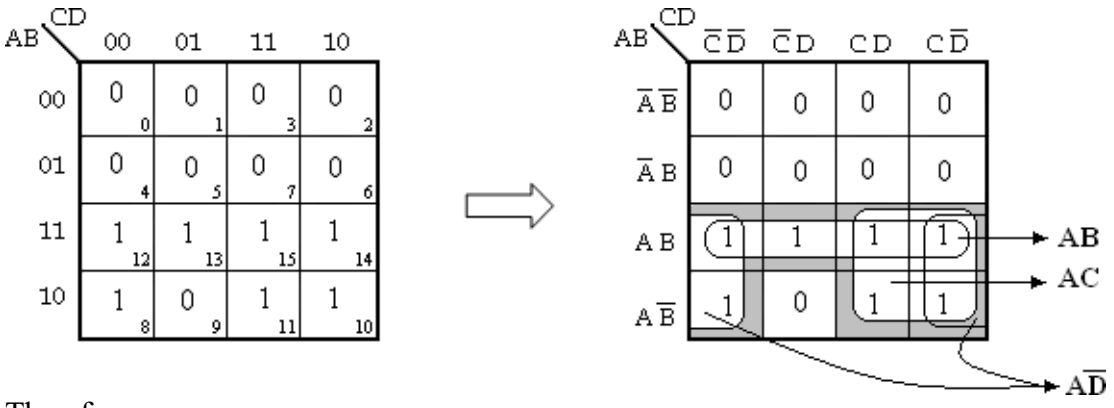
= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ (ABC+ ABC') (D+ D')

= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ ABCD+ ABCD'+ ABC'D+ ABC'D'

= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ ABCD'+ ABC'D+ ABC'D'

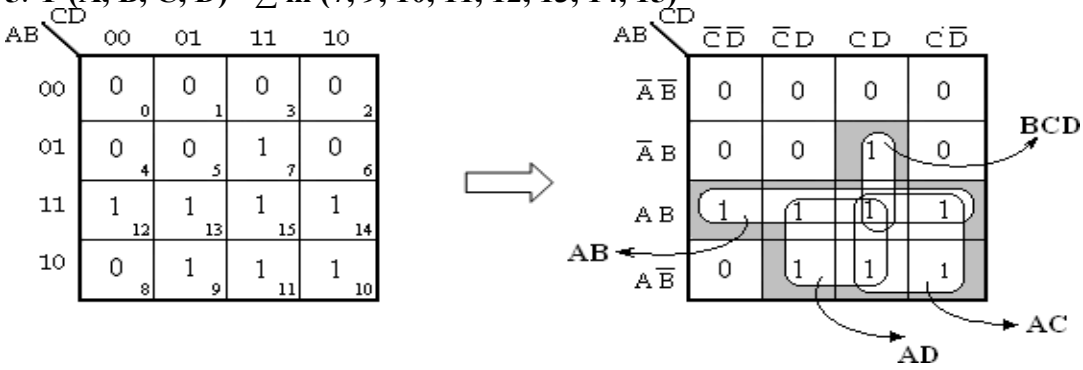
= m_{15}+ m_8+ m_{11}+ m_{10}+ m_{14}+ m_{13}+ m_{12}

= ∑ **m (8, 10, 11, 12, 13, 14, 15)**



Therefore,
Y= AB+ AC+ AD'.

5. Y (A, B, C, D)= ∑ m (7, 9, 10, 11, 12, 13, 14, 15)

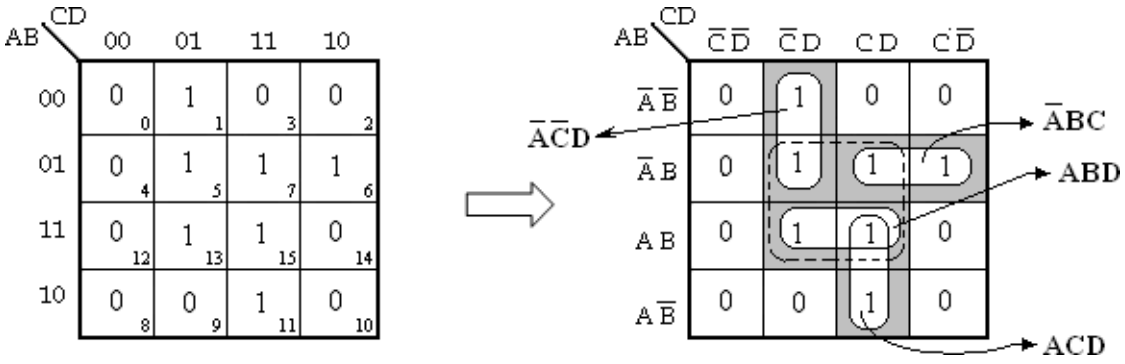


Therefore, **Y= AB+ AC+ AD+BCD.**

6. Y= A'B'C'D+ A'BC'D+ A'BCD+ A'BCD'+ ABC'D+ ABCD+ AB'CD

= m_1+ m_5+ m_7+ m_6+ m_{13}+ m_{15}+ m_{11}

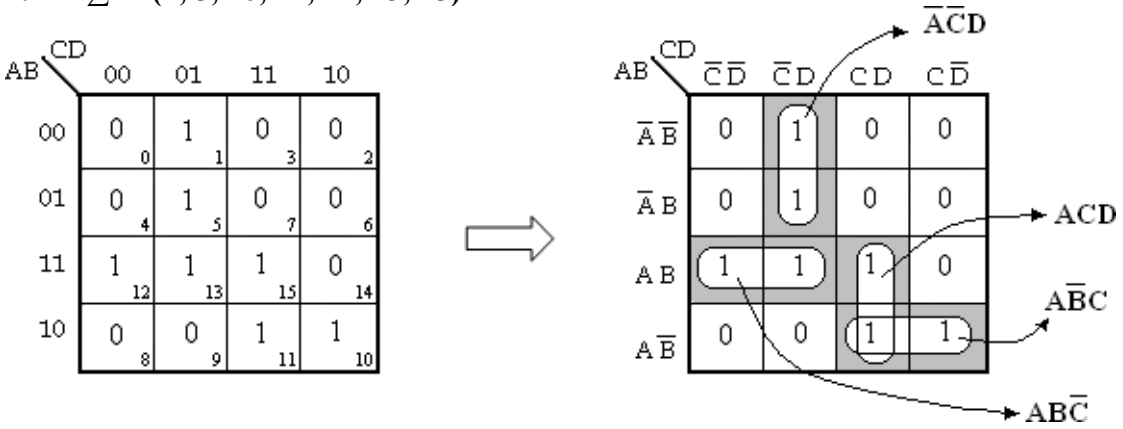
= ∑ **m (1, 5, 6, 7, 11, 13, 15)**



In the above K-map, the cells 5, 7, 13 and 15 can be grouped to form a quad as indicated by the dotted lines. In order to group the remaining 1's, four pairs have to be formed. However, all the four 1's covered by the quad are also covered by the pairs. So, the quad in the above k-map is redundant.
Therefore, the simplified expression will be,

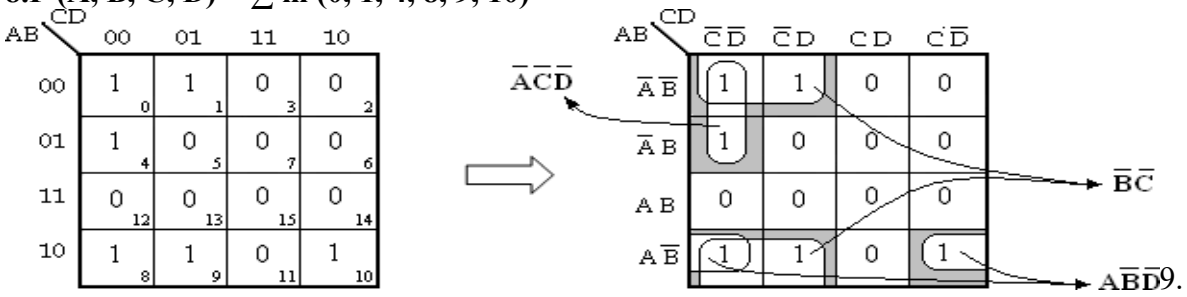
Y = A'C'D+ A'BC+ ABD+ ACD.

7. Y= ∑ m (1, 5, 10, 11, 12, 13, 15)



Therefore, **Y= A'C'D+ ABC'+ ACD+ AB'C.**

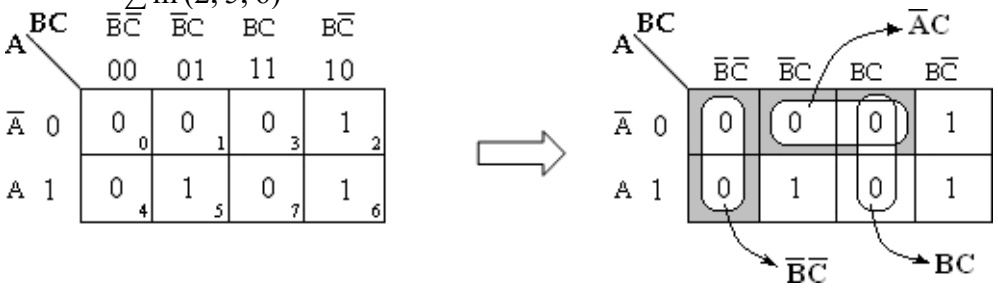
8.F (A, B, C, D) = ∑ m (0, 1, 4, 8, 9, 10)



Therefore, **F= A'C'D'+ AB'D'+ B'C'.**

<u>Simplification of Sum of Products Expressions: (Minimal Sums)</u>

1. **Y= (A+ B+ C') (A+ B'+ C') (A'+ B'+ C') (A'+ B+ C) (A+ B+ C)**
   = $M_1 . M_3 . M_7 . M_4 . M_0$
   = ∏ M (0, 1, 3, 4, 7)
   = ∑ m (2, 5, 6)



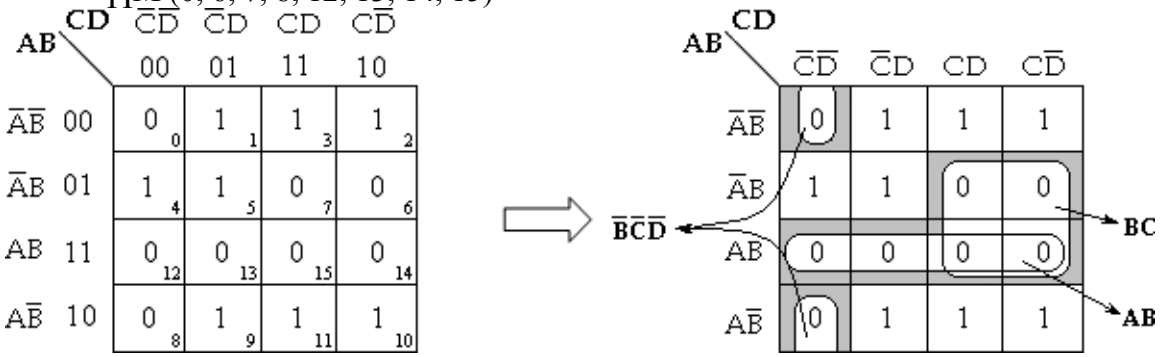
Y' = B'C'+ A'C+ BC.

Y= Y" = (B'C'+ A'C+ BC)'
= (B'C')'. (A'C)'. (BC)'
= (B"+ C"). (A"+C'). (B'+ C')
Y = (B+ C). (A+C'). (B'+ C')

2. **Y= (A'+ B'+ C+ D) (A'+ B'+ C'+ D) (A'+ B'+ C'+ D') (A'+ B+ C+ D) (A+ B'+ C'+ D) (A+ B'+ C'+ D') (A+ B+ C+ D) (A'+ B'+ C+ D')**

= $M_{12}. M_{14}. M_{15}. M_8. M_6. M_7. M_0. M_{13}$

= ∏M (0, 6, 7, 8, 12, 13, 14, 15)



Y' = B'C'D'+ AB+ BC

Y= Y" = (B'C'D'+ AB+ BC)'

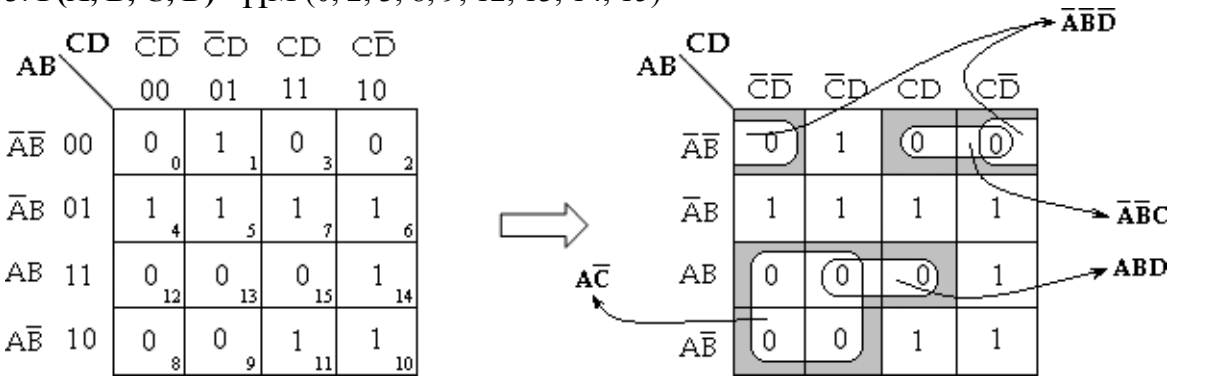
= (B'C'D')'. (AB)'. (BC)'

= (B"+ C"+D"). (A'+B'). (B'+ C')

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

Therefore, **Y= (B+ C+ D). (A'+ B'). (B'+ C')**

3. **F(A, B, C, D)=** ∏M (0, 2, 3, 8, 9, 12, 13, 14, 15)



Y' = A'B'D'+ A'B'C+ ABD+ AC'

Y= Y" = (A'B'D'+ A'B'C+ ABD+ AC')'

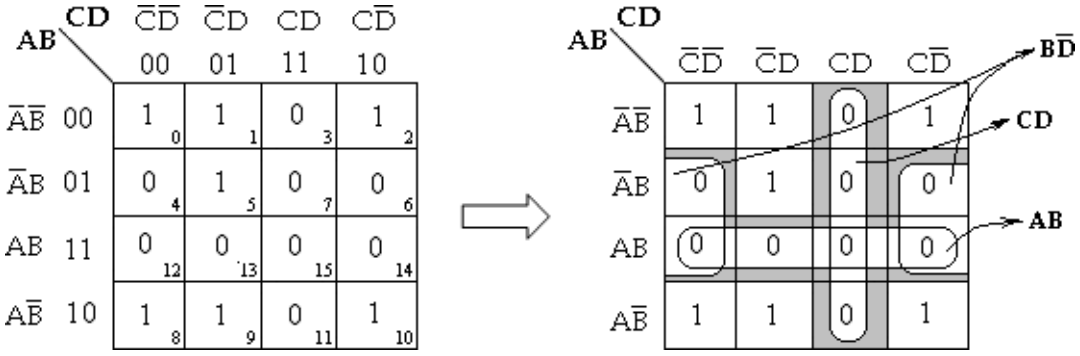
= (A'B'D')'. (A'B'C)'. (ABD)'. (AC')'

= (A"+ B"+ D"). (A"+ B"+C'). (A'+ B'+ D'). (A'+ C")

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

Therefore, **Y= (A+ B+ D). (A+ B+ C'). (A'+ B'+ D'). (A'+ C)**

4. **F(A, B, C, D)**= ∑m (0, 1, 2, 5, 8, 9, 10)
= ∏M (3, 4, 6, 7, 11, 12, 13, 14, 15)

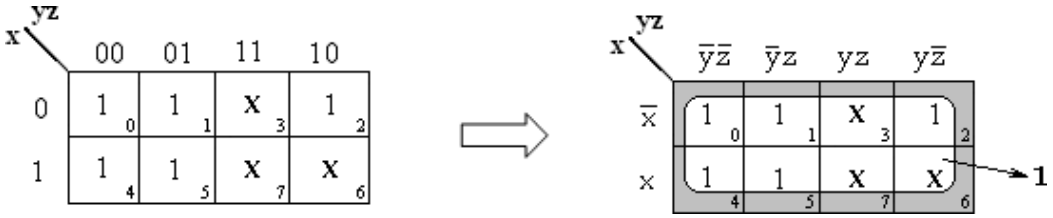


Y' = BD'+ CD+ AB

Y= Y" = (BD'+ CD+ AB)'
= (BD')'. (CD)'. (AB)' = (B'+ D"). (C'+ D'). (A'+ B')
= **(B'+ D). (C'+ D'). (A'+ B')**
Therefore, **Y= (B'+ D). (C'+ D'). (A'+ B')**

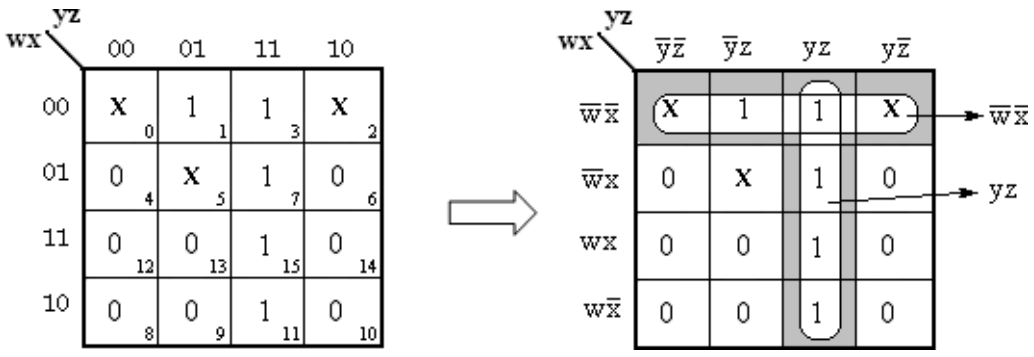
<u>Don't care Conditions</u>:
A don't care minterm is a combination of variables whose logical value is not specified. When choosing adjacent squares to simplify the function in a map, the don't care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

1. F (x, y, z) = ∑m (0, 1, 2, 4, 5)+ ∑d (3, 6, 7)



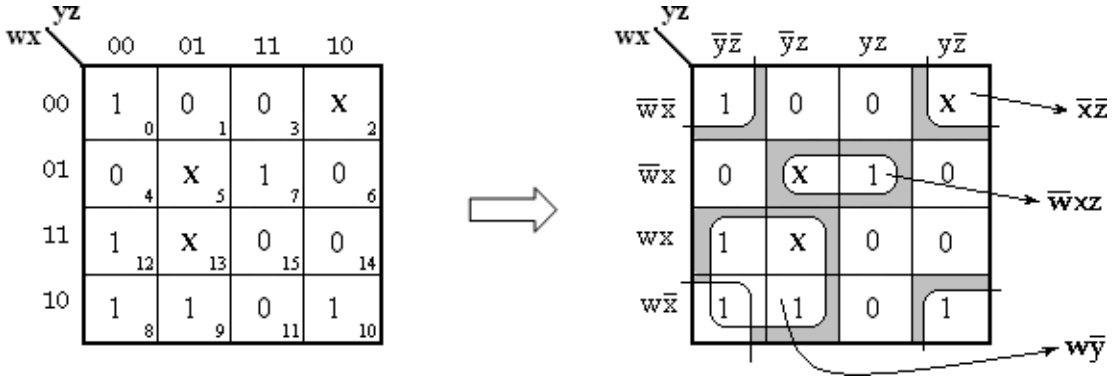
F (x, y, z) = 1

2. F (w, x, y, z) = ∑m (1, 3, 7, 11, 15)+ ∑d (0, 2, 5)



F (w, x, y, z) = w'x'+ yz

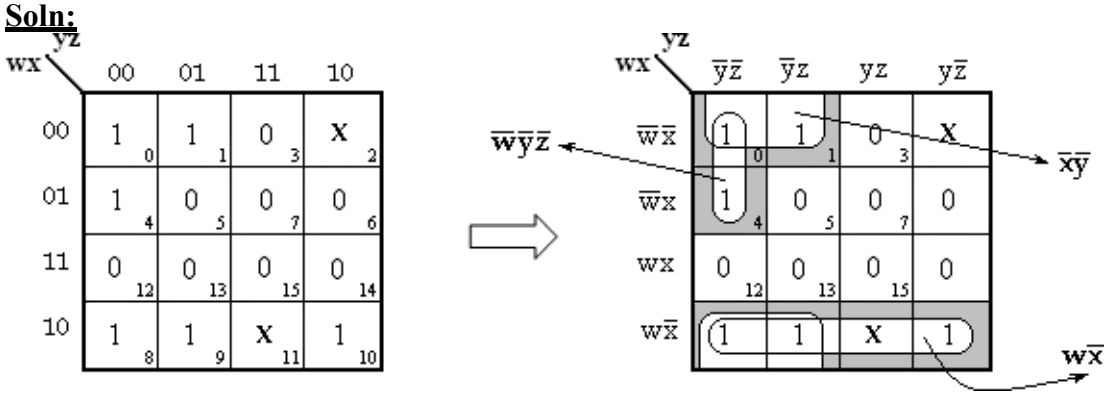
3. F (w, x, y, z) = ∑m (0, 7, 8, 9, 10, 12)+ ∑d (2, 5, 13)



F (w, x, y, z) = w'xz+ wy'+ x'z'.

4. F (w, x, y, z) = ∑m (0, 1, 4, 8, 9, 10)+ ∑d (2, 11)

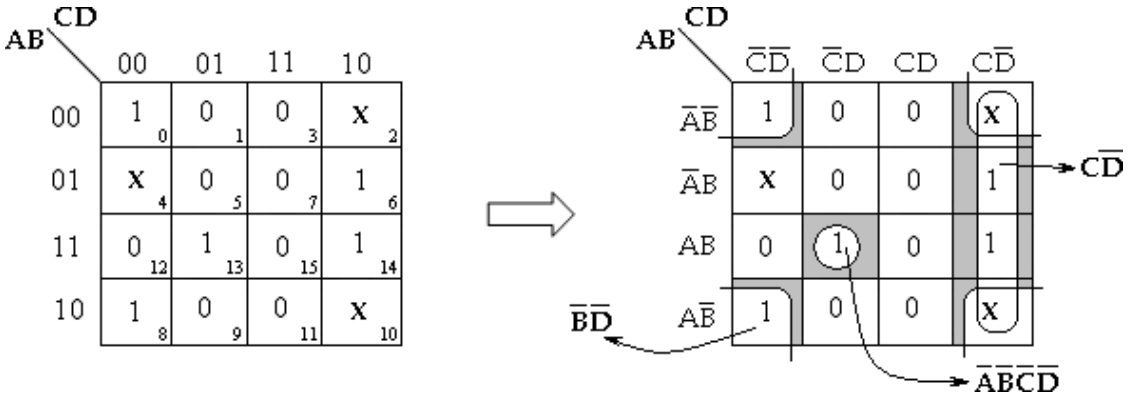
Soln:



F (w, x, y, z) = wx'+ x'y'+ w'y'z'.

5. F(A, B, C, D) = ∑m (0, 6, 8, 13, 14)+ ∑d (2, 4, 10)

Soln:

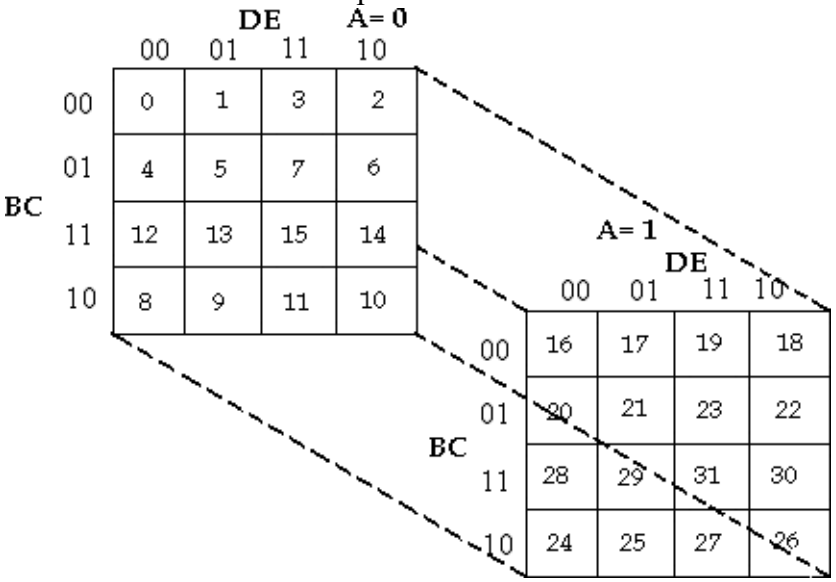


F(A, B, C, D) = CD'+ B'D'+ A'B'C'D'.

Five- Variable Maps:

A 5- variable K- map requires 25= 32 cells, but adjacent cells are difficult to identify on a single 32-cell map. Therefore, two 16 cell K-maps are used.

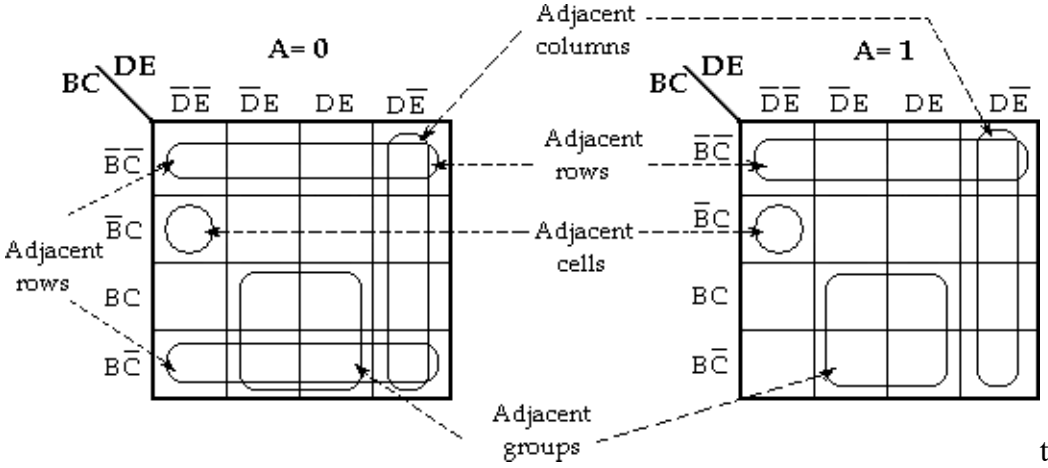
If the variables are A, B, C, D and E, two identical 16- cell maps containing B, C, D and E can be constructed. One map is used for A and other for A'.



In order to identify the adjacent grouping in the 5- variable map, we must imagine the two maps superimposed on one another ie., every cell in one map is adjacent to the corresponding cell in the other map, because only one variable changes between such corresponding cells.

Five- Variable Karnaugh map (Layer Structure)

Thus, every row on one map is adjacent to the corresponding row (the one occupying the same position) on the other map, as are corresponding columns. Also,



the rightmost and leftmost columns within each 16- cell map are adjacent, just as they are in any 16- cell map, as are the top and bottom rows.

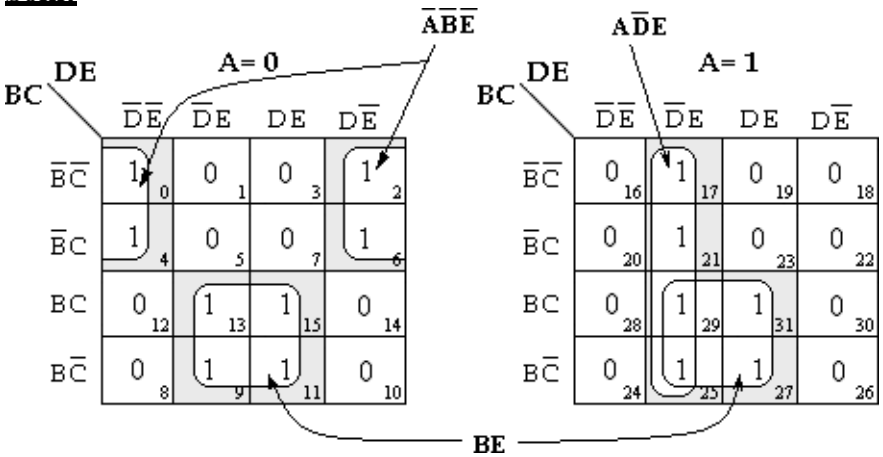
Typical subcubes on a five-variable map

However, the rightmost column of the map is not adjacent to the leftmost column of the other map.

1. Simplify the Boolean function

F (A, B, C, D, E) = ∑m (0, 2, 4, 6, 9, 11, 13, 15, 17, 21, 25, 27, 29, 31)

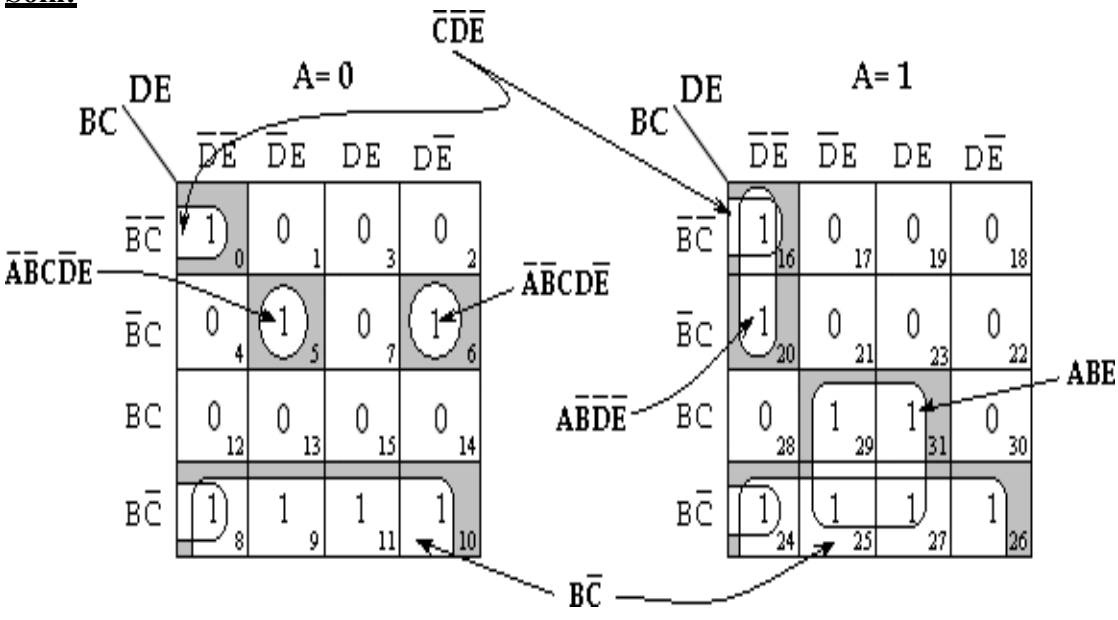
Soln:



F (A, B, C, D, E) = A'B'E'+ BE+ AD'E

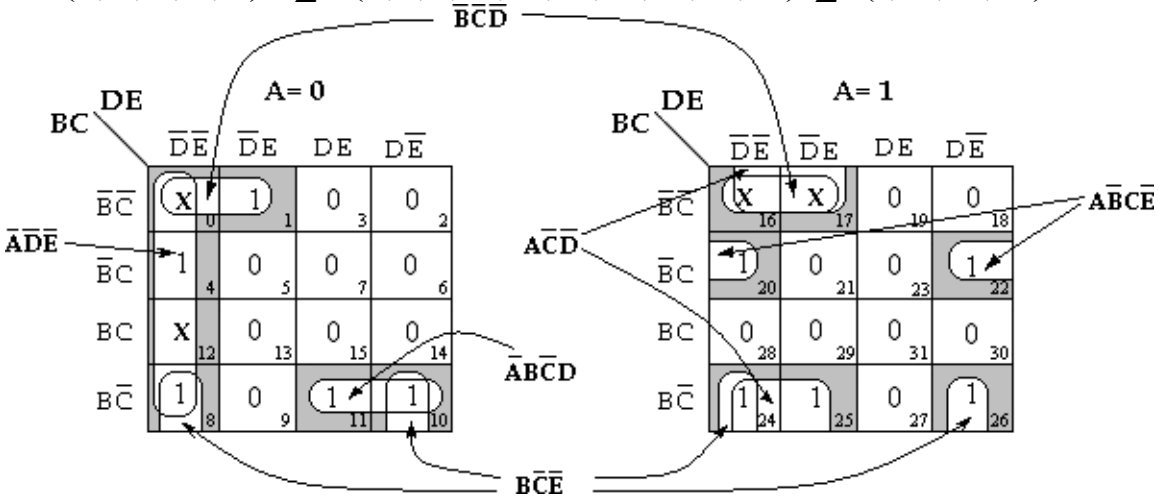
2. F (A, B, C, D, E) = ∑m (0, 5, 6, 8, 9, 10, 11, 16, 20, 24, 25, 26, 27, 29, 31)

Soln:



F (A, B, C, D, E) = C'D'E'+ A'B'CD'E+ A'B'CDE'+ AB'D'E'+ ABE+ BC'

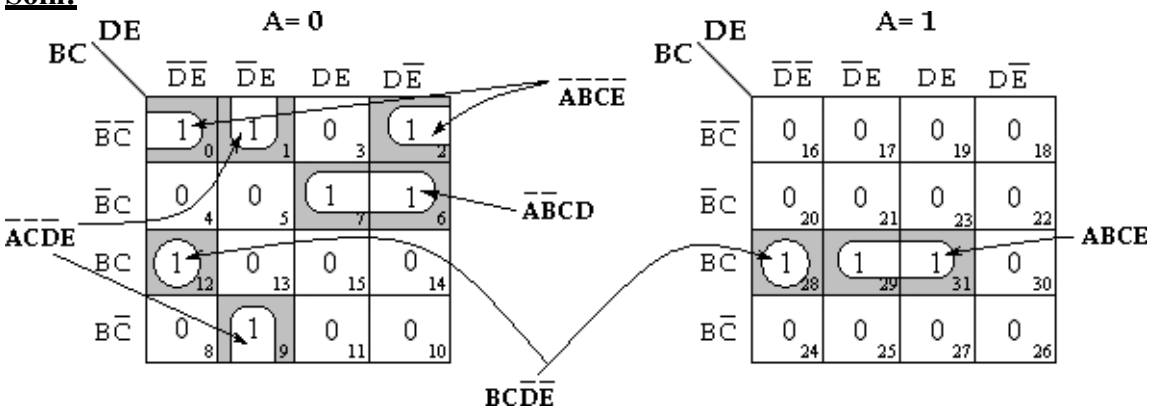
3. F (A, B, C, D, E) = ∑m (1, 4, 8, 10, 11, 20, 22, 24, 25, 26)+∑d (0, 12, 16, 17)



F (A, B, C, D, E) = B'C'D'+ A'D'E'+ BC'E'+ A'BC'D+ AC'D'+ AB'CE'

4. F (A, B, C, D, E) = ∑m (0, 1, 2, 6, 7, 9, 12, 28, 29, 31)

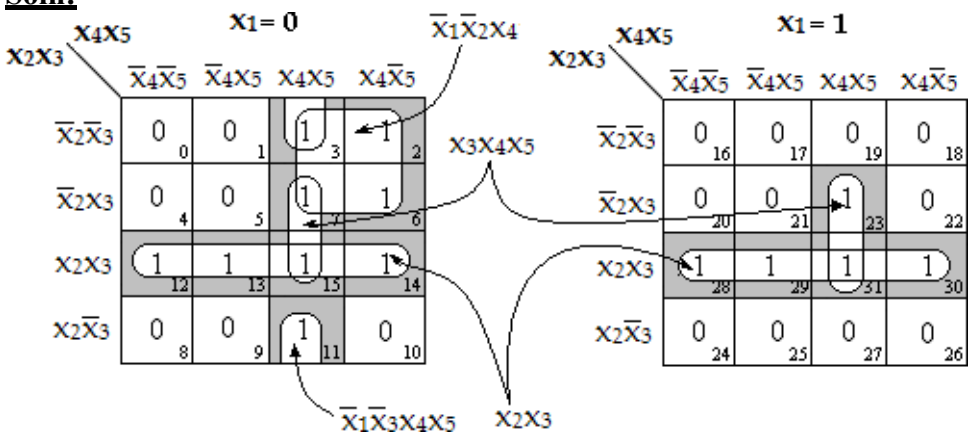
Soln:



F (A, B, C, D, E) = BCD'E'+ ABCE+ A'B'C'E'+ A'C'D'E+ A'B'CD

5. F (x_1, x_2, x_3, x_4, x_5) = ∑m (2, 3, 6, 7, 11, 12, 13, 14, 15, 23, 28, 29, 30, 31)

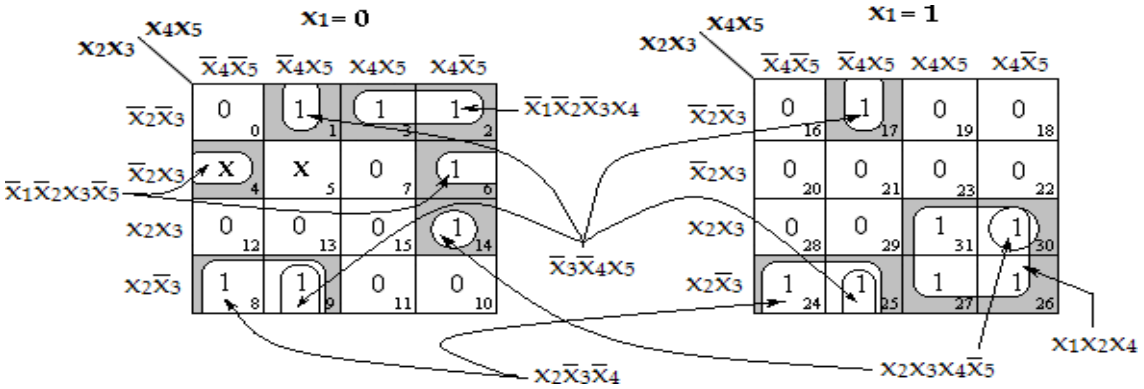
Soln:



F (x_1, x_2, x_3, x_4, x_5) = x_2x_3+ $x_3x_4x_5$+ $x_1'x_2'x_4$+ $x_1'x_3'x_4x_5$

6. F (x_1, x_2, x_3, x_4, x_5) = ∑m (1, 2, 3, 6, 8, 9, 14, 17, 24, 25, 26, 27, 30, 31)+ ∑d (4, 5)

Soln:



F (x_1, x_2, x_3, x_4, x_5) = $x_2x_3'x_4' + x_2x_3x_4x_5' + x_3'x_4'x5 + x_1x_2x_4 + x_1'x_2'x_3x_5' + x_1'x_2'x_3'x_4$

QUINE- MCCLUSKEY METHOD or TABULATION METHOD

The tabular method which is also known as the Quine-McCluskey method is particularly useful when minimising functions having a large number of variables, e.g. The six-variable functions. Computer programs have been developed employing this algorithm. The method reduces a function in standard sum of products form to a set of prime implicants from which as many variables are eliminated as possible. These prime implicants are then examined to see if some are redundant.

The tabular method makes repeated use of the law $A + \bar{A} = 1$. Note that Binary notation is used for the function, although decimal notation is also used for the functions. As usual a variable in true form is denoted by 1, in inverted form by 0, and the abscence of a variable by a dash (-).

RULES OF TABULATION METHOD

1. List all minterms in the binary form.
2. Arrange the minterms according to number of 1's and separate by a horizontal line.
3. Compare each binary number with every term in the adjacent next higher category and if they differ only by one position, put a check mark and copy the term in the next column with '-' in the position that they differed.
4. Apply the same process described in step 3 for the resultant column and continue these until no further elimination of literals.
5. List all the prime implicants.
6. Select the minimum number of prime implicants which must cover all the minterms.

Simplify the given boolean expression using Tabulation method

1. $Y(A, B, C, D) = \sum m(2, 4, 5, 9, 12, 13)$.

Table : 1

Min Term	Binary Representation				No.of 1s	
	A	B	C	D		
2	0	0	1	0	1	✓
4	0	1	0	0	1	✓
5	0	1	0	1	2	✓
9	1	0	0	1	2	✓
12	1	1	0	0	2	✓
13	1	1	0	1	3	✓

Table : 3

Min Term	Binary Representation				
	A	B	C	D	
4,5	0	1	0	_	✓
4,12	_	1	0	0	✓
5,13	_	1	0	1	✓
9,13	1	_	0	1	*
12,13	1	1	0	_	✓

Table : 2

Min Term	Binary Representation				
	A	B	C	D	
2	0	0	1	0	*
4	0	1	0	0	✓
5	0	1	0	1	✓
9	1	0	0	1	✓
12	1	1	0	0	✓
13	1	1	0	1	✓

Table : 4

Min Term	Binary Representation			
	A	B	C	D
4,5,12,13	_	1	0	_
4,12,5,13	_	1	0	_

Table : 4 Reduced

4,5,12,13	_	1	0	_	*

Table: 5 Prime Implicants Table

Min Term	Binary Representation				Product Term		2	4	5	9	12	13
	A	B	C	D								
2	0	0	1	0	A' B' C D'	✓	X					
9,13	1	_	0	1	A C' D	✓				X		X
4, 5, 12, 13	_	1	0	_	B C'	✓		X	X		X	X
Select Single X column for Essential Prime Implecants							✓	✓	✓	✓	✓	

Y = A' B' C D' + A C' D +B C'

2. $f(A, B, C, D) = \sum m(0,1,2,3,5,7,8,10,12,13,15)$.

Table : 1

Min Term	Binary Representation A	B	C	D	No.of 1s	
0	0	0	0	0	0	✓
1	0	0	0	1	1	✓
2	0	0	1	0	1	✓
3	0	0	1	1	2	✓
5	0	1	0	1	2	✓
7	0	1	1	1	3	✓
8	1	0	0	0	1	✓
10	1	0	1	0	2	✓
12	1	1	0	0	2	✓
13	1	1	0	1	3	✓
15	1	1	1	1	4	✓

Table : 2

Min Term	Binary Representation A	B	C	D	
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
12	1	1	0	0	✓
7	0	1	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

Table : 3

Min Term	Binary Representation A	B	C	D	
0,1	0	0	0	_	✓
0,2	0	0	_	0	✓
0,8	_	0	0	0	✓
1,3	0	0	_	1	✓
1,5	0	_	0	1	✓
2,3	0	0	1	_	✓
2,10	_	0	1	0	✓
8,10	1	0	_	0	✓
8,12	1	_	0	0	*
3,7	0	_	1	1	✓
5,7	0	1	_	1	✓
5,13	_	1	0	1	✓
12,13	1	1	0	_	*
7,15	_	1	1	1	✓
13,15	1	1	_	1	✓

Table : 4

Min Term	Binary Representation A	B	C	D	
0, 1, 2, 3	0	0	—	—	
0, 2, 1, 3	0	0	—	—	
0, 2, 8, 10	—	0	—	0	
0, 8, 2, 3	—	0	—	0	
1, 3, 5, 7	0	—	—	1	
1, 5, 3, 7	0	—	—	1	
5, 7, 13, 15	—	1	—	1	
5, 13, 7, 15	—	1	—	1	

Table : 4 Reduced

0, 1, 2, 3	0	0	_	_	*
0, 2, 8, 10	_	0	_	0	*
1, 3, 5, 7	0	_	_	1	*
5, 7, 13, 15	_	1	_	1	*

Table: 5 Prime Implicants Table

Min Term	Binary Representation A	B	C	D	Product Term	0	1	2	3	5	7	8	10	12	13	15
8,12	1	_	0	0	$A\bar{C}\bar{D}$							X		X		
12,13	1	1	0	_	$AB\bar{C}$									X	X	
0, 1, 2, 3	0	0	_	_	$\bar{A}\bar{B}$	X	X	X	X							
0, 2, 8, 10	_	0	_	0	$\bar{B}\bar{D}$ ✓	X		X				X	X			
1, 3, 5, 7	0	_	_	1	$\bar{A}D$		X		X	X	X					
5, 7, 13, 15	_	1	_	1	BD ✓					X	X				X	X
Select Single X column for Essential Prime Implecants													✓			✓

Missing min terms 1, 3, 12 so include 1, 3, 5, 7 & 12, 13 OR 1, 3, 5, 7 & 8, 12

$f(A, B, C, D) = \bar{B}\bar{D} + BD + \bar{A}D + AB\bar{C}$ OR $f(A, B, C, D) = \bar{B}\bar{D} + BD + \bar{A}D + A\bar{C}\bar{D}$

3. $Y(A,B,C,D,E) = \sum m(0, 1, 9, 15, 24, 29, 30) + \sum d(8, 11, 31)$

Table : 1

Min Term	Binary Representation A	B	C	D	E	No.of 1s	
0	0	0	0	0	0	0	✓
1	0	0	0	0	1	1	✓
d8	0	1	0	0	0	1	✓
9	0	1	0	0	1	2	✓
d11	0	1	0	1	1	3	✓
15	0	1	1	1	1	4	✓
24	1	1	0	0	0	2	✓
29	1	1	1	0	1	4	✓
30	1	1	1	1	0	4	✓
d31	1	1	1	1	1	5	✓

Table : 2

Min Term	Binary Representation A	B	C	D	E	
0	0	0	0	0	0	✓
1	0	0	0	0	1	✓
8	0	1	0	0	0	✓
9	0	1	0	0	1	✓
24	1	1	0	0	0	✓
11	0	1	0	1	1	✓
15	0	1	1	1	1	✓
29	1	1	1	0	1	✓
30	1	1	1	1	0	✓
31	1	1	1	1	1	✓

Table : 3

Min Term	Binary Representation					
	A	B	C	D	E	
0,1	0	0	0	0	_	✓
0,8	0	_	0	0	0	✓
1,9	0	_	0	0	1	✓
8,9	0	1	0	0	_	✓
8,24	_	1	0	0	0	*
9,11	0	1	0	_	1	*
11,15	0	1	_	1	1	*
15,31	_	1	1	1	1	*
29,31	1	1	1	_	1	*
30,31	1	1	1	1	_	*

Table : 4

Min Term	Binary Representation				
	A	B	C	D	E
0,1,8,9	0	_	0	0	_
0,8,1,9	0	_	0	0	_

Table : 4 Reduced

0,1,8,9	0	_	0	0	_	*

Table: 5 Prime Implicants Table

Min Term	Binary Representation					Product Term		0	1	d8	9	d11	15	24	29	30	d31
	A	B	C	D	E												
8,24	_	1	0	0	0	B C' D' E'	✓			X				X			
9,11	0	1	0	_	1	A' B C' E					X	X					
11,15	0	1	_	1	1	A' B D E						X	X				
15,31	_	1	1	1	1	B C D E							X				X
29,31	1	1	1	_	1	A B C E	✓								X		X
30,31	1	1	1	1	_	A B C D	✓									X	X
0,1,8,9	0	_	0	0	_	A' C' D'	✓	X	X	X	X						
Select Single X column for Essential Prime Implecants								✓	✓					✓	✓	✓	

Missing min terms 15 so include 11, 15 OR 15,31

Y = B C' D' E' + A B C E + A B C D + A' C' D' + A' B D E OR

Y = B C' D' E' + A B C E + A B C D + A' C' D' + B C D E

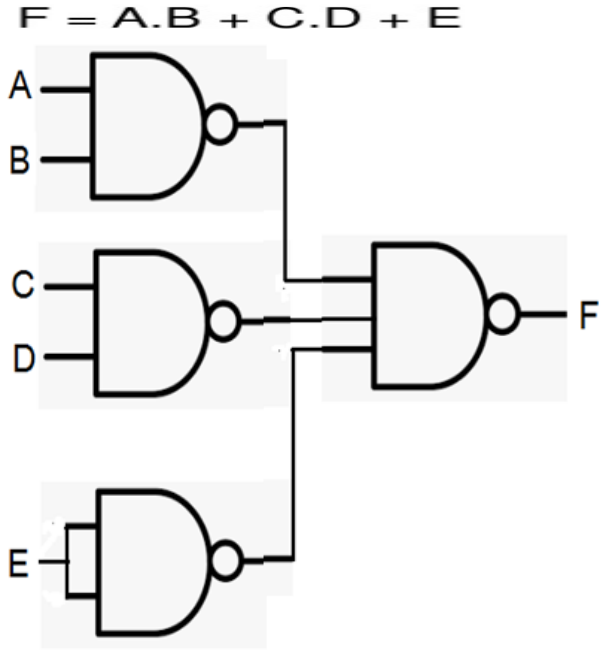
Two Level Gate Network

- **The SOP can be implemented using NAND – NAND logic**
    1. Each product term is connected to NAND gates in level 1
    2. One NAND is connected in the second level 2

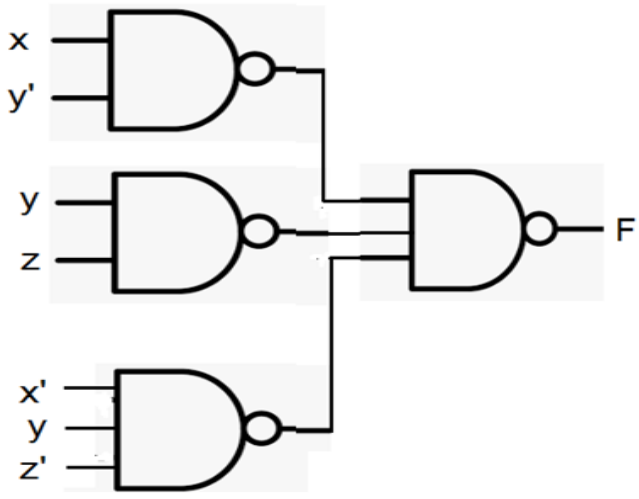
- **The POS can be implemented using NOR – NOR logic**
    1. Each sum term is connected to NOR gates in level 1
    2. One NOR is connected in the second level 2

Implement Using NAND – NAND logic

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

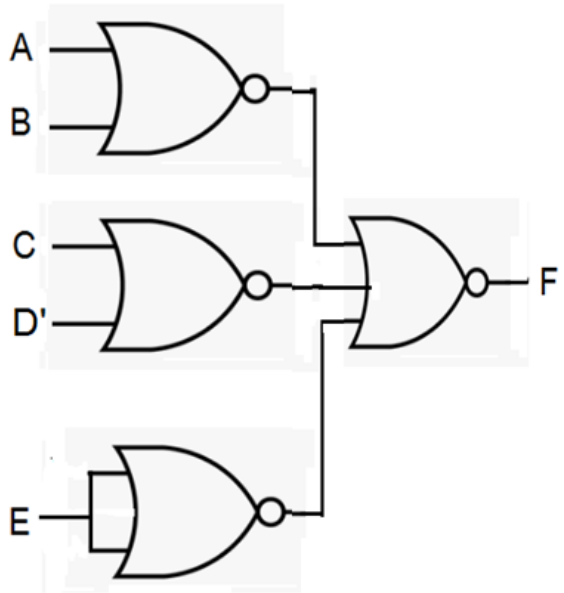


$F = (A+B)(C+D')E$

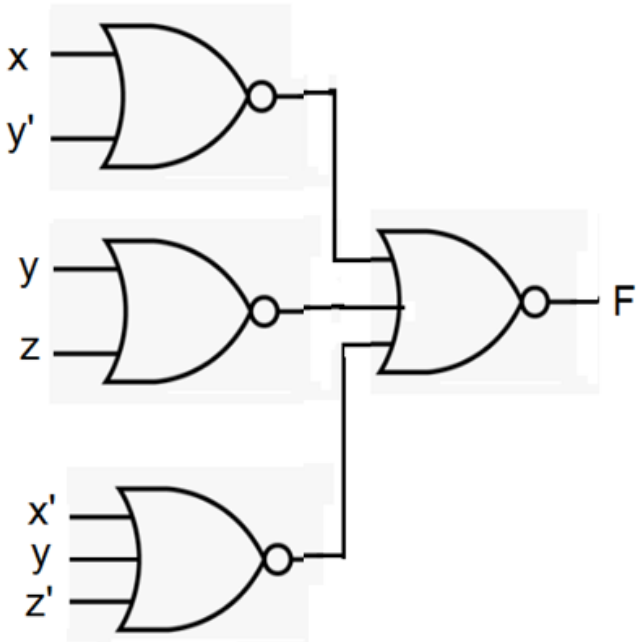


Implement Using NOR – NOR logic

$$F = (A+B)\,(C+D')\,E$$



$$F = (x + y')\,(y + z)\,(x' + y + z')$$



TEXT BOOKS:

1. Morris Mano, “Digital design”, 3rd Edition, Prentice Hall of India, 2008.

REFERENCE BOOKS:

1. Milos Ercegovac, Jomas Lang, “Introduction to Digital Systems”, Wiley publications, 1998.
2. John M. Yarbrough, “Digital logic: Applications and Design”, Thomas – Vikas Publishing House, 2002.
3. R.P.Jain, “Modern digital Electronics”,3rd Edition, TMH, 2003.
4. William H. Gothmann, “Digital Electronics”, Prentice Hall, 2001.

QUESTION BANK

PART-A

1. Distinguish between SOP and POS.
2. Examine Canonical SOP for F=A’C + BC’
3. Define prime implicant.
4. Describe don’t care terms.
5. Define K-Map and its advantages.
6. State the limitations of K-map.
7. Summarize the use of K-map and draw the format of 5 variables K-map.
8. Define the meaning of essential and non-essentialprime implicant.
9. Find Minimized SOP for F (a, b, c) = ∑m(3, 4, 6) + ∑d (0, 2) usingKarnaugh map method.
10. Produce minimized POS for F= π(2,4,6,7)

PART-B

1. Examine the Canonical SOP and Canonical POS for the expression
   F = A + B’C
2. Apply Karnaugh map to Simplify the following Boolean functions.
   a) f(w,x,y,z)=∑ (0,2,5,6,7,8,10)
   b) f(a,b,c,d)=π(1,3,5,7,13,15)
3. Examine and simplify the Boolean function using tabulation methods.
   F= ∑ (1, 2, 3, 8, 10, 11, 14, 15)
4. Apply Quine McCluskey method to simplify the boolean function
   F= ∑ (0,1,2,3,6,7,13,15) and verify using k-map
5. Produce minimum sum of products of
   F(a,b,c,d) =∑m(2,3,6,7,11,13,14,15,23,30,31) using K-map.
6. Produce minimum sum of products form of
   f(A,B,C,D) = π (1,2,3,4,6,8,11,12,27,28)+d(9,16,17,18,19)

UNIT III

COMBINATIONAL CIRCUITS

Introduction to Combinational circuits – Half Adder, Full Adder - Half Subtractor, Full Subtractor- Parallel binary Adder, Parallel binary Subtractor - Carry look ahead Adder- BCD Adder- Decoders- Encoders - Priority Encoder- Multiplexers- MUX as universal combinational modules- Demultiplexers- Code convertors- Magnitude Comparator.

INTRODUCTION:

The digital system consists of two types of circuits, namely

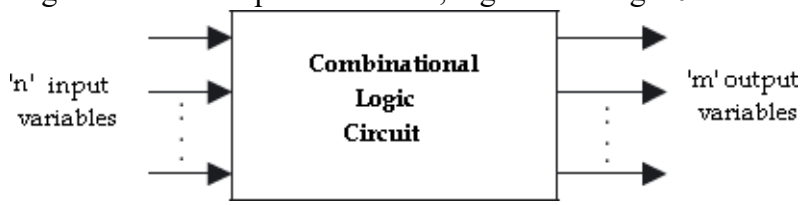
- Combinational circuits
- Sequential circuits

Combinational circuit consists of logic gates whose output at any time is determined from the present combination of inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

Sequential logic circuit comprises both logic gates and the state of storage elements such as flip-flops. As a consequence, the output of a sequential circuit depends not only on present value of inputs but also on the past state of inputs.

In the previous chapter, we have discussed binary numbers, codes, Boolean algebra and simplification of Boolean function and logic gates. In this chapter, formulation and analysis of various systematic designs of combinational circuits will be discussed.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from inputs and output signals are generated according to the logic circuits employed in it. Binary information from the given data transforms to desired output data in this process. Both input and output are obviously the binary signals, *i.e.*, both the input and output signals are of two possible states, logic 1 and logic 0.



Block diagram of a combinational logic circuit

For n number of input variables to a combinational circuit, 2^n possible combinations of binary input states are possible. For each possible combination, there is one and only one possible output combination. A combinational logic circuit can be described by m Boolean functions and each output can be expressed in terms of n input variables.

DESIGN PROCEDURE:

- The problem is stated.
- Identify the input and output variables.
- The input and output variables are assigned letter symbols.
- Construction of a truth table to meet input -output requirements.
- Writing Boolean expressions for various output variables in terms of input variables.
- The simplified Boolean expression is obtained by any method of minimization—algebraic method, Karnaugh map method, or tabulation method.
- A logic diagram is realized from the simplified boolean expression using logic gates.

The following guidelines should be followed while choosing the preferred form for hardware implementation:

- The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.
- There should be a minimum number of interconnections.
- Limitation on the driving capability of the gates should not be ignored.

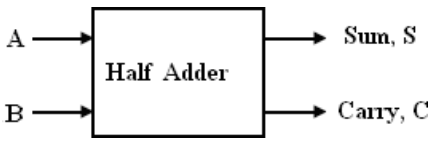
ARITHMETIC CIRCUITS – BASIC BUILDING BLOCKS:

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction.

The basic building blocks that form the basis of all hardware used to perform the arithmetic operations on binary numbers are half-adder, full adder, half-subtractor, full- subtractor.

Half-Adder:

A half-adder is a combinational circuit that can be used to add two binary bits. It has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.



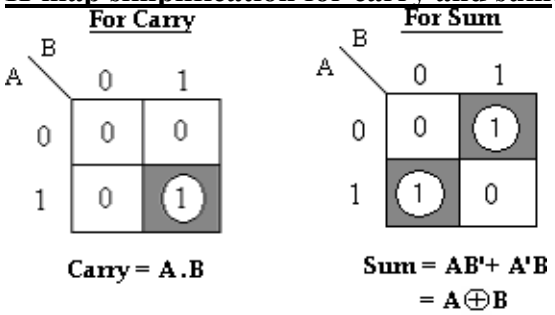
Block schematic of half-adder

The truth table of a half-adder, showing all possible input combinations and the corresponding outputs are shown below.

Truth table of half-adder

Inputs		Outputs	
A	**B**	**Carry (C)**	**Sum (S)**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

K-map simplification for carry and sum:



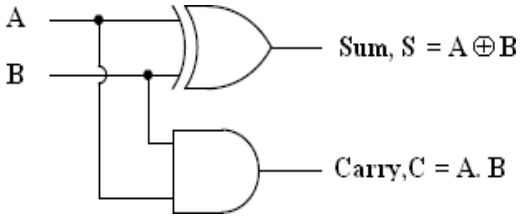
The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A'B+ AB'

Carry, C = A . B

The first one representing the SUM output is that of an EX-OR gate, the second one representing the CARRY output is that of an AND gate.

The logic diagram of the half adder is,

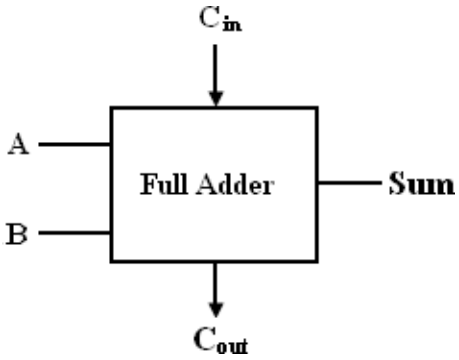


Logic Implementation of Half-adder

Full-Adder:

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of 3 inputs and 2 outputs.

Two of the input variables, represent the significant bits to be added. The third input represents the carry from previous lower significant position. The block diagram of full adder is given by,



Block schematic of full-adder

The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. As there are three input variables, eight different input combinations are possible. The truth table is shown below,

Truth Table:

Inputs			Outputs	
A	**B**	**C_{in}**	**Sum (S)**	**Carry (C_{out})**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

To derive the simplified Boolean expression from the truth table, the Karnaugh map method is adopted as,

For Carry

A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Carry, C_{out} = AB+ AC_{in} + BC_{in}

For Sum

A \ BC_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

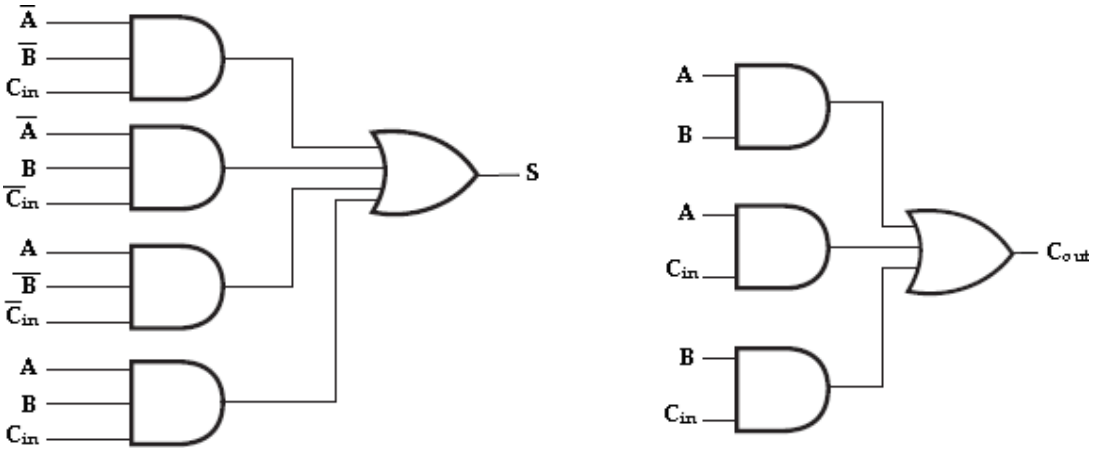
Sum, S = A'B'C_{in}+ A'BC'$_{in}$+ AB'C'$_{in}$+ ABC_{in}

The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A'B'Cin+ A'BC'in + AB'C'in +ABC_{in}

Carry, C_{out} = AB+ AC_{in} +BC_{in}.

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Product

The logic diagram of the full adder can also be implemented with two half- adders and one OR gate. The S output from the second half adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

Sum = $C_{in} \oplus (A \oplus B)$ [$x \oplus y = x'y + xy'$]

$= C_{in} \oplus (A'B + AB')$

= C‘in (A‘B+AB‘) + C_{in} (A‘B+AB‘)‘ [(x‘y+xy‘)‘= (xy+x‘y‘)]

= C‘in (A‘B+AB‘) + Cin (AB+A‘B‘)

= A‘BC‘in + AB‘C‘in + ABC_{in} + A‘B‘Cin .

and the carry output is,

Carry, C_{out} = AB+ C_{in} (A’B+AB’)

= AB+ A‘BCin+ AB‘Cin

= AB (C_{in}+1) + A‘BCin+ AB‘Cin [C_{in}+1= 1]

= ABC_{in}+ AB+ A‘BCin+ AB‘Cin

= AB+ AC_{in} (B+B‘) + A‘BCin

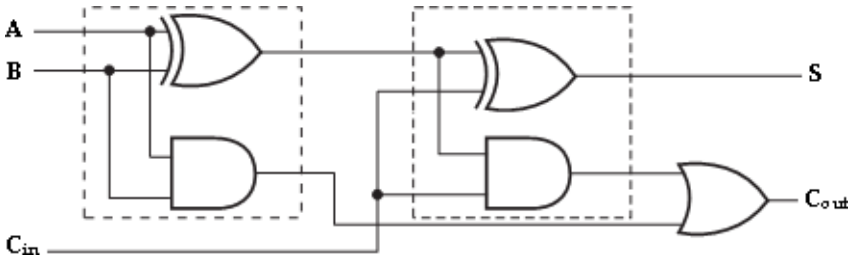
= AB+ AC_{in}+ A‘BCin

= AB (C_{in}+1) + AC_{in}+ A‘BCin [C_{in}+1= 1]

= ABC_{in}+ AB+ AC_{in}+ A‘BCin

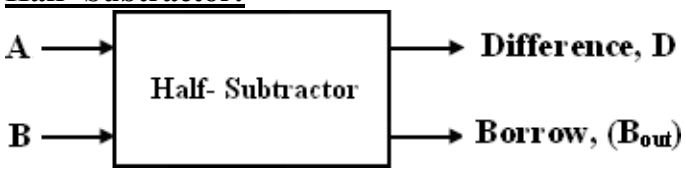
= AB+ AC_{in}+ BC_{in} (A +A‘)

= AB+ AC_{in}+ BC_{in}.



Implementation of full adder with two half-adders and an OR gate

Half -Subtractor:



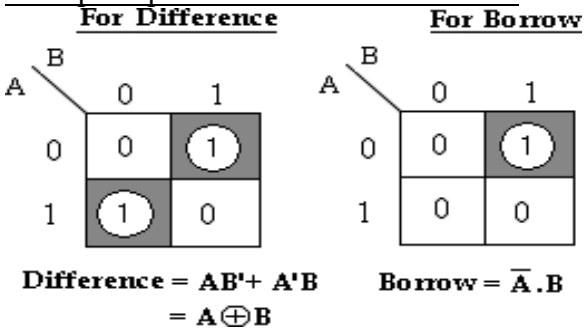
Block schematic of half-subtractor

A half-subtractor is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a _1' has been borrowed to perform the subtraction.

The truth table of half-subtractor, showing all possible input combinations and the corresponding outputs are shown below.

Input		Output	
A	**B**	**Difference (D)**	**Borrow (B_{out})**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map simplification for half subtractor:



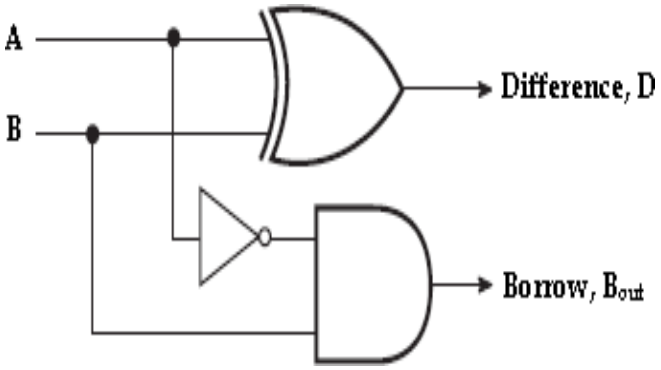
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = A'B+ AB'

Borrow, B_{out} = A' . B

The first one representing the DIFFERENCE (**D**)output is that of an exclusive-OR gate, the expression for the BORROW output (**B_{out}**) is that of an AND gate with input A complemented before it is fed to the gate.

The logic diagram of the half adder is,



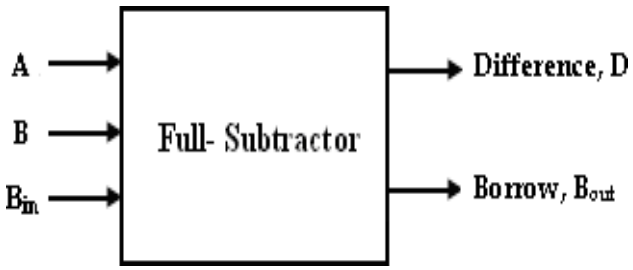
Logic Implementation of Half-Subtractor

Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, ie., the minuend is complemented, an AND gate can be used to implement the BORROW output.

Full Subtractor:

A full subtractor performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a _1' has already been borrowed by the previous adjacent lower minuend bit or not.

As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o. The BORROW output bit tells whether the minuend bit needs to borrow a _1' from the next possible higher minuend bit.



Block schematic of full-adder

The truth table for full-subtractor is,

Inputs			Outputs	
A	**B**	**B_{in}**	**Difference(D)**	**Borrow(B_{out})**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

For Difference

A \ BB_{in}	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

For Borrow

A \ BB_{in}	00	01	11	10
0	0	1	(1)	1
1	0	0	1	0

Difference, $D = A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

Borrow, $B_{out} = A'B + A'B_{in} + BB_{in}$

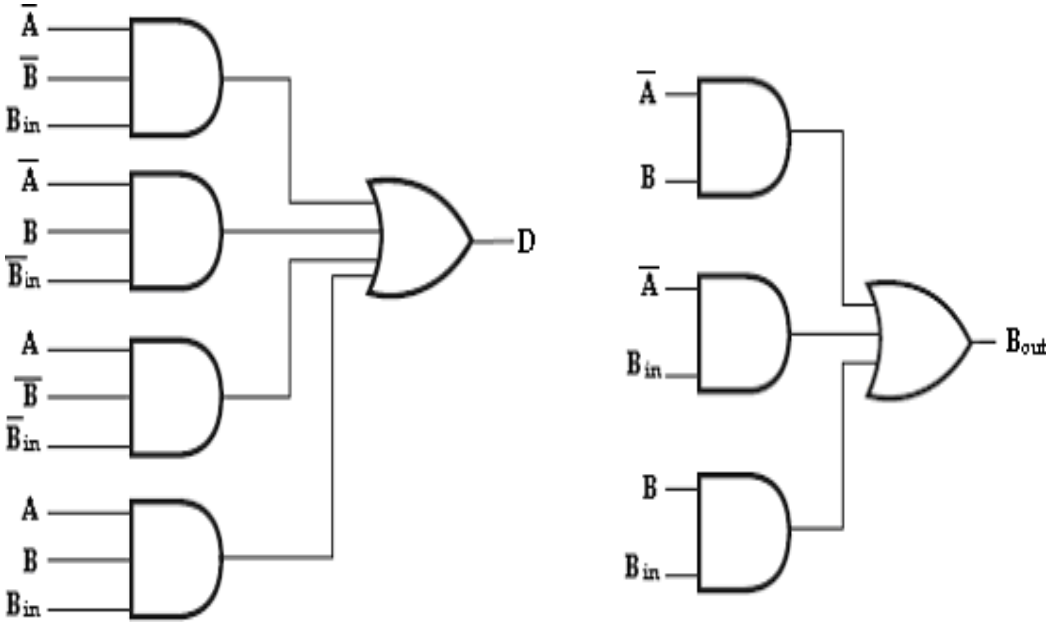
K-map simplification for full-subtractor:

The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = A'B'Bin + A'BB'in + AB'B'in + ABBin

Borrow, B_{out} = A'B + A'Cin + BB_{in}.

The logic diagram for the above functions is shown as,



Implementation of full-Subtractor using Half Subtractors

The logic diagram of the full-subtractor can also be implemented with two half- subtractors and one OR gate. The difference,D output from the second half subtractor is the Ex -OR of B_{in} and the output of the first half-subtractor, giving

Difference,D= $B_{in} \oplus (A \oplus B)$ $[x \oplus y = x'y + xy']$

$= B_{in} \oplus (A'B + AB')$

= B‘in (A‘B+AB‘) + B_{in} (A‘B+AB‘)‘ [(x‘y+xy‘)‘= (xy+x‘y‘)]

= B‘in (A‘B+AB‘) + Bin (AB+A‘B‘)

= A‘BB‘in + AB‘B‘in + ABB_{in} + A‘B‘Bin .

and the borrow output is,

Borrow, B_{out} = A’B+ B_{in} (A’B+AB’)’ [(x‘y+xy‘)‘= (xy+x‘y‘)]

= A‘B+ B_{in} (AB+A‘B‘)

= A‘B+ ABBin+ A‘B‘Bin

= A‘B (Bin+1) +ABB_{in}+ A‘B‘Bin [C_{in}+1= 1]

= A‘BBin+ A‘B+ ABBin+ A‘B‘Bin

= A‘B+ BBin (A+A‘) + A‘B‘Bin [A+A‘= 1]

= A‘B+ BBin+ A‘B‘Bin

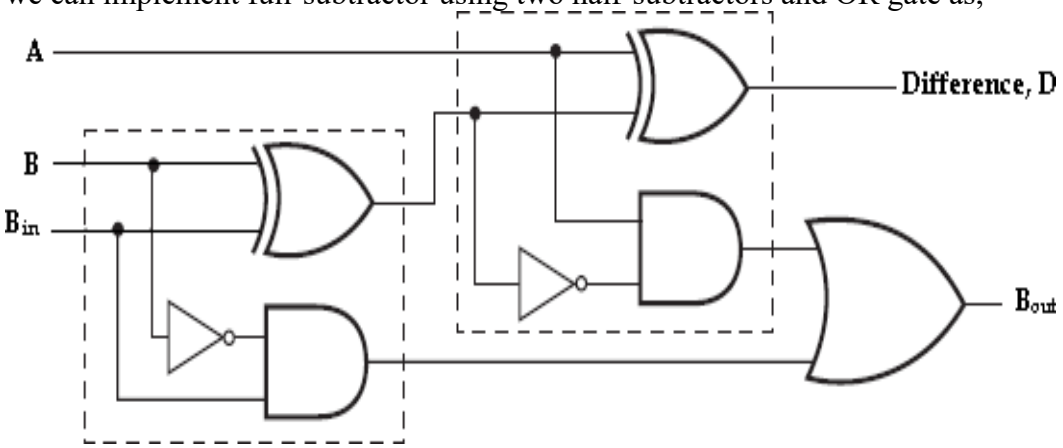
= A‘B (Bin+1) +BB_{in}+ A‘B‘Bin [C_{in}+1= 1]

= A‘BBin+ A‘B+ BBin+ A‘B‘Bin

= A‘B+ BBin+ A‘Bin (B +B‘)

= A‘B+ BBin+ A‘Bin.

Therefore,
we can implement full-subtractor using two half-subtractors and OR gate as,



Implementation of full-subtractor with two half-subtractors and an OR gate

Binary Adder (Parallel Adder):

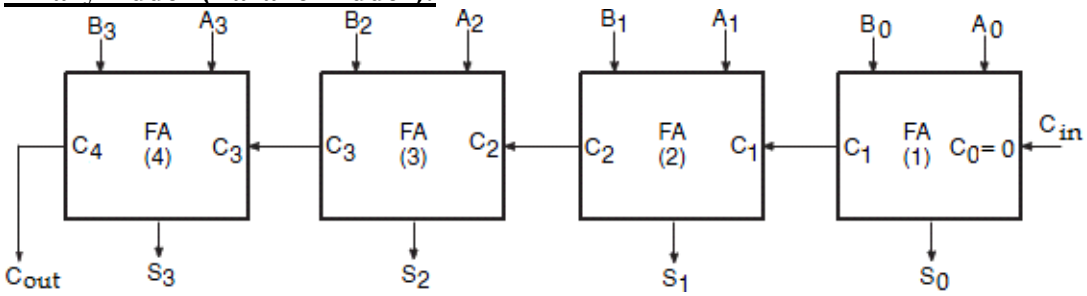


Fig. 4-bit binary parallel Adder

The 4-bit binary adder using full adder circuits is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in figure below. Since all the bits of augend and addend are fed into the adder circuits simultaneously and the additions in each position are taking place at the same time, this circuit is known as parallel adder.

Significant place		4	3	2	1
Input carry		1	1	1	0
Augend word A :		1	1	1	1
Addend word B :		0	0	1	1
	1	0	0	1	0 ← Sum
	↑ Output Carry				

Let the 4-bit words to be added be represented by, $A_3A_2A_1A_0$= 1111 and $B_3B_2B_1B_0$= 0011.

The bits are added with full adders, starting from the least significant position, to form the sum it and carry bit. The input carry C_0 in the least significant position must be

- The carry output of the lower order stage is connected to the carry input of the next higher order stage. Hence this type of adder is called ripple-carry adder.

In the least significant stage, A_0, B_0 and C_0 (which is 0) are added resulting in sum S_0 and carry C_1. This carry C_1 becomes the carry input to the second stage. Similarly in the second stage, A_1, B_1 and C_1 are added resulting in sum S_1 and carry C_2, in the third stage, A_2, B_2 and C_2 are added resulting in sum S_2 and carry C_3, in the third stage, A_3, B_3 and C_3 are added resulting in sum S_3 and C_4, which is the output carry. Thus the circuit results in a sum ($S_3S_2S_1S_0$) and a carry output (C_{out}).

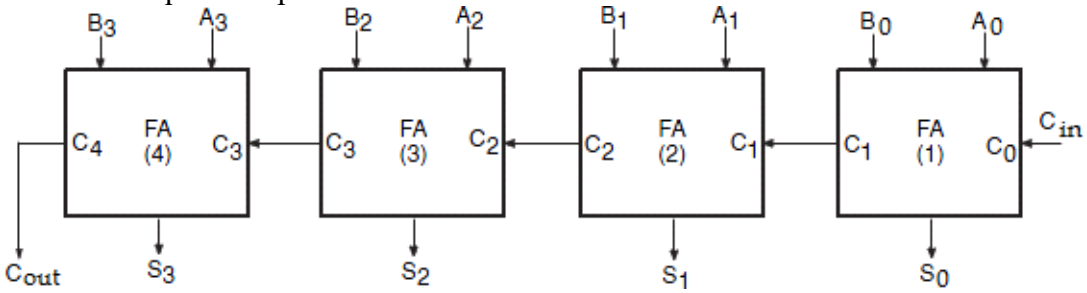
Though the parallel binary adder is said to generate its output immediately after the inputs are applied, its speed of operation is limited by the carry propagation delay through all stages. However, there are several methods to reduce this delay. One of the methods of speeding up this process is look-ahead carry addition which eliminates the ripple-carry delay.

Carry Propagation–Look-Ahead Carry Generator:

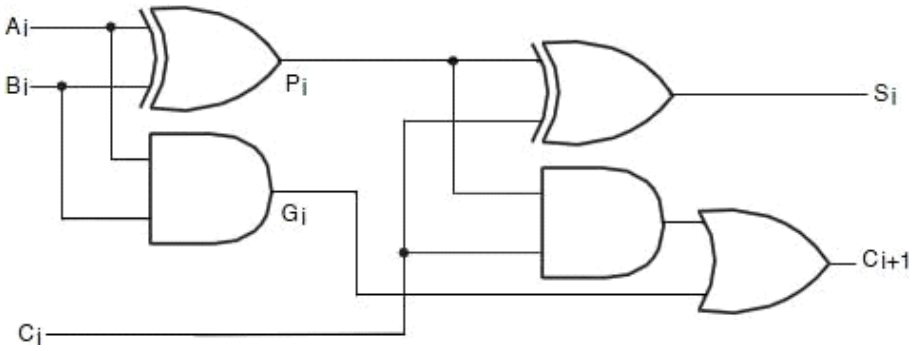
In Parallel adder, all the bits of the augend and the addend are available for computation at the same time. The carry output of each full-adder stage is connected to the carry input of the next high-order stage. Since each bit of the sum output depends on the value of the input carry, time delay occurs in the addition process. This time delay is called as carry propagation delay.

For example, addition of two numbers (0011+ 0101) gives the result as 1000. Addition of the LSB position produces a carry into the second position. This carry when added to the bits of the second position, produces a carry into the third position. This carry when added to bits of the third position, produces a carry into the last position. The sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous position. i.e., the adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example, if each full adder is considered to have a propagation delay of

30nsec, then S_3 will not react its correct value until 90 nsec after LSB is generated. Therefore total time required to perform addition is 90+ 30 = 120nsec.



4-bit Parallel Adder



Full-Adder circuit

The method of speeding up this process by eliminating inter stage carry delay is called look ahead-carry addition. This method utilizes logic gates to look at the lower order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions: carry generate and carry propagate.

Consider the circuit of the full-adder shown above. Here we define two functions:
carry generate (G_i) and carry propagate (P_i) as,

Carry generate, $\mathbf{G_i = A_i \oplus B_i}$

Carry propagate, $\mathbf{P_i = A_i \oplus B_i}$

the output sum and carry can be expressed as,

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i \oplus P_i C_i$$

G_i (carry generate), it produces a carry 1 when both Ai and Bi are 1, regardless of the input carry C_i. Pi (carry propagate) because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

The Boolean functions for the carry outputs of each stage and substitute for each C_i its value from the previous equation:

C_0= input

carry $C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2 (G_1 + P_1G_0 + P_1P_0C_0) = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

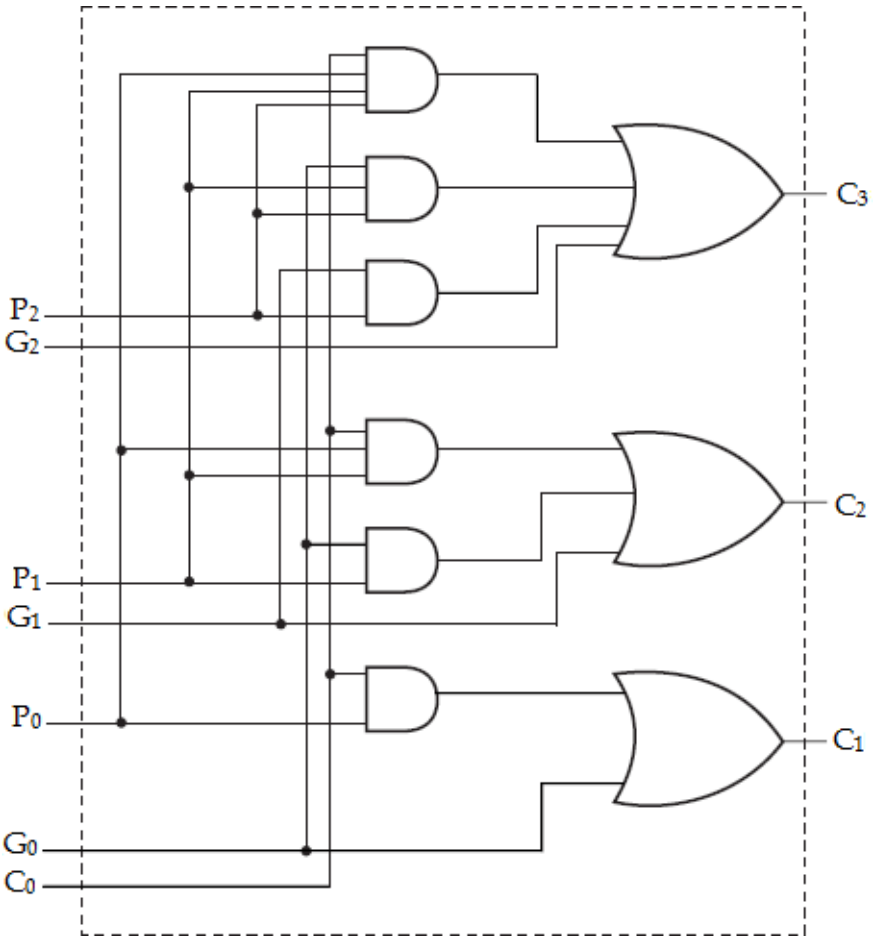


Fig. Logic diagram of Carry Look-ahead Generator

Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate. The three Boolean functions for C_1, C_2 and C_3 are implemented in the carry look-ahead generator as shown below.

Note that C_3 does not have to wait for C_2 and C_1 to propagate; in fact C_3 is propagated at the same time as C_1 and C_2.

Using a Look-ahead Generator we can easily construct a 4-bit parallel adder with a Look-ahead carry scheme. Each sum output requires two exclusive-OR gates. The

output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry look-ahead generator and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.

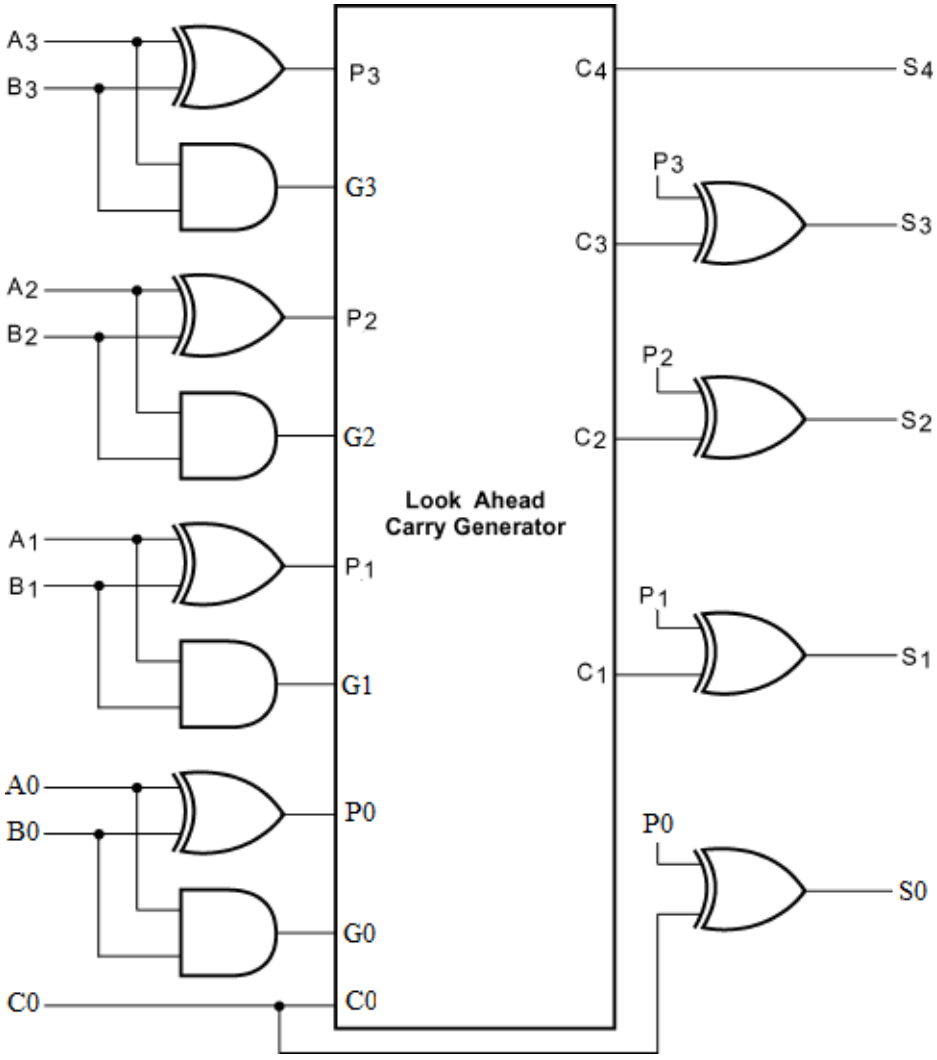


Fig. 4-Bit Adder with Carry Look-ahead

Binary Subtractor (Parallel Subtractor):

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. The subtraction A-B can be done by taking the 2‘s complement of B and adding it to A. The 2‘s complement can be obtained by taking the 1‘s complement and adding

1 to the least significant pair of bits. The 1‘s complement can be implemented with inverters and a 1 can be added to the sum through the input carry.

The circuit for subtracting A-B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when performing subtraction. The operation thus performed becomes A, plus the 1‘s complement of B, plus1. This is equal to A plus the 2‘s complement of B.

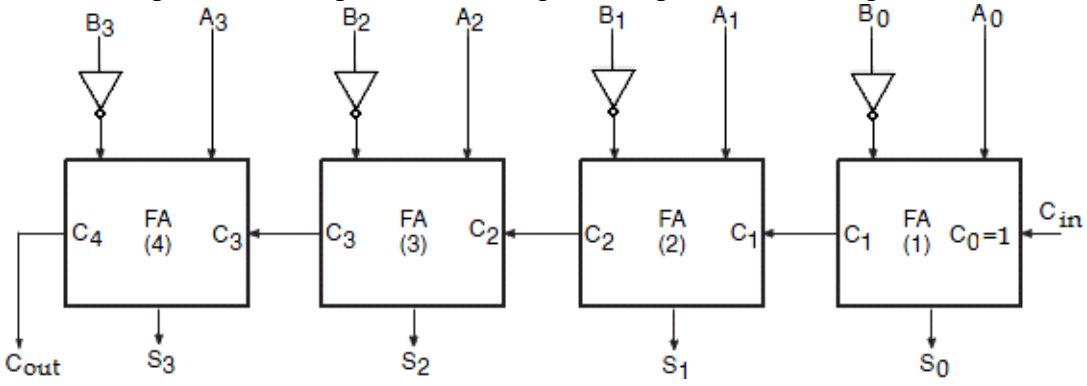


Fig. 4-bit Parallel Subtractor

Parallel Adder/ Subtractor:

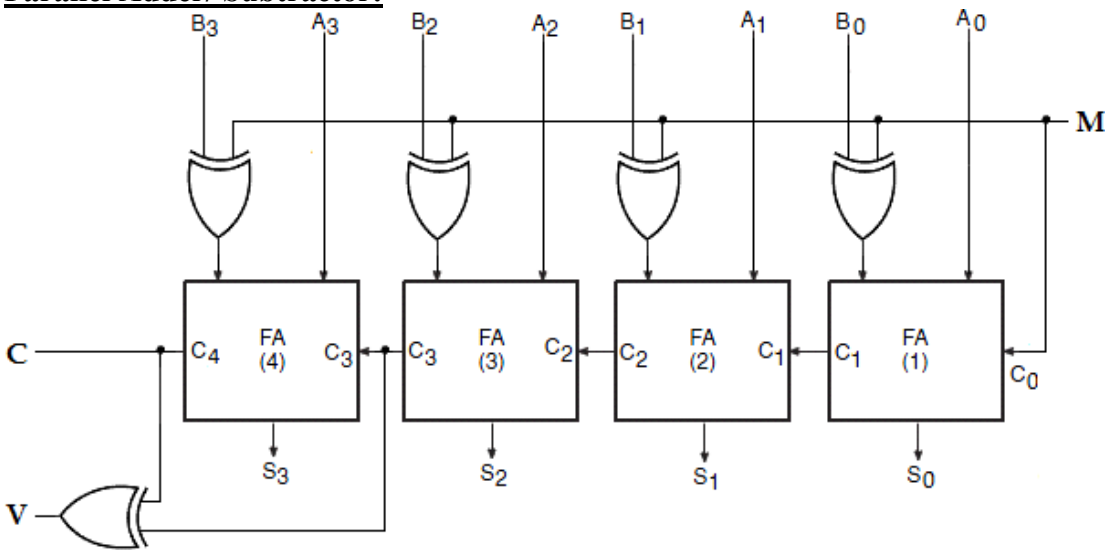


Fig. 4-Bit Adder Subtractor

The addition and subtraction operation can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full adder. A 4-bit adder Subtractor circuit is shown below.

The mode input M controls the operation. When M= 0, the circuit is an adder and when M=1, the circuit becomes a Subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M=0, we have B Ex-OR 0 = B. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M=1, we have B Ex –OR 1= B‘

and C0=1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2‘s complement of B. The exclusive-OR with output V is for detecting an overflow.

Decimal Adder (BCD Adder):

The digital system handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD adder is a circuit that adds two BCD bits and produces a sum digit also in BCD.

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9+ 9+1 = 19; the 1 is the sum being an input carry. The adder will form the sum in binary and produce a result that ranges from 0 through 19.

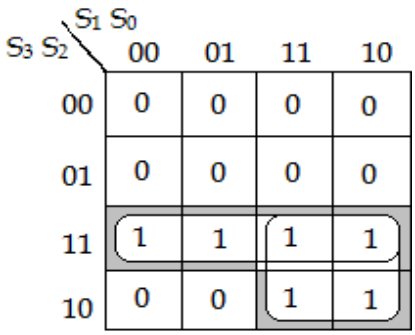
These binary numbers are labeled by symbols K, Z_8, Z_4, Z_2, Z_1, K is the carry. The columns under the binary sum list the binary values that appear in the outputs of the 4- bit binary adder. The output sum of the two decimal digits must be represented in BCD.

Binary Sum					**BCD Sum**					**Decimal**
K	**Z_8**	**Z_4**	**Z_2**	**Z_1**	**C**	**S_8**	**S_4**	**S_2**	**S_1**	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 9 (1001), we obtain a non- valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit to detect sum greater than 9 can be determined by simplifying the boolean expression of the given truth table.

Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



$Y = S_3S_2 + S_3S_1$

To implement BCD adder we require:

- 4-bit binary adder for initialaddition
- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add 0110_2 in the sum if the sum is greater than 9 or carry is

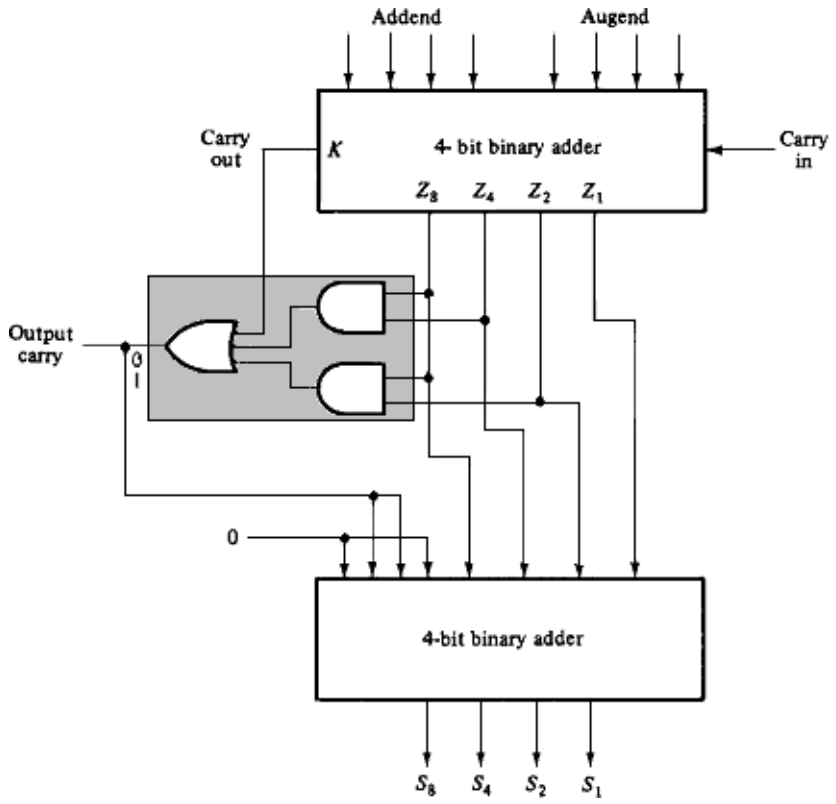


Fig. Logic diagram of BCD adder

The two decimal digits, together with the input carry, are first added in the top4- bit binary adder to provide the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. The output carry from one stage must be connected to the input carry of the next higher-order stage.

MAGNITUDE COMPARATOR:

A magnitude comparator is a combinational circuit that compares two given numbers (A and B) and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions A= B, A>B and A<B, if A and B are the two numbers being compared.

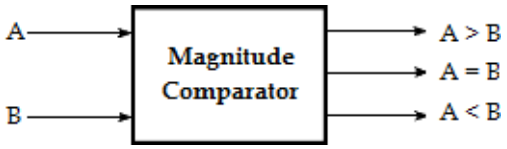


Fig. Block diagram of magnitude comparator

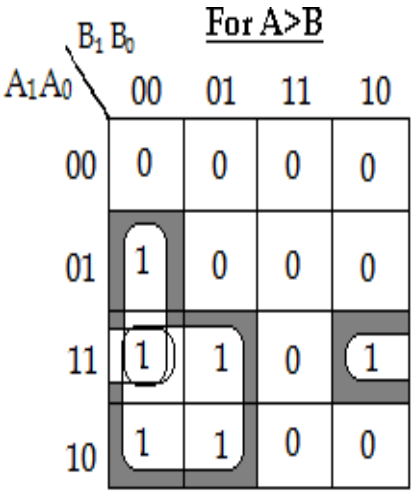
For comparison of two n-bit numbers, the classical method to achieve the Boolean expressions requires a truth table of 2^{2n} entries and becomes too lengthy and cumbersome.

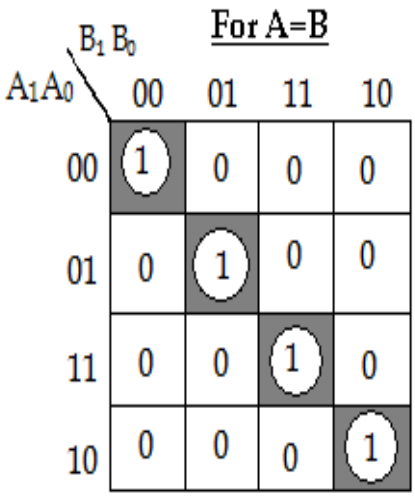
2-bit Magnitude Comparator:

The truth table of 2-bit comparator is given in table below— Truth table:

Inputs				Outputs		
A_3	A_2	A_1	A_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

K-map Simplification:





$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$

$A=B = A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A_0B_1B_0 + A_1A_0'B_1B_0'$

$= A_1'B_1' (A_0'B_0' + A_0B_0) + A_1B_1 (A_0B_0 + A_0'B_0')$

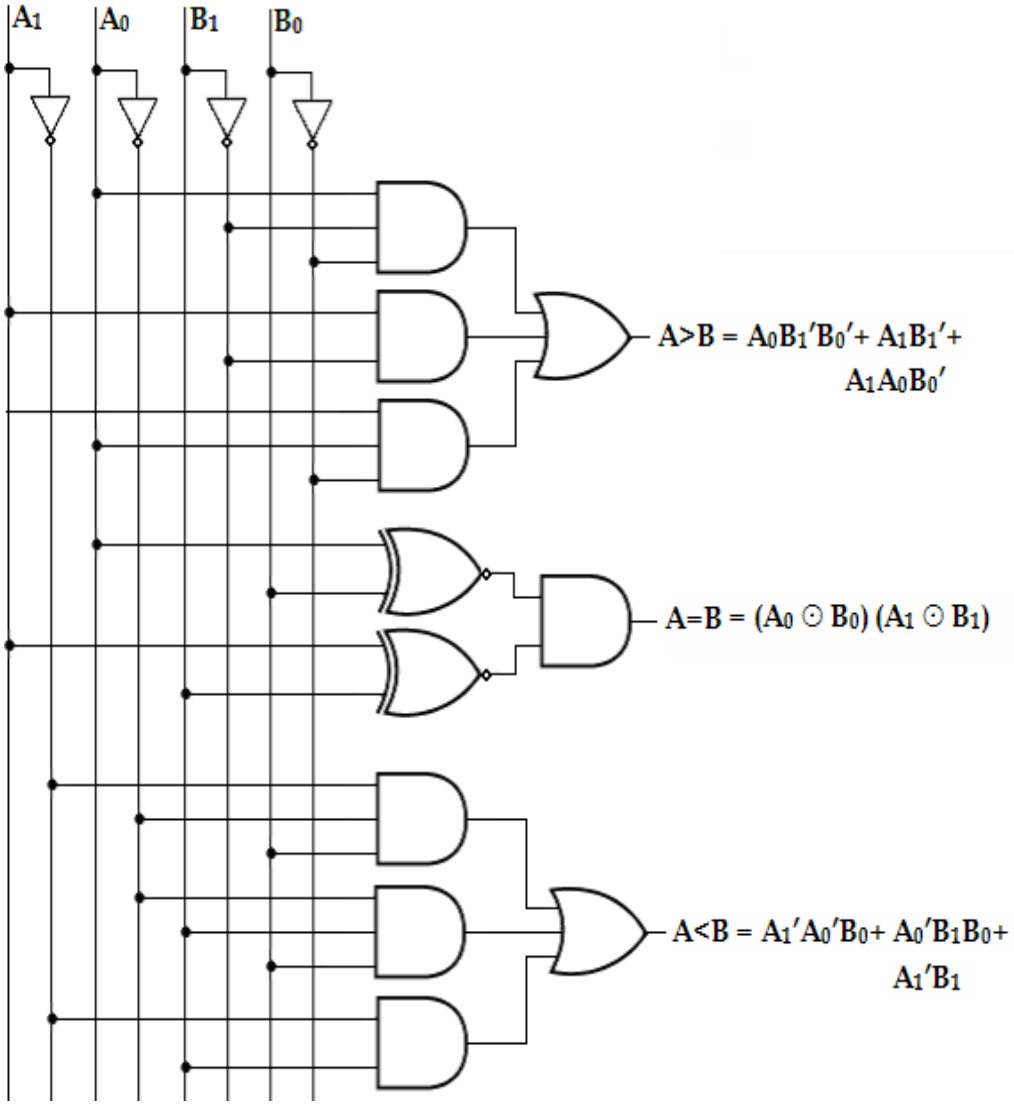
$= (A_0 \odot B_0)(A_1 \odot B_1)$

For A<B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$$

Logic Diagram:



CODE CONVERTERS:

Code to another code of binary code. The following are some of the most commonly used code converters:

- Binary-to-Gray code
- Gray-to-Binary code
- BCD-to-Excess-3
- Excess-3-to-BCD
- Binary-to-BCD
- BCD-to-binary
- Gray-to-BCD
- BCD-to-Gray
- 8 4 -2 -1 to BCD converter

Binary to Gray Converters:

The gray code is often used in digital systems because it has the advantage that only one bit in the numerical representation changes between successive numbers. The truth table for the binary-to-gray code converter is shown below,

Truth table:

Decimal	Binary code				Gray code			
	B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

K-map simplification:

For G_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$G_3 = B_3$

For G_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

$G_2 = B_3'B_2 + B_3B_2'$
$= B_3 \oplus B_2$

For G_1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

$G_1 = B_2'B_1 + B_2B_1'$
$= B_2 \oplus B_1$

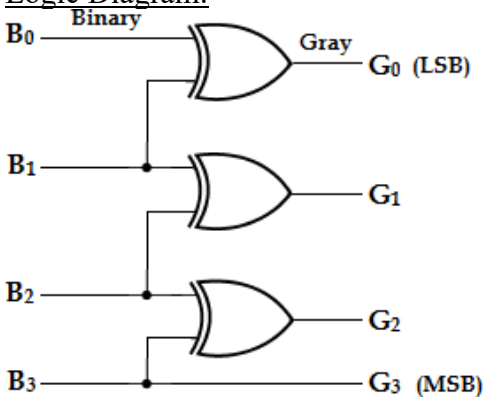
For G_0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

$G_0 = B_1'B_0 + B_1B_0'$
$= B_1 \oplus B_0$

Now, the above expressions can be implemented using EX-OR gates as,

Logic Diagram:



Gray to Binary Converters:

The truth table for the gray-to-binary code converter is shown below,

Truth table:

Gray code				Binary code			
G_3	G_2	G_1	G_0	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

From the truth table, the logic expression for the binary code outputs can be written as,

$G_3 = \sum m\ (8, 9, 10, 11, 12, 13, 14, 15)$

$G_2 = \sum m\ (4, 5, 6, 7, 8, 9, 10, 11)$

$G_1 = \sum m\ (2, 3, 4, 5, 8, 9, 14, 15)$

$G_0 = \sum m\ (1, 2, 4, 7, 8, 11, 13, 14)$

K-map Simplification:

From the above K-map,

$\mathbf{B_3 = G_3}$

$B_2 = G_3'G_2 + G_3G_2'$

$\mathbf{B_2 = G_3 \oplus G_2}$

$B_1 = G_3'G_2'G_1 + G_3'G_2G_1' + G_3G_2G_1 + G_3G_2'G_1'$

$= G_3' (G_2'G_1 + G_2G_1') + G_3 (G_2G_1 + G_2'G_1')$

$= G_3' (G_2 \oplus G_1) + G_3 (G_2 \oplus G_1)'$ $\quad [x \oplus y = x'y + xy'], [(x \oplus y)' = xy + x'y']$

$\mathbf{B_1 = G_3 \oplus G_2 \oplus G_1}$

$B_0 = G_3'G_2' G_1'G_0 + G_3'G_2\_G_1G_0' + G_3G_2G_1\_G_0 + G_3G_2G_1 G_0' + G_3'G_2G_1'G_0' +$
$G_3G_2\_G_1'G_0' + G_3'G_2G_1G_0 + G_3G_2\_G_1 G_0.$

$= G_3'G_2' (G_1'G_0 + G_1G_0') + G_3G_2 (G_1'G_0 + G_1G_0') + G_1'G_0' (G_3'G_2 + G_3G_2') +$
$G_1G_0 (G_3'G_2 + G_3G_2').$

$= G_3'G_2' (G_0 \oplus G_1) + G_3G_2 (G_0 \oplus G_1) + G_1'G_0' (G_2 \oplus G_3) + G_1G_0 (G_2 \oplus G_3).$

$= G_0 \oplus G_1 (G_3'G_2' + G_3G_2) + G_2 \oplus G_3 (G_1'G_0' + G_1G_0)$

$= (G_0 \oplus G_1) (G_2 \oplus G_3)' + (G_2 \oplus G_3) (G_0 \oplus G_1)$ $\quad [x \oplus y = x'y + xy']$

$\mathbf{B_0 = (G_0 \oplus G_1) \oplus (G_2 \oplus G_3).}$

Now, the above expressions can be implemented using EX-OR gates as,

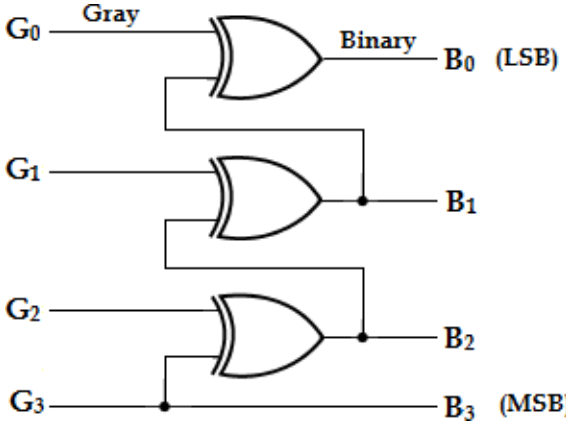


Fig. Logic diagram of 4-bit gray-to-binary converter

BCD –to-Excess-3 Converters:

Excess-3 is a modified form of a BCD number. The excess-3 code can be derived from the natural BCD code by adding 3 to each coded number.

For example, decimal 12 can be represented in BCD as 0001 0010. Now adding 3 to each digit we get excess-3 code as 0100 0101 (12 in decimal). With this information the truth table for BCD to Excess-3 code converter can be determined as,

Truth Table:

Decimal	BCD code				Excess-3 code			
	B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$E_3 = \sum m\ (5, 6, 7, 8, 9) + \sum d\ (10, 11, 12, 13, 14, 15)$

$E_2 = \sum m\ (1, 2, 3, 4, 9) + \sum d\ (10, 11, 12, 13, 14, 15)$

$E_1 = \sum m\ (0, 3, 4, 7, 8) + \sum d\ (10, 11, 12, 13, 14, 15)$

$E_0 = \sum m\ (0, 2, 4, 6, 8) + \sum d\ (10, 11, 12, 13, 14, 15)$

K-map Simplification:

For E_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$E_3 = B_3 + B_2 (B_0 + B_1)$

For E_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	X	X	X	X
10	0	1	X	X

$E_2 = B_2B_1'B_0' + B_2' (B_0 + B_1)$

For E_1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	x	x	x	x
10	1	0	x	x

$E_1 = B_1'B_0' + B_1B_0$

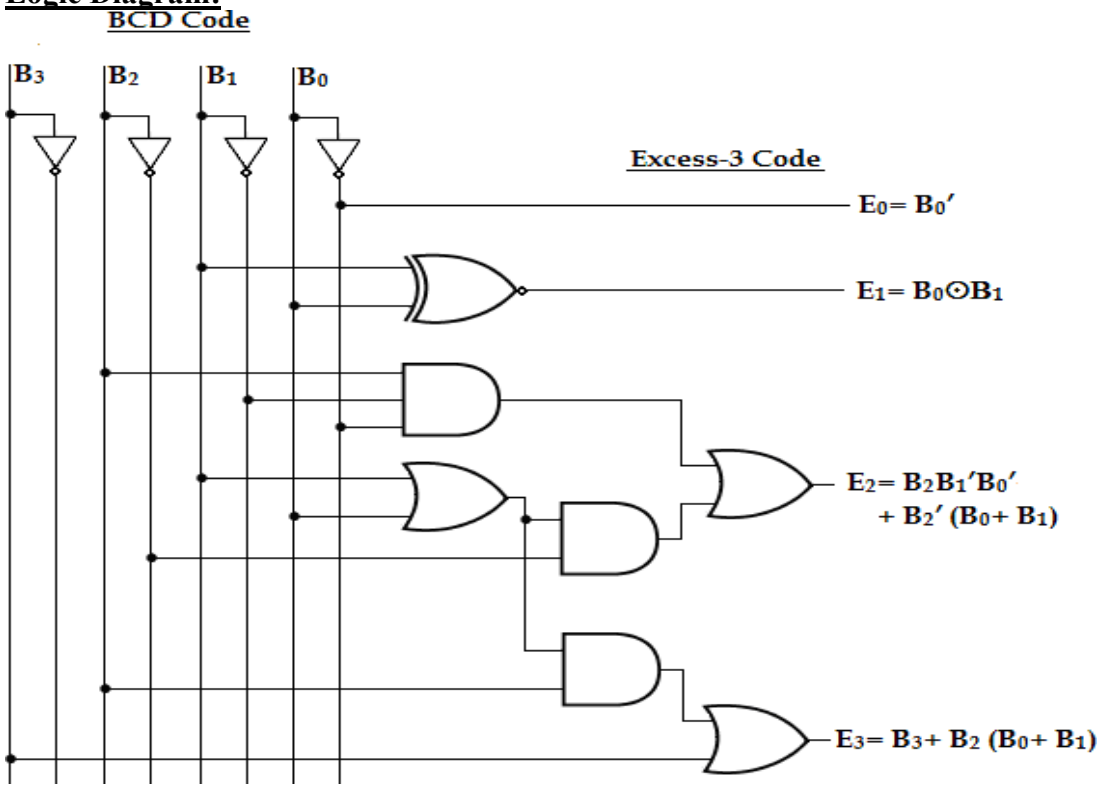
$= B_1 \odot B_0$

For E_0

B_3B_2 \ B_1B_0	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	x	x	x	x
10	1	0	x	x

$E_0 = B_0'$

Logic Diagram:



Excess-3 to BCD Converter: Truth table:

Decimal	Excess-3 code				BCD code			
	E_3	E_2	E_1	E_0	B_3	B_2	B_1	B_0
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	1	0	0

8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	1
11	1	0	1	1	1	0	0	0
12	1	1	0	0	1	0	0	1

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$B_3 = \sum m\,(11, 12) + \sum d\,(0, 1, 2, 13, 14, 15)$

$B_2 = \sum m\,(7, 8, 9, 10) + \sum d\,(0, 1, 2, 13, 14, 15)$

$B_1 = \sum m\,(5, 6, 9, 10) + \sum d\,(0, 1, 2, 13, 14, 15)$

$B_0 = \sum m\,(4, 6, 8, 10, 12) + \sum d\,(0, 1, 2, 13, 14, 15)$

K-map Simplification:

For B_3

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	0	0	0
11	1	X	X	X
10	0	0	1	0

$B_3 = E_3E_2 + E_3E_1E_0$

For B_2

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	0	1	0
11	0	X	X	X
10	1	1	0	1

$B_2 = E_2'E_1' + E_2E_1E_0 + E_3E_1E_0'$

For B_1

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	1	0	1
11	0	X	X	X
10	0	1	0	1

$B_1 = E_1'E_0 + E_1E_0'$
$= E_1 \oplus E_0$

For B_0

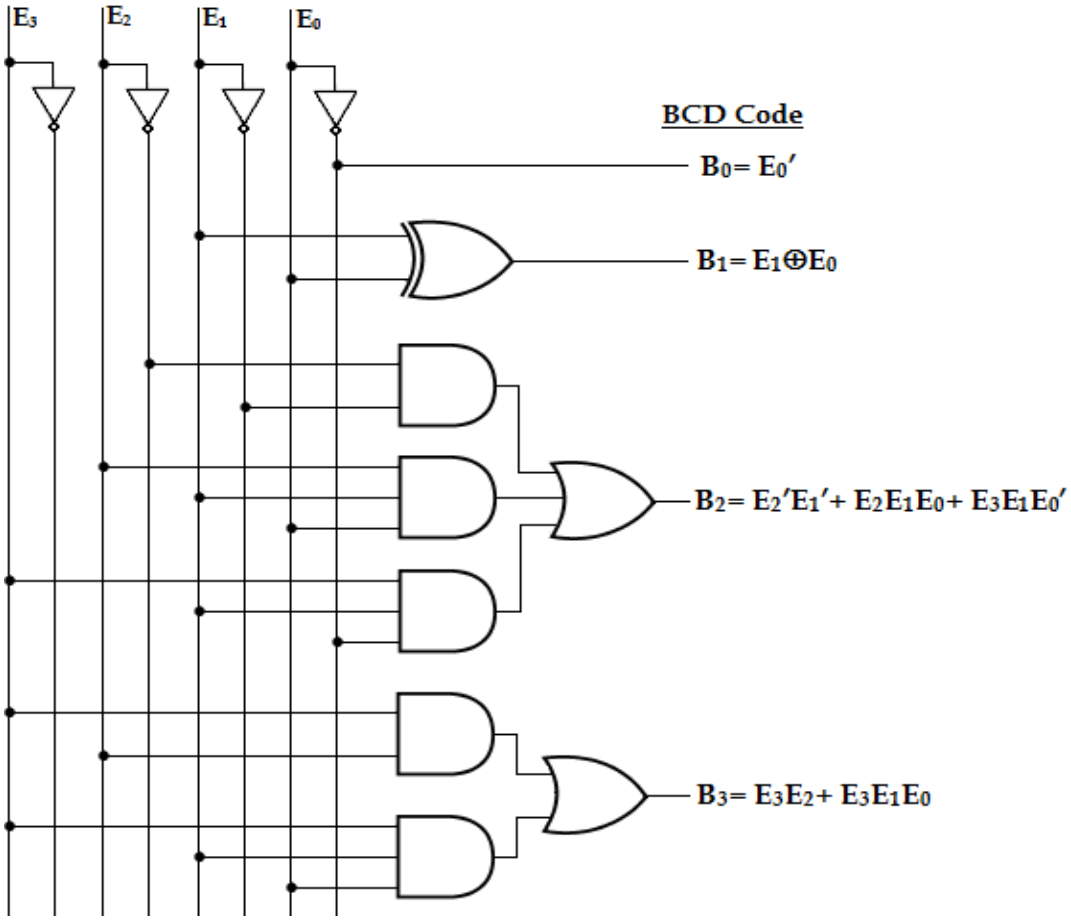
E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	1	0	0	1
11	1	X	X	X
10	1	0	0	1

$B_0 = E_0'$

Now, the above expressions the logic diagram can be implemented as,

Logic Diagram:

Excess-3 Code



BCD –to-Binary Converters:

The steps involved in the BCD-to-binary conversion process are as follows:

- The value of each bit in the BCD number is represented by a binary equivalent or weight.
- All the binary weights of the bits that are 1‘s in the BCD are added.
- The result of this addition is the binary equivalent of the BCD number.

1 9

0001 1001 Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two 4-bit code groups. For example, 19_{10} is represented as,

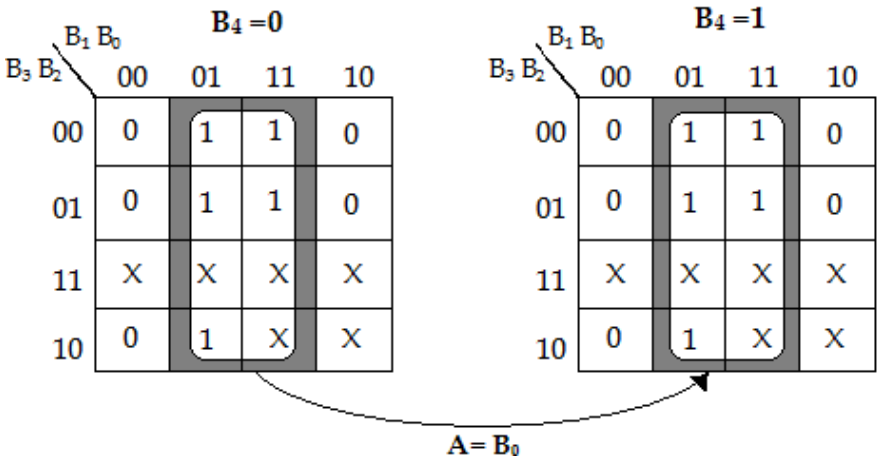
The left-most four-bit group represents 10 and right-most four-bit group represents 9. The binary representation for decimal 19 is $19_{10} = 11001_2$.

BCD Code					Binary				
B_4	B_3	B_2	B_1	B_0	E	D	C	B	A
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1

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

K-map Simplification:

For A



For B

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	X	X	X	X
10	0	0	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	X	X	X	X
10	1	1	X	X

$$B = B_1B_4' + B_1'B_4$$
$$= B_1 \oplus B_4$$

For C

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	X	X	X	X
10	0	0	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$$C = B_4'B_2 + B_2B_1' + B_4B_2'B_1$$

For D

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$$D = B_4'B_3 + B_4B_3'B_2' + B_4B_3'B_1'$$

For E

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	0	0	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	X	X	X	X
10	1	1	X	X

$$E = B_4B_3 + B_4B_2B_1$$

From the above K-map,

A= B_0

B= B_1B_4'+ B1'B4
= B_1 Ex-OR **B_4**

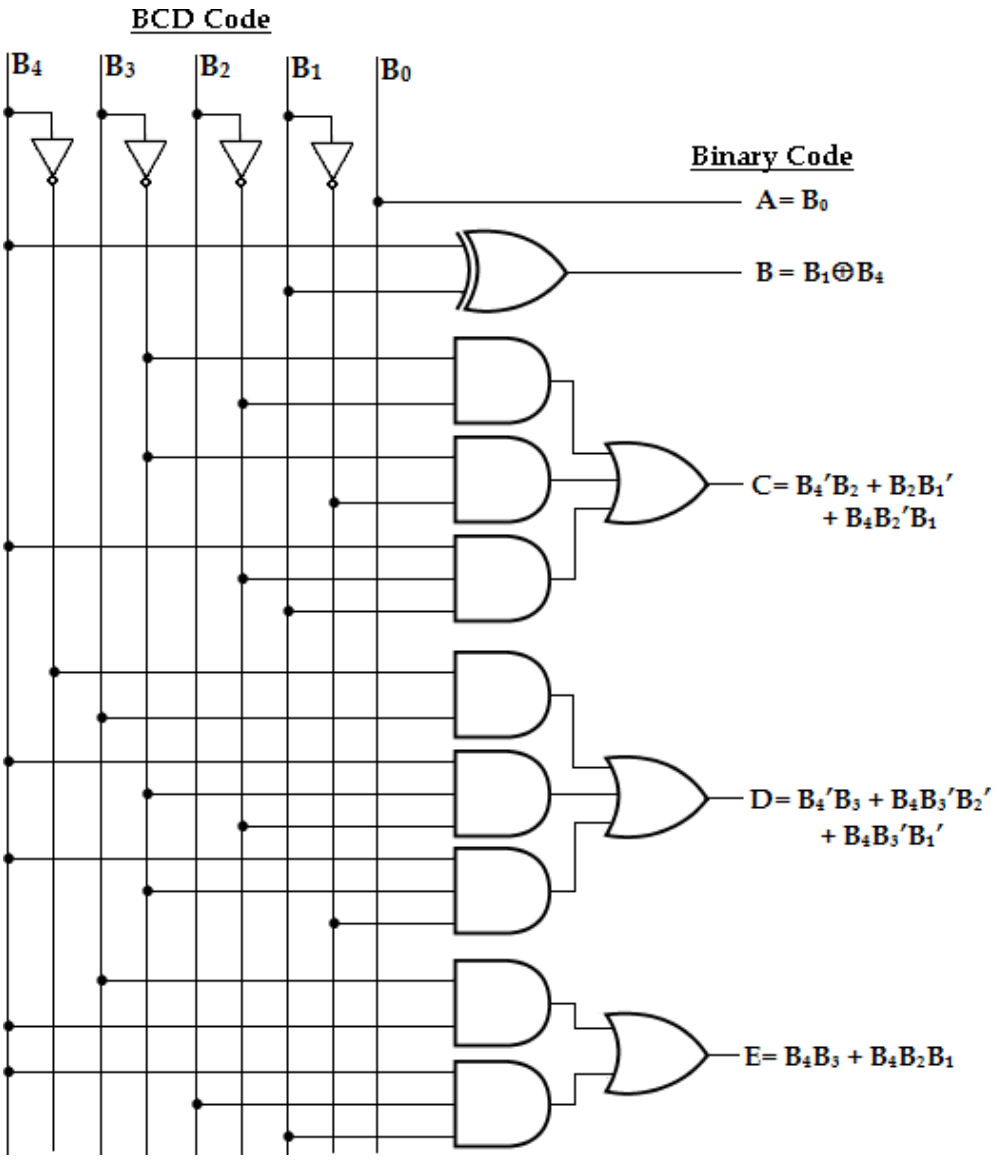
C= B_4'B2 + B_2B_1' + B4B_2'B1

D= B_4'B3 + B_4B_3'B2' + B4B_3'B1'

E= B_4B_3 + $B_4B_2B_1$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



Binary to BCD Converter:

The truth table for binary to BCD converter can be written as,

Truth Table:

Decimal	Binary Code				BCD Code				
	D	C	B	A	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	0	1

From the truth table, the logic expression for the BCD code outputs can be written as,

$B_0 = \sum m\ (1, 3, 5, 7, 9, 11, 13, 15)$

$B_1 = \sum m\ (2, 3, 6, 7, 12, 13)$

$B_2 = \sum m\ (4, 5, 6, 7, 14, 15)$

$B_3 = \sum m\ (8, 9)$

$B_4 = \sum m\ (10, 11, 12, 13, 14, 15)$

K-map Simplification:

For B_0

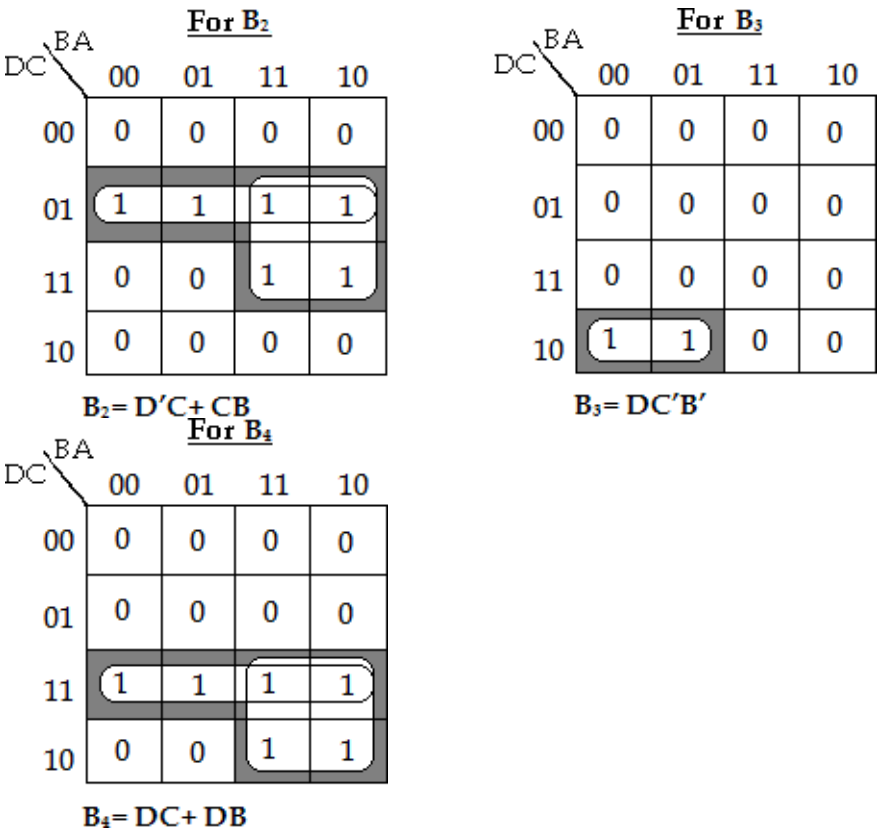
DC \ BA	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	0	0
10	0	0	0	0

$B_1 = DCB' + D'B$



From the above K-map, the logical expression can be obtained as,

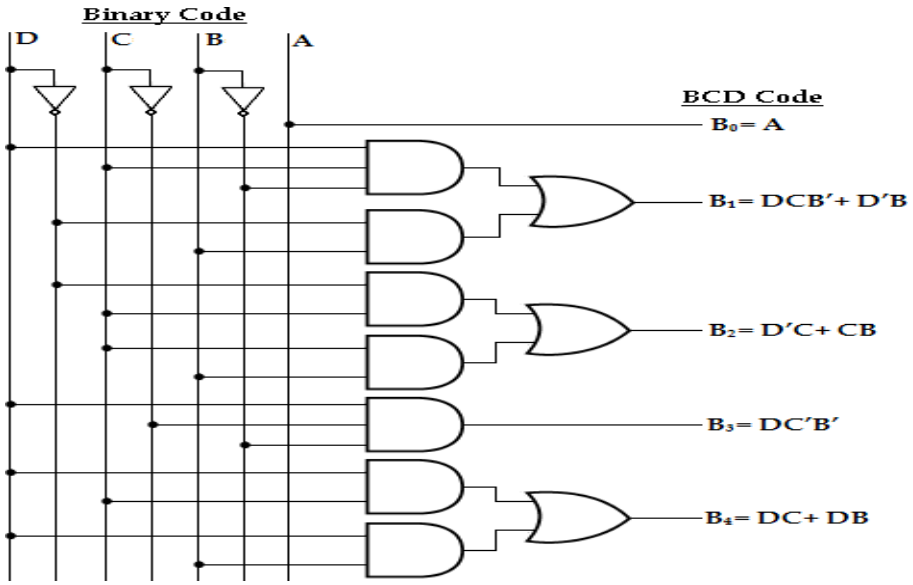
B_0= A

B_1= DCB'+ D'B B_2= D'C+ CB B_3= DC'B'

B_4= DC+ DB

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



Gray to BCD Converter:

The truth table for gray to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	1	1	0	0	0	1	0	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	0	0	0	0	1	1	1
1	1	0	0	0	1	0	0	0
1	1	0	1	0	1	0	0	1
1	1	1	1	1	0	0	0	0
1	1	1	0	1	0	0	0	1
1	0	1	0	1	0	0	1	0
1	0	1	1	1	0	0	1	1
1	0	0	1	1	0	1	0	0
1	0	0	0	1	0	1	0	1

K-map Simplification:

For B_0

G_3G_2 \ G_1G_0	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$B_0 = (G_0 \oplus G_1) \oplus (G_2 \oplus G_3)$

For B_1

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	1	1

$B_1 = G'_2G_1 + G'_3G_2G'_1$

For B_2

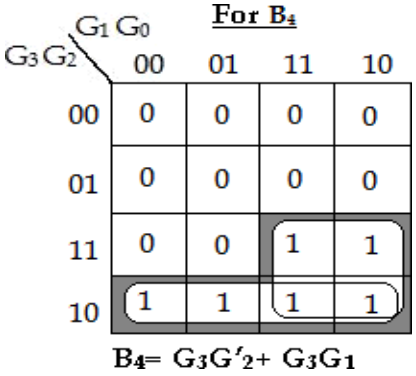
G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	0	0

$B_2 = G'_3G_2 + G_3G'_2G'_1$

For B_3

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	0	0
10	0	0	0	0

$B_3 = G_3G_2G'_1$



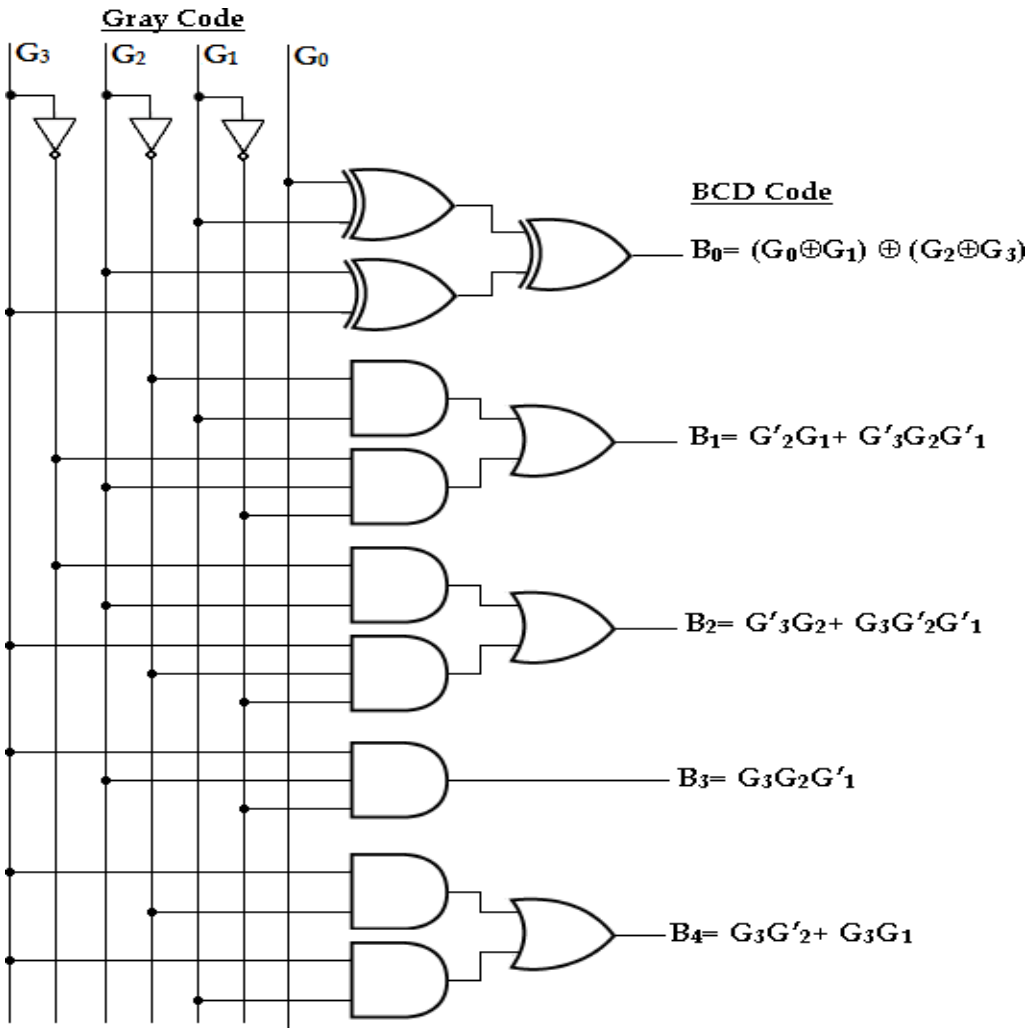
From the above K-map, the logical expression can be obtained as,

B_0= G'2G_1+ G'3G_2G'1 B_2= G'3G_2+ G_3G'2G'1 B_3= G_3G_2G'1

B_4= G_3G'2+ G_3G_1

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



BCD to Gray Converter:

The truth table for gray to BCD converter can be written as,

Truth table:

BCD Code (8421)				Gray code			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

K-map Simplification:

For G_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

$G_3 = B_3$

For G_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

$G_2 = B_3 + B_2$

For G_1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$G_1 = B_2'B_1 + B_2B_1'$
$= B_2 \oplus B_1$

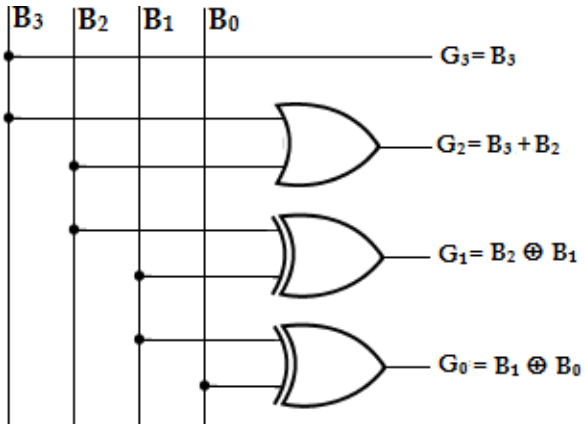
For G_0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	X	X	X	X
10	0	1	X	X

$G_0 = B_1'B_0 + B_1B_0'$
$= B_1 \oplus B_0$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



8 4 -2 -1 to BCD Converter:

The truth table for 8 4 -2 -1 to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
D	**C**	**B**	**A**	**B_4**	**B_3**	**B_2**	**B_1**	**B_0**
0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	1
0	1	1	0	0	0	0	1	0
0	1	0	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	1
1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	1	1	1	0	1	0	0	1
1	1	1	0	1	0	0	0	0
1	1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0

K-map Simplification:

For B_0

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	0	1
11	1	0	0	0
10	0	1	0	1

$B_1 = DCB'A' + D'B'A + D'BA' + C'B'A + C'BA'$
$= A'B'CD + D'(B'A + BA') + C'(B'A + BA')$
$= A'B'CD + D'(A \oplus B) + C'(A \oplus B)$
$= A'B'CD + (A \oplus B)(C' + D')$

For B_2

DC \ BA	00	01	11	10
00	0	X	X	X
01	1	0	0	0
11	0	0	0	0
10	0	1	1	1

$B_2 = D'CB'A' + C'A + C'B$
$= D'CB'A' + C'(A + B)$

For B_3

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	0	0	1	0
10	1	0	0	0

$B_3 = ABCD + A'B'C'D$
$= D(ABC + A'B'C')$

For B_4

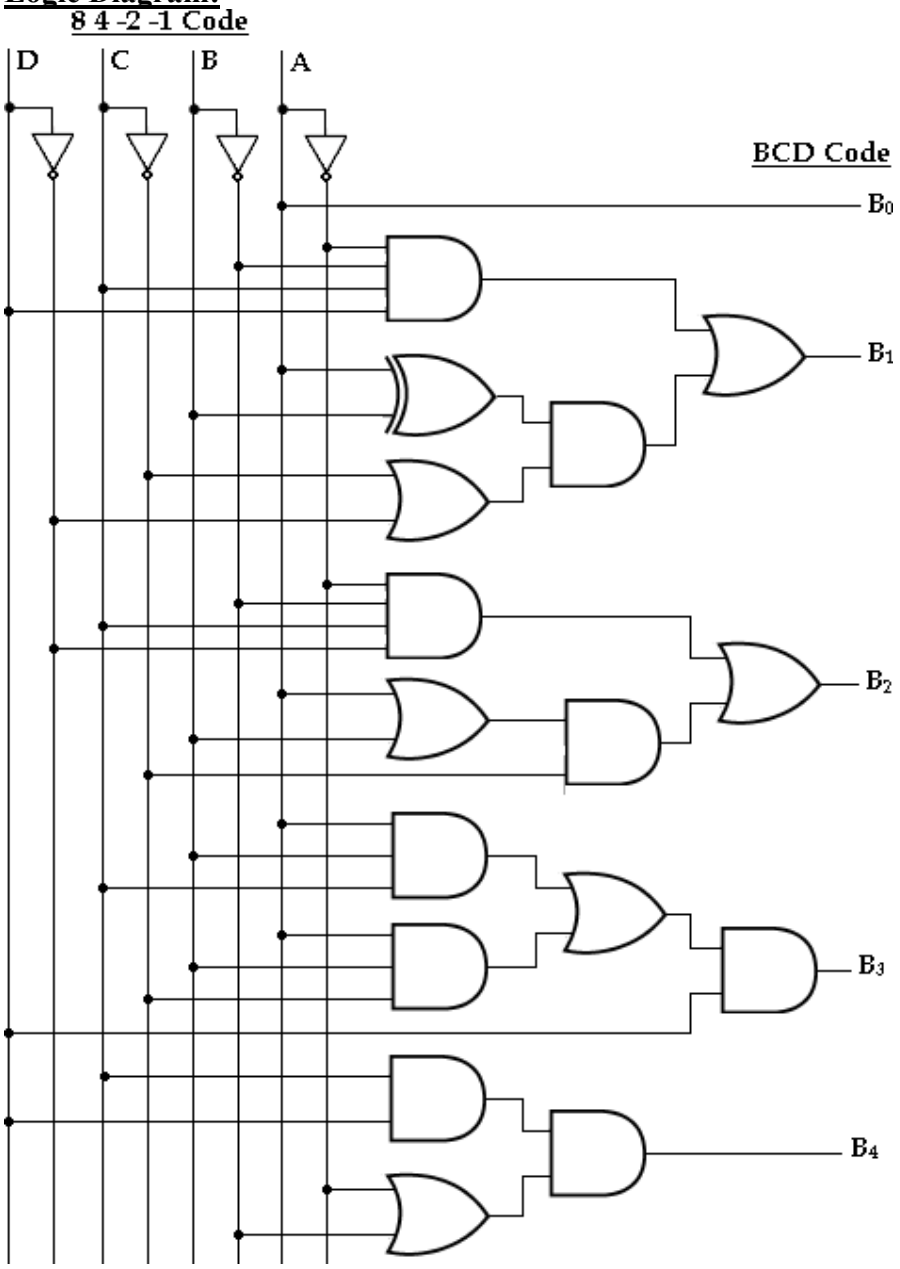
DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	1	1	0	1
10	0	0	0	0

$B_4 = B'CD + A'CD$
$= CD(A' + B')$

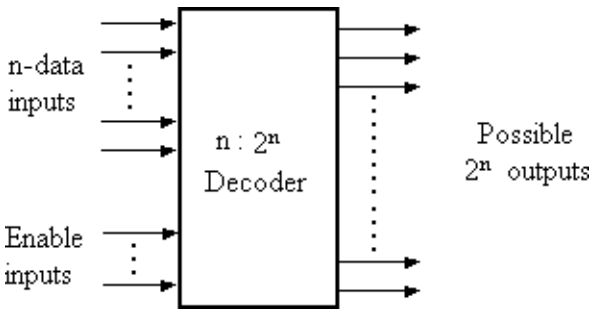
From the above K-map, the logical expression can be obtained as,

B_0= A
B_1= A'B'CD+ (A Ex-OR **B) (C'+D')**
B_2= D'CB'A'+ C' (A+B)
B_3= D (ABC+ A'B'C')
B_4= CD (A'+B')

Logic Diagram:



DECODERS:

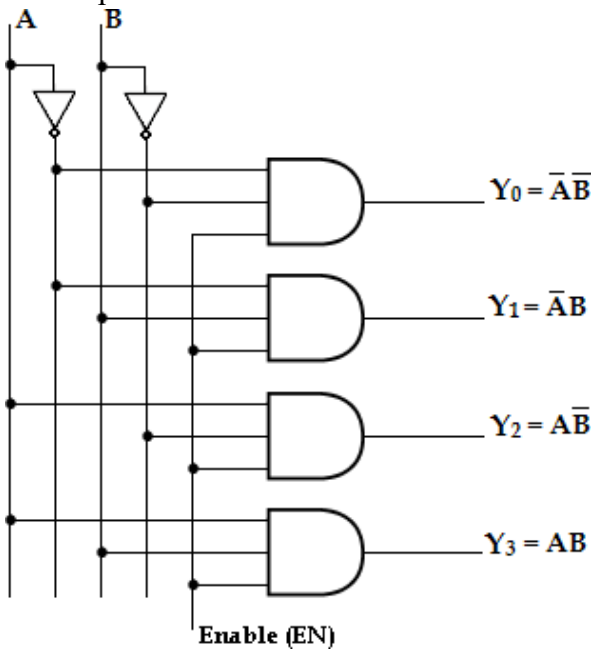


General structure of decoder

A decoder is a combinational circuit that converts binary information from ‗n' input lines to a maximum of ‗2n' unique output lines. The encoded information is presented as ‗n' inputs producing ‗2n' possible outputs. The 2n output values are from 0 through 2n-1. A decoder is provided with enable inputs to activate decoded output based on data inputs. When any one enable input is unasserted, all outputs of decoder are disabled.

Binary Decoder (2 to 4 decoder):

A binary decoder has ‗n' bit binary input and a one activated output out of 2^n outputs. A binary decoder is used when it is necessary to activate exactly one of 2n outputs based on an n-bit input value.



2-to-4 Line decoder

Here the 2 inputs are decoded into 4 outputs, each output representing one of the minterms of the two input variables.

Inputs			Outputs			
Enable	A	B	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

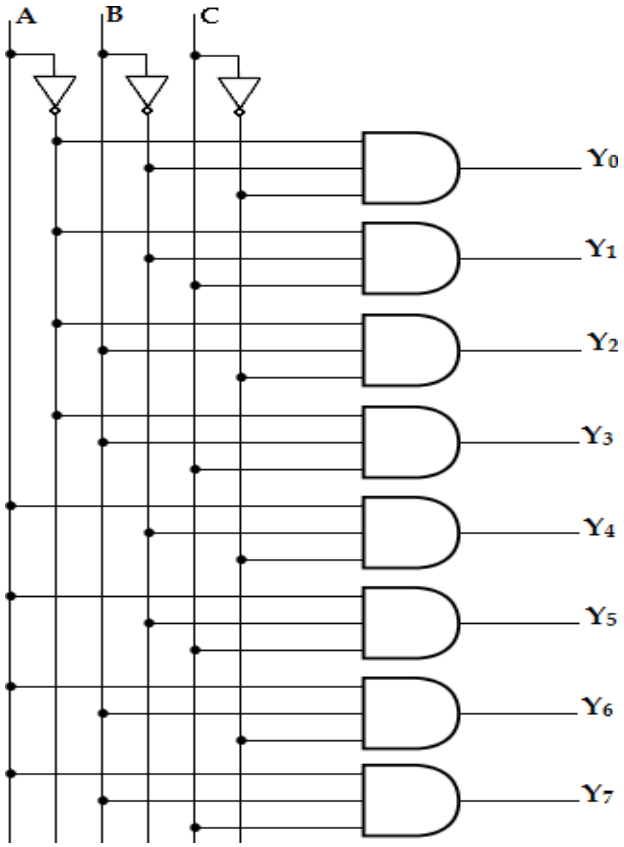
As shown in the truth table, if enable input is 1 (EN= 1) only one of the outputs ($Y_0 - Y_3$), is active for a given input. The output Y_0 is active, ie., Y_0= 1 when inputs A= B= 0, Y_1 is active when inputs, A= 0 and B= 1, Y_2 is active, when input A= 1 and B= 0, Y_3 is active, when inputs A= B= 1.

3 to-8 Line Decoder:

A 3-to-8 line decoder has three inputs (A, B, C) and eight outputs (Y_0- Y_7). Based on the 3 inputs one of the eight outputs is selected.

The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. This decoder is used for binary-to-octal conversion. The input variables may represent a binary number and the outputs will represent the eight digits in the octal number system. The output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

Inputs			**Outputs**							
A	**B**	**C**	**Y_0**	**Y_1**	**Y_2**	**Y_3**	**Y_4**	**Y_5**	**Y_6**	**Y_7**
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



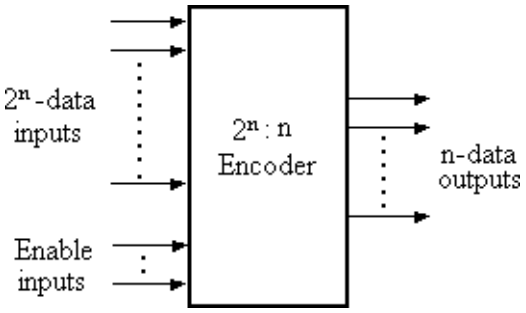
3-to-8 line decoder

Applications of decoders:

- Decoders are used in counter system.
- They are used in analog to digital converter.
- Decoder outputs can be used to drive a display system.

ENCODERS:

An encoder is a digital circuit that performs the inverse operation of a decoder. Hence, the opposite of the decoding process is called encoding. An encoder is a combinational circuit that converts binary information from 2^n input lines to a maximum of _n' unique output lines.



General structure of Encoder

It has 2^n input lines, only one which 1 is active at any time and _n' output lines. It encodes one of the active inputs to a coded binary output with _n' bits. In an encoder, the number of outputs is less than the number of inputs.

Octal-to-Binary Encoder:

It has eight inputs (one for each of the octal digits) and the three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

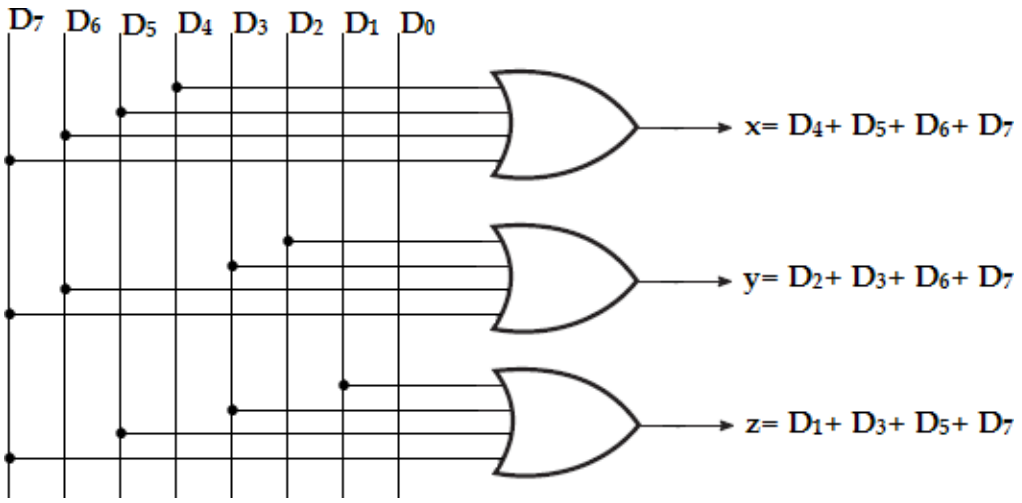
The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1, when the input octal digit is 1 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6, or 7 and the output is 1 for digits 4, 5, 6 or 7. These conditions can be expressed by the following output Boolean functions:

$z = D_1 + D_3 + D_5 + D_7$

$y = D_2 + D_3 + D_6 + D_7$ $x = D_4 + D_5 + D_6 + D_7$

The encoder can be implemented with three OR gates. The encoder defined in the below table, has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination.

For eg., if D_3 and D_6 are 1 simultaneously, the output of the encoder may be 111. This does not represent either D_6 or D_3. To resolve this problem, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers and if D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.



Octal-to-Binary Encoder

Another problem in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; this output is same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate that atleast one input is equal to 1.

Priority Encoder:

A priority encoder is an encoder circuit that includes the priority function. In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

In addition to the two outputs x and y, the circuit has a third output, V (valid bit indicator). It is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0.

The higher the subscript number, higher the priority of the input. Input D_3, has the highest priority. So, regardless of the values of the other inputs, when D_3 is 1, the output for xy is 11.

D_2 has the next priority level. The output is 10, if D_2= 1 provided D_3= 0. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Truth table:

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

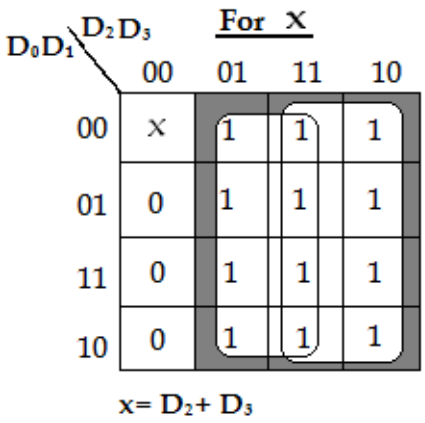
Although the above table has only five rows, when each don't care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the third row in the

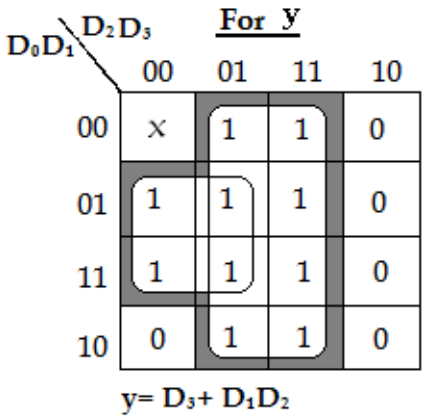
table with X100 represents minterms 0100 and 1100. The don‘t care condition is replaced by 0 and 1 as shown in the table below.

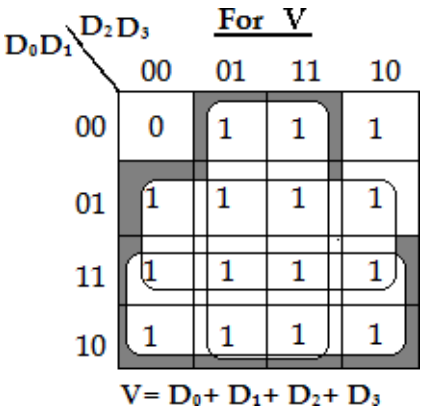
Modified Truth table:

Inputs				Outputs		
D_0	D_1	D_2	D_3	X	y	V
0	0	0	0	X	x	0
1	0	0	0	0	0	1
0	1	0	0	0	1	1
1	1	0	0			
0	0	1	0			
0 1	1 0	1 1	0 0	1	0	1
1	1	1	0			
0	0	0	1			
0	0	1	1			
0	1	0	1			
0	1	1	1	1	1	1
1	0	0	1			
1	0	1	1			
1 1	1 1	0 1	1 1			

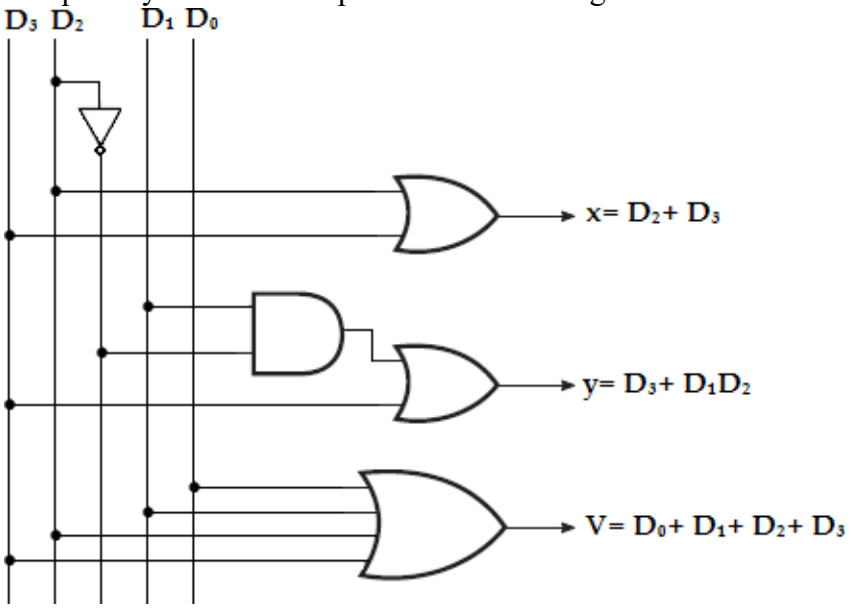
K-map Simplification:







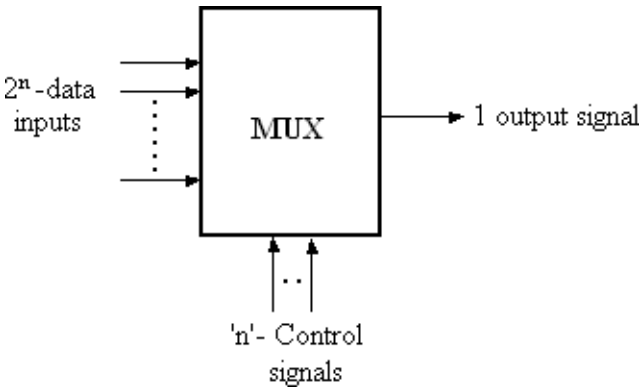
The priority encoder is implemented according to the above Boolean functions.



• **Logic Diagram of Priority Encoder**

MULTIPLEXER: (Data Selector)

A multiplexer or MUX, is a combinational circuit with more than one input line, one output line and more than one selection line. A multiplexer selects binary information present from one of many input lines, depending upon the logic status of the selection inputs, and routes it to the output line. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected. The multiplexer is often labeled as MUX in block diagrams.

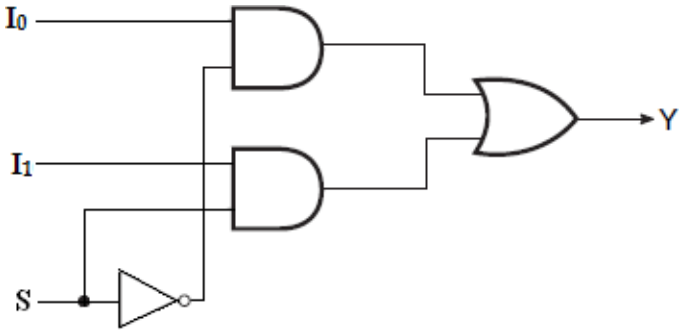


Block diagram of Multiplexer

A multiplexer is also called a **data selector**, since it selects one of many inputs and steers the binary information to the output line.

2-to-1- line Multiplexer:

The circuit has two data input lines, one output line and one selection line, S. When S= 0, the upper AND gate is enabled and I_0 has a path to the output.
When S=1, the lower AND gate is enabled and I_1 has a path to the output.



Logic diagram

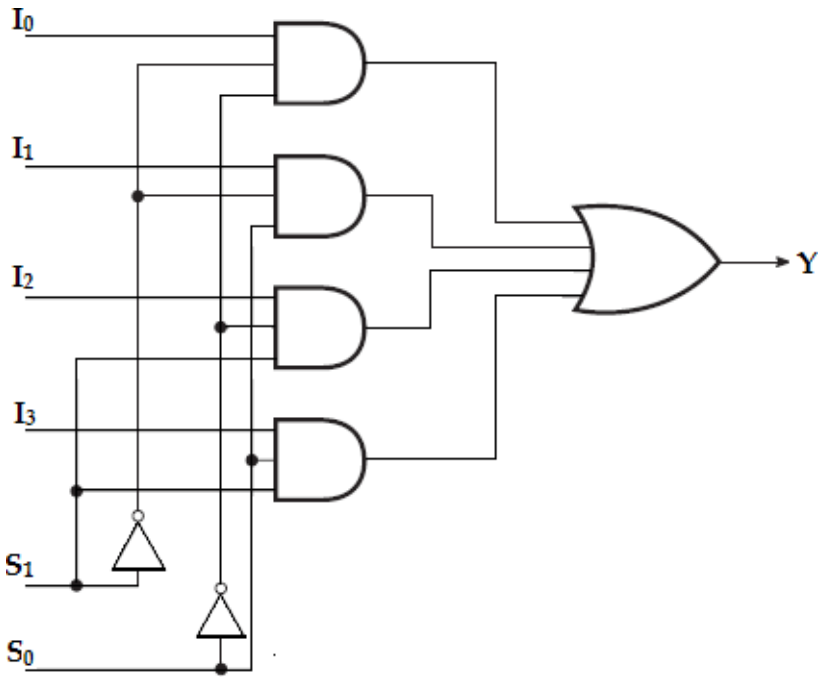
The multiplexer acts like an electronic switch that selects one of the two sources.

Truth table:

S	Y
0	I_0
1	I_1

4-to-1-line Multiplexer:

A 4-to-1-line multiplexer has four (2^n) input lines, two (n) select lines and one output line. It is the multiplexer consisting of four input channels and information of one of the channels can be selected and transmitted to an output line according to the select inputs combinations. Selection of one of the four input channel is possible by two selection inputs.



4-to-1-Line Multiplexer

Each of the four inputs I_0 through I_3, is applied to one input of AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gate are applied to a single OR gate that provides the 1-line output.

Function table:

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

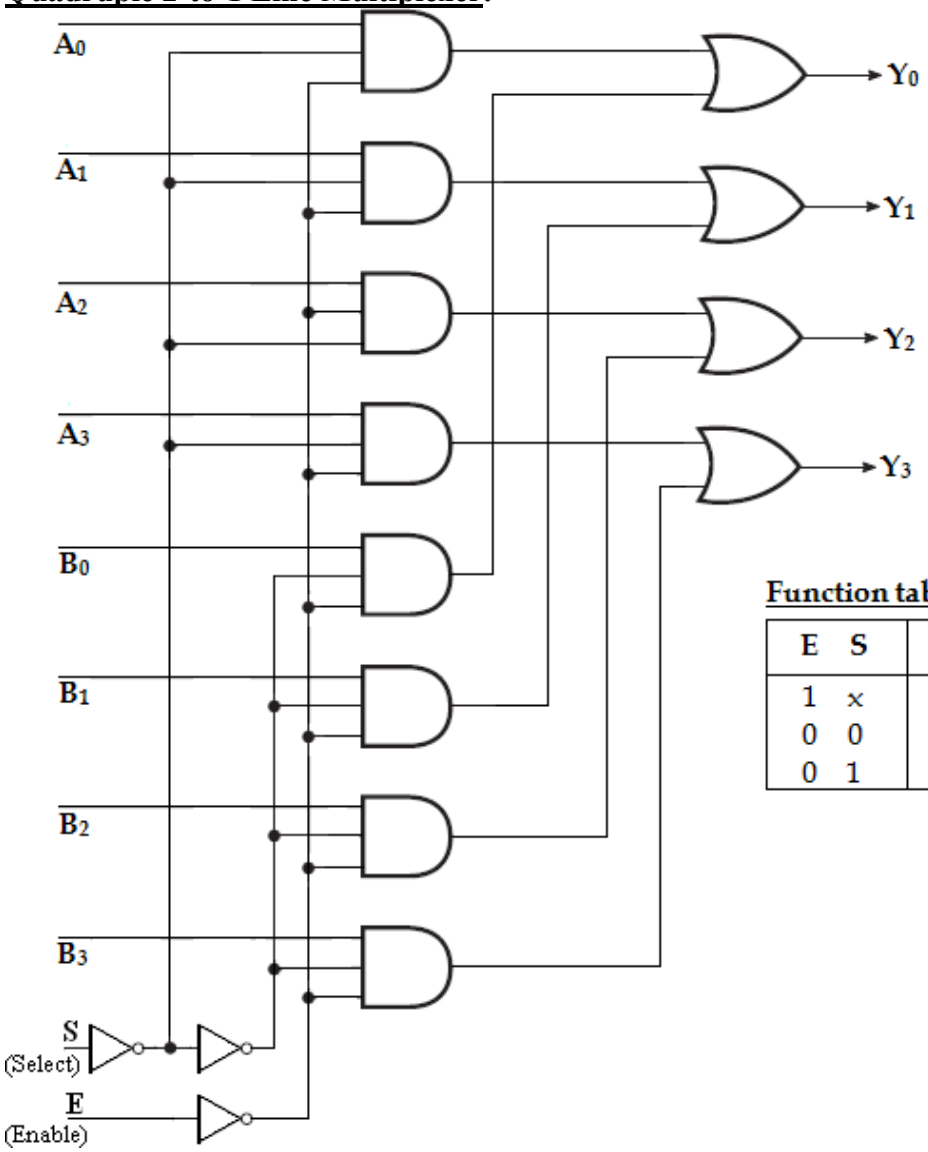
To demonstrate the circuit operation, consider the case when S_1S_0= 10. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have atleast one input equal to 0, which makes their outputs equal to 0. The OR output is now equal to the value of I_2, providing a path from the selected input to the output.

The data output is equal to I_0 only if S_1= 0 and S_0= 0; Y= I_0S_1'S0'. The data output is equal to I_1 only if S_1= 0 and S_0= 1; Y= I_1S_1'S0. The data output is equal to I_2 only if S_1= 1 and S_0= 0; Y= $I_2S_1S_0$'. The data output is equal to I_3 only if S_1= 1 and S_0= 1; Y= $I_3S_1S_0$.
When these terms are ORed, the total expression for the data output is,

Y= I_0S_1'S0'+ I1S_1'S0 +$I_2S_1S_0$'+ I3S_1S_0.

As in decoder, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

Quadruple 2-to-1 Line Multiplexer:



Function table:

E	S	Output Y
1	x	All 0's
0	0	Select A
0	1	Select B

This circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either A0 or B0. Similarly, output Y1 may have the value of A1 or B1, and so on. Input selection line, S selects one of the lines in each of the four multiplexers. The enable input E must be active for normaloperation.

Although the circuit contains four 2-to-1-Line multiplexers, it is viewed as a circuit that selects one of two 4-bit sets of data lines. The unit is enabled when E= 0. Then if S= 0, the four A inputs have a path to the four outputs. On the other hand, if S=1, the four B inputs are applied to the outputs. The outputs have all 0's when E= 1, regardless of the value ofS.

Application:

The multiplexer is a very useful MSI function and has various ranges of applications in data communication. Signal routing and data communication are the important applications of a multiplexer. It is used for connecting two or more sources to guide to a single destination among computer units and it is useful for constructing a common bus system. One of the

general properties of a multiplexer is that Boolean functions can be implemented by this device.

Implementation of Boolean Function using MUX:

Any Boolean or logical expression can be easily implemented using a multiplexer. If a Boolean expression has (n+1) variables, then _n' of these variables can be connected to the select lines of the multiplexer. The remaining single variable along with constants 1 and 0 is used as the input of the multiplexer. For example, if C is the single variable, then the inputs of the multiplexers are C, C', 1 and 0. By this method any logical expression can be implemented. In general, a Boolean expression of (n+1) variables can be implemented using a multiplexer with 2^n inputs.

1) **Implement the following boolean function using 4: 1 multiplexer,**
F (A, B, C) = ∑m (1, 3, 5, 6).

Solution:

Variables, n= 3 (A, B, C) Select lines= n-1 = 2 (**S_1, S_0**)

2^{n-1} to MUX i.e., 22 to 1 = 4 to 1 MUX

Input lines= $2^{n-1} = 2^2 = 4$ (**D_0, D_1, D_2, D_3**)

Implementation table:

Apply variables A and B to the select lines. The procedures for implementing the function are:

- List the input of the multiplexer
- List under them all the minterms in two rows as shown below.

The first half of the minterms is associated with A' and the second half with A. The given function is implemented by circling the minterms of the function and applying the following rules to find the values for the inputs of the multiplexer.

- If both the minterms in the column are not circled, apply 0 to the corresponding input.
- If both the minterms in the column are circled, apply 1 to the corresponding input.
- If the bottom minterm is circled and the top is not circled, apply C to the input.
- If the top minterm is circled and the bottom is not circled, apply C' to the input.

Implementation Table:

	D_0	D_1	D_2	D_3
$\overline{C}$	0	(1)	2	(3)
C	4	(5)	(6)	7
	0	**1**	**C**	$\overline{C}$

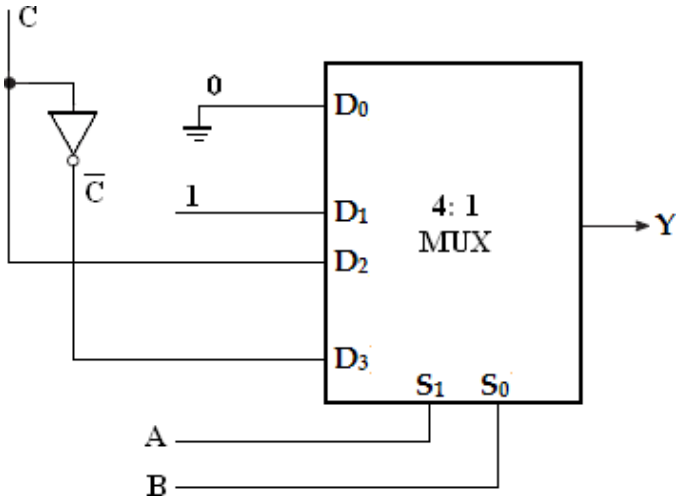


Fig. Multiplexer Implementation

2. F (x, y, z) = ∑m (1, 2, 6, 7)

Solution: Implementation table:

	D_0	D_1	D_2	D_3
$\overline{z}$	0	(1)	(2)	3
z	4	5	(6)	(7)
	0	$\overline{z}$	1	z

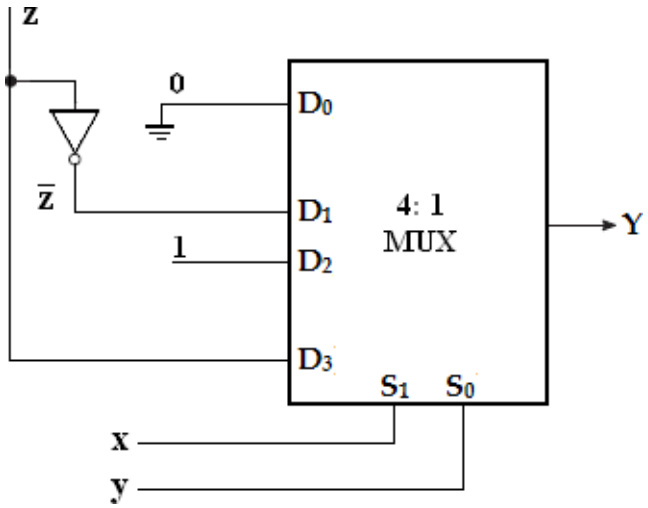


Fig. Multiplexer Implementation

3. **F (A, B, C) = ∑m (1, 2, 4, 5)**

Solution:

Variables, n= 3 (A, B, C) Select lines= n-1 = 2 (**S_1, S_0**)

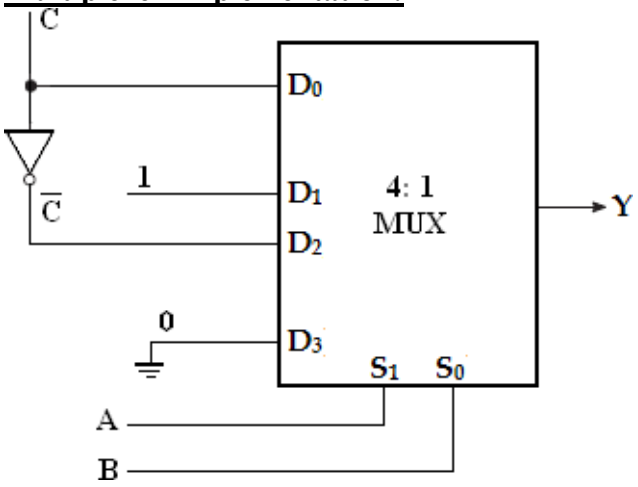
$_2$n-1 to MUX i.e., 22 to 1 = 4 to 1 MUX

Input lines= $2^{n-1} = 2^2 = 4$ (**D_0, D_1, D_2, D_3**)

	D_0	D_1	D_2	D_3
$\overline{C}$	0	1	2	3
C	4	5	6	7
	C	1	$\overline{C}$	0

Fig. Implementation table

Multiplexer Implementation:



4. F(P, Q, R, S)= ∑m (0, 1, 3, 4, 8, 9, 15)

Solution:

Variables, n= 4 (P, Q, R, S) Select lines= n-1 = 3 (**S_2, S_1, S_0**)

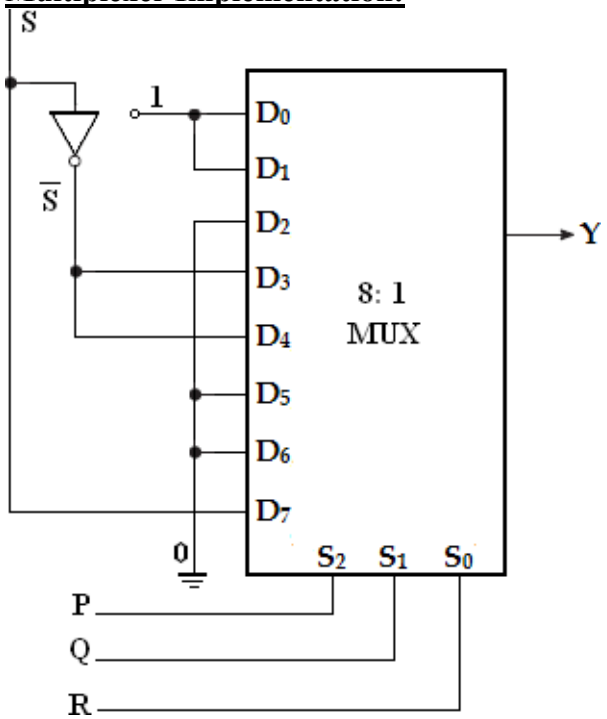
$_2$n-1 to MUX i.e., 23 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{S}$	(0)	(1)	2	(3)	(4)	5	6	7
S	(8)	(9)	10	11	12	13	14	(15)
	1	**1**	**0**	$\overline{S}$	$\overline{S}$	**0**	**0**	**S**

Multiplexer Implementation:



5)Implement the Boolean function using 8: 1 and also using 4:1 multiplexer
F (A, B, C, D) = ∑m (0, 1, 2, 4, 6, 9, 12, 14)
Solution:

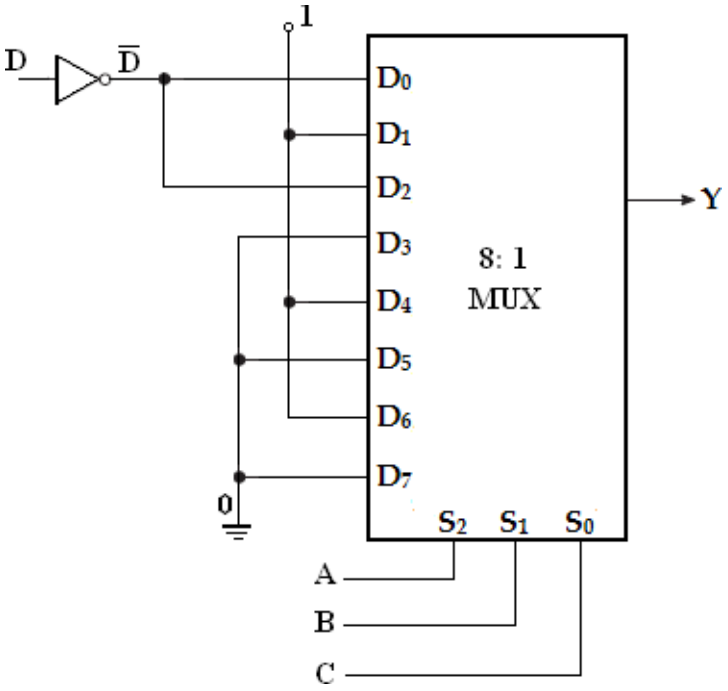
Variables, n= 4 (A, B, C, D) Select lines= n-1 = 3 (**S_2, S_1, S_0**)
2^{n-1} to MUX i.e., 23 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

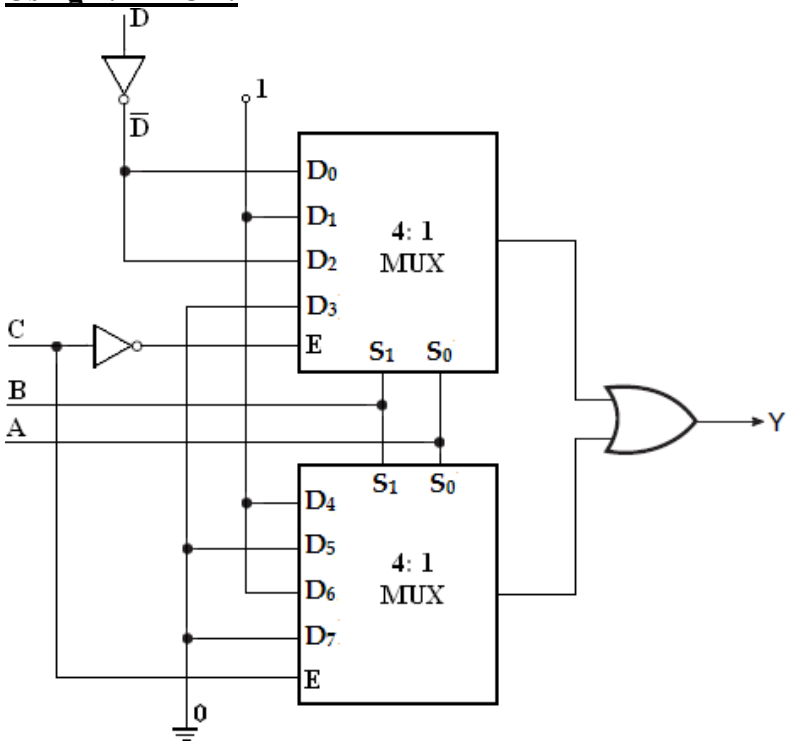
Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	(0)	(1)	(2)	3	(4)	5	(6)	7
D	8	(9)	10	11	(12)	13	(14)	15
	$\overline{D}$	**1**	$\overline{D}$	**0**	**1**	**0**	**1**	**0**

Multiplexer Implementation (Using 8: 1 MUX):



Using 4: 1 MUX:



6. F (A, B, C, D) = ∑m (1, 3, 4, 11, 12, 13, 14, 15)

Solution:

Variables, n= 4 (A, B, C, D) Select lines= n-1 = 3 (**S_2, S_1, S_0**)

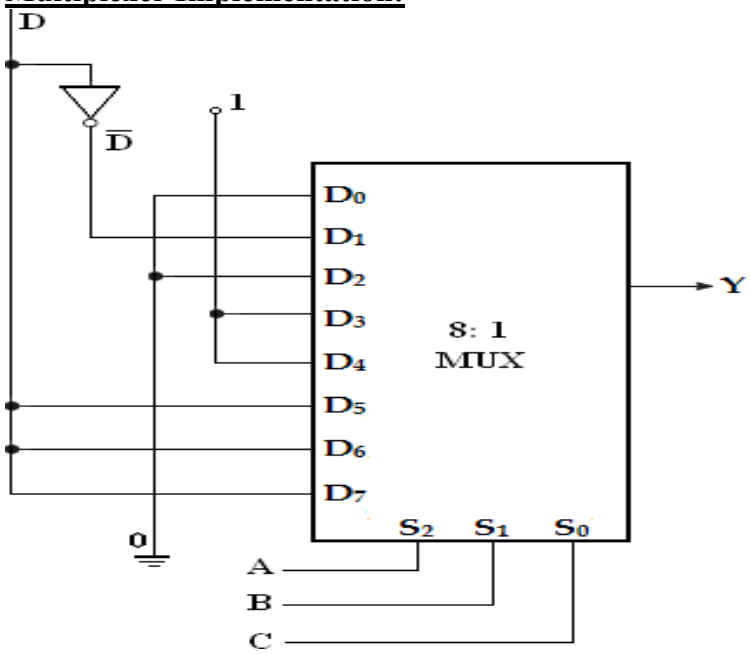
2^{n-1} to MUX i.e., 23 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	2	(3)	(4)	5	6	7
D	8	9	10	(11)	(12)	(13)	(14)	(15)
	0	$\overline{D}$	**0**	**1**	**1**	D	D	D

Multiplexer Implementation:



7)Implement the Boolean function using 8: 1multiplexer.
F (A, B, C, D) = A'BD' + ACD + B'CD + A'C'D.

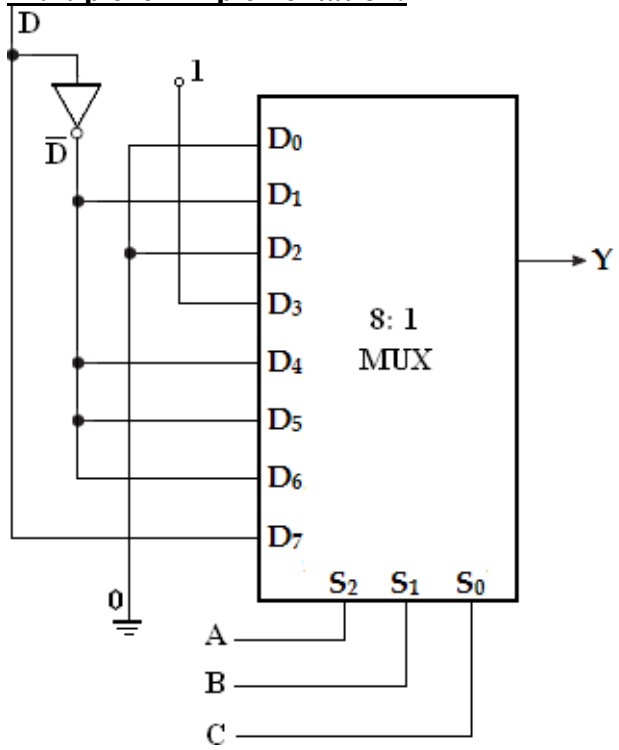
Solution:
Convert into standard SOP form,
= A'BD' (C'+C) + ACD (B'+B) + B'CD (A'+A) + A'C'D (B'+B)
= A'BC'D' + A'BCD'+ AB'CD + ABCD +A'B'CD + AB'CD +A'B'C'D+ A'BC'D

= A'BC'D' + A'BCD'+ AB'CD + ABCD +A'B'CD +A'B'C'D+ A'BC'D
= m4+ m6+ m11+ m15+ m3+ m1+ m5
= ∑m (1, 3, 4, 5, 6, 11, 15)

Implementation table:

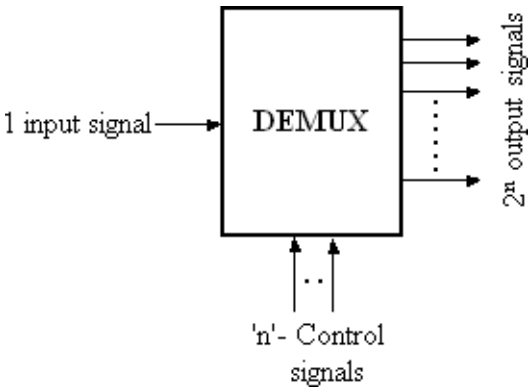
	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	2	(3)	(4)	(5)	(6)	7
D	8	9	10	(11)	12	13	14	(15)
	0	$\overline{D}$	**0**	**1**	$\overline{D}$	$\overline{D}$	$\overline{D}$	D

Multiplexer Implementation:



DEMULTIPLEXER:

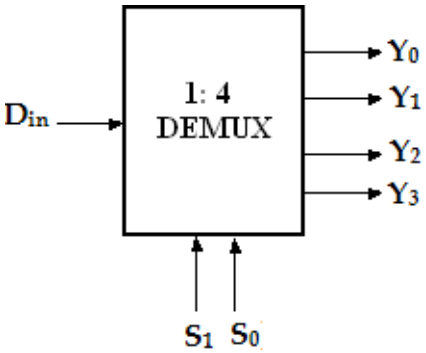
Demultiplex means one into many. Demultiplexing is the process of taking information from one input and transmitting the same over one of several outputs.
A demultiplexer is a combinational logic circuit that receives information on a single input and transmits the same information over one of several (2^n) output lines.



Block diagram of Demultiplexer

The block diagram of a demultiplexer which is opposite to a multiplexer in its operation is shown above. The circuit has one input signal, ‗n' select signals and 2^n output signals. The select inputs determine to which output the data input will be connected. As the serial data is changed to parallel data, i.e., the input caused to appear on one of the n output lines, the demultiplexer is also called a ―data distributer‖ or a ―serial-to-parallel converter‖ .

1-to-4 Demultiplexer:



Logic Symbol

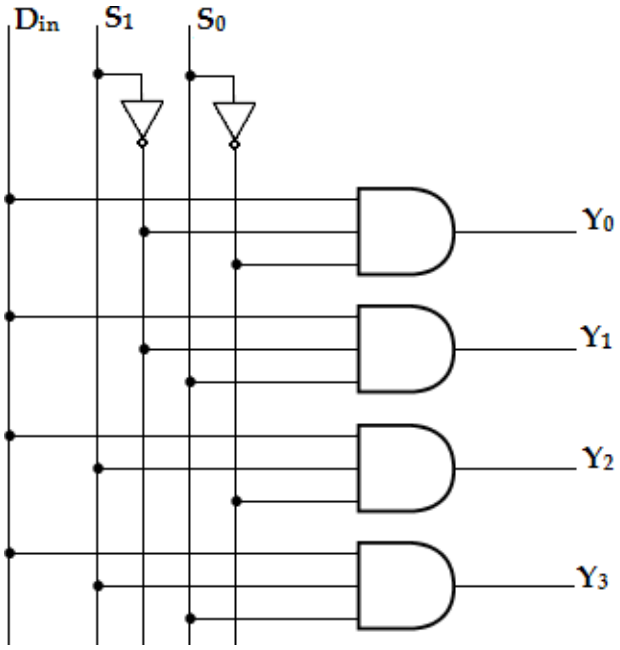
A 1-to-4 demultiplexer has a single input, $\mathbf{D_{in}}$, four outputs ($\mathbf{Y_0}$ **to** $\mathbf{Y_3}$) and two select inputs ($\mathbf{S_1}$ **and** $\mathbf{S_0}$). The input variable D_{in} has a path to all four outputs, but the input information is directed to only one of the output lines. The truth table of the 1-to-4 demultiplexer is shown below.

Truth table of 1-to-4 demultiplexer

Enable	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3
0	x	x	X	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0

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

From the truth table, it is clear that the data input, D_{in} is connected to the output Y_0, when $S_1= 0$ and $S_0= 0$ and the data input is connected to output Y_1 when $S_1= 0$ and $S_0= 1$. Similarly, the data input is connected to output Y_2 and Y_3 when $S_1= 1$ and $S_0= 0$ and when $S_1= 1$ and $S_0= 1$, respectively. Also, from the truth table, the expression for outputs can be written as follows,



Logic diagram of 1-to-4 demultiplexer

$Y_0= S_1'S0'Din$ $Y_1= S_1'S0D_{in}$ $Y_2= S_1S_0'Din$ Y3= S1S0Din

Now, using the above expressions, a 1-to-4 demultiplexer can be implemented using four 3-input AND gates and two NOT gates. Here, the input data line D_{in}, is connected to all the AND gates. The two select lines S_1, S_0 enable only one gate at a time

and the data that appears on the input line passes through the selected gate to the associated output line.

<u>1-to-8 Demultiplexer</u>:

A 1-to-8 demultiplexer has a single input, **D_{in}**, eight outputs (**Y_0 to Y_7**) and three select inputs (**S_2, S_1 and S_0**). It distributes one input line to eight output lines based on the select inputs. The truth table of 1-to-8 demultiplexer is shown below.

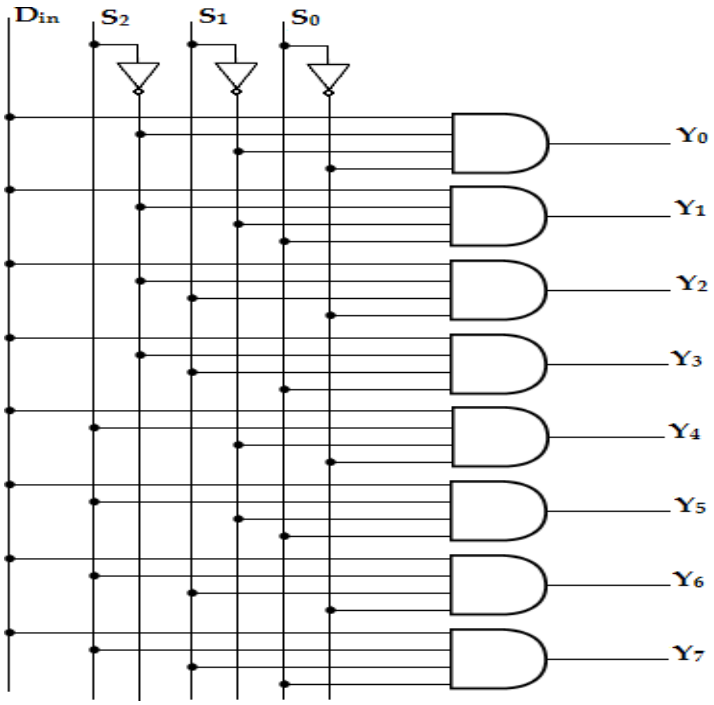
Truth table of 1-to-8 demultiplexer

D_{in}	S_2	S_1	S_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

From the above truth table, it is clear that the data input is connected with one of the eight outputs based on the select inputs. Now from this truth table, the expression for eight outputs can be written as follows:

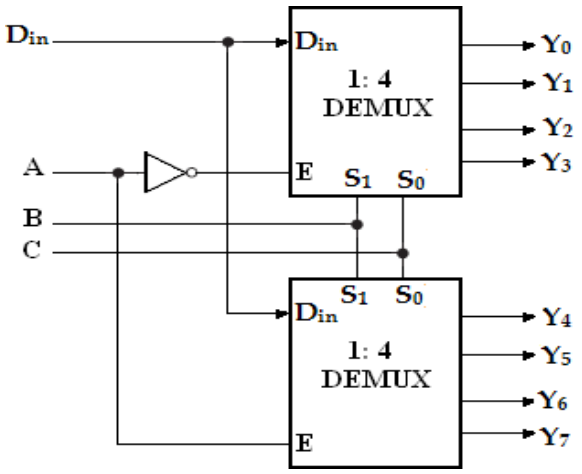
$Y_0 = S_2‘S1‘S0‘Din$ $Y_4 = S_2\ S_1‘S0‘Din$ $Y_1 = S_2‘S1‘S0D_{in}$ $Y_5 = S_2\ S_1‘S0D_{in}$
$Y_2 = S_2‘S1S_0‘Din$ $Y_6 = S_2\ S_1S_0‘Din$ $Y_3 = S_2‘S1S_0D_{in}$ $Y_7 = S_2S_1S_0D_{in}$

Now using the above expressions, the logic diagram of a 1-to-8 demultiplexer can be drawn as shown below. Here, the single data line, D_{in} is connected to all the eight AND gates, but only one of the eight AND gates will be enabled by the select input lines. For example, if $S_2S_1S_0 = 000$, then only AND gate-0 will be enabled and thereby the data input, D_{in} will appear at Y_0. Similarly, the different combinations of the select inputs, the input D_{in} will appear at the respective output.



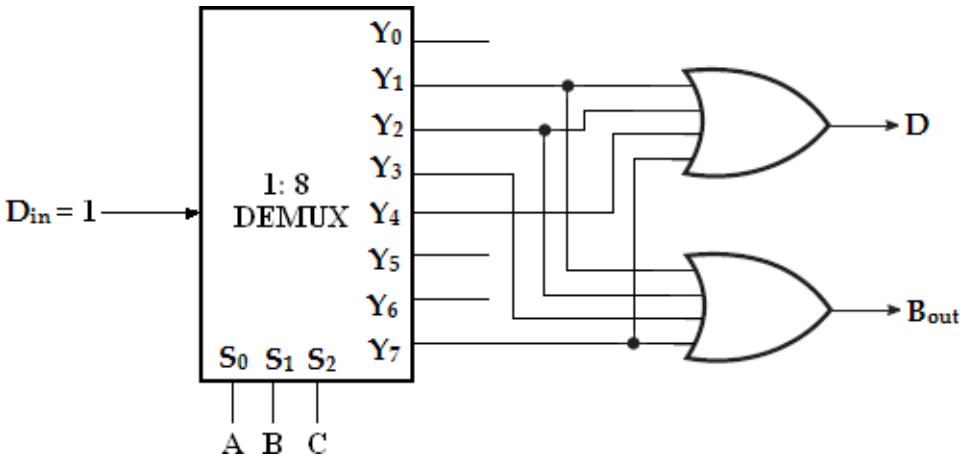
Logic diagram of 1-to-8 demultiplexer

1)Design 1:8 demultiplexer using two 1:4 DEMUX.



2)Implement full subtractor using demultiplexer.

Inputs			Outputs	
A	**B**	**B_{in}**	**Difference(D)**	**Borrow(B_{out})**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



TEXT BOOKS:

1. Morris Mano, “Digital design”, 3rd Edition, Prentice Hall of India, 2008.

REFERENCE BOOKS:

1. Milos Ercegovac, Jomas Lang, “Introduction to Digital Systems”, Wiley publications, 1998.
2. John M. Yarbrough, “Digital logic: Applications and Design”, Thomas – Vikas Publishing House, 2002.
3. R.P.Jain, “Modern digital Electronics”,3rd Edition, TMH, 2003.
4. William H. Gothmann, “Digital Electronics”, Prentice Hall, 2001.

QUESTION BANK

PART-A

1. Describe combinational logic circuits.
2.Definedemultiplexer.
3.Illustrate the logic diagram of full adder.
4.Relate the other name for multiplexer and demultiplexer.
5.Illustrate a 4 : 1 MUX.
6.List the limitations of encoder.
7.Express the truth table for 2 bit subtractor.
8.Design MUX for the expression
$F(a,b,c) = \pi (0,3,4,5)$
9.Illustrate 3 to 8 line decoder.
10.Define encoder.

PART-B

1. A combination circuit is defined by following three Boolean functions. Design the circuit with a decoder f_1=x’y’z’+xz,f_2= xy’z’+x’y, f_3=x’y’z+xy
2. Design 4:2 Priority Encoder.
3. Design and implement 8:3 Encoder circuit.
4. Design and implement 1:8 Demultiplexer circuit.
5. Interpret the following function with a multiplexer,
$F(a,b,c,d) = \sum (0,1,4,8,9,15)$
6. Design a look ahead carry generator.
7. Design and implement 8:1 MUX circuit.
8. Design and implement a full adder and full subtractor circuit.
9. Design and implement 4 bit BCD adder circuit.
10. Design BCD to 7- segment display.

UNIT IV
SEQUENTIAL CIRCUITS

Introduction to Sequential circuits – Flip flops – SR, JK, D and T flip flops, Master Slave flip flop, Characteristic and excitation table – Realization of one flip flop with other flip flops – Regisrers – Shift registers – Counters – Synchronous and Asynchronous counters – Modulus counters – Ring Counter – Johnson Counter – State diagram, State table, State minimization – Hazards.

.INTRODUCTION

In sequential logic circuits, it consists of combinational circuits to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information either 1 or 0.

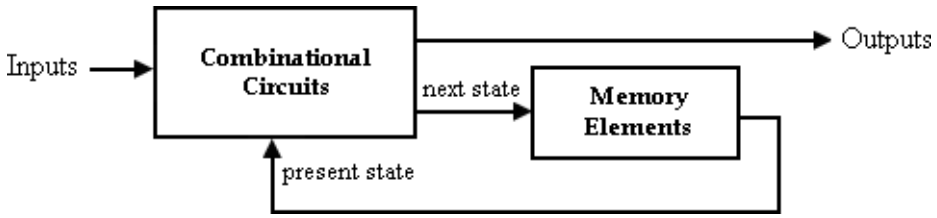


Fig. Sequential Circuit- Block Diagram

The information stored in the memory elements at any given time defines the present state of the sequential circuit. The present state and the external circuit determine the output and the next state of sequential circuits.

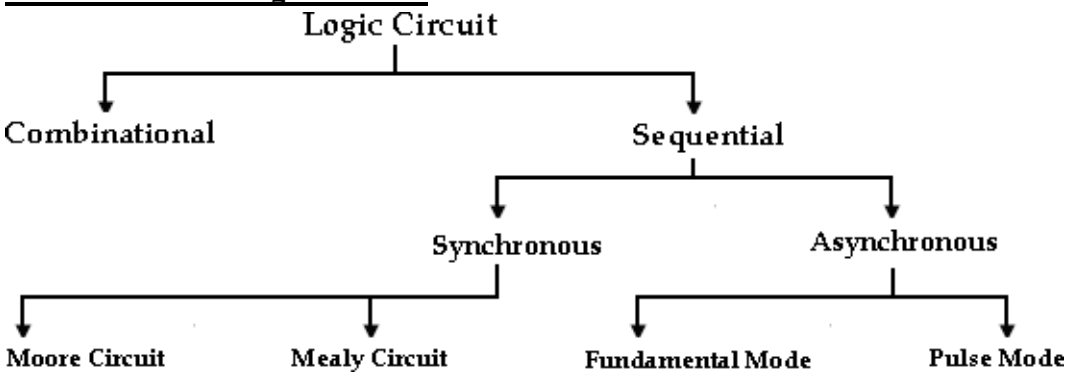
Thus in sequential circuits, the output variables depend not only on the present input variables but also on the past history of input variables. The rotary channel selected knob on an old-fashioned TV is like a combinational. Its output selects a channel based only on its current input – the position of the knob. The channel-up and channel-down push buttons on a TV is like a sequential circuit. The channel selection depends on the past sequence of up/down pushes.

The comparison between combinational and sequential circuits is given in table below.

S.No	Combinational logic	Sequential logic
1	The output variable, at all times depends on the combination of input variables.	The output variable depends not only on the present input but also depend upon the past history of inputs.
2	Memory unit is not required	Memory unit is required to store the past history of input variables.

3	Faster in speed	Slower than combinational circuits.
4	Easy to design	Comparatively harder to design.
5	Eg. Parallel adder	Eg. Serial adder

Classification of Logic Circuits



The sequential circuits can be classified depending on the timing of their signals:

- Synchronous sequential circuits
- Asynchronous sequential circuits.

In synchronous sequential circuits, signals can affect the memory elements only at discrete instants of time. In asynchronous sequential circuits change in input signals can affect memory element at any instant of time. The memory elements used in both circuits are Flip-Flops, which are capable of storing 1- bit information.

S.No	Synchronous sequential circuits	Asynchronous sequential circuits
1	Memory elements are clocked Flip-Flops	Memory elements are either unclocked Flip-Flops or time delay elements.
2	The change in input signals can affect memory element upon activation of clock signal.	The change in input signals can affect memory element at any instant of time.
3	The maximum operating speed of clock depends on time delays involved.	Because of the absence of clock, it can operate faster than synchronous circuits.
4	Easier to design	More difficult to design

LATCHES:

Latches and Flip-Flops are the basic building blocks of the most sequential circuits. Latches are used for a sequential device that checks all of its inputs continuously and changes

its outputs accordingly at any time independent of clocking signal. Enable signal is provided with the latch. When enable signal is active output changes occur as the input changes. But when enable signal is not activated input changes do not affect the output.
Flip-Flop is used for a sequential device that normally samples its inputs and changes its outputs only at times determined by clocking signal.

SR Latch:

The simplest type of latch is the set-reset (SR) latch. It can be constructed from either two NOR gates or two NAND gates.

SR latch using NOR gates:

The two NOR gates are cross-coupled so that the output of NOR gate 1 is connected to one of the inputs of NOR gate 2 and vice versa. The latch has two outputs Q and Q' and two inputs, set and reset.

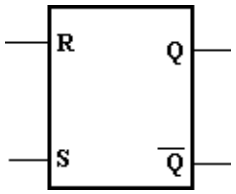
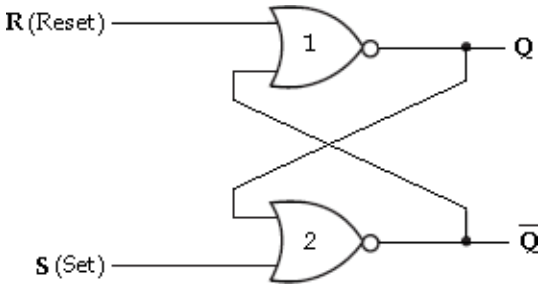


Fig. Logic Symbol



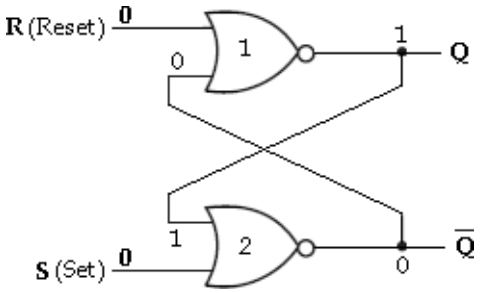
Logic Diagram

Before going to analyse the SR latch, we recall that a logic 1 at any input of a NOR gate forces its output to a logic 0. Let us understand the operation of this circuit for various input/ output possibilities.

Case 1: S= 0 and R= 0

Initially, Q= 1 and Q'= 0

Let us assume that initially Q=1 and Q'=0. With Q'=0, both inputs to NOR gate 1 are at logic 0. So, its output, Q is at logic 1. With Q=1, one input of NOR gate 2 is at logic



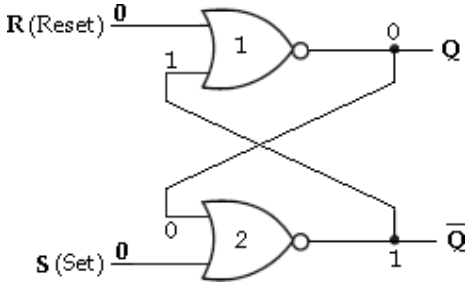
Hence its output, Q' is at logic 0. This shows that when S and R both are low, the output does not change.

Initially, Q= 0 and Q'= 1

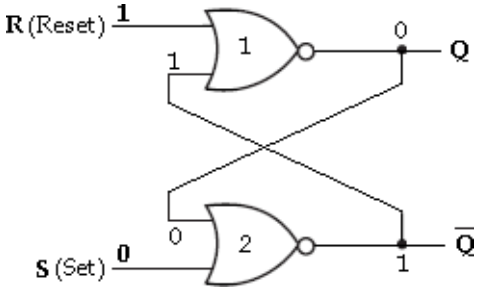
With Q'=1, one input of NOR gate 1 is at logic 1, hence its output, Q is at logic

- With Q=0, both inputs to NOR gate 2 are at logic 0. So, its output Q' is at logic 1. In

this case also there is no change in the output state.

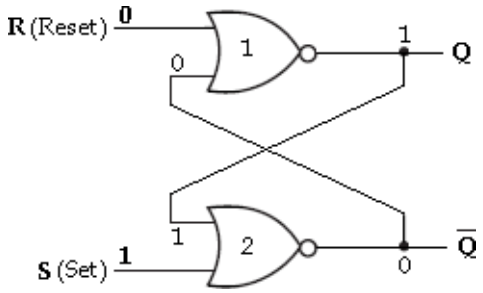


Case 2: S= 0 and R= 1



In this case, R input of the NOR gate 1 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 2 are now at logic 0. So that its output, Q' is at logic 1.

Case 3: S= 1 and R= 0



In this case, S input of the NOR gate 2 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 1 are now at logic 0. So that its output, Q is at logic 1.

Case 4: S= 1 and R= 1

When R and S both are at logic 1, they force the outputs of both NOR gates to the low state, i.e., (Q=0 and Q'=0). So, we call this an indeterminate or prohibited state, and represent this condition in the truth table as an asterisk (*). This condition also violates the basic definition of a latch that requires Q to be complement of Q'. Thus in normal operation this condition must be avoided by making sure that 1's are not applied to both the inputs simultaneously.
We can summarize the operation of SR latch as follows:

- When S= 0 and R= 0, the output, Q_{n+1} remains in its present state, Q_n.
- When S= 0 and R= 1, the latch is reset to 0.
- When S= 1 and R= 1, the latch is set to 1.
- When S= 1 and R= 1, the output of both gates will produce0. i.e., **Q_{n+1}= Q_{n+1}'= 0.**

S	R	Q_n	Q_{n+1}	State
0	0	0	0	No Change
0	0	1	1	(NC)
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	x	Indeterminate
1	1	1	x	*

SR latch using NAND gates:

The SR latch can also be implemented using NAND gates. The inputs of this Latch are S and R. To understand how this circuit functions, recall that a low on any input to a NAND gate forces its output high.

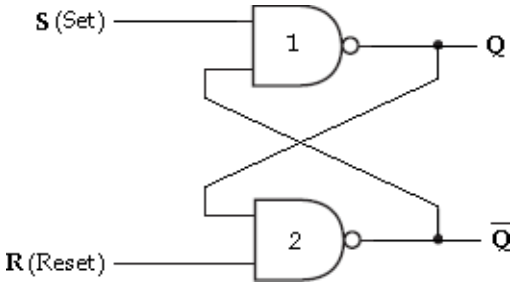


Fig. Circuit diagram

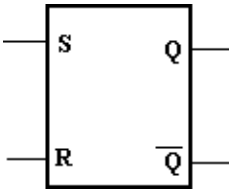


Fig. Logic Symbol

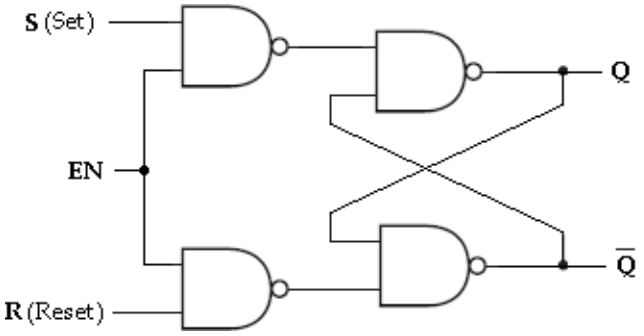
We can summarize the operation of SR latch as follows:

- When S= 0 and R= 0, the output of both gates will produce 0. i.e., **Q_{n+1}= Q_{n+1}'= 1.**
- When S= 0 and R= 1, the latch is reset to 0.
- When S= 1 and R= 0, the latch is set to 1.
- When S= 1 and R= 1, the output, Q_{n+1} remains in its present state, Q_n.

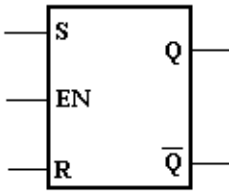
S	R	Q_n	Q_{n+1}	State
0	0	0	x	Indeterminate
0	0	1	x	*
0	1	0	1	Set
0	1	1	1	
1	0	0	0	Reset
1	0	1	0	
1	1	0	0	No Change
1	1	1	1	(NC)

Gated SR Latch:

In the SR latch, the output changes occur immediately after the input changes i.e, the latch is sensitive to its S and R inputs all the time. A latch that is sensitive to the inputs only when an enable input is active. Such a latch with enable input is known as gated SR latch. The circuit behaves like SR latch when EN= 1. It retains its previous state when EN= 0



SR Latch with enable input using NAND gates

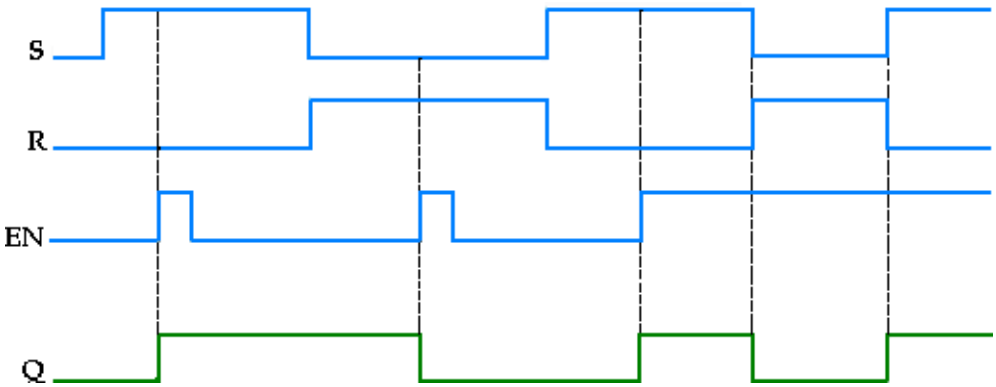


Logic Symbol

The truth table of gated SR latch is show below.

EN	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	
1	1	1	0	x	Indeterminate
1	1	1	1	x	*
0	x	x	0	0	No Change (NC)
0	x	x	1	1	

When S is HIGH and R is LOW, a HIGH on the EN input sets the latch. When S is LOW and R is HIGH, a HIGH on the EN input resets the latch.



D Latch

In SR latch, when both inputs are same (00 or 11), the output either does not change or it is invalid. In many practical applications, these input conditions are not required. These input conditions can be avoided by making them complement of
each other. This modified SR latch is known as **D latch.**

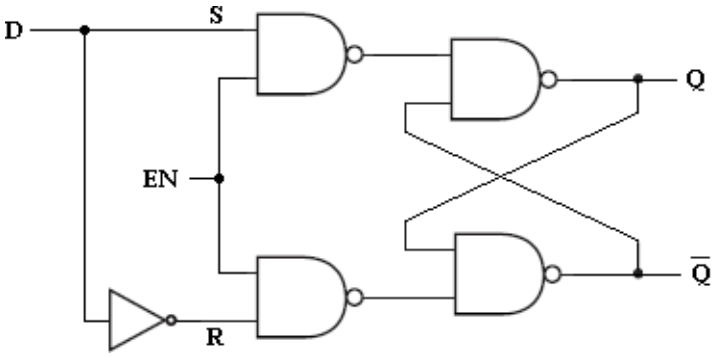


Fig. Logic Diagram

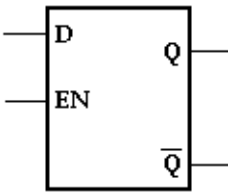
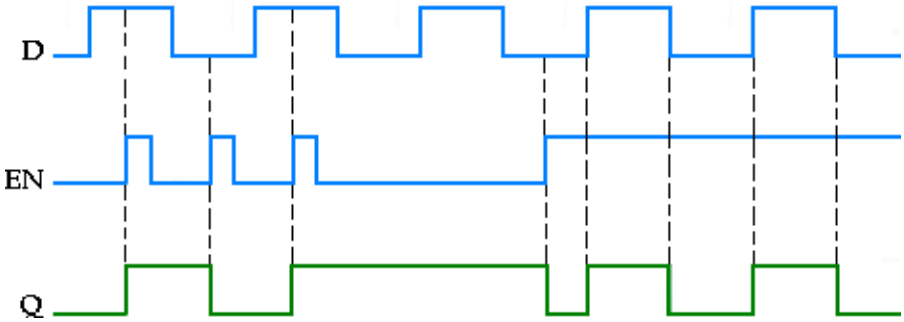


Fig. Logic Symbol

As shown in the figure, D input goes directly to the S input, and its complement is applied to the R input. Therefore, only two input conditions exists, either S=0 and R=1 or S=1 and R=0. The truth table for D latch is shown below.

EN	D	Q_n	Q_{n+1}	State
1	0	x	0	Reset
1	1	x	1	Set
0	x	x	Q_n	No Change (NC)

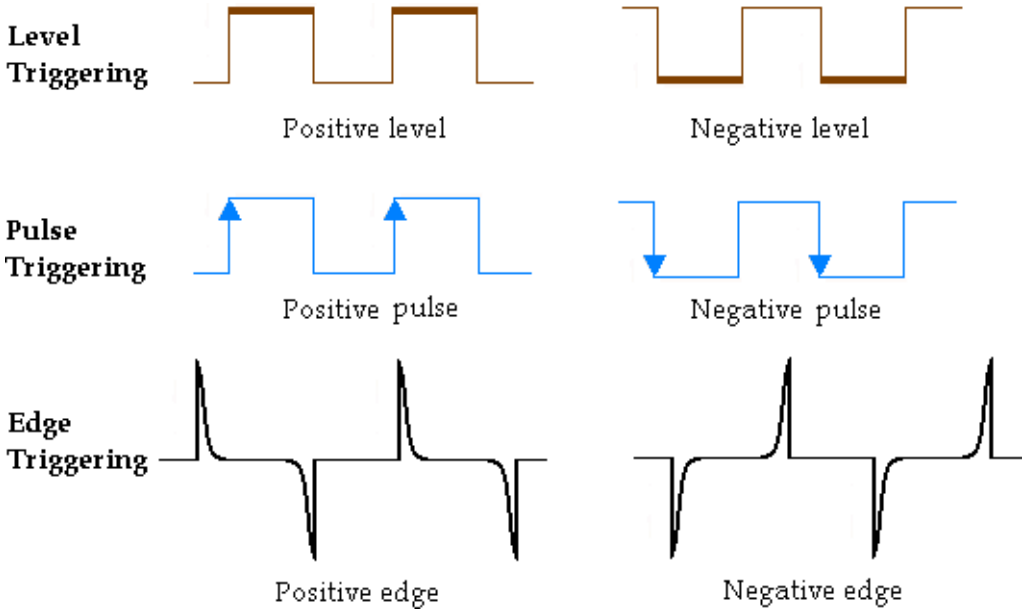
As shown in the truth table, the Q output follows the D input. For this reason, D latch is called transparent latch.



When D is HIGH and EN is HIGH. Q goes HIGH. When D is LOW and EN is HIGH, Q goes LOW. When EN is LOW, the state of the latch is not affected by the D input.

TRIGGERING OF FLIP-FLOPS

The state of a Flip-Flop is switched by a momentary change in the input signal. This momentary change is called a trigger and the transition it causes is said to trigger the Flip-Flop. Clocked Flip-Flops are triggered by pulses. A clock pulse starts from an initial value of 0, goes momentarily to 1and after a short time, returns to its initial 0 value. Latches are controlled by enable signal, and they are level triggered, either positive level triggered or negative level triggered. The output is free to change according to the S and R input values, when active level is maintained at the enable input. Flip-Flops are different from latches. Flip-Flops are pulse or clock edge triggered instead of level triggered.



EDGE TRIGGERED FLIP-FLOPS :

Flip-Flops are synchronous bistable devices (has two outputs Q and Q'). In this case, the term synchronous means that the output changes state only at a specified point on the triggering input called the clock (CLK), i.e., changes in the output occur in synchronization with the clock. An edge-triggered Flip-Flop changes state either at the positive edge (rising

edge) or at the negative edge (falling edge) of the clock pulse and is sensitive to its inputs only at this transition of the clock. The different types of edge-triggered Flip- Flops are

- S-R Flip-Flop,
- J-K Flip-Flop,
- D Flip-Flop,
- T Flip-Flop.

Although the S-R Flip-Flop is not available in IC form, it is the basis for the D and J-K Flip-Flops. Each type can be either positive edge-triggered (no bubble at C input) or negative edge-triggered (bubble at C input). The key to identifying an edge- triggered Flip-Flop by its logic symbol is the small triangle inside the block at the clock (C) input. This triangle is called the dynamic input indicator.

S-R Flip-Flop

The S and R inputs of the S-R Flip-Flop are called ***synchronous*** inputs because data on these inputs are transferred to the Flip-Flop's output only on the triggering edge of the clock pulse. The circuit is similar to SR latch except enable signal is replaced by clock pulse (CLK). On the positive edge of the clock pulse, the circuit responds to the S and R inputs.

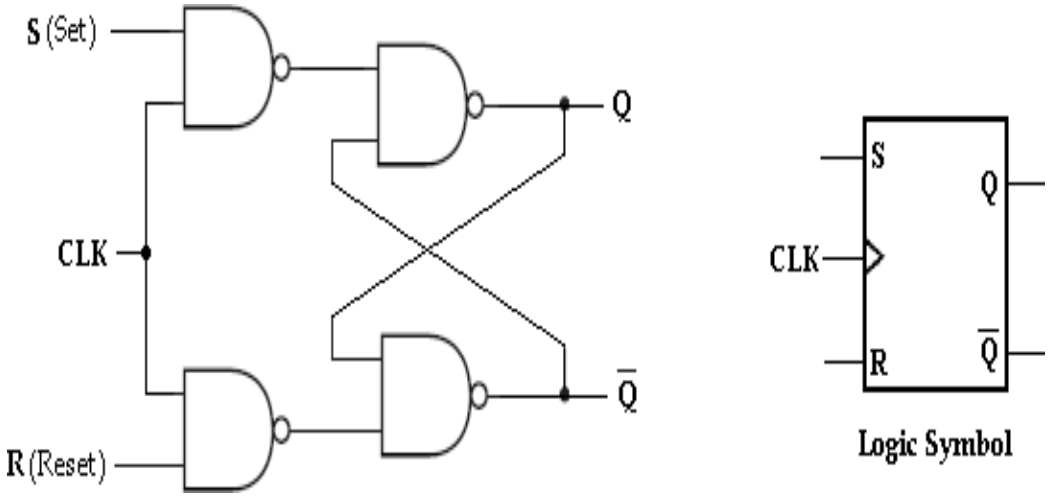
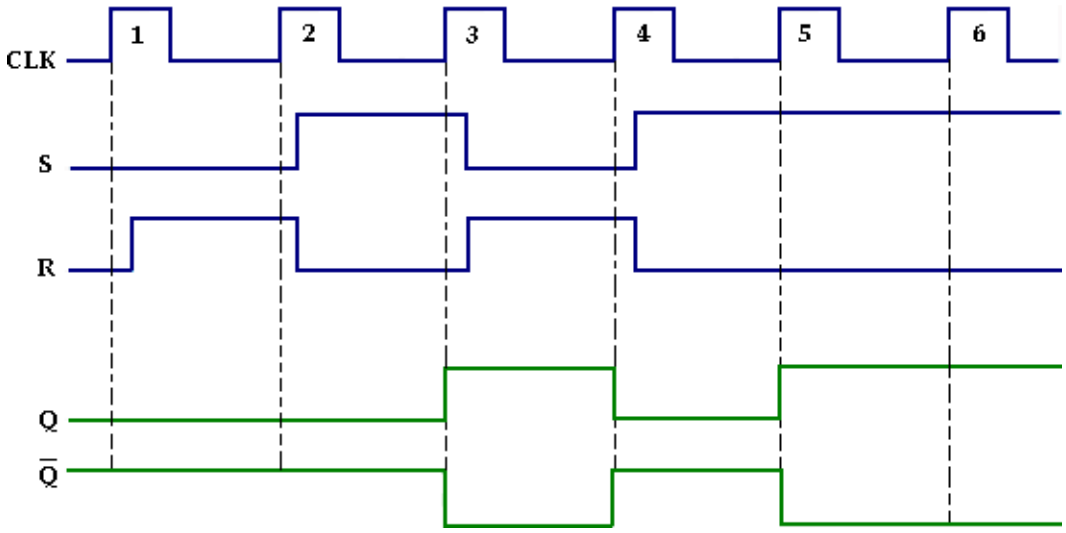


Fig. Logic diagram

When S is HIGH and R is LOW, the Q output goes HIGH on the triggering edge of the clock pulse, and the Flip-Flop is SET. When S is LOW and R is HIGH, the Q output goes LOW on the triggering edge of the clock pulse, and the Flip-Flop is RESET. When both S and R are LOW, the output does not change from its prior state. An invalid condition exists when both S and R are HIGH.

Truth table for SR Flip-Flop

CLK	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	
1	1	1	0	x	Indeterminate
1	1	1	1	x	*
0	x	x	0	0	No Change (NC)
0	x	x	1	1	



Input and output waveforms of SR Flip-Flop

J-K Flip-Flop:

JK means Jack Kilby, Texas Instrument (TI) Engineer, who invented IC in 1958. JK Flip-Flop has two inputs J(set) and K(reset). A JK Flip-Flop can be obtained from the clocked SR Flip-Flop by augmenting two AND gates as shown below.

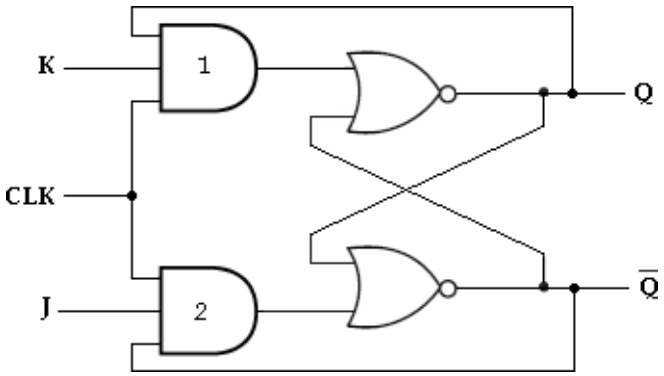
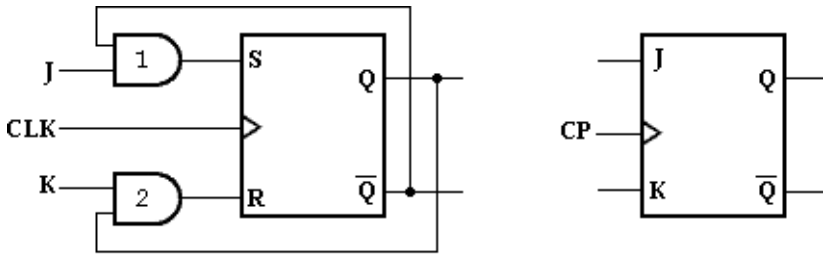


Fig. Logic Diagram

The data input J and the output Q' are applied o the first AND gate and its output (JQ') is applied to the S input of SR Flip-Flop. Similarly, the data input K and the output Q are applied to the second AND gate and its output (KQ) is applied to the R input of SR Flip-Flop.



(a) Using SR flipflop (b) Graphic symbol

J= K= 0, When J=K= 0, both AND gates are disabled. Therefore clock pulse have no effect, hence the Flip-Flop output is same as the previous output.

J= 0, K= 1, When J= 0 and K= 1, AND gate 1 is disabled i.e., S= 0 and R= 1. This condition will reset the Flip-Flop to 0.

J= 1, K= 0, When J= 1 and K= 0, AND gate 2 is disabled i.e., S= 1 and R= 0. Therefore the Flip-Flop will set on the application of a clock pulse.

J= K= 0, When J=K= 1, it is possible to set or reset the Flip-Flop. If Q is High, AND gate 2 passes on a reset pulse to the next clock. When Q is low, AND gate 1 passes on a set pulse to the next clock. Eitherway, Q changes to the complement of the last state i.e., toggle. Toggle means to switch to the opposite state.

The truth table of JK Flip-Flop is given below.

CLK	Inputs		Output	State
	J	K	Q_{n+1}	
1	0	0	Q_n	No Change
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	Q_n'	Toggle

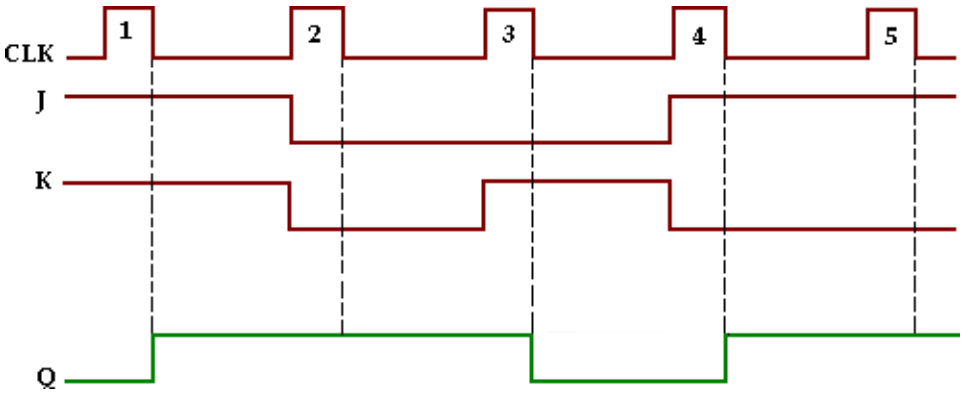


Fig. Input and output waveforms of JK Flip-Flop

Characteristic table and Characteristic equation:

The characteristic table for JK Flip-Flop is shown in the table below. From the table, K-map for the next state transition (Q_{n+1}) can be drawn and the simplified logic expression which represents the characteristic equation of JK Flip-Flop can be found.

Characteristic table

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

K-map Simplification:

Q_n \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

Characteristic equation: **Q_{n+1}= JQ'+ K'Q.**

D Flip-Flop:

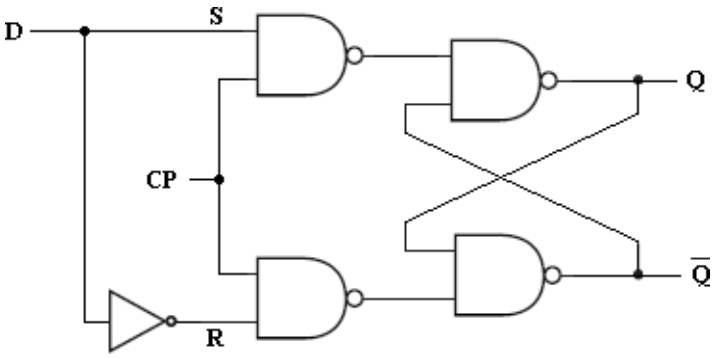
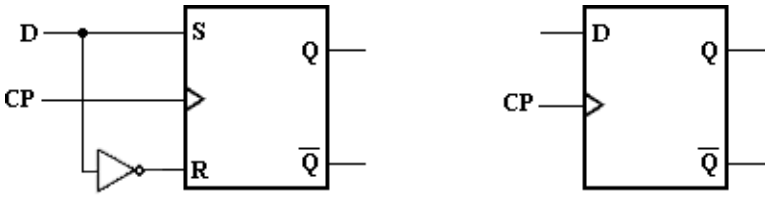


Fig. Logic Diagram

Like in D latch, in D Flip-Flop the basic SR Flip-Flop is used with complemented inputs. The D Flip-Flop is similar to D-latch except clock pulse is used instead of enable input. To eliminate the undesirable condition of the indeterminate state in the RS Flip-Flop is to ensure that inputs S and R are never equal to 1 at the same time. This is done by D Flip-Flop. The D (delay) Flip-Flop has one input called delay input and clock pulse input. The D Flip-Flop using SR Flip-Flop is shown below.



(a) Using SR flipflop (b) Graphic symbol

The truth table of D Flip-Flop is given below.

Clock	D	Q_{n+1}	State
1	0	0	Reset
1	1	1	Set
0	x	Q_n	No Change

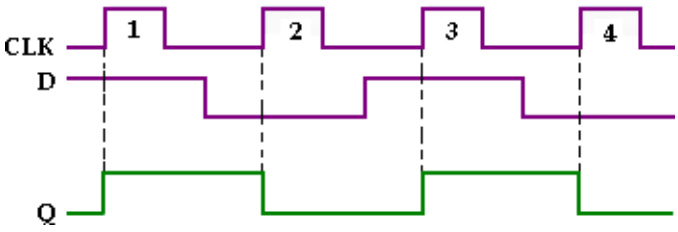


Fig. Input and output waveforms of clocked D Flip-Flop

Looking at the truth table for D Flip-Flop we can realize that Q_{n+1} function follows the D input at the positive going edges of the clock pulses.

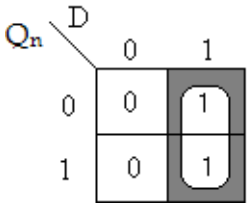
Characteristic table and Characteristic equation:

The characteristic table for D Flip-Flop shows that the next state of the Flip- Flop is independent of the present state since Q_{n+1} is equal to D. This means that an input pulse will transfer the value of input D into the output of the Flip-Flop independent of the value of the output before the pulse was applied.

The characteristic equation is derived from K-map.

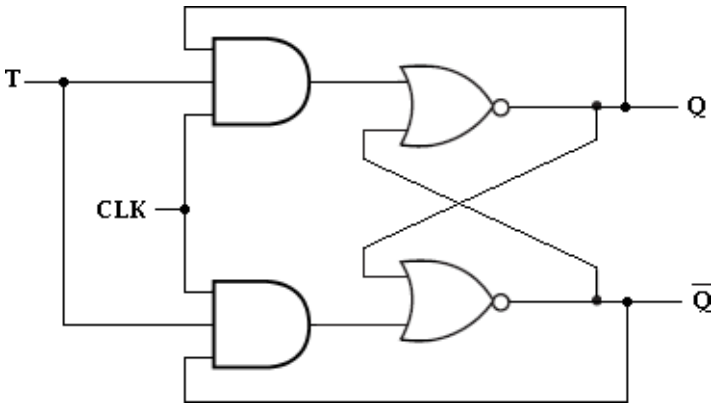
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

K-map simplification



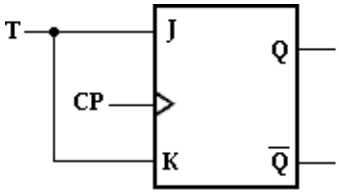
Characteristic equation: $\mathbf{Q_{n+1}= D.}$

T Flip-Flop

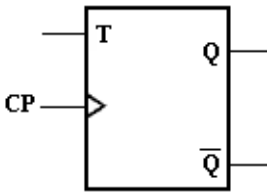


The T (***Toggle***) Flip-Flop is a modification of the JK Flip-Flop. It is obtained from JK Flip-Flop by connecting both inputs J and K together, i.e., single input. Regardless of the present state, the Flip-Flop complements its output when the clock pulse occurs while input T= 1.

When T= 0, $Q_{n+1}= Q_n$, ie., the next state is the sameas the present state and no change occurs.
When T= 1, $Q_{n+1}= Q_n$',ie., the next state is the complement of the present state.



(a) Using JK flipflop



(b) Graphic symbol

The truth table of T Flip-Flop is given below.

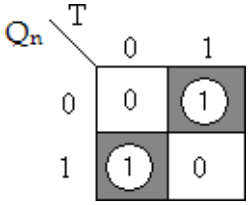
T	Q_{n+1}	State
0	Q_n	No Change
1	Q_n'	Toggle

Characteristic table and Characteristic equation:

The characteristic table for T Flip-Flop is shown below and characteristic equation is derived using K-map.

Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

K-map Simplification:



Characteristic equation: $\mathbf{Q_{n+1} = TQ_n' + T'Qn.}$

Master-Slave JK Flip-Flop

A master-slave Flip-Flop is constructed using two separate JK Flip-Flops. The first Flip-Flop is called the master. It is driven by the positive edge of the clock pulse. The second Flip-Flop is called the slave. It is driven by the negative edge of the clock pulse. The logic diagram of a master-slave JK Flip-Flop is shown below.

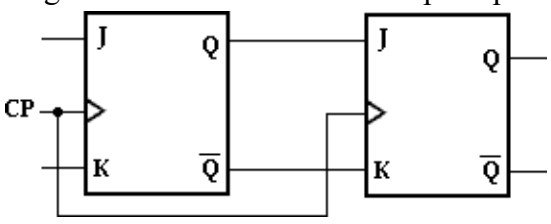


Fig. Logic diagram

When the clock pulse has a positive edge, the master acts according to its J- K inputs, but the slave does not respond, since it requires a negative edge at the clock input. When the clock input has a negative edge, the slave Flip-Flop copies the master outputs. But the master does not respond since it requires a positive edge at its clock input.

The clocked master-slave J-K Flip-Flop using NAND gates is shown below.

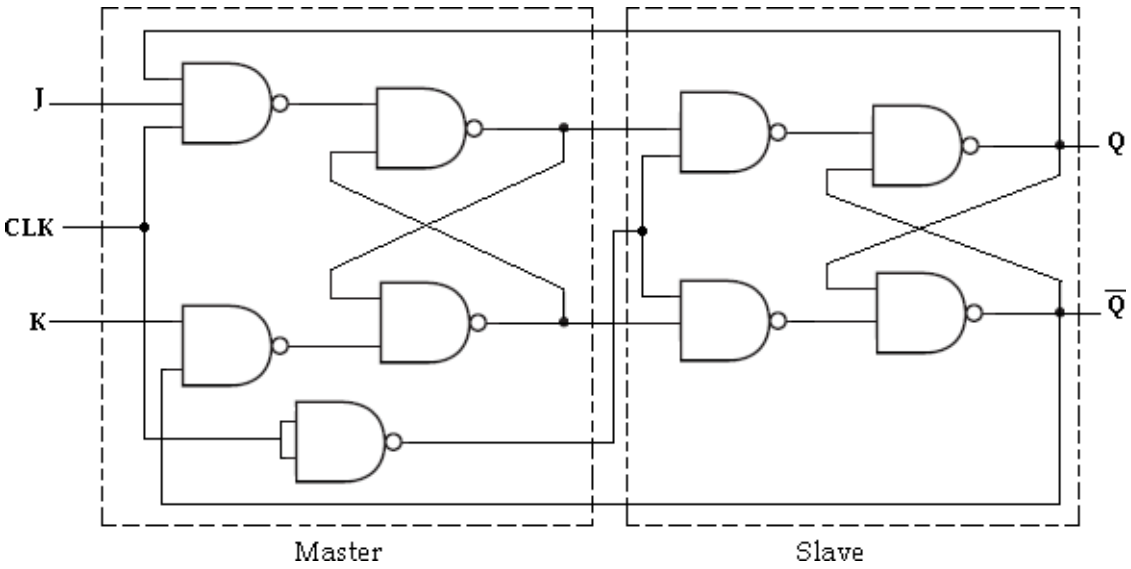


Fig. Circuit diagram of Master-Slave JK Flip-Flop

APPLICATION TABLE (OR) EXCITATION TABLE:

The characteristic table is useful for analysis and for defining the operation of the Flip-Flop. It specifies the next state (Q_{n+1}) when the inputs and present state are known. The excitation or application table is useful for design process. It is used to find the Flip-Flop input conditions that will cause the required transition, when the present state (Q_n) and the next state (Q_{n+1}) are known.

Characteristic Table

Present State	Inputs		Next State
Q_n	S	R	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	X
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	X

Excitation Table

Present State	Next State	Inputs		Inputs	
Q_n	Q_{n+1}	S	R	S	R
0	0	0	0	0	x
0	0	0	1		
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	0	x	0
1	1	1	0		

Modified Table

Present State	Next State	Inputs	
Q_n	Q_{n+1}	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

The above table presents the excitation table for SR Flip-Flop. It consists of present state (Q_n), next state (Q_{n+1}) and a column for each input to show how the required transition is achieved.

There are 4 possible transitions from present state to next state. The required Input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol ‘x’ denotes the don’t care condition, it does not matter whether the input is 0 or 1.

Characteristic Table

Present State	Inputs		Next State
Q_n	J	K	Q_{n+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Modified Table

Present State	Next State	Inputs		Inputs	
Q_n	Q_{n+1}	J	K	J	K
0	0	0	0	0	X
0	0	0	1		
0	1	1	0	1	X
0	1	1	1		
1	0	0	1	x	1
1	0	1	1		
1	1	0	0	x	0
1	1	1	0		

Excitation Table

Present State	Next State	Inputs	
Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	x	1
1	1	x	0

Characteristic Table

Present State	Input	Next State
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

Excitation Table

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

T Flip-Flop

Characteristic Table

Present State	Input	Next State
Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Modified Table

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

REALIZATION OF ONE FLIP-FLOP USING OTHER FLIP-FLOPS

It is possible to convert one Flip-Flop into another Flip-Flop with some additional gates or simply doing some extra connection. The realization of one Flip- Flop using other Flip-Flops is implemented by the use of characteristic tables and excitation tables. Let us see few conversions among Flip-Flops.

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

SR Flip-Flop to D Flip-Flop:

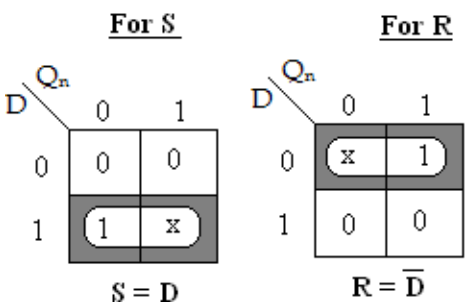
- Write the characteristic table for required Flip-Flop (D Flip-Flop).
- Write the excitation table for given Flip-Flop (SR Flip-Flop).
- Determine the expression for the given Flip-Flop inputs (S and R) by using K- map.
- Draw the Flip-Flop conversion logic diagram to obtain the required Flip- Flop (D Flip-Flop) by using the above obtained expression.

The excitation table for the above conversion is

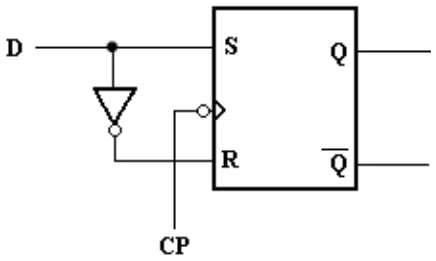
Required Flip-Flop (D)			Given Flip-Flop (SR)	
Input	Present state	Next state	Flip-Flop Inputs	
D	Q_n	Q_{n+1}	S	R
0	0	0	0	x
0	1	0	0	1

1	0	1	1	0
1	1	1	x	0

K-map simplification



Logic diagram

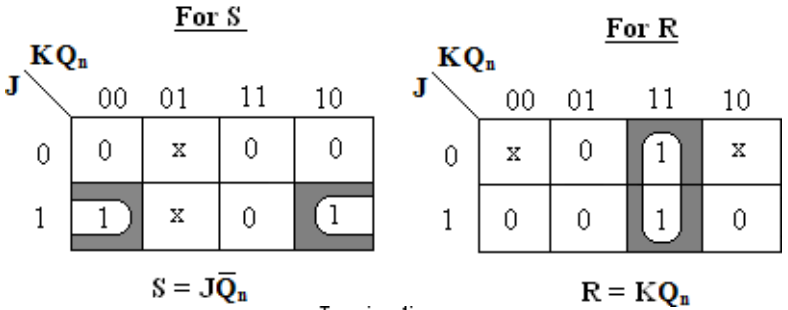


SR Flip-Flop to JK Flip-Flop

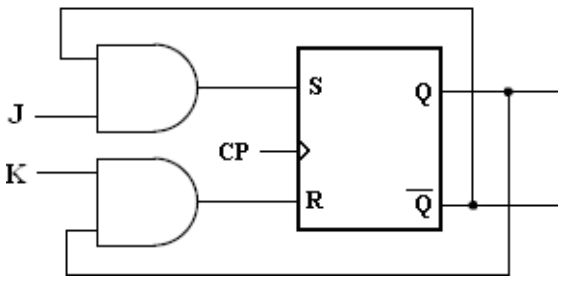
The excitation table for the above conversion is,

Inputs		Present state	Next state	Flip-Flop Input	
J	**K**	**Q_n**	**Q_{n+1}**	**S**	**R**
0	0	0	0	0	x
0	0	1	1	x	0
0	1	0	0	0	x
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	x	0
1	1	0	1	1	0
1	1	1	0	0	1

K-map simplification



Logic diagram



SR Flip-Flop to T Flip-Flop

The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**S**	**R**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	0
1	1	0	0	1

K-map simplification

For S

T \ Q_n	0	1
0	0	x
1	1	0

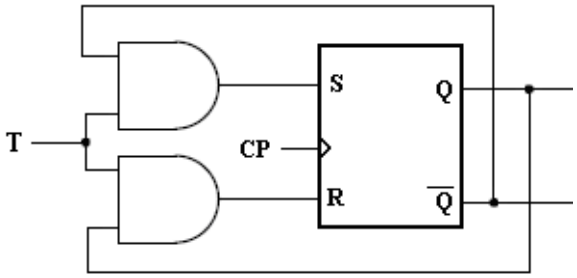
$S = T\overline{Q}_n$

For R

T \ Q_n	0	1
0	x	0
1	0	1

$R = TQ_n$

Logic diagram



JK Flip-Flop to T Flip-Flop

The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**J**	**K**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

K-map simplification

For J

T \ Q_n	0	1
0	0	x
1	1	x

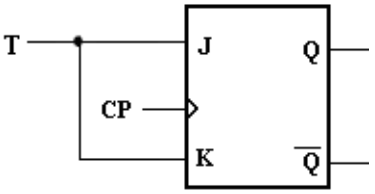
$J = T$

For K

T \ Q_n	0	1
0	x	0
1	x	1

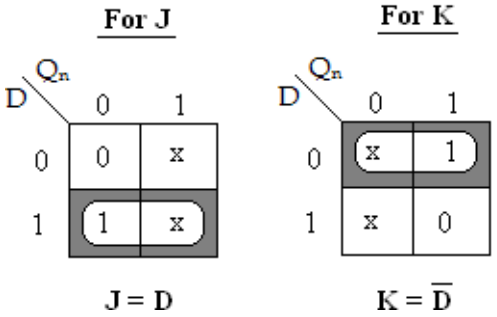
$K = T$

Logic diagram

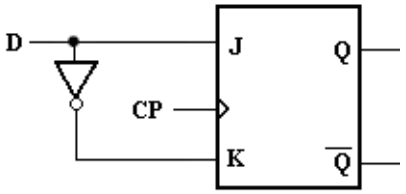


Input	Present state	Next state	Flip-Flop Inputs	
D	**Q_n**	**Q_{n+1}**	**J**	**K**
0	0	0	0	x
0	1	0	x	1
1	0	1	1	X
1	1	1	x	0

K-map simplification



Logic diagram

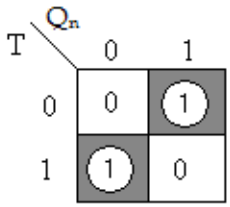


D Flip-Flop to T Flip-Flop

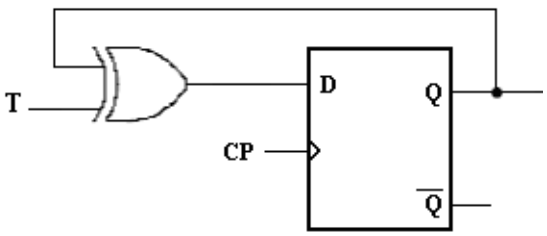
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Input
T	**Q_n**	**Q_{n+1}**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

K-map simplification

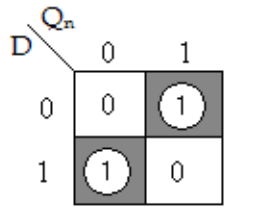


Logic diagram



Input	Present state	Next state	Flip-Flop Input
D	**Q_n**	**Q_{n+1}**	**T**
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

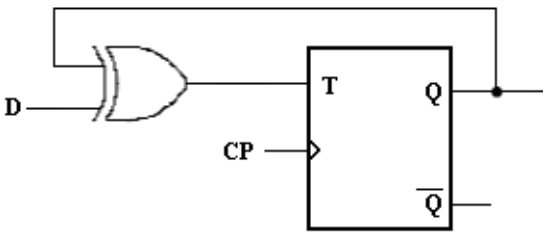
K-map simplification



$T = D\overline{Q_n} + \overline{D}Q_n$

$= D \oplus Q_n$

Logic diagram



SHIFT REGISTERS:

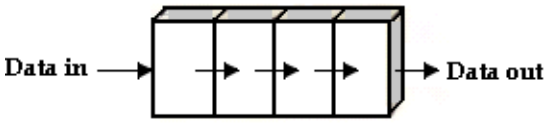
A register is simply a group of Flip-Flops that can be used to store a binary number. There must be one Flip-Flop for each bit in the binary number. For instance, a register used to store an 8-bit binary number must have 8 Flip-Flops.

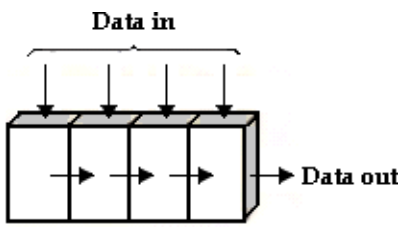
The Flip-Flops must be connected such that the binary number can be entered (shifted) into the register and possibly shifted out. A group of Flip-Flops connected to provide either or both of these functions is called a shift register.

The bits in a binary number (data) can be removed from one place to another in either of two ways. The first method involves shifting the data one bit at a time in a serial fashion, beginning with either the most significant bit (MSB) or the least significant bit (LSB). This technique is referred to as serial shifting. The second method involves shifting all the data bits simultaneously and is referred to as parallel shifting.

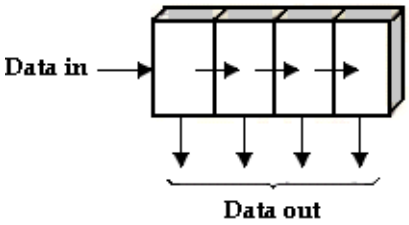
There are two ways to shift into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic register type

- Serial in- serial out,
- Serial in- parallel out,
- Parallel in- serial out,
- Parallel in- parallel out.

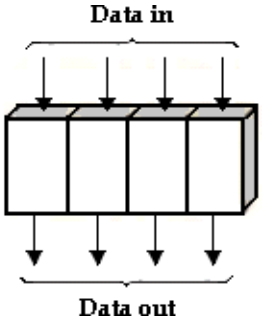




Serial in- serial out



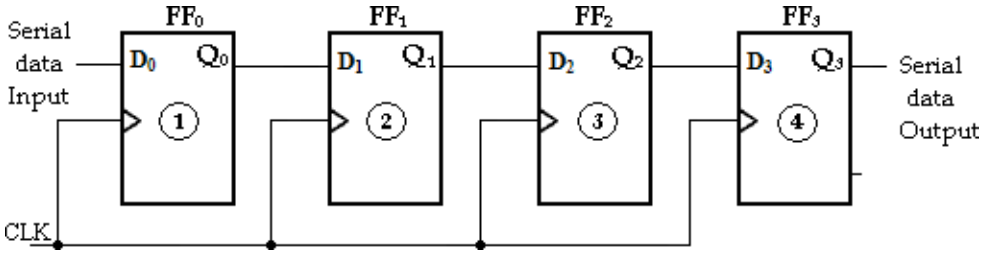
Parallel in- serial out



Serial in- parallel out

Parallel in- parallel out

Serial-In Serial-Out Shift Register:



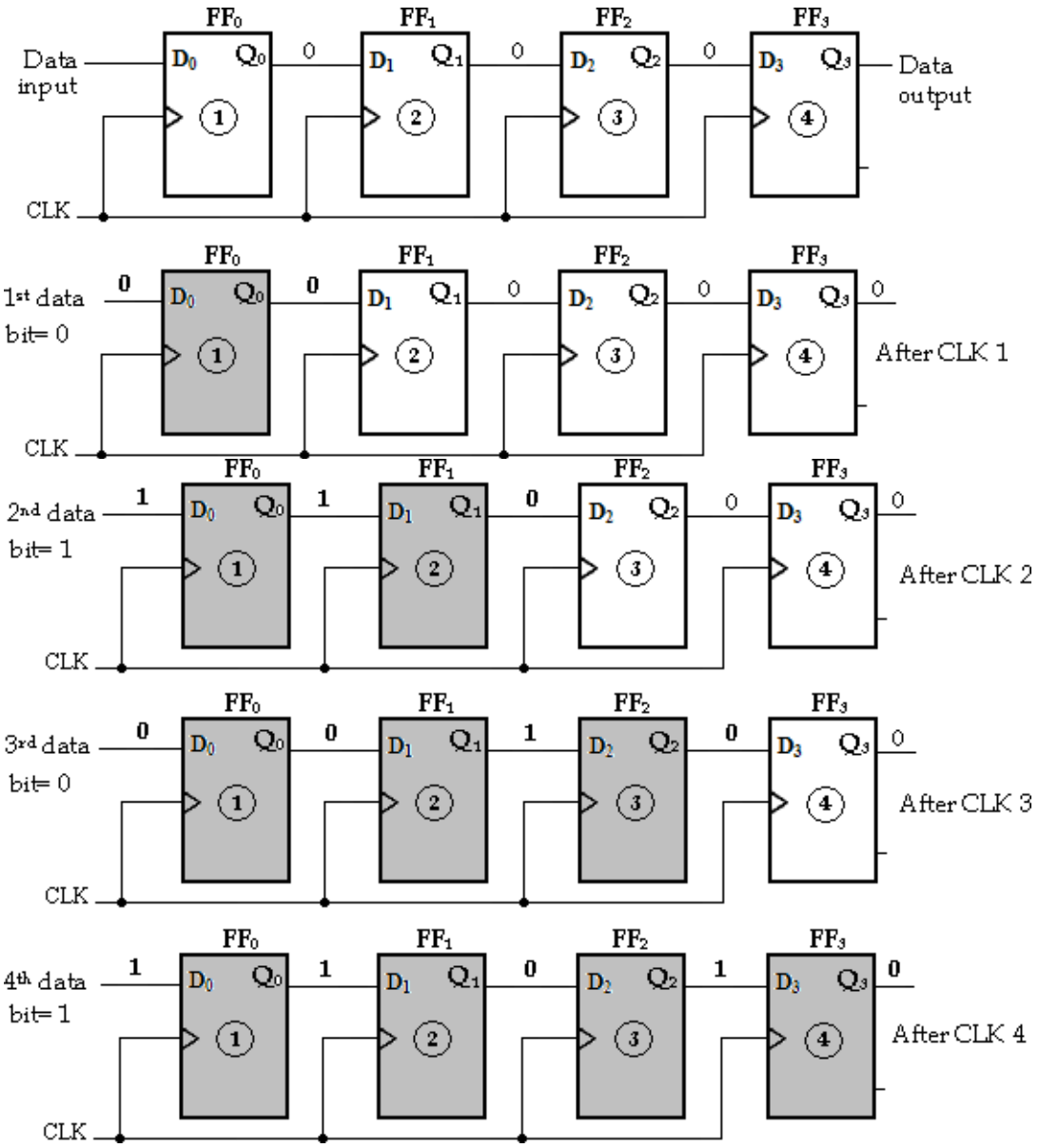
The serial in/serial out shift register accepts data serially, i.e., one bit at a time on a single line. It produces the stored information on its output also in serial form.

The entry of the four bits 1010 into the register is illustrated below, beginning with the right-most bit. The register is initially clear. The 0 is put onto the data input line, making D=0 for FF_0. When the first clock pulse is applied, FF_0 is reset, thus storing the 0.

Next the second bit, which is a 1, is applied to the data input, making D=1 for FF_0 and D=0 for FF_1 because the D input of FF_1 is connected to the Q_0 output. When the second clock pulse occurs, the 1 on the data input is shifted into FF0, causing FF0 to set; and the 0 that was in FF0 is shifted into FFl.

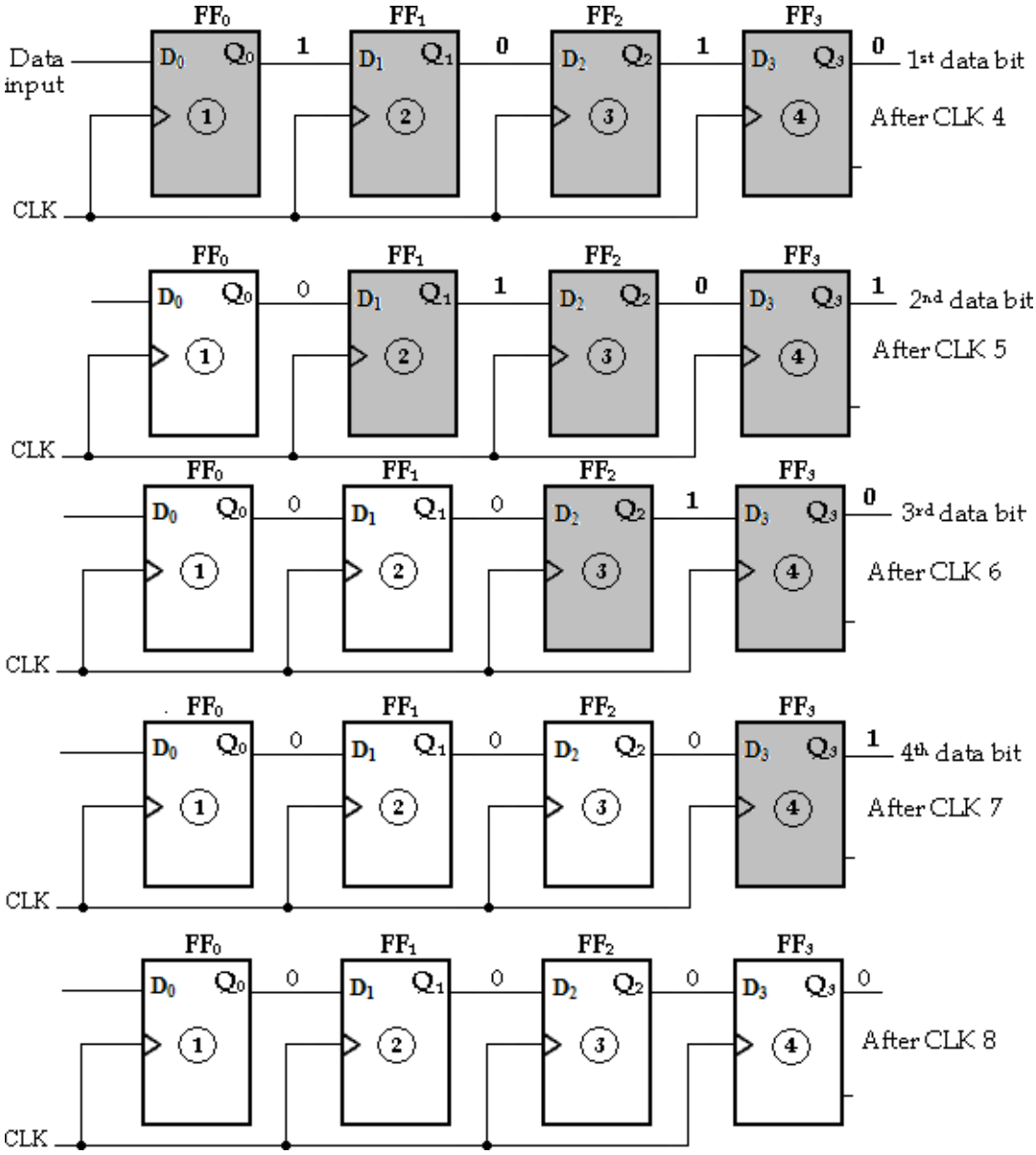
The third bit, a 0, is now put onto the data-input line, and a clock pulse is applied. The 0 is entered into FF_0, the 1 stored in FF_0 is shifted into FF_l, and the 0 stored in FF_1 is shifted into FF_2.

The last bit, a 1, is now applied to the data input, and a clock pulse is applied. This time the 1 is entered into FF0, the 0 stored in FF0 is shifted into FFl, the 1 stored in FF1 is shifted into FF2, and the 0 stored in FF2 is shifted into FF3. This completes the serial entry of the four bits into the shift register, where they can be stored for any length of time as long as the Flip-Flops have dc power.



Four bits (1010) being entered serially into the register

To get the data out of the register, the bits must be shifted out serially and taken off the Q3 output. After CLK4, the right-most bit, 0, appears on the Q3 output.

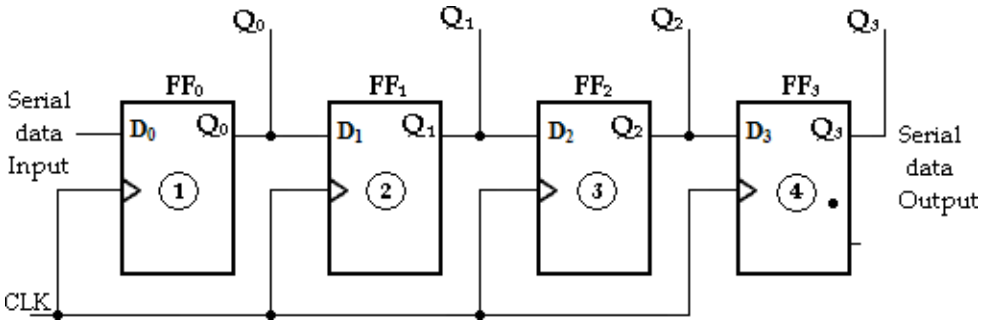


When clock pulse CLK5 is applied, the second bit appears on the Q3 output. Clock pulse CLK6 shifts the third bit to the output, and CLK7 shifts the fourth bit to the output. While the original four bits are being shifted out, more bits can be shifted in. All zeros are shown being shifted out, more bits can be shifted in.

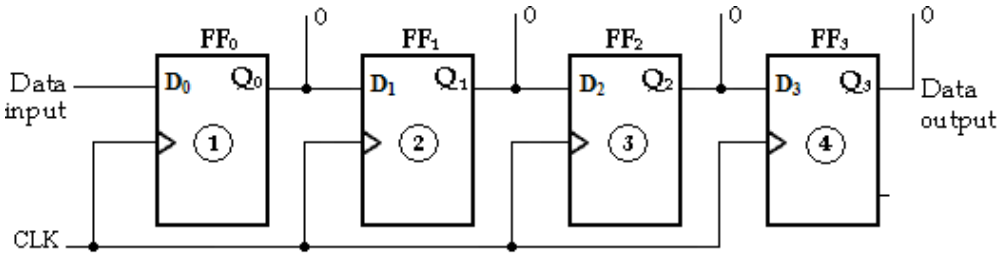
Four bits (1010) being entered serially-shifted out of the register and replaced by all zeros

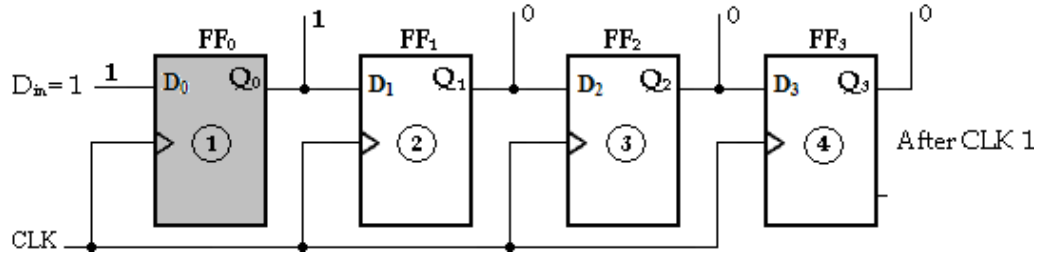
Serial-In Parallel-Out Shift Register:

In this shift register, data bits are entered into the register in the same as serial-in serial-out shift register. But the output is taken in parallel. Once the data are stored, each bit appears on its respective output line and all bits are available simultaneously instead of on a bit-by-bit.



Serial-In parallel-Out Shift Register





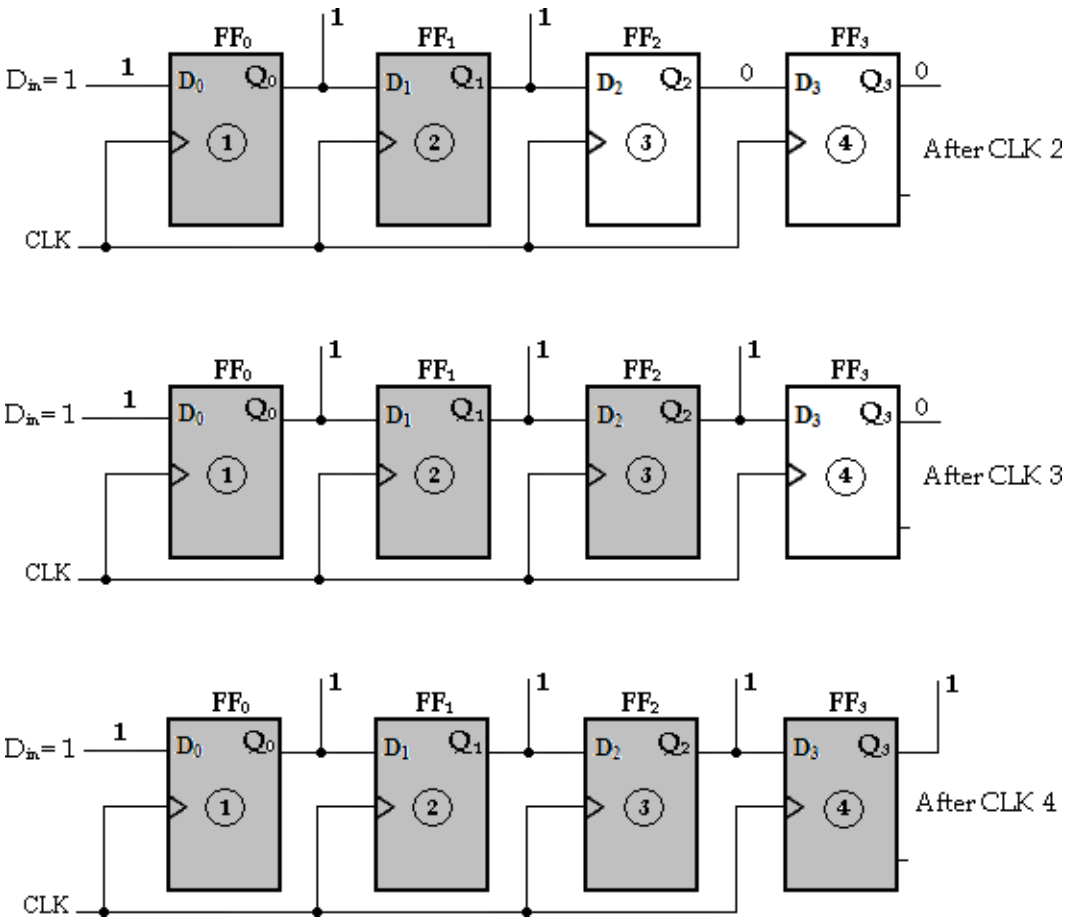


Fig. Four bits (1111) being serially entered into the register

Parallel-In Serial-Out Shift Register:

In this type, the bits are entered in parallel i.e., simultaneously into their respective stages on parallel lines. A 4-bit parallel-in serial-out shift register is illustrated below. There are four data input lines, X_0, X_1, X_2 and X_3 for entering data in parallel into the register. SHIFT/ LOAD input is the control input, which allows four bits of data to **load** in parallel into the register. When SHIFT/LOAD is LOW, gates G_1, G_2, G_3 and G_4 are enabled, allowing each data bit to be applied to the D input of its respective Flip-Flop. When a clock pulse is applied, the Flip-Flops with D = 1 will **set** and those with D = 0 will **reset,** thereby storing all four bits simultaneously.

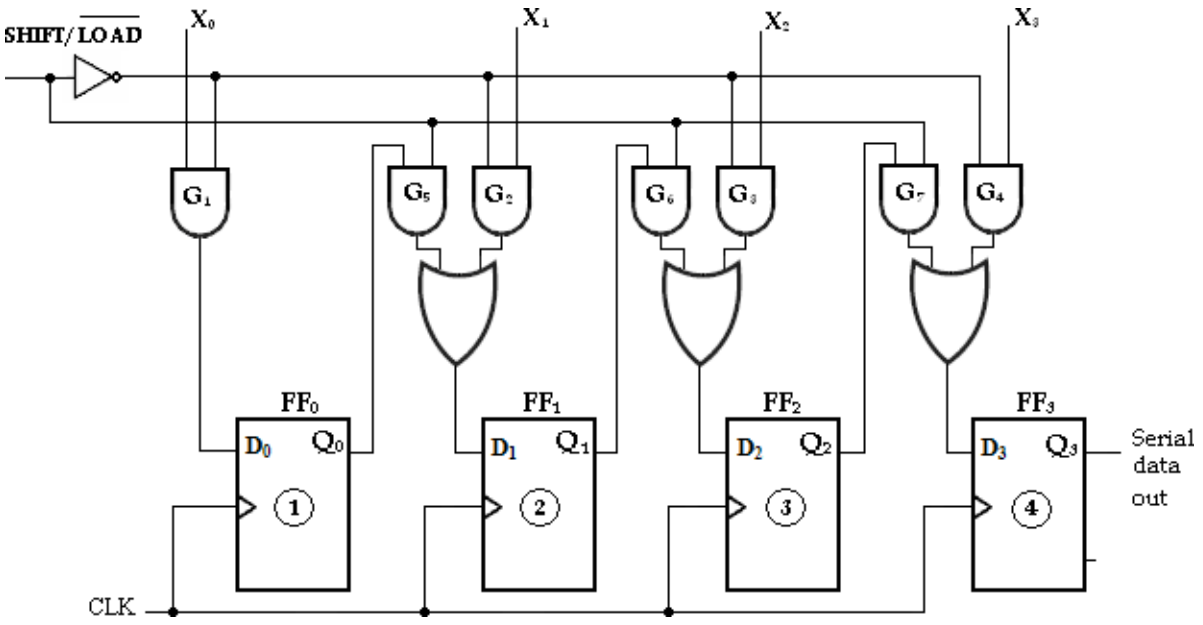


Fig. Parallel-In Serial-Out Shift Register

When SHIFT/LOAD is HIGH, gates G_1, G_2, G_3 and G_4 are disabled and gates G_5, G_6 and G_7 are enabled, allowing the data bits to shift right from one stage to the next. The OR gates allow either the normal shifting operation or the parallel data- entry operation, depending on which AND gates are enabled by the level on the SHIFT/LOAD input.

Parallel-In Parallel-Out Shift Register:

In this type, there is simultaneous entry of all data bits and the bits appear on parallel outputs simultaneously.

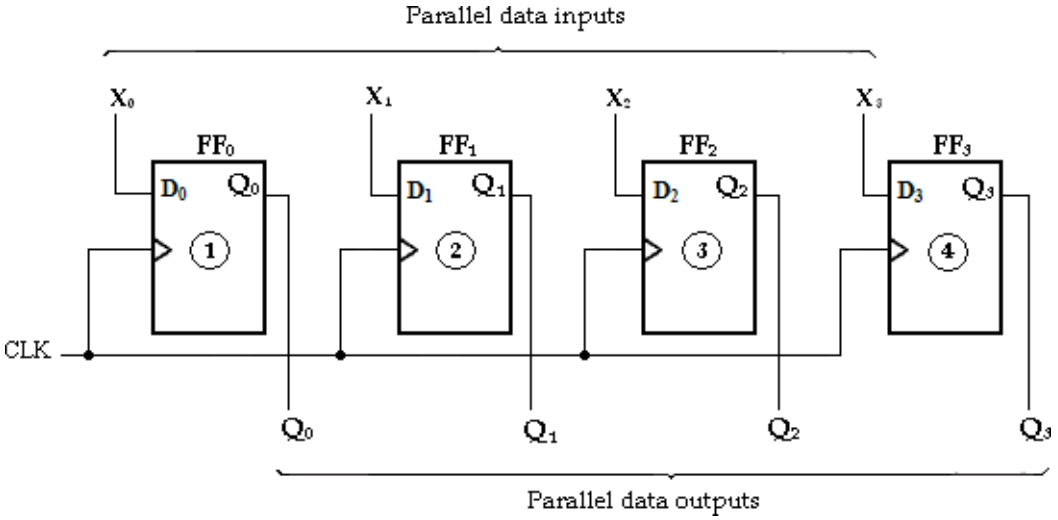


Fig. Parallel-In Parallel-Out Shift Register

UNIVERSAL SHIFT REGISTERS

If the register has shift and parallel load capabilities, then it is called a shift register with parallel **load or universal shift register. Shift** register can be used for converting serial data to parallel data, and vice-versa. If a parallel load capability is added to a shift register, the data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

The functions of universal shift register are:

- A clear control to clear the register to 0.
- A clock input to synchronize the operations.
- A shift-right control to enable the shift right operation and the serial input and output lines associated with the shift right.
- A shift-left control to enable the shift left operation and the serial input and output lines associated with the shift left.
- A parallel-load control to enable a parallel transfer and the n input lines associated with the parallel transfer.
- 'n' parallel output lines.
- A control line that leaves the information in the register unchanged even though the clock pulses re continuously applied.

It consists of four D-Flip-Flops and four 4 input multiplexers (MUX). S_0 and S_1 are the two selection inputs connected to all the four multiplexers. These two selection inputs are used to select one of the four inputs of each multiplexer.

The input 0 in each MUX is selected when S_1S_0= 00 and input 1 is selected when S_1S_0= 01. Similarly inputs 2 and 3 are selected when S_1S_0= 10 and S_1S_0= 11 respectively. The inputs S1 and S0 control the mode of the operation of the register.

When S_1S_0= 00, the present value of the register is applied to the D-inputs of the Flip-Flops. This is done by connecting the output of each Flip-Flop to the 0 input of the respective multiplexer. The next clock pulse transfers into each Flip-Flop, the binary value is held previously, and hence no change of state occurs.

When S_1S_0= 01, terminal 1 of the multiplexer inputs has a path to the D inputs of the Flip-Flops. This causes a shift-right operation with the lefter serial input transferred into Flip-Flop FF_3.

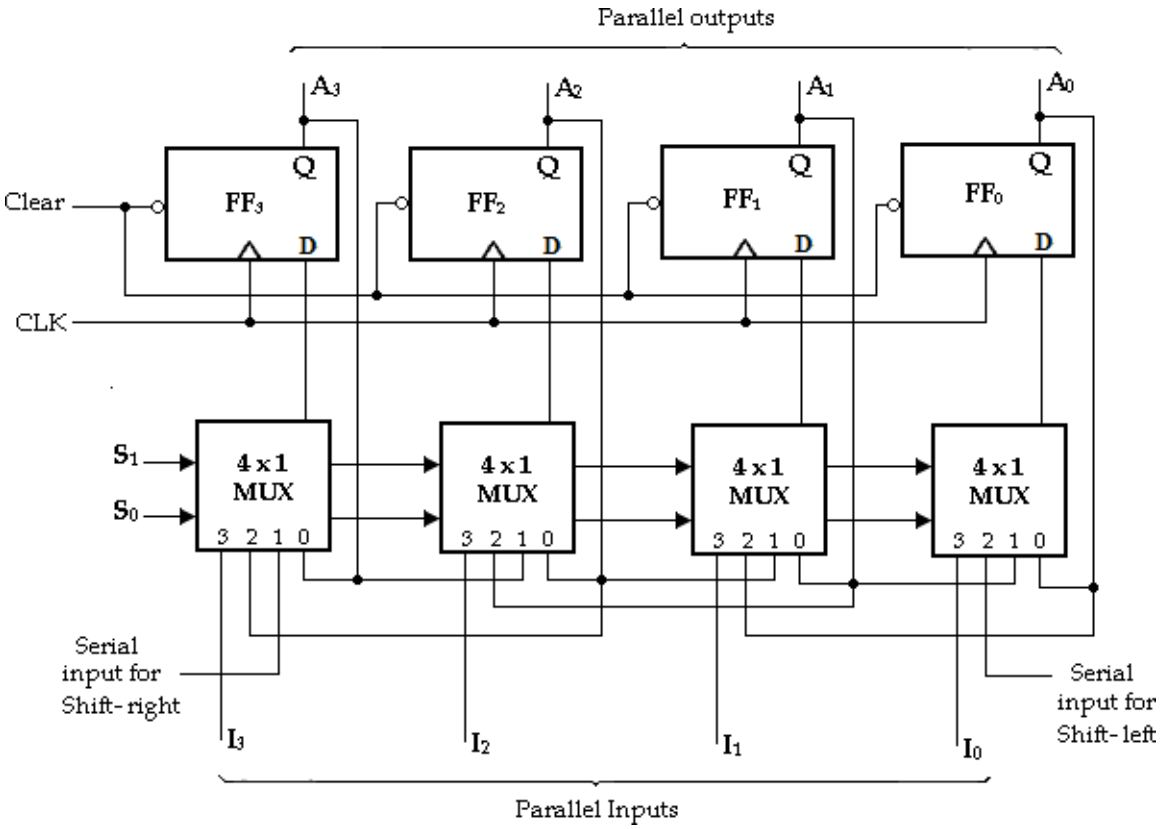


Fig. Block 4-Bit Universal Shift Register

When S_1S_0= 10, a shift-left operation results with the right serial input going into Flip-Flop FF_1. Finally when S_1S_0= 11, the binary information on the parallel input lines (I_1, I_2, I_3 and I_4) are transferred into the register simultaneously during the next clock pulse.

The function table of bi-directional shift register with parallel inputs and parallel outputs is shown below.

Mode Control		**Operation**
S_1	**S_0**	
0	0	No change
0	1	Shift-right
1	0	Shift-left
1	1	Parallel load

BI-DIRECTION SHIFT REGISTERS:

A bidirectional shift register is one in which the data can be shifted either left or right. It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left depending on the level of a control line.
A 4-bit bidirectional shift register is shown below. A HIGH on the RIGHT/LEFT control input allows data bits inside the register to be shifted to the right, and a LOW enables data bits inside the register to be shifted to the left.

When the RIGHT/LEFT control input is **HIGH**, gates G_1, G_2, G_3 and G_4 are enabled, and the state of the Q output of each Flip-Flop is passed through to the D input of the following Flip-Flop. When a clock pulse occurs, the data bits are shifted one place to the right.

When the RIGHT/LEFT control input is **LOW**, gates G_5, G_6, G_7 and G_8 are enabled, and the Q output of each Flip-Flop is passed through to the D input of the preceding Flip-Flop. When a clock pulse occurs, the data bits are then shifted one place to the left.

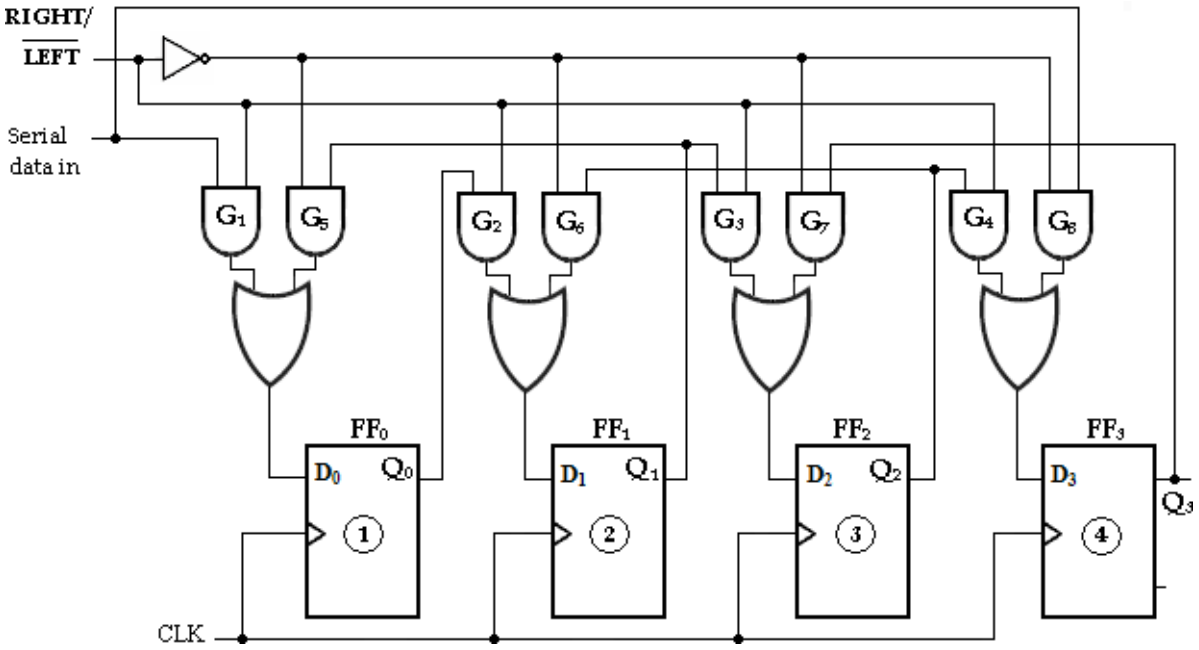


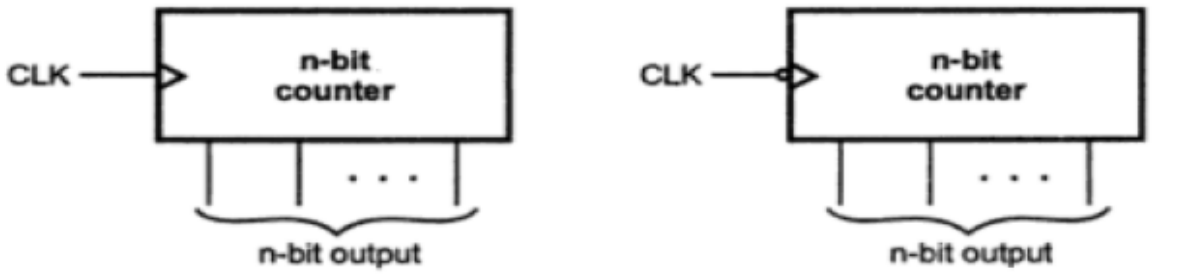
Fig. 4-bit bi-directional shift register

Counters

A counter is a register capable of counting the number of clock pulses arriving at its clock input. Count represents the number of clock pulses arrived. On arrival of each clock pulse,

- In case of Up counter, the counter is incremented by one
- In case of down counter, it is decremented by one

The Fig. shows the logic symbol of a binary counter. External clock is applied to the clock input of the counter. The counter can be positive edge triggered or negative edge triggered. The n-bit binary counter has n flip-flops, and it has 2^n distinct states of outputs. For example, 2-bit counter has 2 flip-flops and it has $4(2^2)$ distinct states : 00, 01, 10 and 11. Similarly, the 3-bit binary counter has 3 flip-flops and it has $8(2^3)$ distinct states : 000, 001, 010, 011, 100, 101 110 and 111.



(a) Positive edge triggered n-bit counter

(b) Negative edge triggered n-bit counter

Fig. Logic symbol of counter

Types of Counters

1) Asynchronous or ripplecounters 2) Synchronous counters

Asynchronous or ripplecounters

A binary ripple / asynchronous counter consists of a series connection of complementing flip –flops, with the output of each flip – flop connected to the clock input of the next higher order flip- flop. The flip-flop holding the least significant bit receives the incoming clock pulses.

Synchronous counters

When counter is clocked such that each flip –flop in the counter is triggered at the same time, the counter is called as synchronous counter.

S.No	Asynchronous (ripple) counter	Synchronous counter
1	All the Flip-Flops are not clocked simultaneously.	All the Flip-Flops are clocked simultaneously.
2	The delay times of all Flip- Flops are added. Therefore there is considerable propagation delay.	There is minimum propagation delay.
3	Speed of operation is low	Speed of operation is high.
4	Logic circuit is very simple even for more number of states.	Design involves complex logic circuit as number of state increases.
5	Minimum numbers of logic devices are needed.	The number of logic devices is more than ripple counters.
6	Cheaper than synchronous counters.	Costlier than ripple counters.

SYNCHRONOUS COUNTERS

Flip-Flops can be connected together to perform counting operations. Such a group of Flip- Flops is a counter. The number of Flip-Flops used and the way in which they are connected determine the number of states (called the modulus) and also the specific sequence of states that the counter goes through during each complete cycle.

Counters are classified into two broad categories according to the way they are clocked: Asynchronous counters, Synchronous counters.

In asynchronous (ripple) counters, the first Flip-Flop is clocked by the external clock pulse and then each successive Flip-Flop is clocked by the output of the preceding Flip-Flop.

In synchronous counters, the clock input is connected to all of the Flip-Flops so that they are clocked simultaneously. Within each of these two categories, counters are classified primarily by the type of sequence, the number of states, or the number of Flip-Flops in the counter.

The term 'synchronous' refers to events that have a fixed time relationship with each other. In synchronous counter, the clock pulses are applied to all Flip- Flops simultaneously. Hence there is minimum propagation delay.

2-Bit Synchronous Binary Counter

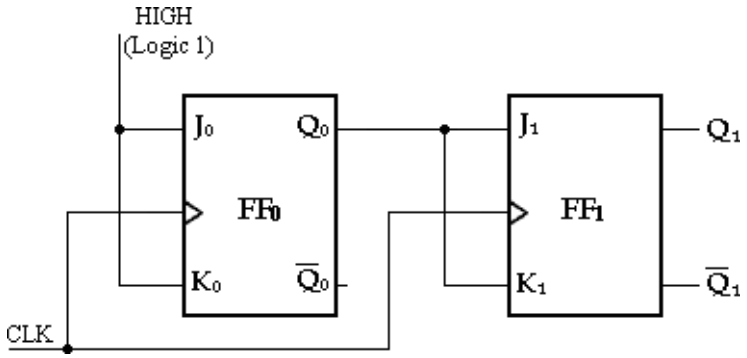


Fig. Logic diagram of 2-Bit Synchronous Binary Counter

In this counter the clock signal is connected in parallel to clock inputs of both the Flip-Flops (FF_0 and FF_1). The output of FF_0 is connected to J_1 and K_1 inputs of the second Flip-Flop (FF_1).

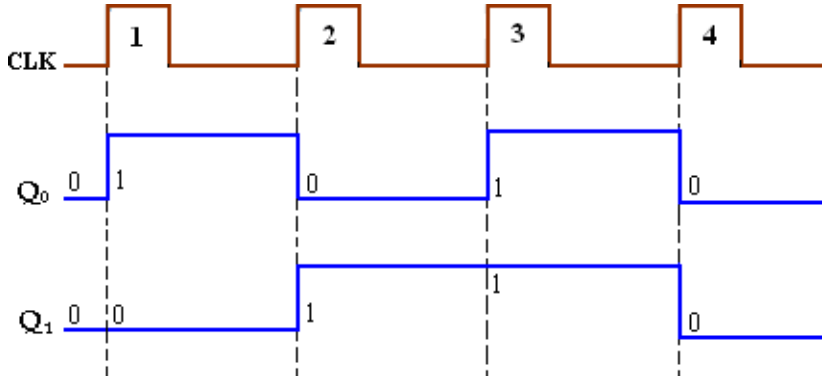
Assume that the counter is initially in the binary 0 state: i.e., both Flip-Flops are RESET. When the positive edge of the first clock pulse is applied, FF_0 will toggle because $J_0 = k_0 = 1$, whereas FF_1 output will remain 0 because $J_1 = k_1 = 0$. After the first clock pulse $Q_0 = 1$ and $Q_1 = 0$.

When the leading edge of CLK2 occurs, FF_0 will toggle and Q_0 will go LOW. Since FF_1 has a HIGH ($Q_0 = 1$) on its J_1 and K_1 inputs at the triggering edge of this clock pulse, the Flip-Flop toggles and Q_1 goes HIGH. Thus, after CLK2,

$Q_0 = 0$ and $Q_1 = 1$.

When the leading edge of CLK3 occurs, FF_0 again toggles to the SET state ($Q_0 = 1$), and FF_1 remains SET ($Q_1 = 1$) because its J_1 and K_1 inputs are both LOW ($Q_0 = 0$). After this triggering edge, $Q_0 = 1$ and $Q_1 = 1$.

Finally, at the leading edge of CLK4, Q_0 and Q_1 go LOW because they both have a toggle condition on their J_1 and K_1 inputs. The counter has now recycled to its original state, $Q_0 = Q_1 = 0$.



Timing diagram

3-Bit Synchronous Binary Counter

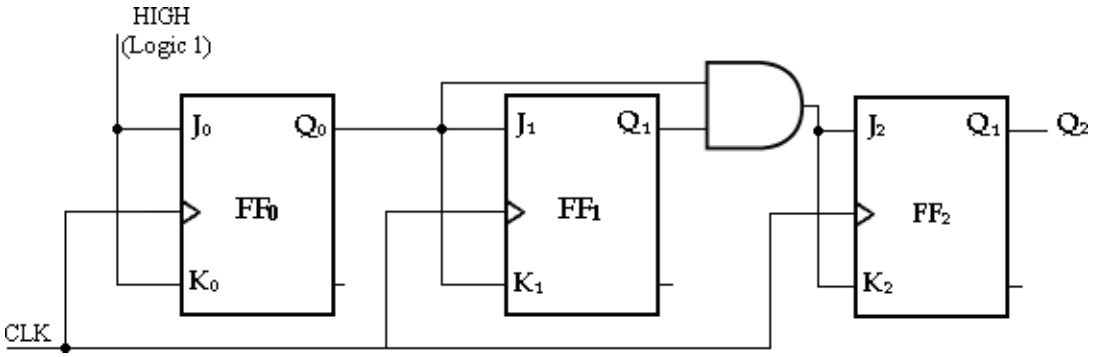


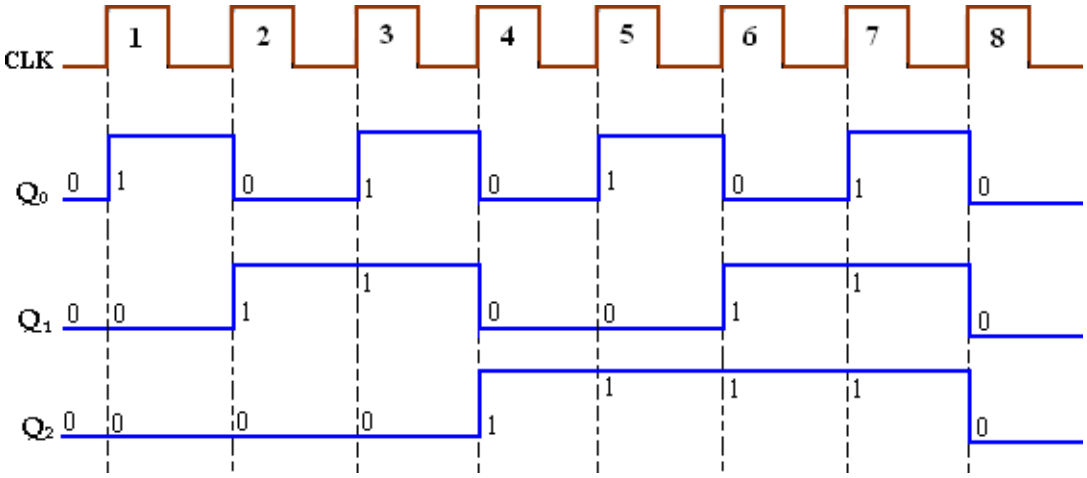
Fig. Logic diagram of 3-Bit Synchronous Binary Counter

A 3 bit synchronous binary counter is constructed with three JK Flip-Flops and an AND gate. The output of FF_0 (Q_0) changes on each clock pulse as the counter progresses from its original state to its final state and then back to its original state. To produce this operation, FF_0 must be held in the toggle mode by constant HIGH, on its J_0 and K_0inputs.

The output of FF_1 (Q_1) goes to the opposite state following each time Q_0= 1. This change occurs at CLK2, CLK4, CLK6, and CLK8. The CLK8 pulse causes the counter to recycle. To produce this operation, Q_0 is connected to the J_1 and K_1 inputs of FF_1. When Q_0= 1 and a clock pulse occurs, FF_1 is in the toggle mode and therefore changes state. When Q_0= 0, FF_1 is in the no-change mode and remains in its present state.

The output of FF_2 (Q_2) changes state both times; it is preceded by the unique condition in which both Q_0 and Q_1 are HIGH. This condition is detected by the AND gate and applied to the J_2 and K_2 inputs of FF_3. Whenever both outputs Q_0= Q_1= 1, the output of the AND gate makes the J_2= K_2= 1 and FF_2 toggles on the following clock pulse. Otherwise, the J_2 and K_2 inputs of FF2 are held LOW by the AND gate output, FF_2 does not change state.

CLOCK Pulse	Q_2	Q_1	Q_0
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0



Timing diagram

4-Bit Synchronous Binary Counter

This particular counter is implemented with negative edge-triggered Flip- Flops. The reasoning behind the J and K input control for the first three Flip- Flops is the same as previously discussed for the 3-bit counter. For the fourth stage, the Flip- Flop has to change the state when $Q_0 = Q_1 = Q_2 = 1$. This condition is decoded by AND gate G_3.

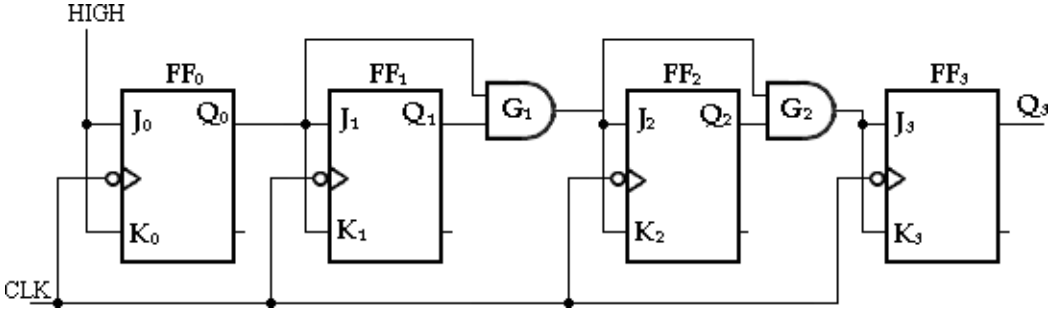
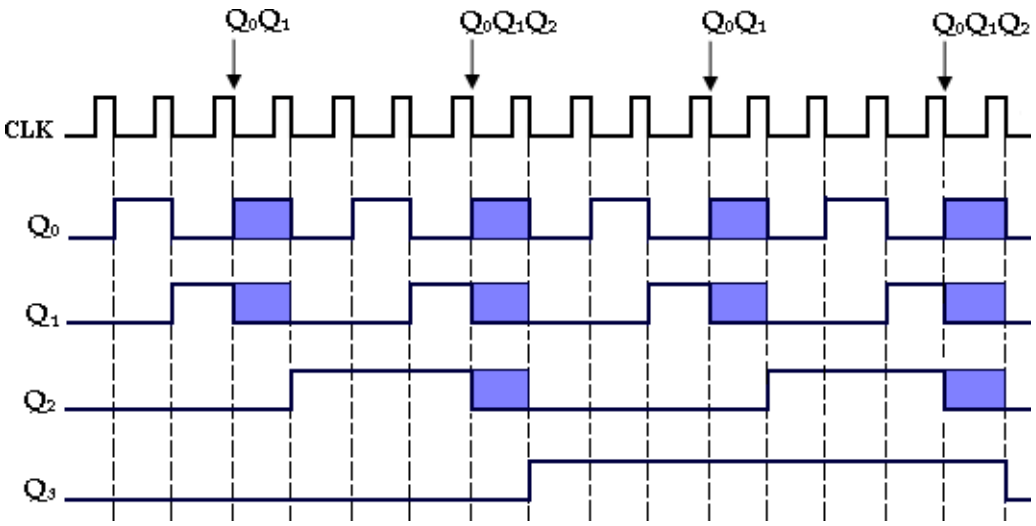


Fig. Logic diagram of 4-Bit Synchronous Binary Counter

Therefore, when $Q_0 = Q_1 = Q_2 = 1$, Flip-Flop FF_3 toggles and for all other times it is in a no-change condition. Points where the AND gate outputs are HIGH are indicated by the shaded areas.



Timing diagram

4-Bit Synchronous Decade Counter: (BCD Counter):

BCD decade counter has a sequence from 0000 to 1001 (9). After 1001 state it must recycle back to 0000 state. This counter requires four Flip-Flops and AND/OR logic as shown below.

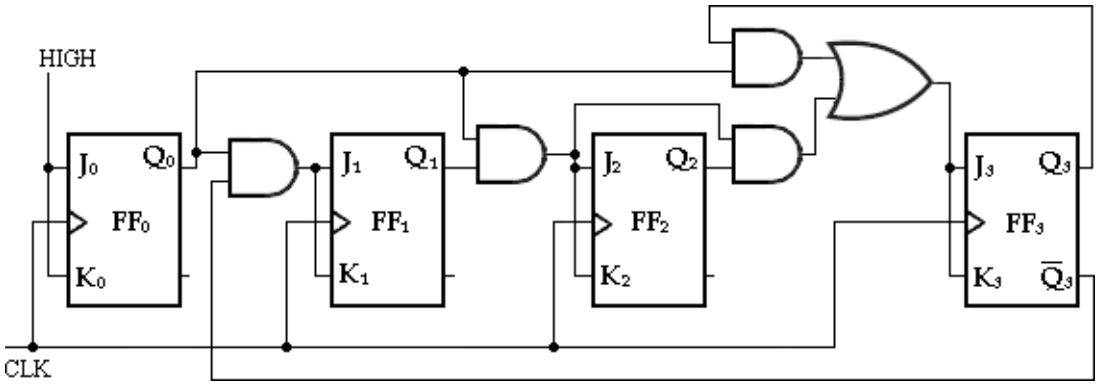


Fig. 4-Bit Synchronous Decade Counter

CLOCK Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0

3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10(recycles)	0	0	0	0

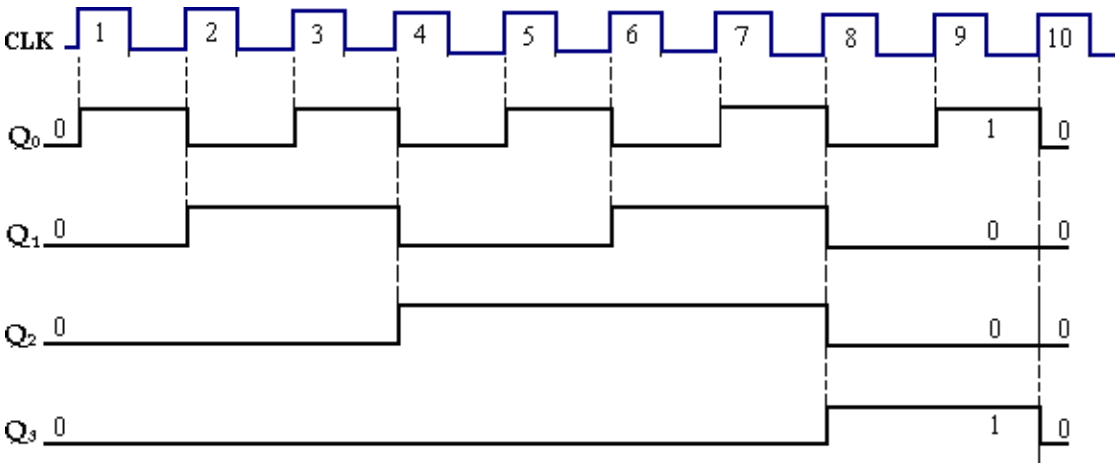
First, notice that FF_0 (Q_0) toggles on each clock pulse, so the logic equation for its J_0 and K_0 inputs is **$J_0 = K_0 = 1$**

This equation is implemented by connecting J_0 and K_0 to a constant HIGH level. Next, notice from table, that FF_1 (Q_1) changes on the next clock pulse each time Q0 = 1 and Q3 = 0, so the logic equation for the J_1 and K_1 inputs is **$J_1 = K_1 = Q_0Q_3'$**

This equation is implemented by ANDing Q_0 and Q_3 and connecting the gate output to the J_1 and K_1 inputs of FF_1. Flip-Flop 2 (Q_2) changes on the next clock pulse each time both $Q_0 = Q_1 = 1$. This requires an input logic equation as follows: **$J_2 = K_2 = Q_0Q_1$**

This equation is implemented by ANDing Q_0 and Q_1 and connecting the gate output to the J_2 and K_2 inputs of FF_3. Finally, FF_3 (Q_3) changes to the opposite state on the next clock pulse each time $Q_0 = 1$, $Q_1 = 1$, and $Q_2 = 1$ (state 7), or when $Q_0 = 1$ and $Q_1 = 1$ (state 9). The equation for this is as follows: **$J_3 = K_3 = Q_0Q_1Q_2 + Q_0Q_3$**

This function is implemented with the AND/OR logic connected to the J_3 and K_3 inputs of FF_3.



Timing diagram

Synchronous UP/DOWN Counter

An up/down counter is a bidirectional counter, capable of progressing in either direction through a certain sequence. A 3-bit binary counter that advances upward through its

sequence (0, 1, 2, 3, 4, 5, 6, 7) and then can be reversed so that it goes through the sequence in the opposite direction (7, 6, 5, 4, 3, 2, 1,0) is an illustration of up/down sequential operation.

The complete up/down sequence for a 3-bit binary counter is shown in table below. The arrows indicate the state-to-state movement of the counter for both its UP and its DOWN modes of operation. An examination of Q_0 for both the up and down sequences shows that FF_0 toggles on each clock pulse. Thus, the J_0 and K_0 inputs of FF_0 are, **$J_0 = K_0 = 1$**

CLOCK PULSE	UP	Q_2	Q_1	Q_0	DOWN
0		0	0	0	
1		0	0	1	
2		0	1	0	
3		0	1	1	
4		1	0	0	
5		1	0	1	
6		1	1	0	
7		1	1	1	

To form a synchronous UP/DOWN counter, the control input (UP/DOWN) is used to allow either the normal output or the inverted output of one Flip-Flop to the J and K inputs of the next Flip-Flop. When UP/DOWN= 1, the MOD 8 counter will count from 000 to 111 and UP/DOWN= 0, it will count from 111 to 000.

When UP/DOWN= 1, it will enable AND gates 1 and 3 and disable AND gates 2 and 4. This allows the Q0 and Q1 outputs through the AND gates to the J and K inputs of the following Flip-Flops, so the counter counts up as pulses are applied.
When UP/DOWN= 0, the reverse action takes place.

$J_1 = K_1 = (Q_0.UP) + (Q_0'.DOWN)$

$J_2 = K_2 = (Q_0.\ Q_1.UP) + (Q_0'.Q1'.DOWN)$

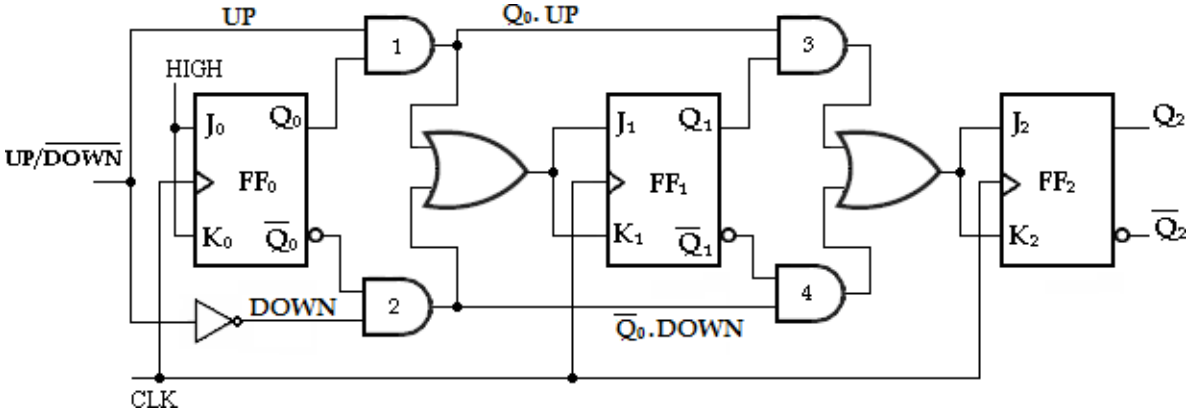


Fig. Circuit diagram of 3-bit UP/DOWN Synchronous Counter

Design of Synchronous MOD Counter

The counter with 'n' Flip-Flops has maximum MOD number 2^n. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

$$2^n \geq N$$

For example, a 3 bit binary counter is a MOD 8 counter. The basic counter can be modified to produce MOD numbers less than 2^n by allowing the counter to skin those are normally part of counting sequence.

n= 3 , N= 8 , $2^n = 23 = 8 = N$

MOD 5 Counter:

$2^n = N$, $2^n = 5$, $2^2 = 4$ less than N., $2^3 = 8 > N(5)$

Therefore, 3 Flip-Flops are required.

MOD 10 Counter:

$2^n = N = 10$, $2^3 = 8$ less than N, $2^4 = 16 > N(10)$.

To construct any MOD-N counter, the following methods can be used.

- Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation, $2^n \geq N$.
- Connect all the Flip-Flops as a required counter.
- Find the binary number for N.
  - Connect all Flip-Flop outputs for which Q= 1 when the count is N, as inputs to NAND gate.
  - Connect the NAND gate output to the CLR input of each Flip-Flop.

When the counter reaches N^{th} state, the output of the NAND gate goes LOW, resetting all Flip-Flops to 0. Therefore the counter counts from 0 through N-1.

For example, MOD-10 counter reaches state 10 (1010). i.e., $Q_3Q_2Q_1Q_0$= 1 0 1 0. The outputs Q_3 and Q_1 are connected to the NAND gate and the output of the NAND gate goes LOW and resetting all Flip-Flops to zero. Therefore MOD-10 counter counts from 0000 to 1001. And then recycles to the zero value.

The MOD-10 counter circuit is shown below.

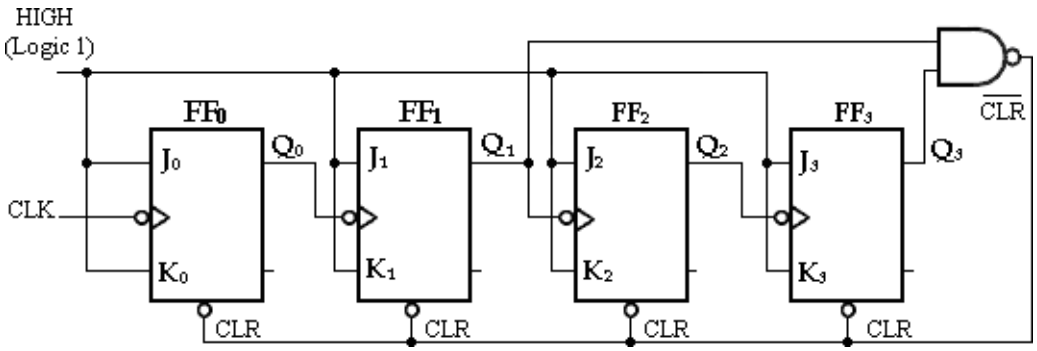


Fig. Logic diagram of MOD-10 (Decade) Counter

Design a MOD- 5 synchronous counter using JK flip flops and implement it, Also draw the timing diagram

Step 1 : Determine the number of flipflop needed. Her N=5, $2^n >= N$, n=3 number of flipflops.
Step 2: type of flipflop is JK
Step 3: Determine the execution table
Step 4: Find the input of the Flipflop using K-Map
Step 5: Draw the Circuit diagram

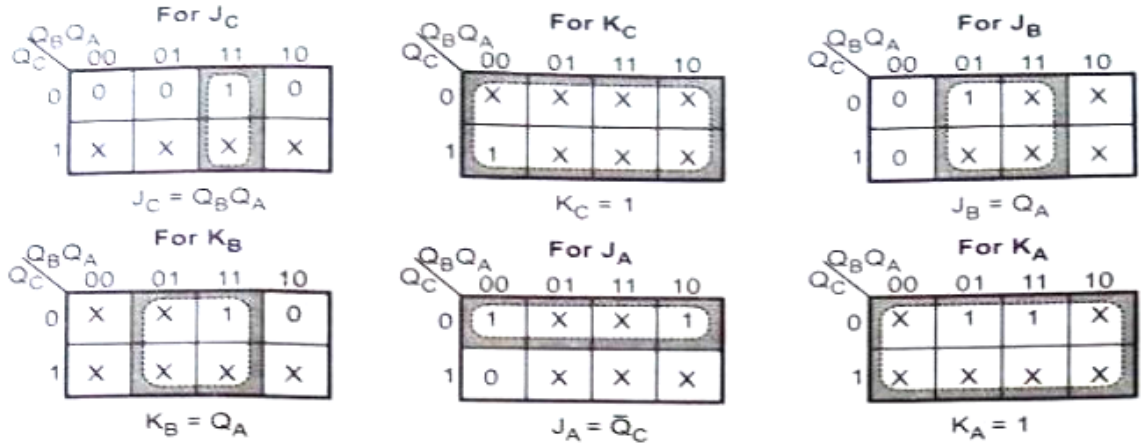
Excitation table of JK flip flop

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

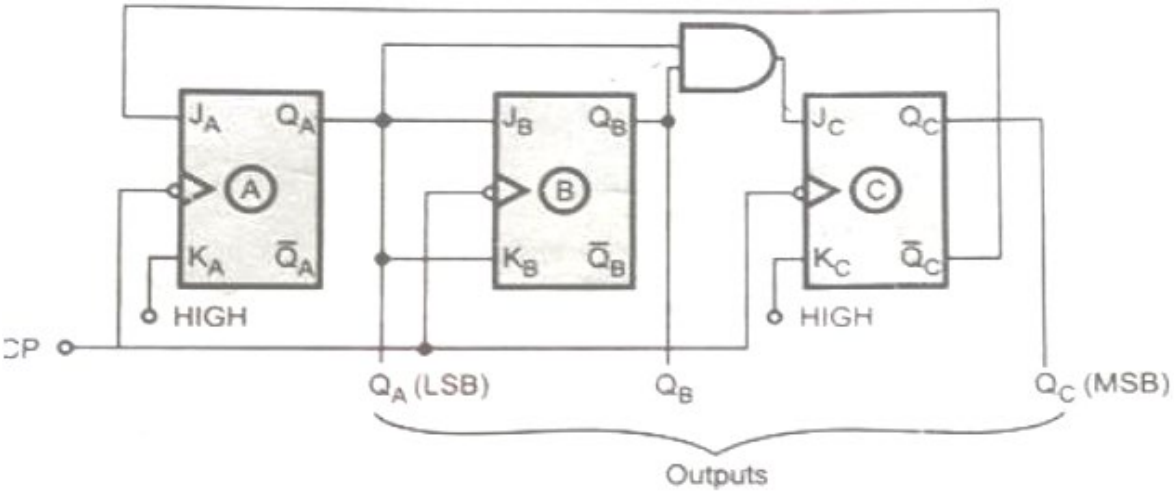
Excitation table for the counter

Present State			Next State			Flip- flop inputs					
Q_C	Q_B	Q_A	Q_{C+1}	Q_{B+1}	Q_{A+1}	J_C	K_C	J_B	K_B	J_A	K_A
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	0	0	0	X	1	0	X	0	X
1	0	1	X	X	X	X	X	X	X	X	X
1	1	0	X	X	X	X	X	X	X	X	X
1	1	1	X	X	X	X	X	X	X	X	X

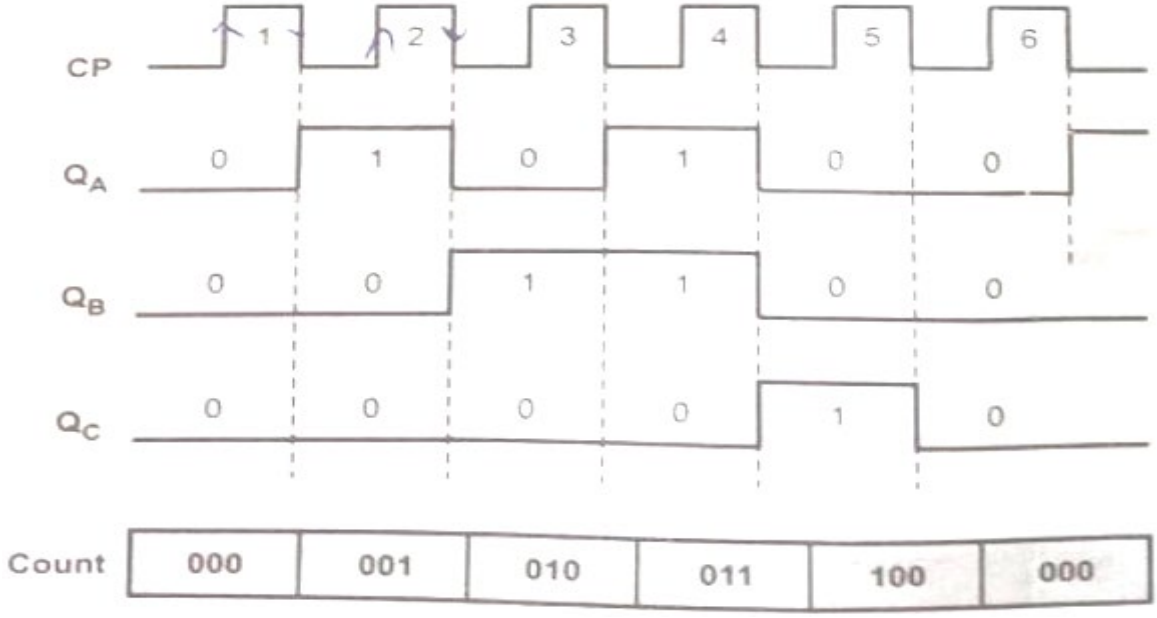
Step 4: Simplification using K - Map



Step 5: Draw the Logic diagram



Timing Diagram



Asynchronous or ripple counters

Asynchronous 2 bit Up counter:

A binary ripple/asynchronous counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the clock input of the next higher-order flip-flop. The flip-flop holding the least significant bit receives the incoming clock pulses. A complementing flip-flops can be obtained from a JK flip-flop with the J and K inputs tied together as shown in the Fig. or from a T flip-flop. A third alternative is to use a D flip-flop with the complement output connected to the D input.

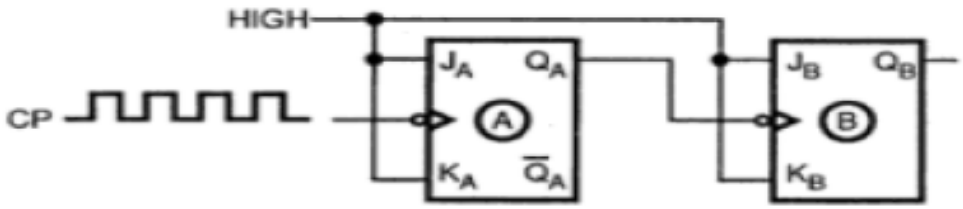


Fig. (a) A two-bit asynchronous binary counter

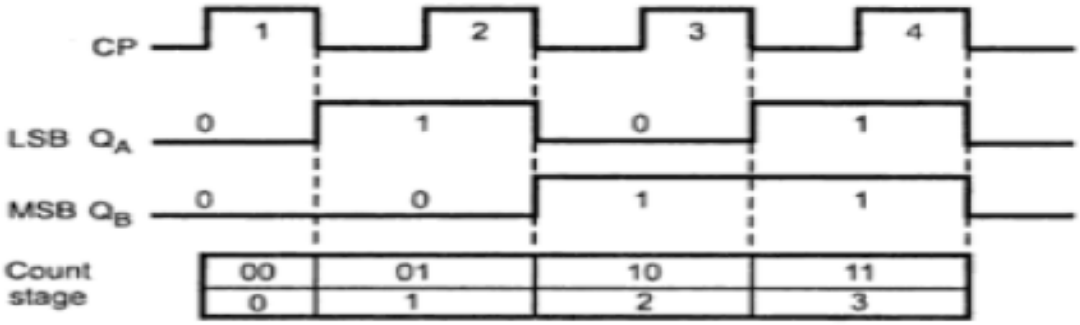


Fig. (b) Timing diagram for the counter

Fig. (a) shows 2-bit asynchronous counter using JK flip-flops. As shown in Fig. the clock signal is connected to the clock input of only first stage flip-flop. The clock input of the second stage flip-flop is triggered by the Q_A output of the first stage . Because of the inherent propagation delay time through a flip-flop, a transition of the input clock pulse and a transition of the Q_A output of first stage can never occur at exactly the same time. Therefore, the two flip-flops are never simultaneously triggered, which results in asynchronous counter operation.

Fig. (b) shows the timing diagram for two-bit asynchronous counter. It illustrates the changes in the state of the flip-flop outputs in response to the clock. J and K input of JK flip-flops are tied to logic HIGH hence output will toggle for each negative edge of the clock input.

Operation

Initially let both the FFs be in the reset state, $Q_AQ_B = 00$

As soon as the first negative clock edge is applied, FF-A will toggle and Q_A will change from 0 to 1. But at the instant of application of negative clock edge, Q_A , $J_B = K_B = 0$. Hence FF-B will not change its state. So Q_B will remain0. $Q_BQ_A = 01$ after the first clock pulse.

On the arrival of second negative clock edge, FF-A toggles again and Q_A changes from 1 to 0. But at this instant Q_A was 1. So $J_B = K_B = 1$ and FF-B will toggle. Hence Q_B changes from 0 to1. $Q_BQ_A = 10$ after the second clock pulse.

On application of the third falling clock edge, FF-A will toggle from 0 to 1 but there is no change of state for FF-B. $Q_BQ_A = 11$ after the third clock pulse. The output of counter is incremented by one for each clock transition. So,such counters are called as up counters.

Asynchronous 3 bit Up counter :

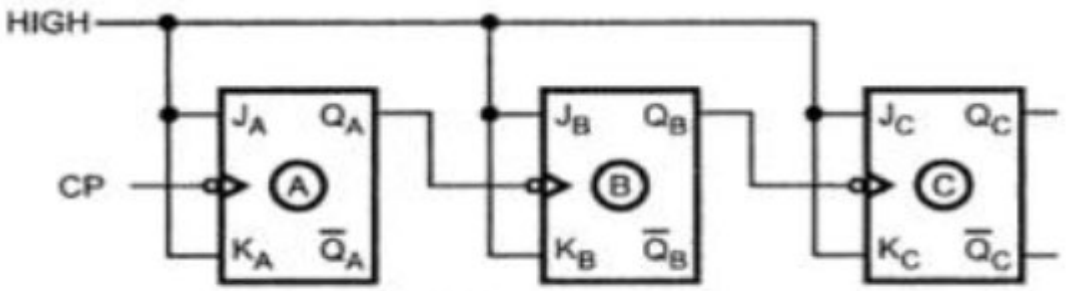


Fig. (a) Logic diagram

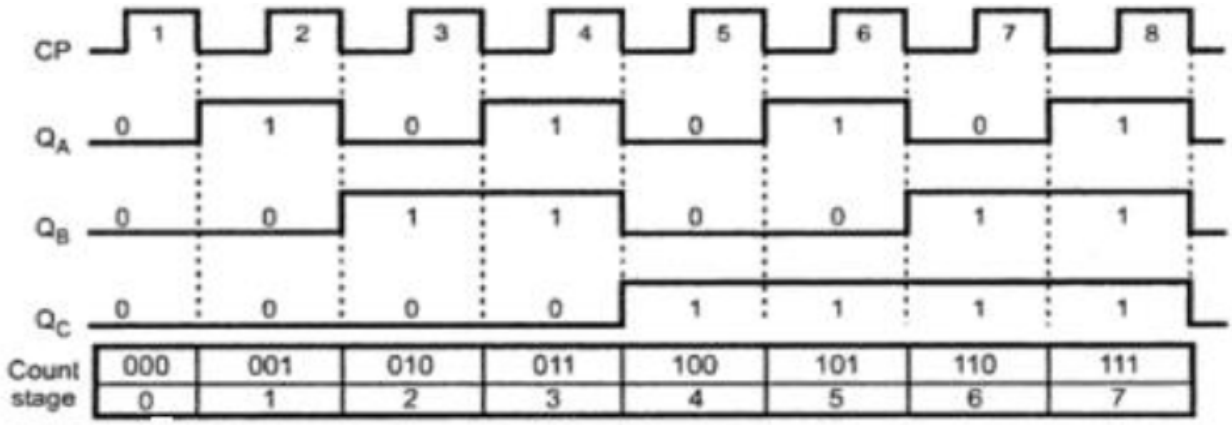
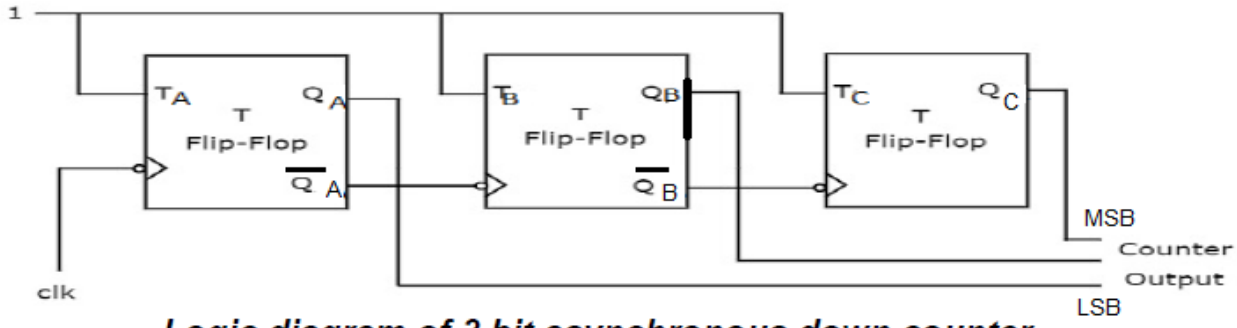


Fig. (b) Output waveforms for 3-bit asynchronous counter

Asynchronous 3 bit Down counter

An 'N' bit asynchronous binary down counter consists of 'N' T flip-flops. It counts from $2^N - 1$ to 0. The block diagram of 3-bit Asynchronous binary down counter is shown in figure.



Logic diagram of 3 bit asynchronous down counter

Down counters are not as widely used as up counters. They are used in situations where it must be known when a desired number of input pulses has occurred. In these situations the down counter is preset to the desired number and then allowed to count down as the pulses are applied. When the counter reaches the zero state it is detected by a logic gate whose output then indicates that the preset number of pulses have occurred.

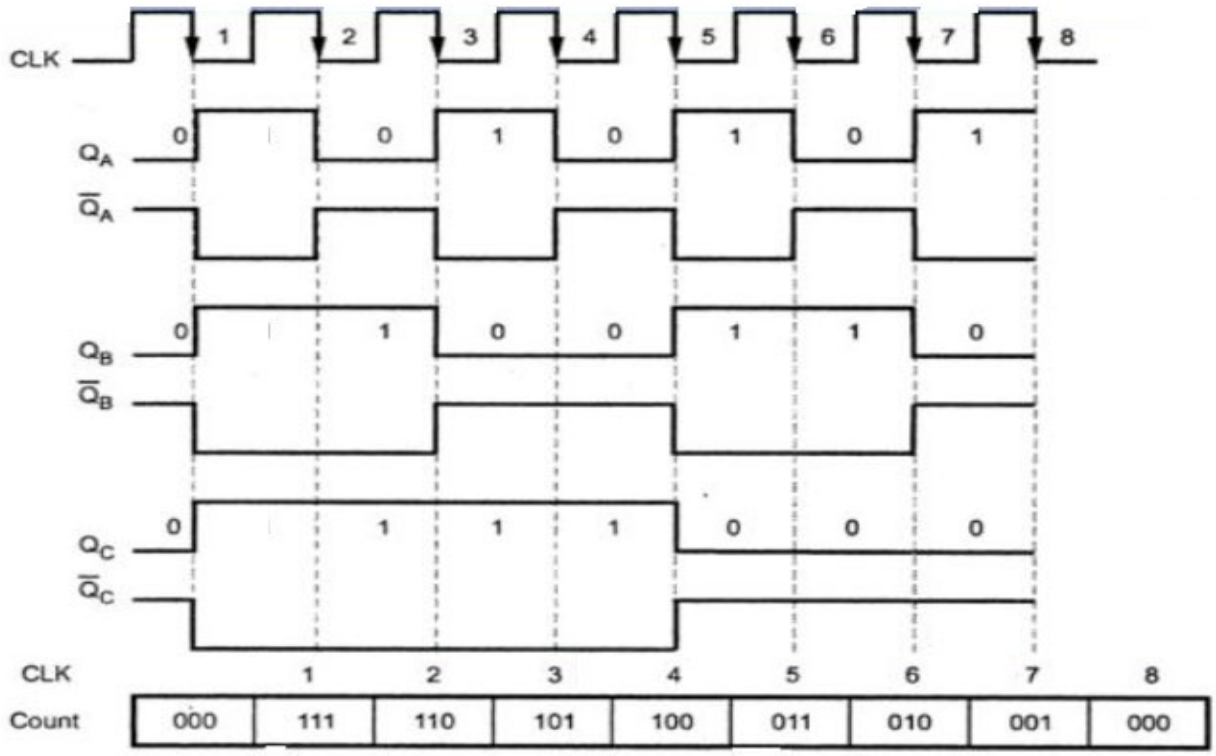
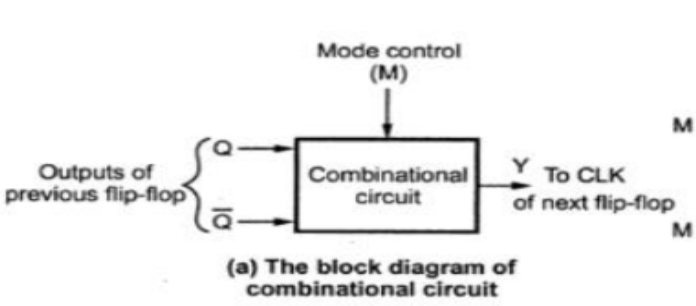


Fig. Timing diagram for 3-bit DOWN ripple counter

Asynchronous Up / Down counter

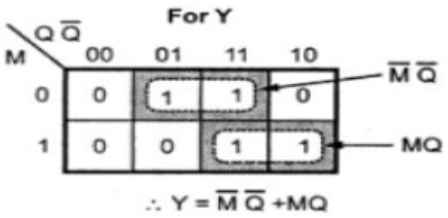
To form an asynchronous up/$\overline{\text{down}}$ counter one control input say M is necessary to control the operation of the up/$\overline{\text{down}}$ counter. When M = 1, the counter will count up and when M = 0, the counter will count down. To achieve this the M input should be used to control whether the normal flip-flop output (Q) or the inverted flip-flop output ($\overline{Q}$) is fed to drive the clock signal of the successive stage flip-flop, as shown in Fig. (a). The truth table for such combinational circuit is shown in Fig. (b).



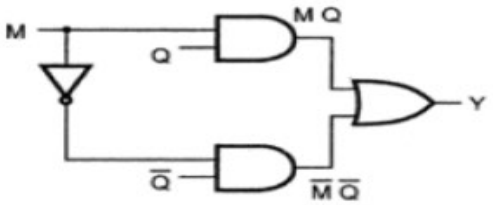
(a) The block diagram of combinational circuit

Inputs M	Q	$\overline{Q}$	Output Y	
M = 0	0	0	0	Y = $\overline{Q}$ for down counting
	0	1	1	
	1	0	0	
	1	1	1	
M = 1	0	0	0	Y = Q for up counting
	0	1	0	
	1	0	1	
	1	1	1	

(b) Truth table



a) K-map simplification



b) Logic diagram

The Fig. shows the 3-bit up/down counter that will count from 000 up to 111 when the mode control input M is 1 and from 111 down to 000 when mode control input M is 0.

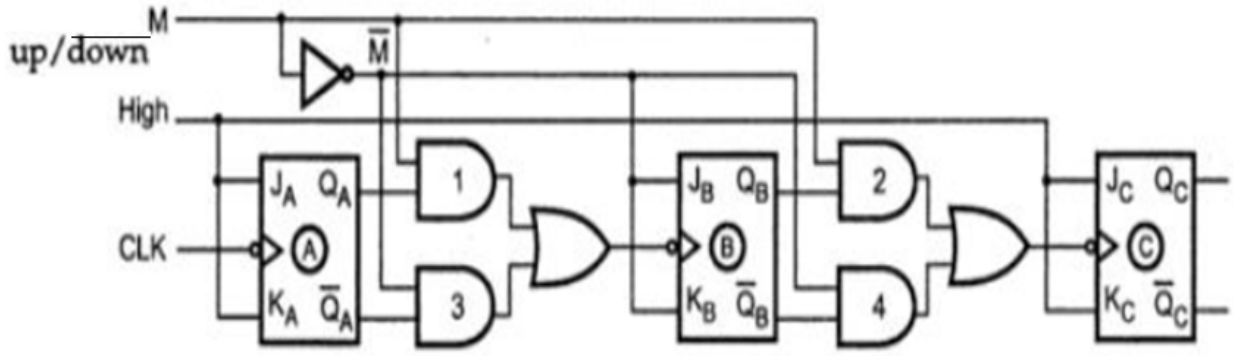


Fig. 3-bit asynchronous up/down counter

A logic 1 on M enables AND gates 1 and 2 and disables AND gates 3 and 4. This allows the Q_A and Q_B outputs to drive the clock inputs of their respective next stages. So that counter will count up. When M is logic 0, AND gates 1 and 2 are disabled and AND gete 3 and 4 are enabled. This allows the $\overline{Q}_A$ and $\overline{Q}_B$ outputs to drive the clock inputs of their respective next stages so that counter will count down. .

Design of Asynchronous MOD Counter

To construct any MOD-N counter, the following methods can be used.

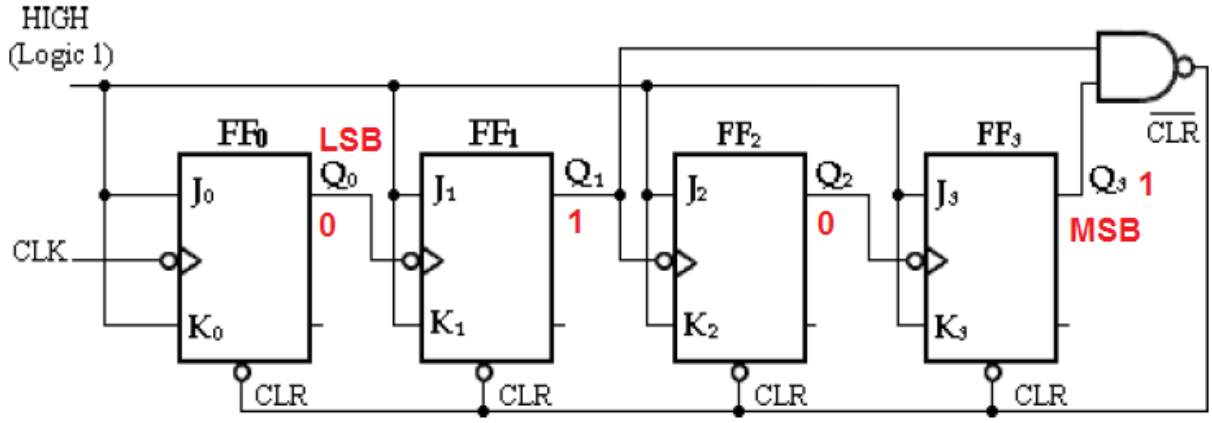
1. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

   $2^n \geq N$.
2. Connect all the Flip-Flops as a required counter.
3. Find the binary number for N.
4. Connect all Flip-Flop outputs for which Q= 1 when the count is N, as inputs to NAND gate.
5. Connect the NAND gate output to the CLR input of each Flip-Flop.

Asynchronous BCD counter or MOD- 10 counter or Decade Counter

BCD decade counter has a sequence from 0000 to 1001. After 1001 state it must recycle back to 0000 state. This counter requites four Flip Flops and AND / OR logis as shown below.

- **Count from 0 – 9 (0000 – 1001)**
- **From 10 – 15 (1010 – 1111) Reset**

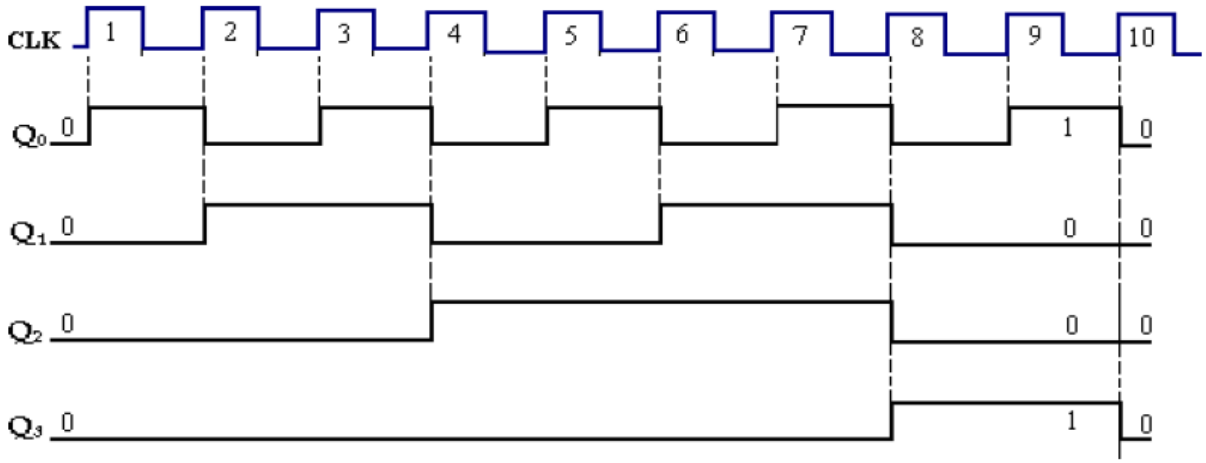


MOD-10 (Decade) Counter

Output $Q_3 Q_2 Q_1 Q_0$

Truth table:

CLOCK Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10(recycles)	0	0	0	0



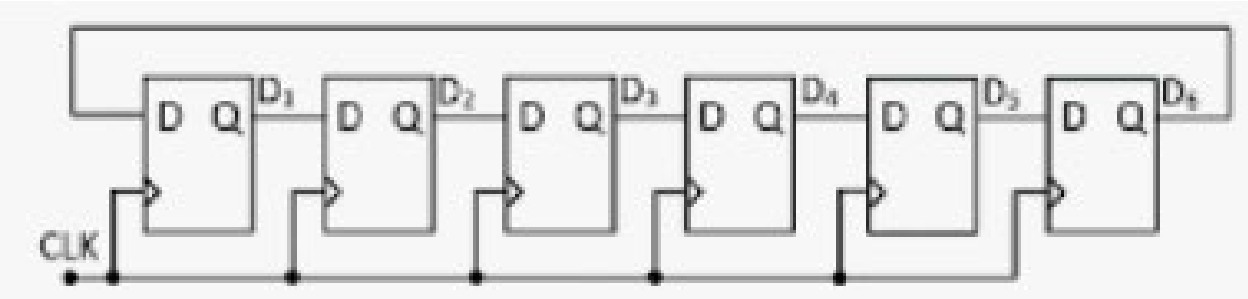
Timing diagram

Ring Counter

Count = Number of flipflops

Used to identify the active state with Logic 1

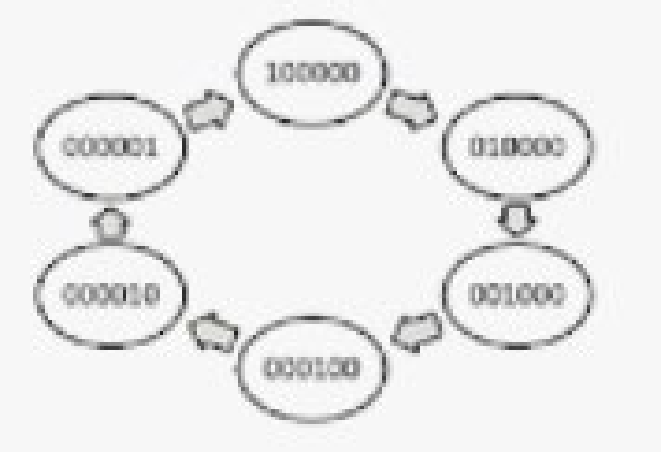
6 – bit Ring Counter circuit Diagram



Truth Table

Clk	D_1	D_2	D_3	D_4	D_5	D_6
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0
6	0	0	0	0	0	1

State Diagram

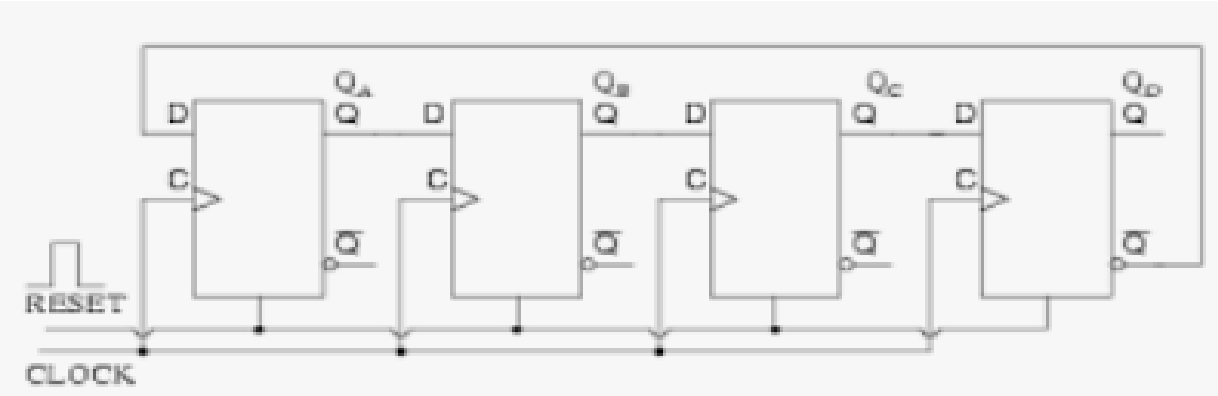


Johnson Counter

Count = Number of Flipflops * 2

Reduce the Number of Flipflops

Circuit Diagram



Truth Table

Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
0	0	0	0

HAZARDS

Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays.

Hazards occur in combinational circuits, where they may cause a temporary false-output value. When this condition occurs in asynchronous sequential circuits, it may result in a transition to a wrong stable state.

Hazards in Combinational Circuits:

A hazard is a condition where a single variable change produces a momentary output change when no output change should occur.

Types of Hazards:

Static hazard Dynamic hazard

- **Static Hazard**

In digital systems, there are only two possible outputs, a '0' or a '1'. The hazard may produce a wrong '0' or a wrong '1'. Based on these observations, there are three types,

Static- 0 hazard,

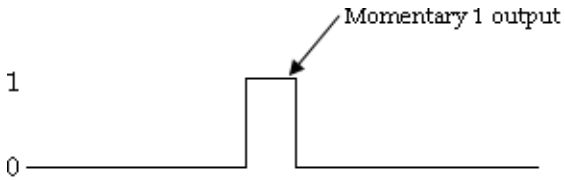
Static- 1 hazard,

Static- 0 hazard:

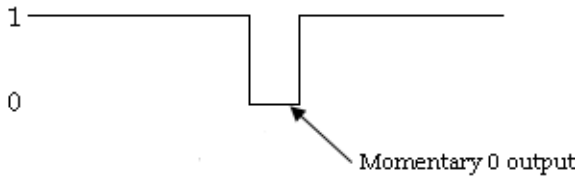
When the output of the circuit is to remain at 0, and a momentary 1 output is possible during the transmission between the two inputs, then the hazard is called a static 0-hazard.

Static- 1 hazard:

When the output of the circuit is to remain at 1, and a momentary 0 output is possible during the transmission between the two inputs, then the hazard is called a static 1-hazard.



(a) Static 0-hazard



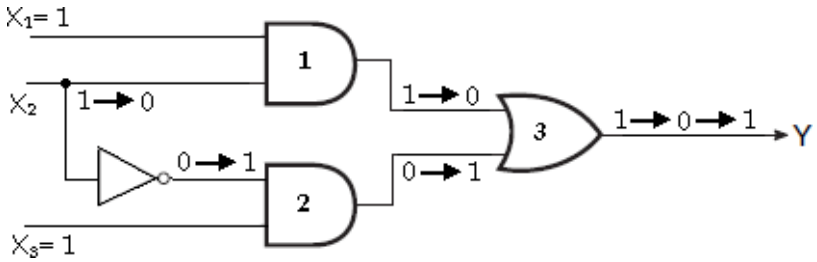
(b) Static 1-hazard

The below circuit demonstrates the occurrence of a static 1-hazard. Assume that all three inputs are initially equal to 1 i.e., $X_1X_2X_3$= 111. This causes the output of the gate 1 to be 1, that of gate 2 to be 0, and the output of the circuit to be equal to 1. Now consider a change of X_2 from 1 to 0 i.e., $X_1X_2X_3$= 101. The output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. The output may momentarily go to 0 if the propagation delay through the inverter is taken into consideration.

The delay in the inverter may cause the output of gate 1 to change to 0 before the output of gate 2 changes to 1. In that case, both inputs of gate 3 are momentarily equal to 0, causing the output to go to 0 for the short interval of time that the input signal from X_2 is delayed while it is propagating through the inverter circuit.

Thus, a static 1-hazard exists during the transition between the input states

$X_1X_2X_3$= 111 and $X_1X_2X_3$= 101.



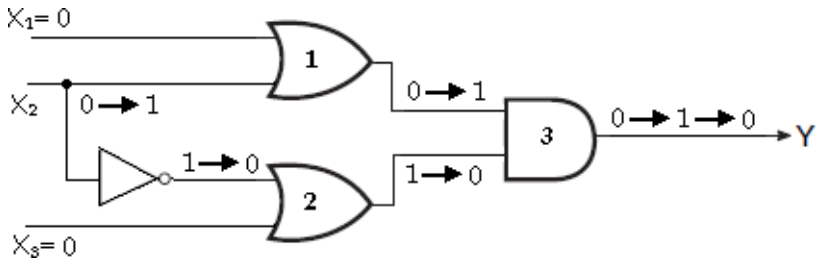
Circuit with static-1 hazard

Now consider the below network, and assume that the inverter has an appreciably greater propagation delay time than the other gates. In this case there is a static 0-hazard in the transition between the input states $X_1X_2X_3$= 000 and $X_1X_2X_3$= 010 since it is possible for a logic-1 signal to appear at both input terminals of the AND gate for a short duration.

The delay in the inverter may cause the output of gate 1 to change to 1 before the output of gate 2 changes to 0. In that case, both inputs of gate 3 are momentarily equal to 0, causing the output to go to 1 for the short interval of time that the input signal from X_2 is delayed while it is propagating through the inverter circuit.

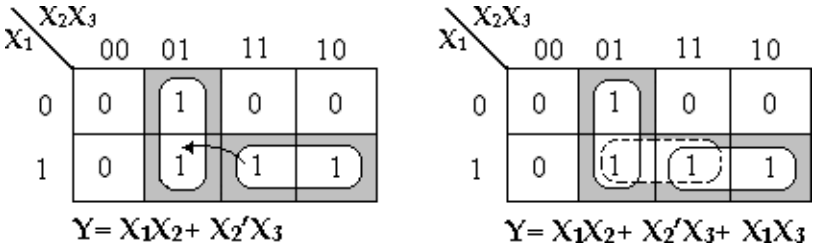
Thus, a static 0-hazard exists during the transition between the input states

$X_1X_2X_3$= 000 and $X_1X_2X_3$= 010.



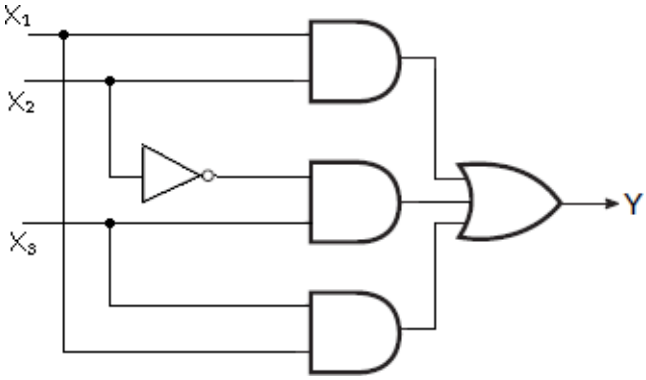
Circuit with static-0 hazard

A hazard can be detected by inspection of the map of the particular circuit. To illustrate, consider the map in the circuit with static 0-hazard, which is a plot of the function implemented. The change in X_2 from *1* to 0 moves the circuit from minterm 111 to minterm 101. The hazard exists because the change in input results in a different product term covering the two minterrns.



Maps demonstrating a Hazard and its Removal

The minterm 111 is covered by the product term implemented in gate 1 and minterm 101 is covered by the product term implemented in gate 2. Whenever the circuit must move from one product term to another, there is a possibility of a momentary interval when neither term is equal to 1, giving rise to an undesirable 0 output. The remedy for eliminating a hazard is to enclose the two minterms in question with another product term that overlaps both groupings. This situation is shown in the *map* above, where the two terms that causes the hazard are combined into one product term. The hazard- free circuit obtained by this combinational is shown below.



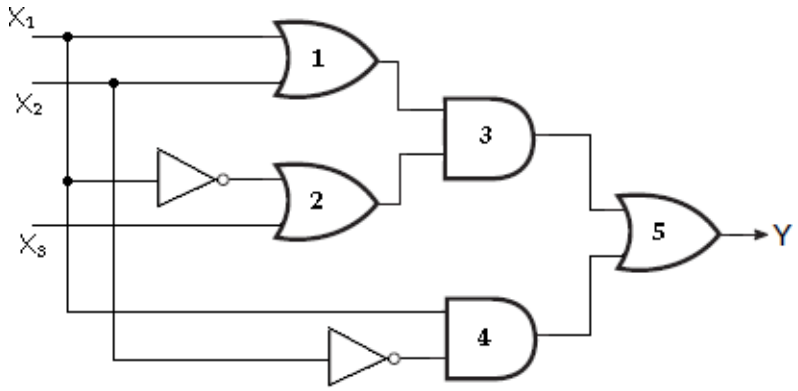
Hazard-free Circuit

The extra gate in the circuit generates the product term X_1X_4. The hazards in combinational circuits can be removed by covering any two minterms that may produce a hazard with a product term common to both. The removal of hazards requires the addition of redundant gates to the circuit.

- **Dynamic Hazard**

A dynamic hazard is defined as a transient change occurring three or more times at an output terminal of a logic network when the output is supposed to change only once during a transition between two input states differing in the value of one variable.

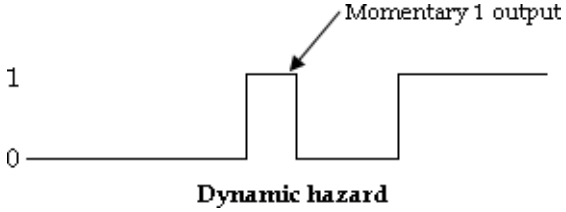
Now consider the input states $X_1X_2X_3$= 000 and $X_1X_2X_3$= 100. For the first input state, the steady state output is 0; while for the second input state, the steady state output is 1. To facilitate the discussion of the transient behavior of this network, assume there are no propagation delays through gates G_3 and G_5 and that the propagation delays of the other three gates are such that G_1 can switch faster than G_2 and G_2 can switch faster than G_4.



Circuit with Dynamic hazard

When X_1 changes from 0 to 1, the change propagates through gate G_1 before gate G_2 with the net effect that the inputs to gate G_3 are simultaneously 1 and the network output changes from 0 to 1. Then, when X_1 change propagates through gate G_2, the lower input to gate G_3 becomes 0 and the network output changes back to 0.

Finally, when the X_1= 1 signal propagates through gate G_4, the lower input to gate G_5 becomes 1 and the network output again changes to 1. It is therefore seen that during the change of X_1 variable from 0 to 1 the output undergoes the sequence, 0 to 1 to 0 to 1, which results in three changes when it should have undergone only a single change.



Dynamic hazard

- **Essential Hazard**

An essential hazard is caused by unequal delays along two or more paths that originate from the same input. An excessive delay through an inverter circuit in comparison to the delay associated with the feedback path may cause such a hazard. **Essential hazards elimination:** Essential hazards can be eliminated by adjusting the amount of delays in the

affected path. To avoid essential hazards, each feedback loop must be handled with individual care to ensure that the delay in the feedback path is long enough compared with delays of other signals that originate from the input terminals.

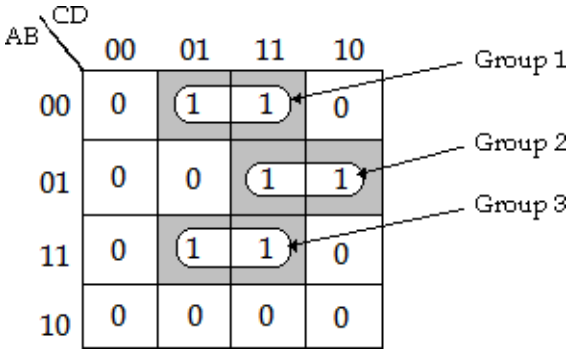
Design Of Hazard Free Circuits

Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (1, 3, 6, 7, 13, 15)

Soln:

- K-map Implementation and grouping

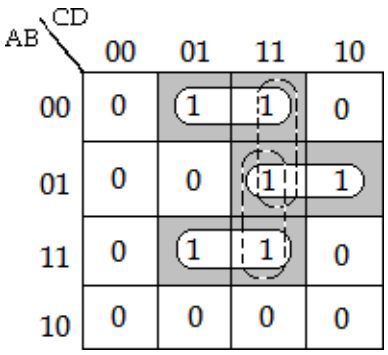


F=A'B'D+ A'BC+ ABD

- Hazard- free realization

The first additional product term A'CD, overlapping two groups (group 1 &

2) and the second additional product term, BCD, overlapping the two groups (group 2 & 3).



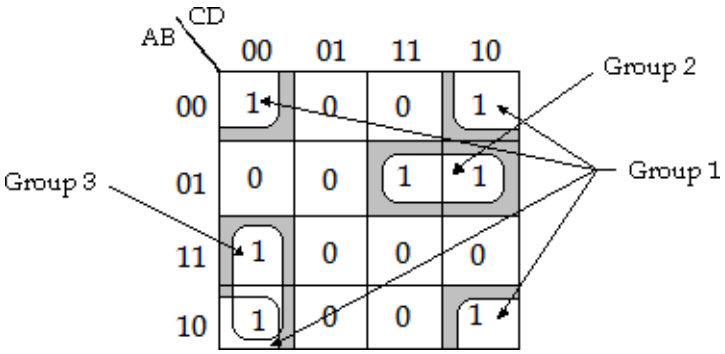
F=A'B'D+ A'BC+ ABD+ A'CD+ BCD

Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (0, 2, 6, 7, 8, 10, 12).

Soln:

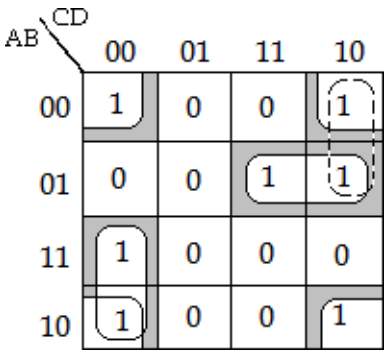
K-map Implementation and grouping



F= B'D'+ A'BC+ AC'D'

- Hazard- free realization

The additional product term, A'CD' overlapping two groups (group 1 & 2) for hazard free realization. Group 1 and 3 are already overlapped hence they do not require additional minterm for grouping.

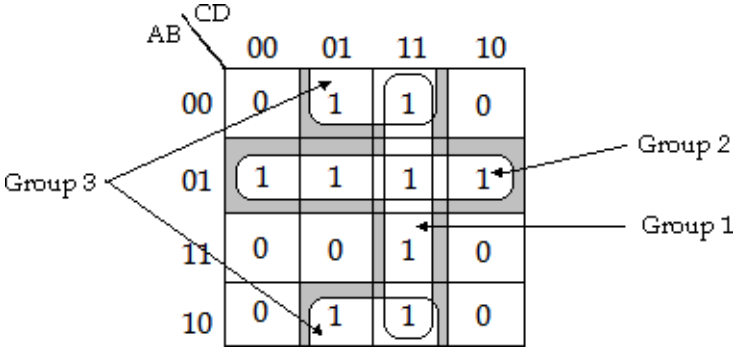


F= B'D'+ A'BC+ AC'D'+ A'CD'

<u>Design a hazard-free circuit to implement the following function.</u>

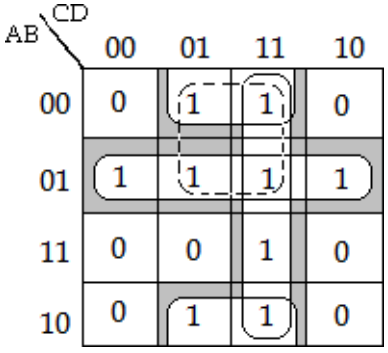
F (A, B, C, D) = ∑m (1, 3, 4, 5, 6, 7, 9, 11, 15).

- <u>K-map Implementation and grouping</u>



F= CD+ A'B+ B'D

- <u>Hazard- free realization</u>



The additional product term, A'D overlapping two groups (group 2 & 3) for hazard free realization. Group 1 and 2 are already overlapped hence they do not require additional minterm for grouping.

F= CD+ A'B+ B'D+ A'D

Design a hazard-free circuit to implement the following function.

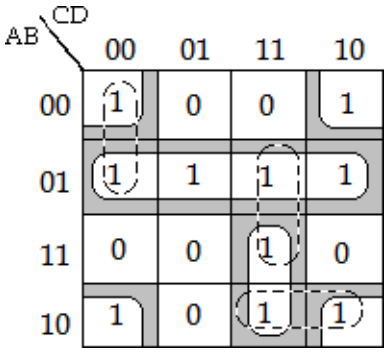
F (A, B, C, D) = ∑m (0, 2, 4, 5, 6, 7, 8, 10, 11, 15).

Soln:

K-map Implementation and grouping

F= B'D'+ A'B+ ACD

Hazard- free realization

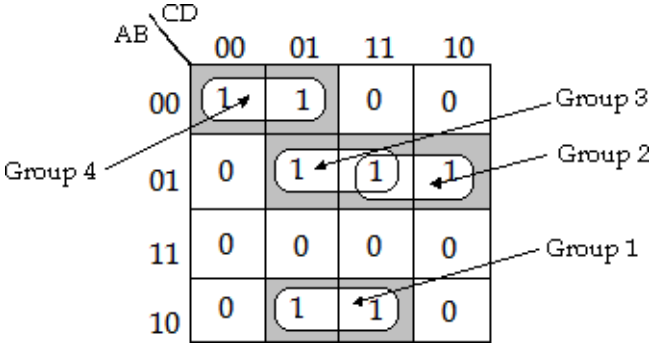


F= B'D'+ A'B+ ACD+ A'C'D'+ BCD+ AB'C

Design a hazard-free circuit to implement the following function.

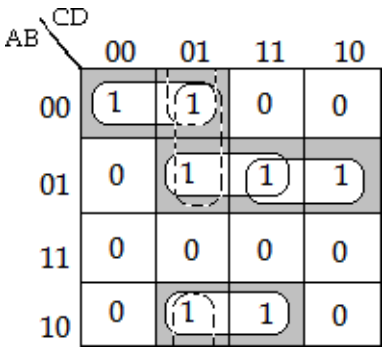
F (A, B, C, D) = ∑m (0, 1, 5, 6, 7, 9, 11).

a) K-map Implementation and grouping



F= AB'D+ A'BC+ A'BD+ A'B'C'

b) Hazard- free realization:



F= AB'D+ A'BC+ A'BD+ A'B'C'+ A'C'D+ B'C'D

RACES:

A race condition is said to exist in an asynchronous sequential circuit when two or more binary state variables change value in response to a change in an input variable.
Races are classified as:

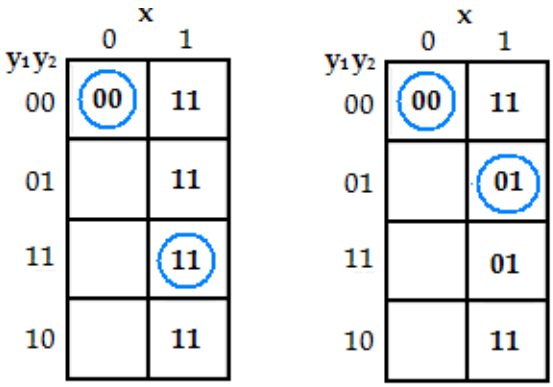
- Non-critical races
- Critical races.

Non-critical races:

the final stable state that the circuit reaches does not depend on the order in which the state variables change, the race is called a non-critical race.

If a circuit, whose transition table (a) starts with the total stable state y_1y_2x= *000* and then change the input from 0 to 1. The state variables must then change from 00 to 11, which define a race condition. The possible transitions are:

00	11	
00	01	11
00	10	11



(a) Possible transitions:

00 → 11
00 → 01 → 11
00 → 10 → 11

(b) Possible transitions:

00 → 11 → 01
00 → 01
00 → 10 → 11 → 01

In all cases, the final state is the same, which results in a non-critical condition. In (a), the final state is (y_1y_2x= 111), and in (b), it is (y_1y_2x= 011).

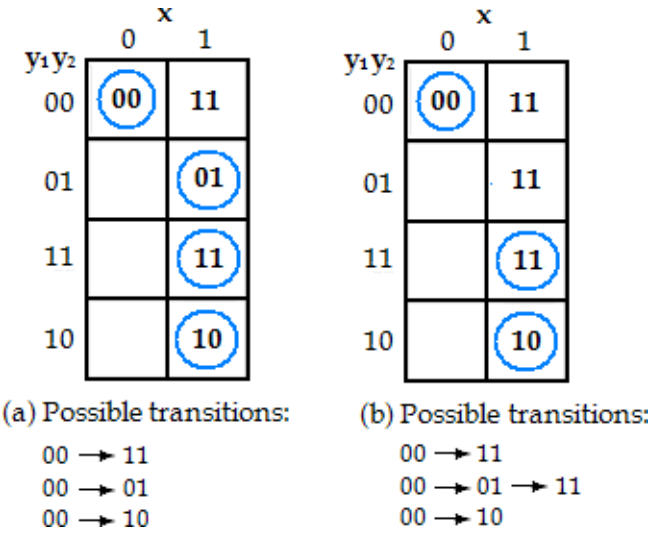
Examples of Non-critical Races

Critical races:

A race becomes critical if the correct next state is not reached during a state transition. If it is possible to end up in two or more different stable states, depending on the order in which the state variables change, then it is a critical race. For proper operation, critical races must be avoided.

The below transition table illustrates critical race condition. The transition table (a) starts in stable state (y_1y_2x= 000), and then change the input from 0 to 1. The state variables must then change from 00 to 11. If they change simultaneously, the final total stable state is 111. In the transition table (a), if, because of unequal propagation delay, Y_2 changes to 1 before Y_1 does, then the circuit goes to the total stable state 011 and remains there. If, however, Y_1 changes first, the internal state becomes 10 and the circuit will remain in the stable total state 101.

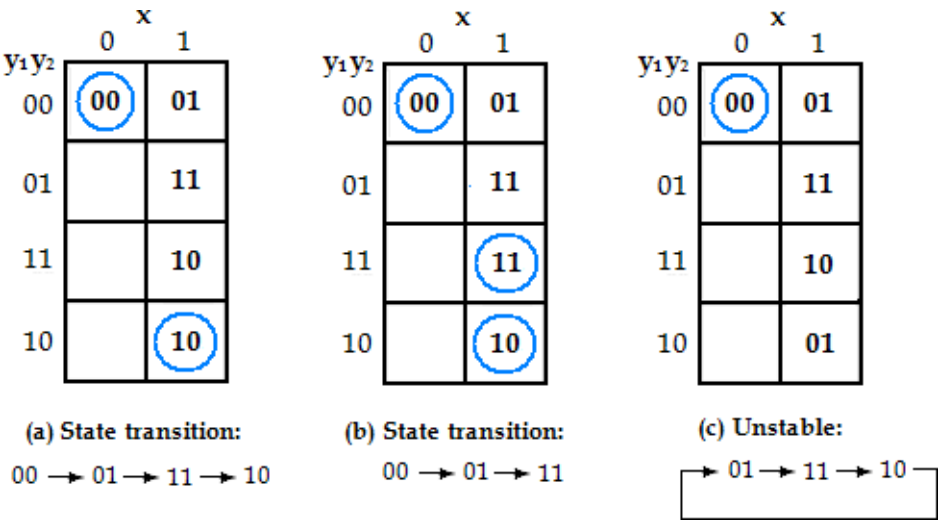
Hence, the race is critical because the circuit goes to different stable states, depending on the order in which the state variables change.



Examples of Critical Races

CYCLES

Races can be avoided by directing the circuit through intermediate unstable states with a unique state-variable change. When a circuit goes through a unique sequence of unstable states, it is said to have a cycle.



`Again, we start with y_1y_2 = 00 and change the input from *0* to 1. The transition table (a) gives a *unique* sequence that terminates in a total stable state 101. The table in (b) shows that even though the state variables change from 00 to 11, the cycle provides a unique transition from *00* to 01 and then to 11, Care must be taken when using a cycle that terminates with a stable state. If a cycle does not terminate with a stable state, the circuit will keep going from one unstable state to another, making the entire circuit unstable. This is demonstrated in the transition table (c).

TEXT BOOKS:

1. Morris Mano, “Digital design”, 3rd Edition, Prentice Hall of India, 2008.

REFERENCE BOOKS:

1. Milos Ercegovac, Jomas Lang, “Introduction to Digital Systems”, Wiley publications, 1998.
2. John M. Yarbrough, “Digital logic: Applications and Design”, Thomas – Vikas Publishing House, 2002.
3. R.P.Jain, “Modern digital Electronics”,3rd Edition, TMH, 2003.
4. William H. Gothmann, “Digital Electronics”, Prentice Hall, 2001.

QUESTION BANK

PART-A

1. Describeasynchronous sequential circuit.
2. Define state diagram.
3. Discuss about the conversion of SR flip flop to D flip flop.
4. Define Hazard
5. Recall the excitation table of JK flip flop.
6. Relate JK flip flop and T flip flop.
7. Illustrate the logic diagram of master slave JK flip flop.
8. Distinguish between a latch and a Flip flop.
9. Define Race in synchronous sequential circuits
10. Distinguish between asynchronous and synchronous counters.

PART-B

1. Describe flip – flop, Explain the principles and operation of SR flip flop and D flip flop..
2. Design a synchronous counter which count through the sequences
   0, 3, 6, 9, 12, 15, 0… use D flip flops.
3. Design and implement 4 bit asynchronous up counter
4. Illustrate the logic circuit of 4 bit binary sync. Counter with ripple carry using JK FF and explain its operation.
5. a) Explain JK flip flop
   b) Explain T flip flop and master slave JK flip flop.
6. Illustrate the logic diagram of a four bit SISO and PIPO shift register and explain the working principle.
7. Illustrate the logic diagram of a four bit SIPO and PISO shift register and explain the working principle.
8. Design MOD-6 synchronous counter using T ff.

UNIT V
DIGITAL LOGIC FAMILIES, MEMORIES AND PROGRAMMABLE DEVICES

Classification and characteristics of logic family – Bipolar logic family – Saturated logic family – Non saturated family – Unipolar family – MOS, CMOS logic families. Classification and Organization of memories – Programmable Logic Devices – Programmable Logic Array(PLA) – Programmable Array Logic (PAL) – Field Programmable Gate Arrays (FPGA) .

DIGITAL LOGIC FAMILIES

The switching characteristics of semiconductor devices have been discussed. Basically, there are two types of semiconductor devices: bipolar and unipolar. Based on these devices, digital integrated circuits have been made which are commercially available. Various digital functions are being fabricated in a variety of forms using bipolar and unipolar technologies. A group of compatible ICs with the same logic levels and supply voltages for performing various logic functions have been fabricated using a specific circuit configuration which is referred to as a logic family.

Classification of logic family
Bipolar Logic Families

The main elements of a bipolar IC are resistors, diodes (which are also capacitors) and transistors. Basically, there are two types of operations in bipolar ICs:

- Saturated, and Non-saturated.

In saturated logic, the transistors in the IC are driven to saturation, whereas in the case of non-saturated logic, the transistors are not driven into saturation.

The saturated bipolar logic families are:

- Resistor–transistor logic (RTL)
- Direct–coupled transistor logic (DCTL)
- Integrated–injection logic (I L)
- Diode–transistor logic (DTL)
- High–threshold logic (HTL)
- Transistor-transistor logic (TTL)

The non-saturated bipolar logic families are:

- Schottky TTL, and
- Emitter-coupled logic (ECL).

Unipolar Logic Families

MOS devices are unipolar devices and only MOSFETs are employed in MOS logic circuits. The MOS logic families are:

- PMOS
- NMOS
- CMOS

While in PMOS only *p*-channel MOSFETs are used and in NMOS only *n*-channel MOSFETs are used, in complementary MOS (CMOS), both *p*- and *n*-channel MOSFETs are employed and are fabricated on the same silicon chip.

CHARACTERISTICS:

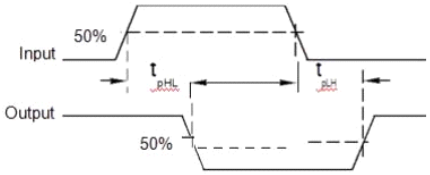
With the widespread use of ICs in digital systems and with the development of various tech-nologies for the fabrication of ICs, it has become necessary to be familiar with the characteris-tics of IC logic families and their relative advantages and disadvantages. Digital ICs are classi-fied either according to the complexity of the circuit, as the relative number of individual basic gates (2-input NAND gates) it would require to build the circuit to accomplish the same logic function or the number of components fabricated on the chip.

The various characteristics of digital ICs used to compare their performances are:

- Speed of operation
- Power dissipation
- Figure of merit
- Fan-out
- Current and voltage parameters
- Noise immunity
- Operating temperature range
- Power supply requirements
- Flexibilities available

1.Speed of Operation

The speed of a digital circuit is specified in terms of the propagation delay time. The input and output waveforms of a logic gate are shown in below Figure. The delay times are measured between the 50 per cent voltage levels of input and out-put waveforms. There are two delay times: t_{pHL} - when the output goes from the HIGH state to the LOW state and $t_{p\,LH}$ – corresponding to the output making a transition from the LOW state to the HIGH state. The propagation delay time of the logic gate



2.Power Dissipation

This is the amount of power dissipated in an IC. It is determined by the current, ICC , that it draws from the VCC supply, and is given by VCC * ICC . ICC is the average value of ICC (0) and ICC (1). This power is specified in milliwatts.

3.Figure of Merit

The figure of merit of a digital IC is defined as the product of speed and power. The speed is specified in terms of propagation delay time expressed in nanoseconds.

Figure of merit = propagation delay time (ns) *power (mW)

It is specified in pico joules (pJ) A low value of speed-power product is desirable. In a digital circuit, if it is desired to have high speed, i.e. low propagation delay, then there is a corresponding increase in the power dissipation and vice-versa.

4.**Fan-Out**

This is the number of similar gates which can be driven by a gate. High fan-out is advantageous because it reduces the need for additional drivers to drive more gates.

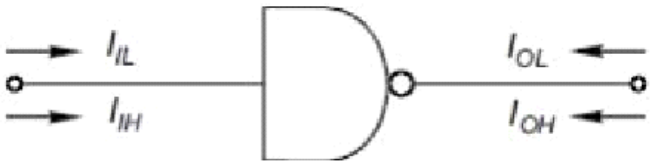
5. Current and Voltage Parameters

The following currents and voltages are specified which are very useful in the design of digital systems.

- High-level input voltage, VIH : This is the minimum input voltage which is recognized by the gate as logic 1.
- Low-level input voltage, VIL: This is the maximum input voltage which is recognized by the gate as logic 0.
- High-level output voltage, VOH : This is the minimum voltage available at the output corre-sponding to logic 1.
- Low-level output voltage, VOL: This is the maximum voltage available at the output correspond-ing to logic 0.
- High-level input current, IIH : This is the minimum current which must be supplied by a driving source corresponding to 1 level voltage.

- Low-level input current, IIL: This is the minimum current which must be supplied by a driving source corresponding to 0 level voltage.
- High-level output current, IOH : This is the maximum current which the gate can sink in 1 level.
- Low-level output current, IOL: This is the maximum cur- rent which the gate can sink in 0 level
- High-level supply current, ICC (1): This is the supply cur- rent when the output of the gate is at logic 1. Low-level supply current, ICC (0): This is the supply cur- rent when the output of the gate is at logic (0).

The current directions are illustrated in below Figure



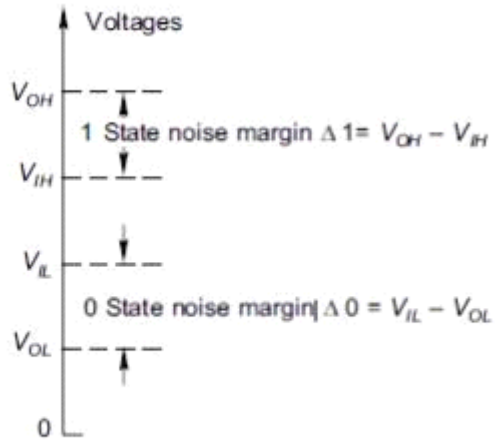
A gate with current directions marked.

6.Noise Immunity

The input and output voltage levels defined above are shown in Fig. Stray electric and magnetic fields may induce unwanted voltages, known as noise, on the connecting wires between logic circuits. This may cause the voltage at the input to a logic circuit to drop below VIH or rise above VIL and may produce undesired operation. The circuit's ability to tolerate noise signals is referred to as the noise immunity, a quantitative measure of which is called noise margin. Noise margins are illustrated in Fig. The noise margins defined above are referred to as dc noise margins. Strictly speaking, the noise is generally thought of as an a.c. signal with amplitude and pulse width. For high speed ICs, a pulse width of a few microseconds is extremely long in comparison to the propagation delay time of the circuit and therefore, may be treated as d.c. as far as the response of the logic circuit is concerned. As the noise pulse width decreases and approaches the propagation delay time of the circuit, the pulse duration is too short for the circuit to respond. Under this condition, a large pulse amplitude would be required to produce a change in the circuit output. This means that a logic circuit can effectively tolerate a large noise amplitude if the noise is of a very short duration. This is referred to as ac noise margin and is substantially greater than the dc noise margin. It is generally supplied by the manufacturers in the form of a curve between noise margin and noise pulse width.

7.Operating Temperature

The temperature range in which an IC functions properly must be known. The accepted temperature ranges are: 0 to + 70 °C for consumer and industrial applications and – 55 °C to + 125 °C for military purposes.



Voltage levels and noise margins of ICs.

8. Power Supply Requirements

The supply voltage(s) and the amount of power required by an IC are important characteristics required to choose the proper power supply.

9. Flexibilities Available

Various flexibilities are available in different IC logic families and these must be considered while selecting a logic family for a particular job. Some of the flexibilities available are:

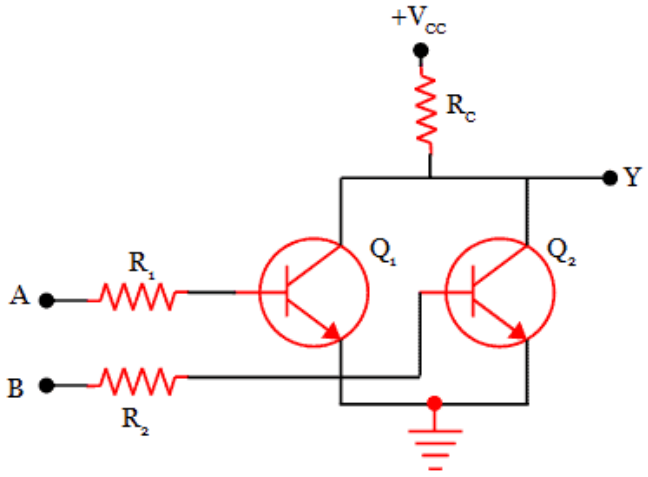
- *The breadth of the series:* Type of different logic functions available in the series.
- *Popularity of the series:* The cost of manufacturing depends upon the number of ICs manufactured. When a large number of ICs of one type are manufactured, the cost per function will be very small and it will be easily available because of multiple sources.
- *Wired-logic capability:* The outputs can be connected together to perform additional logic without any extra hardware.
- *Availability of complement outputs:* This eliminates the need for additional inverters. 5. *Type of output:* Passive pull-up, active pull-up, open-collector/drain, and tristate.

SATURATED BIPOLAR LOGIC FAMILIES

RESISTOR-TRANSISTOR LOGIC (RTL)

RTL consists of resistors and transistors. In RTL, transistors operate in cut-off region or saturation region as per the input voltage applied. The circuit of a two-inputs resistor-

transistor logic NOR gate is given below. Here A and B are the inputs of the gate and Y is the output.



2-input RTL NOR gate

Operation

- When the transistor operates in saturation region, maximum current flows through resistor *RC*. The output voltage *VY* = *V*CEsat (*V*CEsat = 0.2 V for silicon and 0.1 V for germanium); it is logic 0 level voltage. When the transistor operates in cut-off, no current flows through resistor *RC* and the output voltage *VY* = *V*CC = +5 V; it is logic 1 level voltage.
- When both the inputs are in logic 0, transistors *T*1 and *T*2 operate in cut-off, and the output is +*V*CC, i.e. +5 V (logic 1).
- When any one of the inputs is at logic 1 level, the corresponding transistor operates in saturation, and the output is *VY* = 0.2 V (logic 0).
- When both the inputs are at logic 1 level, both the transistors operate in saturation and the output is *VY* = 0.2 V (logic 0).

The operation of circuit is summarized in the below table

V_A	V_B	*Transistor T1*	*Transistor T2*	V_Y
Logic 0	Logic 0	Cut-off	Cut-off	Logic 1
Logic 0	Logic 1	Cut-off	Saturation	Logic 0
Logic 1	Logic 0	Saturation	Cut-off	Logic 0
Logic 1	Logic 1	Saturation	Saturation	Logic 0

In terms of 0 and 1, the above table can be written as in the below Table

Operation of RTL NOR gate

VA	*VB*	*VY*
0	0	1
0	1	0
1	0	0
1	1	0

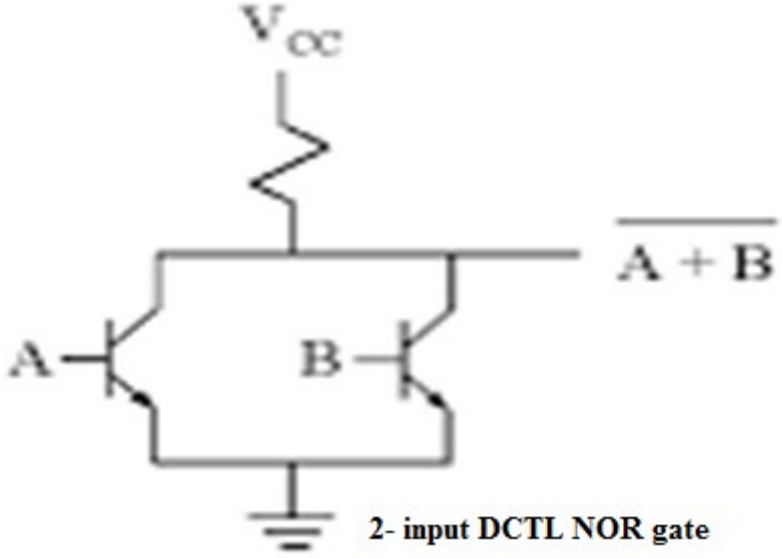
The circuit diagram acts as a two-inputs NOR gate and the above Table is the truth table of NOR gate.

The RTL suffers from a few drawbacks as listed below:

- Low noise margin (Typically 0.1 V)
- Fan-out is poor (Typically 5)
- Propagation delay is high and the speed of operation is low (Typically 12 ns)
- High power dissipation (Typically 12 mW)

DIRECT COUPLED TRANSISTOR LOGIC (DCTL)

In direct coupled transistor logic, the input signal is directly given to the base of the transistor. DCTL is simple than RTL. In DCTL, the transistor operates in saturation or cut-off region. The circuit of a two-inputs DCTL NOR gate is given below.



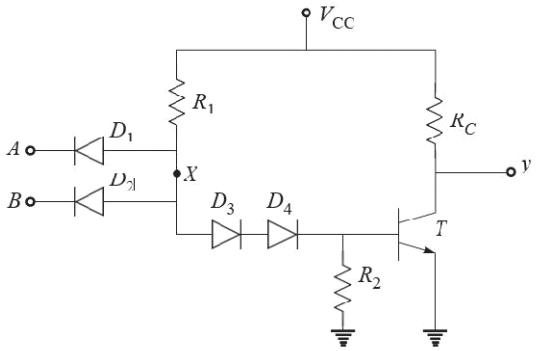
2- input DCTL NOR gate

Two-inputs DCTL NOR gate Operation:

The Operation of DCTL is same as that of RTL. When both the inputs are in logic 0,the transistors operate in cut-off, and the output is logic 1. When anyone of the inputs or both the inputs are in logic 1, the corresponding transistor or transistors operate in saturation and the output logic is 0. Although DCTL is simpler than RTL, it is not popular because of current hogging problem.

DIODE-TRANSISTOR LOGIC (DTL)

The circuit of a DTL consists of diodes and transistors. The circuit of a two-inputs diode-transistor logic NAND gate is shown below.



Two-inputs DTL NAND gate Operation

- When the transistor operates in saturation, the output voltage $V(0) = V\text{CEsat} = 0.2$ V; and when it operates in cut-off, the output voltage $V(1) = V\text{CC} = +5$ V.
- When both the inputs are in logic 0, $V(0) = V\text{CEsat} = 0.2$ V, the input diodes are forward biased, voltage at point x is $Vx = V(0) + VD = 0.2 + 0.7 = 0.9$ V which is not sufficient to drive the transistor in saturation, because the voltage desired at point x to drive the transistor in saturation should be $V\text{BEsat} + V\text{D4} + V\text{D3} = 0.8 + 0.7 + 0.9 = 2.2$ V. The transistor operates in cut-off and the output voltage is in logic 1 state.
- When any one of the inputs is in logic 1, the corresponding diode is forward biased. Voltage at point x is $Vx = 0.2$ V + 0.7 V = 0.9 V; the transistor operates in cut-off and the output voltage is in logic 1 state.
- When all the inputs are in logic 1 state, the diodes $D1$ and $D2$ are reverse biased. The resistances $R1$ and $R2$ are selected such that the transistor operates in saturation and the output voltage is in logic 0 state.

The operation of the circuit is summarized in Table below

Inputs		**Diodes**		**Transistor Output**	
A	**B**	**D1**	**D2**	**T**	**Y**
Logic 0	Logic 0	Forward biased	Forward biased	Cut-off	Logic 1
Logic 0	Logic 1	Forward biased	Reverse biased	Cut-off	Logic 1
Logic 1	Logic 0	Reverse biased	Forward biased	Cut-off	Logic 1
Logic 1	Logic 1	Reverse biased	Reverse biased	Saturation	Logic 0

In terms of 0 and 1, the above Table can be written as in Table below

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

Following are the advantages and disadvantages of DTL over RTL.

Advantages

- Fan-out is high
- Power dissipation is 8–12 mW
- Noise immunity is good

Disadvantages

- More elements are required
- Propagation delay is more (typically 30 ns) and hence the speed of operation is less

TRANSISTOR-TRANSISTOR LOGIC (TTL)

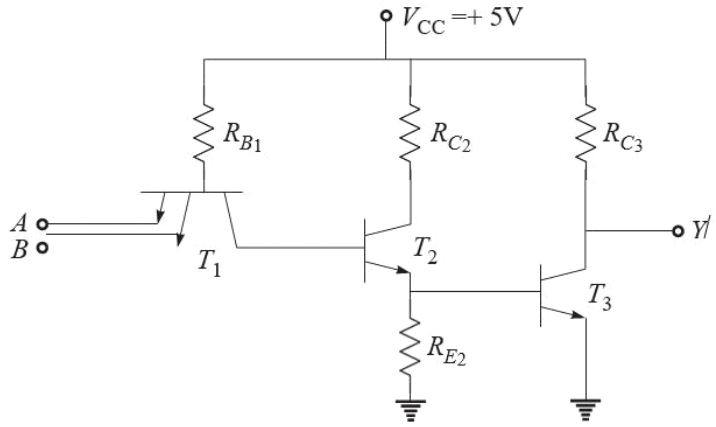
Transistor-transistor logic is one of the popular saturated logic families. Transistor is the basic element of this logic family, which operates either in cut-off or saturation region. The first version of TTL is known as the standard TTL.

Standard TTLs are available in various forms:

- TTL with passive pull-up
- TTL with totem-pole output
- TTL with open collector output
- Tristate TTL

TTL with Passive Pull-Up

Below diagram represents a two-input TTL NAND gate with passive pull-up. Transistor *T*1 has two emitter terminals. These terminals act as the inputs of the gate, that is, input *A* and input *B*. The input voltages are logic 0 or logic 1, where logic 0 corresponds to 0.2 V and logic 1 correspond to +5 V.



Two-input TTL NAND gate with passive pull-up Operation

- When both the inputs (*A* and *B*) are in logic 0, *V*(0) = *V*CEsat = 0.2 V, the emitter junctions of transistor *T*1 are forward biased and the voltage at the base of transistor *T*1 is *V*B1 = *V*(0) + *V*BE = 0.2 + 0.7 = 0.9 V. The minimum voltage required at the base of *T*1, so that *T*2 and *T*3 start to conduct, is *V*BEcut (in) + *V*BEcut (in) + 0.7 = 0.5 + 0.5 + 0.7 = 1.7 V. The required voltage is greater than the voltage available at the base of *T*1 and hence *T*2 and *T*3 are in cut-off and the output voltage is equal to the supply voltage *V*CC (logic 1 level), output is in logic 1 state.
- When any one of the inputs is at logic 0 level, the corresponding emitter junction of *T*1 is forward biased and the voltage at the base of *T*1 is *V*B1 = *V*(0) + *V*BE = 0.2 + 0.7 = 0.9 V. The minimum voltage required at the base of *T*1, so that *T*2 and *T*3 start to conduct, is *V*BEcut (in) + *V*BEcut (in) + 0.7 = 0.5 + 0.5 + 0.7 = 1.7 V. The required voltage is greater than the voltage available at the base of *T*1 and hence *T*2 and *T*3 are in cut-off and the output voltage is equal to the supply voltage *V*CC, output is in logic 1 state.
- When all the inputs are in logic 1 state, the emitter junctions of *T*1 are reverse biased and the current supply by the source is sufficient to operate *T*2 and *T*3 in saturation and the output is in logic 0 state.

Operation of TTL NAND gate

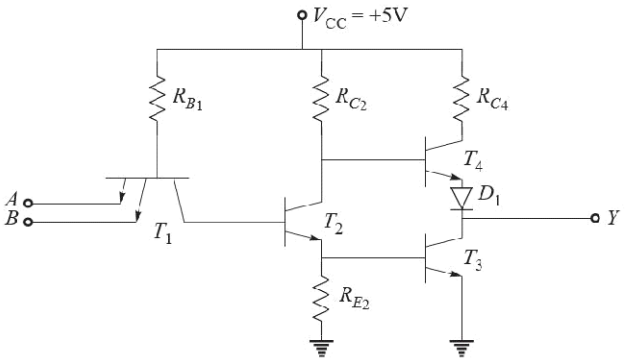
Inputs		**Transistors T1**		**Output**	
A	**B**	**Junction A**	**Junction B**	**T2, T3**	**Y**
Logic 0	Logic 0	Forward biased	Forward biased	Cut-off	Logic 1
Logic 0	Logic 1	Forward biased	Reverse biased	Cut-off	Logic 1
Logic 1	Logic 0	Reverse biased	Forward biased	Cut-off	Logic 1
Logic 1	Logic 1	Reverse biased	Reverse biased	Saturation	Logic 0

In terms of 0 and 1, Table above can be written as follows:

VA	***VB***	***VY***
0	0	1
0	1	1
1	0	1
1	1	0

TTL with Totem-Pole Output

The diagram below shows the circuit of a two-input TTL NAND gate with totem-pole output. It is possible in TTL to improve the speed of operation by reducing the time constant without increasing power dissipation with the help of active pull- up. TTL with active pull-up is known as TTL with totem-pole output.



Two-input TTL NAND gate with totem-pole output Operation

When both the inputs are in logic 0, $V(0) = V\text{CEsat} = 0.2$ V, emitter junctions of $T1$ are forward biased and the voltage at the base of $T1$ is $VB1 = V(0) + V\text{BE} = 0.2 + 0.7 = 0.9$ V. The minimum voltage required at the base of $T1$, so that $T2$ and $T3$ start to conduct, is $V\text{BEcut (in)} + V\text{BEcut (in)} + 0.7 = 0.5 + 0.5 + 0.7 = 1.7$ V. The required voltage is greater than the voltage available at the base of $T1$ and hence $T2$ and $T3$ are in cut-off and the output voltage is equal to the supply voltage $V\text{CC}$ (logic 1 level), output is in logic 1 state. Since $T2$ is in cut-off region, the current supply by the source $V\text{CC}$ through RC_2 is sufficient to operate $T4$ in saturation.

- When any one of the inputs is at logic 0 level, the corresponding emitter junction of $T1$ is forward biased and the voltage at the base of $T1$ is $VB1 = V(0) + V\text{BE} = 0.2 + 0.7 = 0.9$ V. The minimum voltage required at the base of $T1$, so that $T2$ and $T3$ start to conduct, is $V\text{BEcut (in)} + V\text{BEcut (in)} + 0.7 = 0.5 + 0.5 + 0.7 = 1.7$ V. The required voltage is greater than the voltage available at the base of $T1$ and hence $T2$ and $T3$ are in cut-off and the output voltage is equal to the supply voltage $V\text{CC}$ (logic 1 level), output is in logic 1 state. Since $T2$ is in cut-off region, the current supply by the source $V\text{CC}$ through RC_2 is sufficient to operate $T4$ in saturation.
- When all the inputs are in logic 1 state, the emitter junctions of $T1$ are reverse biased and the current supply by the source is sufficient to operate $T2$ and $T3$ in saturation and the output is logic 0 state. Since $T2$ is in saturation region, the voltage at the collector of $T2$ is low and $T4$ operates in cut-off.

Operation of TTL NAND gate

Inputs		Transistors T1		Output		
A	**B**	**Junction A**	**Junction B**	**T2, T3**	**T4**	**Y**
Logic 0	Logic 0	Forward biased	Forward biased	Cut-off	Saturation	Logic 1
Logic 0	Logic 1	Forward biased	Reverse biased	Cut-off	Saturation	Logic 1
Logic 1	Logic 0	Reverse biased	Forward biased	Cut-off	Saturation	Logic 1
Logic 1	Logic 1	Reverse biased	Reverse biased	Saturation	Cut-off	Logic 0

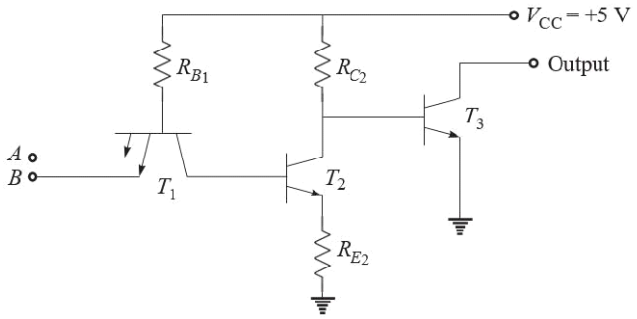
In terms of 0 and 1, Table Above can be written as in Table below.

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

The circuit shown above acts as a two-input NAND gate and its truth table is given in the above Table.

TTL with Open Collector Output

TTL with totem-pole output has a major problem that the two outputs of the two gates cannot be connected together. This problem of TTL with totem-pole output is overcome in TTL with open collector output. Figure below shows the circuit of a TTL NAND gate with open collector output.

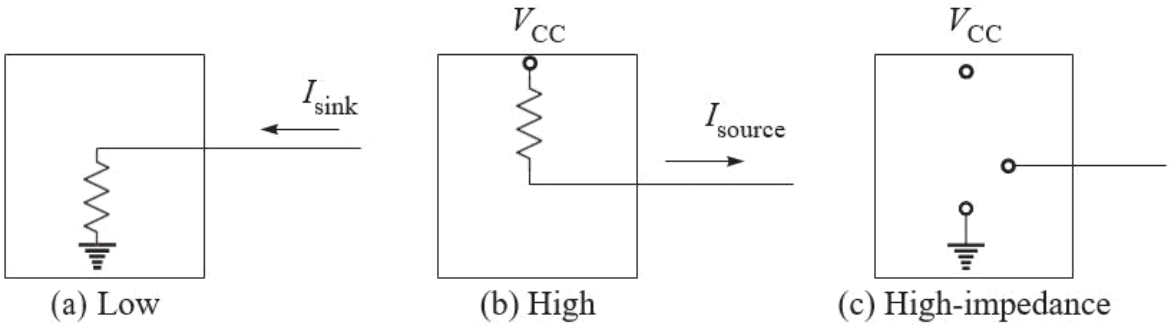


TTL NAND gate with open collector output

The collector terminal of $T3$ is available outside the IC where the external resistor is to be connected. The circuit acts as a TTL with passive pull-up and hence the advantages of active pull-up cannot be achieved in the circuit but wired-AND connection is possible.

Tri-state TTL

A normal digital circuit has two output states: Low and High. The output is either in high state or low state. If the output is not in the low state, it is definitely in the high state. The tri-state TTL has three output states: High, Low, and High- impedance. In TTL with totem-pole output, $T3$ is ON when the output is low and $T4$ is ON when the output is high. In high-impedance state, both $T3$ and $T4$ in totem-pole arrangement are turned OFF and as a result, the output is open or floating. When the output is low, the driver gate sinks the load current as shown in below Fig a. When the output is high, the driver gate supplies the current to the load as shown in Fig b below. When the output is in high-impedance state, it acts as open or floating and there is no sink and source current as shown in Fig c below.



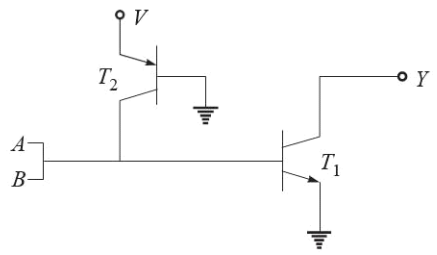
Tri-state Logic

INTEGRATED INJECTION LOGIC (I^2L)

The integrated injection logic uses only transistors for the construction of a gate and hence it becomes possible to integrate a large number of gates in a single package. This IC is easier and cheaper to fabricate. The figure of merit of I^2L circuits is quite small (4 PJ).

I^2L NAND Gate

Figure below shows the I^2L NAND gate. When inputs A and B are low or any one of the inputs is low, the current provided by $T2$ is sinked by the source, $T1$ is OFF, and the output is high. When both the inputs are high, the base current of $T1$ is the sum of currents provided by the source and $T2$, transistor $T1$ is ON and the output is low.

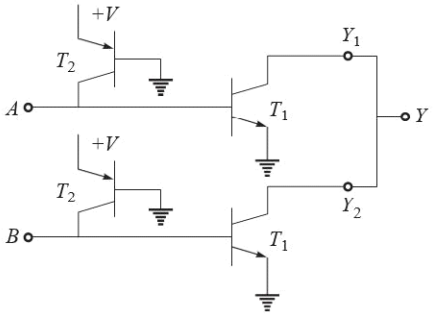


Truth Table for NOR gate

Inputs		*Output*
A	***B***	***Y***
0	0	1
0	1	1
1	1	1
1	1	0

I^2L NOR Gate

Figure below shows the I^2L NOR gate. The circuit has two inverters with their outputs connected together. When both or any one of the inputs is high, the output of the corresponding inverter is low and the resulting output is low. When both inputs are low, the output of both the inverters is high and the result is also high.

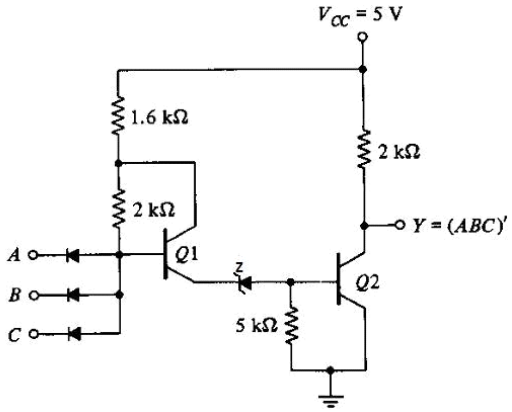


Truth Table for NOR gate

Inputs		**Output**		
A	**B**	**Y1**	**Y2**	**Y**
Logic 0	Logic 0	1	1	1
Logic 0	Logic 1	1	0	0
Logic 1	Logic 0	0	1	0
Logic 1	Logic 1	0	0	0

High Threshold Logic (**HTL**) is a variant of Diode–transistor logic which is used in such environments where noise is very high.

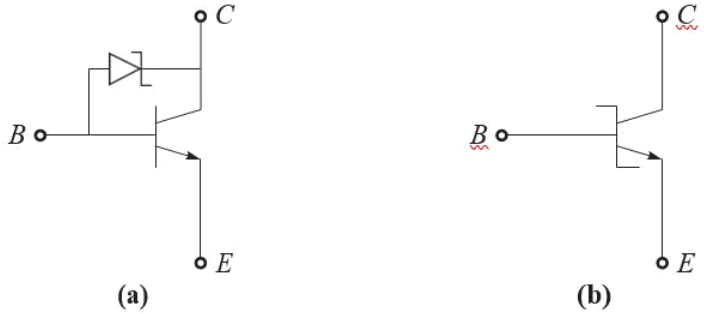
Operation : The threshold values at the input to a logic gate determine whether a particular input is interpreted as a logic 0 or a logic 1.(e.g. anything less than 1 V is a logic 0 and anything above 3 V is a logic 1. In this example, the threshold values are 1V and 3V). HTL incorporates Zener diodeto create a large offset between logic 1 and logic 0 voltage levels. These devices usually ran off a 15 V power supply and were found in industrial control, where the high differential was intended to minimize the effect of noise.



Non -Saturated bipolar logic families:

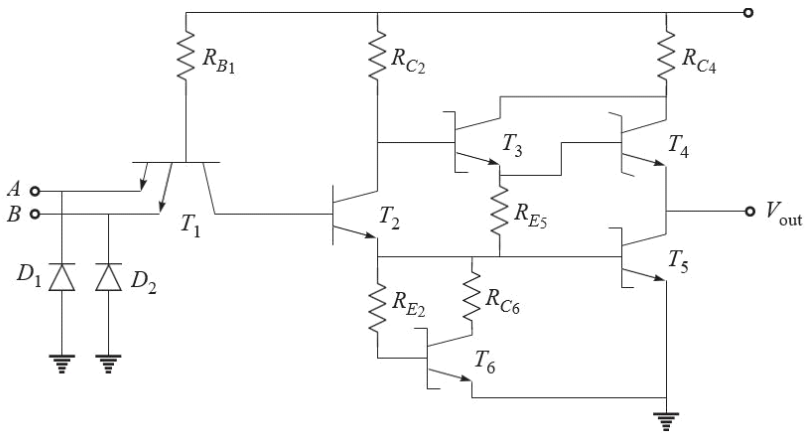
Schottky TTL

All the transistors in the circuits of standard TTL, low power TTL, and high speed TTL operate in saturation or cut-off region. When the transistor is in saturation, it stores the charge and the operation causes a storage-time delay during the transistor transition from ON to OFF; and this limits the circuit's switching speed. In Schottky TTL families, Schottky transistors are used instead of normal transistors. The Schottky transistor is operated in active region or cut-off region, it never goes into saturation and the storage time delay is negligible. The Schottky transistor is obtained by using a Schottky barrier diode between the base and the collector terminals of the transistor as shown in Fig.(a). The Schottky diode has a forward biased voltage of 0.25V. Because of this diode connected between the base and the collector terminals of the transistor, the collector junction of the transistor cannot get forward biased and the transistor never goes in saturation; the transistor operates in cut-off or active region. The symbol of Schottky transistor is shown in Fig.(b).



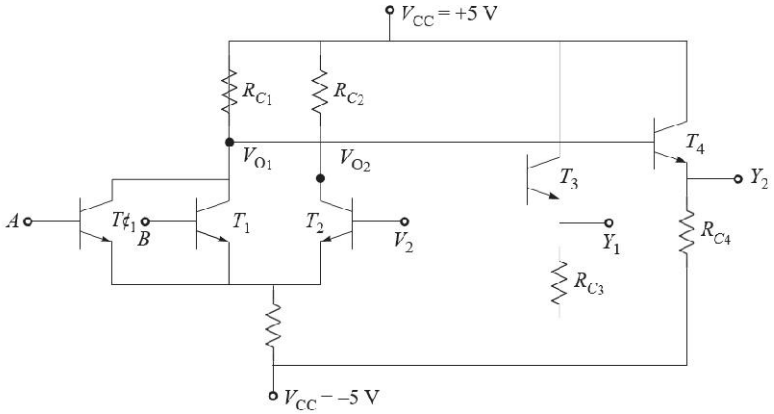
Schottky transistor and Schottky symbol

The 74S series is an example of Schottky TTL. The propagation delay of Schottky TTL is 3 ns only, which is twice as fast as the 74H series. Figure below shows a basic NAND gate in Schottky TTL series.



EMITTER COUPLED LOGIC

Emitter coupled logic (ECL) is faster than TTL family. The transistors of an emitter coupled logic are operated in cut-off or active region, it never goes in saturation and therefore the storage time is eliminated. Emitter coupled logic family is an example of unsaturated logic family. Figure below shows the circuit of an emitter- coupled logic OR/NOR gate. The circuit consists of difference amplifiers and emitter followers. Emitter terminals of the two transistors are connected together and hence it is called as emitter coupled logic. The emitter followers are used at the output of difference amplifier to shift the DC level. The circuit has two outputs Y1 and Y2, which are complementary. Y1 corresponds to OR logic and Y2 corresponds to NOR logic.



Emitter coupled Logic OR/NOR gate Operation

The input voltage of T_2 is $V_2 = V_{ref} = -1.15$ V.

- When both the inputs are in logic 0, $T_1$and T_1operate in cut-off and T_2operates in active region, voltage V_{O_1} is high, T_3 is ON, and the output at Y_2 is logic 1, voltage V_{O_2} is low, T_4 operates in cut-off and the output at Y_1 is logic 0.
- When any one of the inputs is in logic 1 level, the corresponding transistors T_1 or T_1 are operated in active region and T_2 operates in cut-off, voltage V_{O_1} is low, T_3 operates in cut-off and Y_2 is logic 0, voltage V_{O_2} is high, T_4 operates in active region and Y_1 is logic 1.

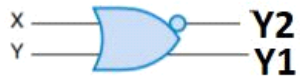
- When both the inputs are in logic 1 state, T_1 and T ρperate in active region and T_2 operates in cut-off, voltage V_{O_1} is low, T_3 operates in cut-off and Y_2 is logic 0, voltage V_{O_2} is high, T_4 operates in active region and Y_1 is logic 1.

The operation of the circuit is summarized in Table below.

Inputs		*Transistors*				*Output*	
A	***B***	***T1***	***T2***	***T3***	***T4***	***Y1***	***Y2***
Logic 0	Logic 0	Cut-off	Active	Active	Cut-off	Logic 0	Logic 1
Logic 0	Logic 1	Cut-off	Cut-off	Cut-off	Active	Logic 1	Logic 0
Logic 1	Logic 0	Activ	Cut-off	Cut-off	Active	Logic 1	Logic 0
Logic 1	Logic 1	Activ	Cut-off	Cut-off	Active	Logic 1	Logic 0

In terms of 0 and 1, Table above can be written as in Table below.

A	***B***	***Y1***	***Y2***
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0



Symbol of OR / NOR gate

The circuit shown in above Fig acts as a two-input OR/NOR gate and its truth table is given in Table above. Also, the logic symbol of emitter coupled logic OR / NOR gate is shown here.

Classification of memories

Introduction: Memory is a collection of cells capable of storing a large quantity of binary information. In to which binary information is transferred for storage and from which information is available when needed for processing.

Memory Device: Device to which binary information is transferred for storage, and from which information is available for processing as needed.

Memory Unit: Is a collection of cells capable of storing a large quantity of binary Information Computer memory is broadly divided into two groups and they are:

- Primary /main memory
- Secondary memory/External Memory

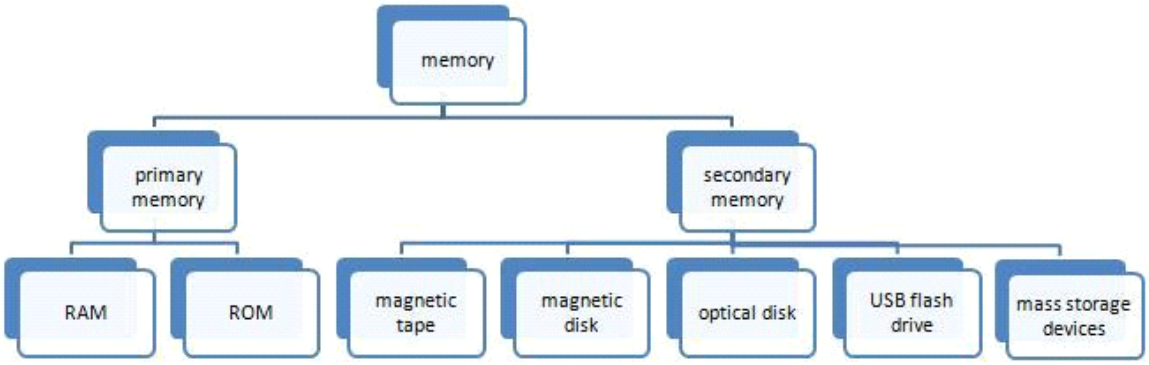


Fig Classification of Memory

Primary memory:

Primary memory is the only type of memory which is directly accessed by the CPU. The CPU continuously reads instructions stored in the primary memory and executes them. Any data that has to be operated by the CPU is also stored. The information is transferred to various locations through the BUS. Primary memories are of two types. They are:

- RAM
- ROM

RAM: It stands for Random Access Memory. Here data can be stored temporarily, so this type of memory is called as temporary memory or volatile memory because when power fails the data from RAM will be erased. The information stored in the RAM is basically loaded from the computer's disk and includes information related to the operating system and applications that are currently executed by the processor. RAM is considered random access because any memory cell can be directly accessed if its address is known. RAM is of distinct types like SRAM, DRAM, and VRAM.

ROM: It stands for Read Only Memory. In this, the data will be furnished by the manufacturers regarding the system, so this information can simply be read by the user but cannot add new data or it cannot be modified.

Types of ROM:

The required paths in a ROM may be programmed in four different ways.

- Mask programming
- Read-only memory (PROM)
- Erasable PROM (EPROM)
- Electrically-erasable PROM (EEPROM)

PROM **:** The PROM units contain all the fuses intact initially. Fuses are blown by application of a high voltage pulse to the device through a special pin by special instruments called PROM programmers. The program is once Written / programmed then it is irreversible. In A mask Programmable ROM, the data array is permanently stored during fabrication. This is done by selectively including switching element where a 1 is desired in

the data array. The designer of the circuit should provide the ROM program, which is simply the content of the storage array to the IC manufacture. Once the ROM is fabricated, the data array cannot be charged. Mask prorgammable ROMs are used when the ROM contents are not expected to change during the lifetime of the ROM.

EPROM: Floating gates served as programmed connections. When placed under ultraviolet light, short wave radiation discharges the gates and makes the EPROM returns to its initial state. It is reprogrammable. EPROMs use a special charge storage mechanism to enable or disable the switching element in the data array. A PROM programmer is used to store the charge at the selected switching elements while the EPROMs is programmed .The charge is retained by the EPROM. Thereby retaining the program until the EPROM is erased by using an ultraviolet light. Once erased, the EPROM can be reprogrammed. This type of ROM is useful in the early development phased of Digital circuit design, when it is often necessary to modify the data array

EEPROM: Erasable with an electrical signal instead of ultraviolet light. Longer time is needed to write flash ROM. It has limited times of write operations.

Random Access Memory (RAM)

The Random access memory, called "RAM" in short, is also known as the primary memory of the computer. RAM is considered as random access because of the fact that it can access any memory cell directly with the knowledge of the point of intersection of the row and column at that cell. It is also called as read write memory or the main memory or the primary memory. The programs and data that the CPU requires during execution of a program are stored in this memory. It is a volatile memory as the data loses when the power is turned off. RAM is further classified into two types- SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory).

Secondary memory: Secondary memory or auxiliary memory consists of slower and less expensive device that communicates indirectly with CPU via main memory. The secondary memory stores the data and keeps it even when the power fails. It is used to store or save large data or programs or other information. The secondary storage devices are explained below:

- Magnetic disks
- Magnetic tape
- Optical disk
- USB flash drive
- Mass storage devices

Magnetic disks: Magnetic disks are made of rigid metals or synthetic plastic material. The disk platter is coated on both the surfaces with magnetic material and both the surfaces can be used for storage. The magnetic disk furnishes direct access and is for both small and large computer systems. The magnetic disk comes in two forms: Floppy disks and Hard disks

Magnetic tape: magnetic tape is serial access storage medium and it can store a large volume of data at low costs. The conventional magnetic tape is in reels of up to 3600 feet made of Mylar plastic tape. The tape is one-half inch in width and is coated with magnetic material on one side. The reel of tape is loaded on a magnetic tape drive unit. During any read/write operation, the tape is moved from one spool to another in the same way as in the audiocassette tape recorder. The magnetic tape is densely packed with magnetic spots in frames across its width.

Optical drives: optical drives are a storage medium from which data is read and to which it is written by lasers. Optical disks can store much more data up to 6GB. Optical store devices are the most widely used and reliable storage devices. The most widely used type of optical storage devices are explained below:

- CD – ROM
- DVD – ROM
- CD – RECORDABLE
- CD – REWRITABLE
- PHOTO – CD

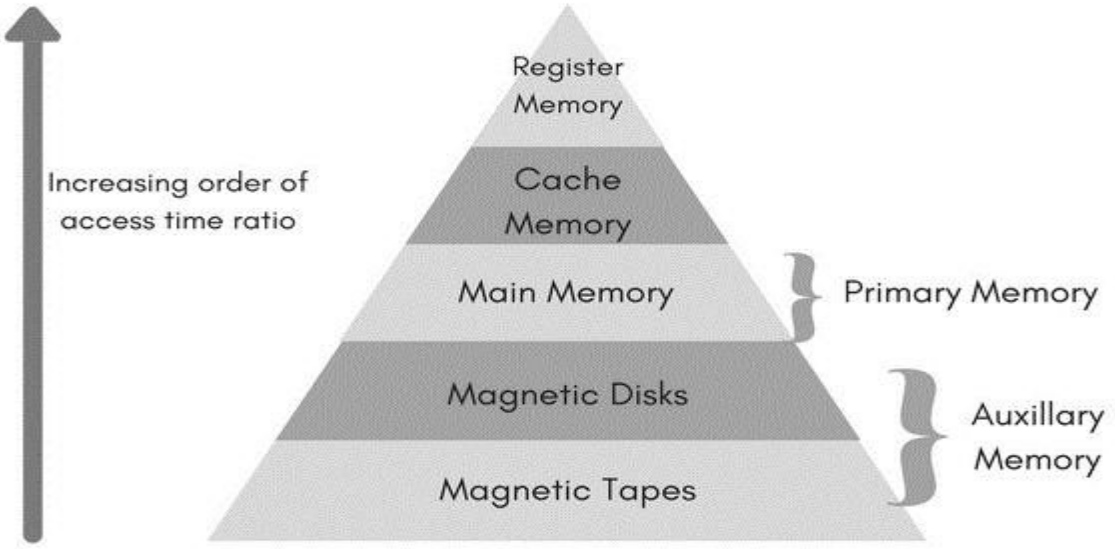
USB flash drives: USB flash drives are removable, rewritable and are physically much smaller drives, which have the weight of less than 30g. In the year of 2010, the storage capacity of the USB flash drives was as large as 256GB. Such devices are a good substitute for floppy disks and CD – ROMs as they are smaller, faster, have thousands of times more capacity, and are more durable and reliable. Until 2005, most desktop and laptop computers had floppy disk drives, but nowadays floppy disk drives have been abandoned in favor of USB ports. The USB connector is often protected inside a removable cap, although it is not likely to be damaged if unprotected. USB flash drives draw power from the computer through external USB connection. The most widely used USB flash drives are the memory cards.

Mass storage devices: Mass storage devices refer to the saving of huge data in a persistent manner. Mass storage machines can store up to several trillion bytes of data and hence are used to store or save large databases, such as the information of customers of a big retail chain and library transactions of students in a college. Some of the commonly used mass storage devices are explained below:

- Disk array
- Automated tape
- CD – ROM jukebox

Memory Hierarchy : The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices

contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.



Comparison between RAM and ROM:

S.No	RAM	ROM
1	RAMs have both read and write capability.	ROMs have only read operation.
2	RAMs are volatile memories.	ROMs are non-volatile memories.
3	They lose stored data when the power is turned OFF.	They retain stored data even if power is turned off.
4	RAMs are available in both bipolar and MOS technologies.	RAMs are available in both bipolar and MOS technologies.
5	**Types**: SRAM, DRAM, EEPROM	**Types**: PROM, EPROM.

Comparison between SRAM and DRAM:

S.No	Static RAM	Dynamic RAM
1	It contains less memory cells per unit area.	It contains more memory cells per unit area.
2	Its access time is less, hence faster memories.	Its access time is greater than static RAM

3	It consists of number of flip-flops. Each flip-flop stores one bit.	It stores the data as a charge on the capacitor. It consists of MOSFET and capacitor for each cell.
4	Refreshing circuitry is not required.	Refreshing circuitry is required to maintain the charge on the capacitors every time after every few milliseconds. Extra hardware is required to control refreshing.
5	Cost is more	Cost is less.

Comparison of Types of Memories:

Memory type	Non- Volatile	High Density	One- Transistor cell	In-system writ ability
SRAM	No	No	No	Yes
DRAM	No	Yes	Yes	Yes
ROM	Yes	Yes	Yes	No
EPROM	Yes	Yes	Yes	No
EEPROM	Yes	No	No	Yes

PROGRAMMABLE LOGIC DEVICES (PLDs)

INTRODUCTION:

A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation. The PLD‘s can be reprogrammed in few seconds and hence gives more flexibility to experiment with designs. Reprogramming feature of PLDs also makes it possible to accept changes/modifications in the previously design circuits.

The advantages of using programmable logic devices are:

- Reduced space requirements.
- Reduced power requirements.
- Design security.
- Compact circuitry.
- Short design cycle.
- Low development cost.

- Higher switching speed.
- Low production cost for large-quantity production.

According to architecture, complexity and flexibility in programming in PLD‘s are classified as

- PROMs : Programmable Read Only memories,
- PLAs : Programmable Logic Arrays,
- PAL : Programmable Logic Array,
- FPGA : Field Programmable Gate Arrays,
- CPLDs : Complex Programmable Logic Devices.

Programmable Arrays:

All PLDs consists of programmable arrays. A programmable array is essentially a grid of conductors that form rows and columns with a fusible link at each cross point. Arrays can be either fixed or programmable.

The OR Array:

It consists of an array of OR gates connected to a programmable matrix with fusible links at each cross point of a row and column, as shown in the figure below. The array can be programmed by blowing fuses to eliminate selected variables from the output functions. For each input to an OR gate, only one fuse is left intact in order to connect the desired variable to the gate input. Once the fuse is blown, it cannot be reconnected.
Another method of programming a PLD is the antifuse, which is the opposite of the fuse. Instead of a fusible link being broken or opened to program a variable, a normally open contact is shorted by ―melting‖ the antifuse material to form a connection.

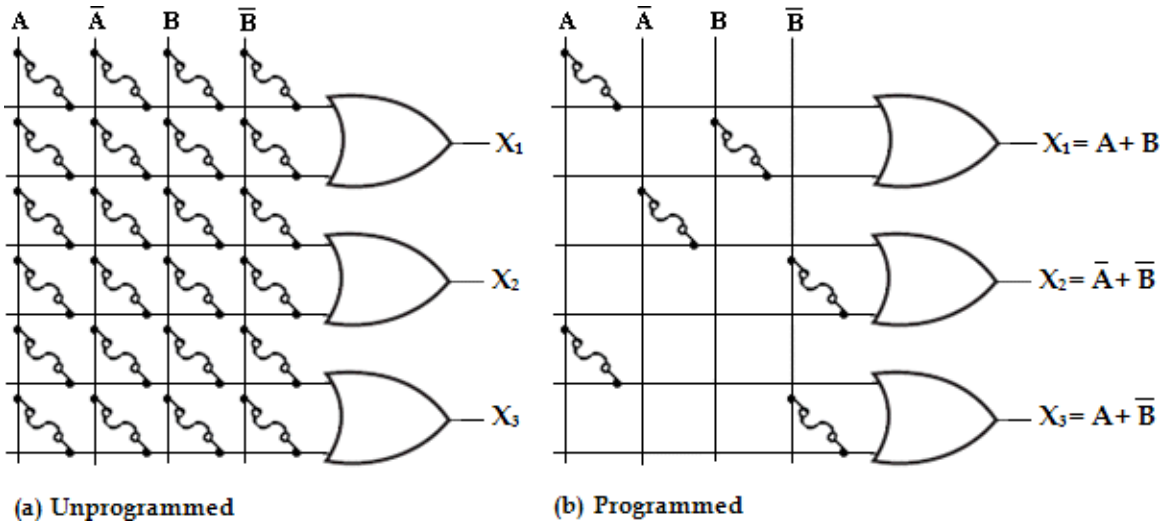


Fig. An example of a basic programmable OR array

The AND Array:

This type of array consists of AND gates connected to a programmable matrix with fusible links at each cross points, as shown in the figure below. Like the OR array, the AND array can be programmed by blowing fuses to eliminate selected variables from the output functions. For each input to an AND gate, only one fuse is left intact in order to connect the desired variable to the gate input. Also, like the OR array, the AND array with fusible links or with antifuses is one-time programmable.

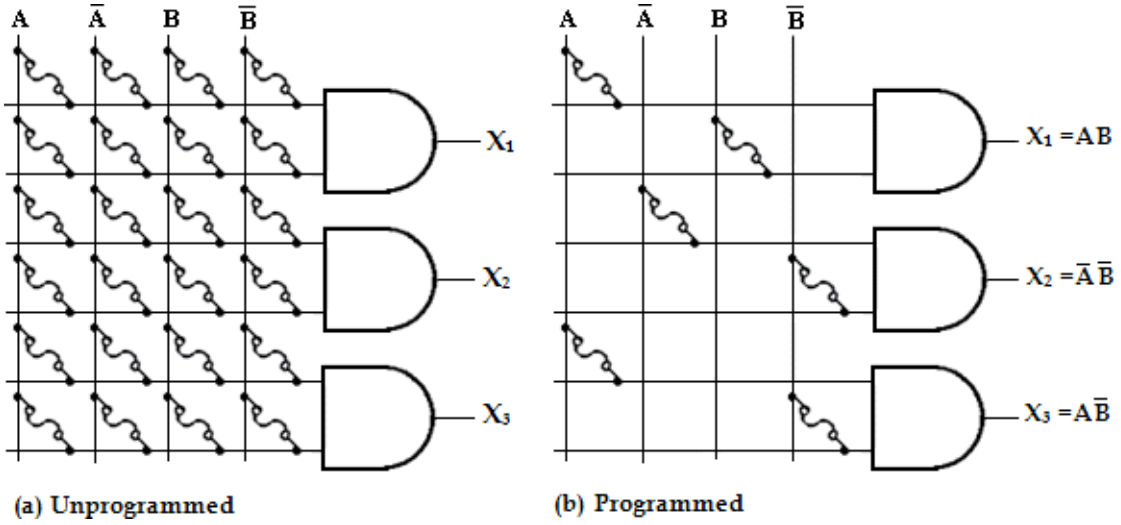


Fig. An example of a basic programmable AND array

Classification of PLDs

There are three major types of combinational PLDs and they differ in the placement of the programmable connections in the AND-OR array. The configuration of the three PLDs is shown below.

Programmable Read-Only Memory (PROM):

A PROM consists of a set of fixed (non-programmable) AND array constructed as a decoder and a programmable OR array. The programmable OR gates implement the Boolean functions in sum of minterms

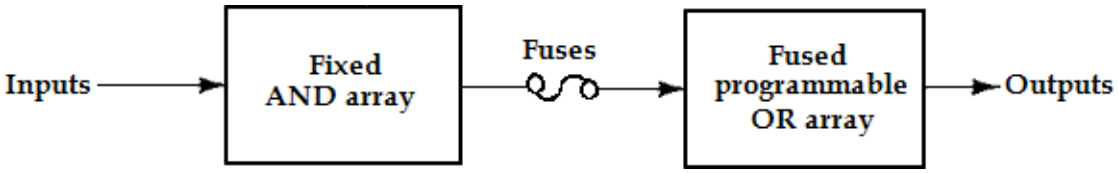


Fig. Programmable read- only memory (PROM)

.

Programmable Logic Array (PLA):

A PLA consists of a programmable AND array and a programmable OR array.
The product terms in the AND array may be shared by any OR gate to provide the required sum of product implementation. The PLA is developed to overcome some of the limitations of the PROM. The PLA is also called an FPLA (Field Programmable Logic Array) because the user in the field, not the manufacturer, programs it.

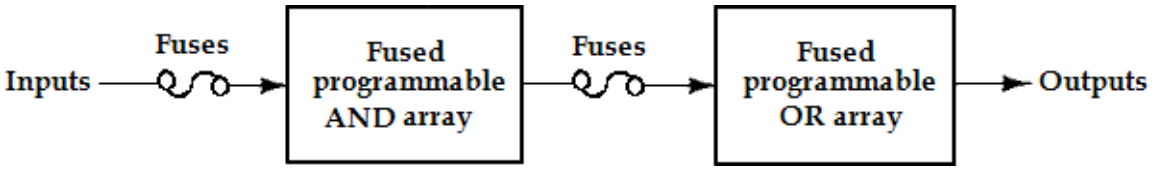


Fig. Programmable Logic Array (PLA)

Programmable Array Logic (PAL):

The basic PAL consists of a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate. It is developed to overcome certain disadvantages of the PLA, such as longer delays due to the additional fusible links that result from using two programmable arrays and more circuit complexity.

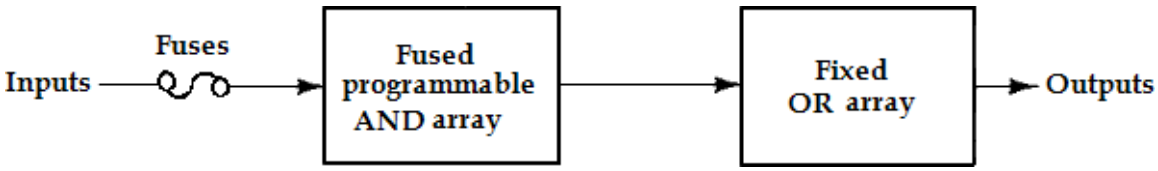
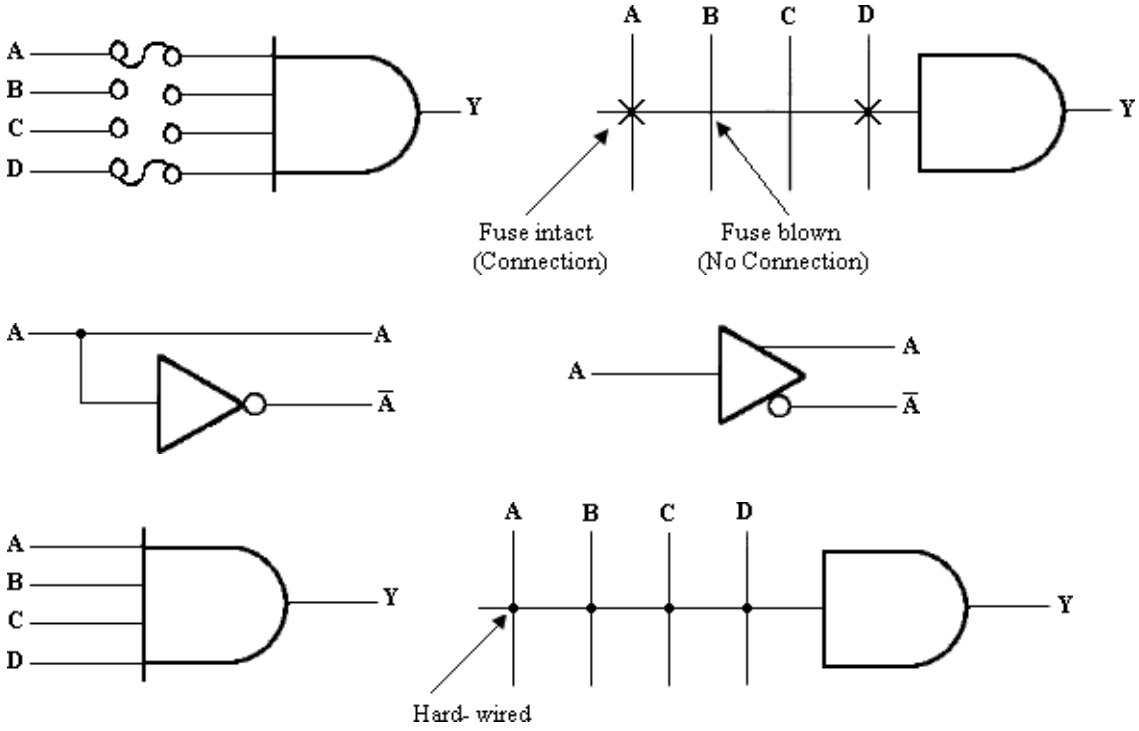


Fig. Programmable Array Logic (PAL)

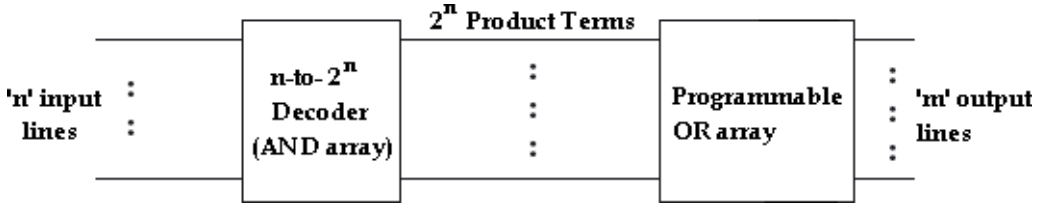
Array logic Symbols:

PLDs have hundreds of gates interconnected through hundreds of electronic fuses. It is sometimes convenient to draw the internal logic of such device in a compact form referred to as array logic.



PROGRAMMABLE ROM:

PROMs are used for code conversions, generating bit patterns for characters and as look-up tables for arithmetic functions. As a PLD, PROM consists of a fixed AND-array and a programmable OR array. The AND array is an n-to-2^n decoder and the OR array is simply a collection of programmable OR gates. The OR array is also called the memory array. The decoder serves as a minterm generator. The n-variable minterms appear on the 2n lines at the decoder output. The 2n outputs are connected to each of the _m' gates in the OR array via programmable fusible links.



2^n x m PROM

Implementation of Combinational Logic Circuit using PROM

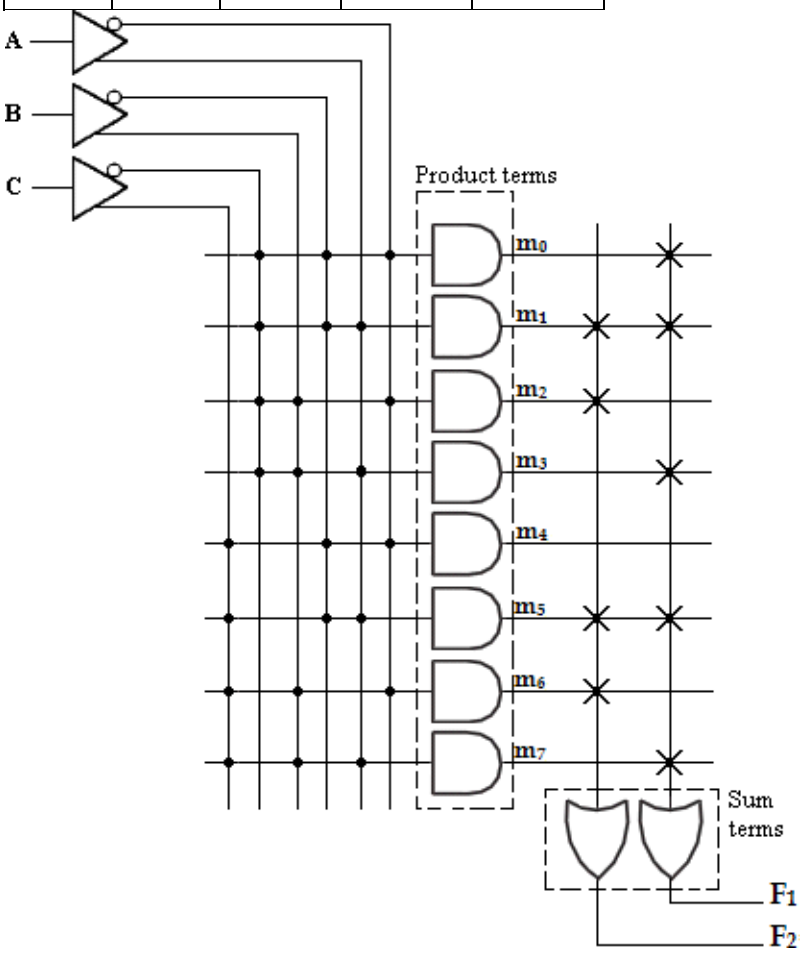
1.Using PROM realize the following expression

F_1 (A, B, C) = ∑m (0, 1, 3, 5, 7)

F_2 (A, B, C) = ∑m (1, 2, 5, 6)

Truth table for the given function

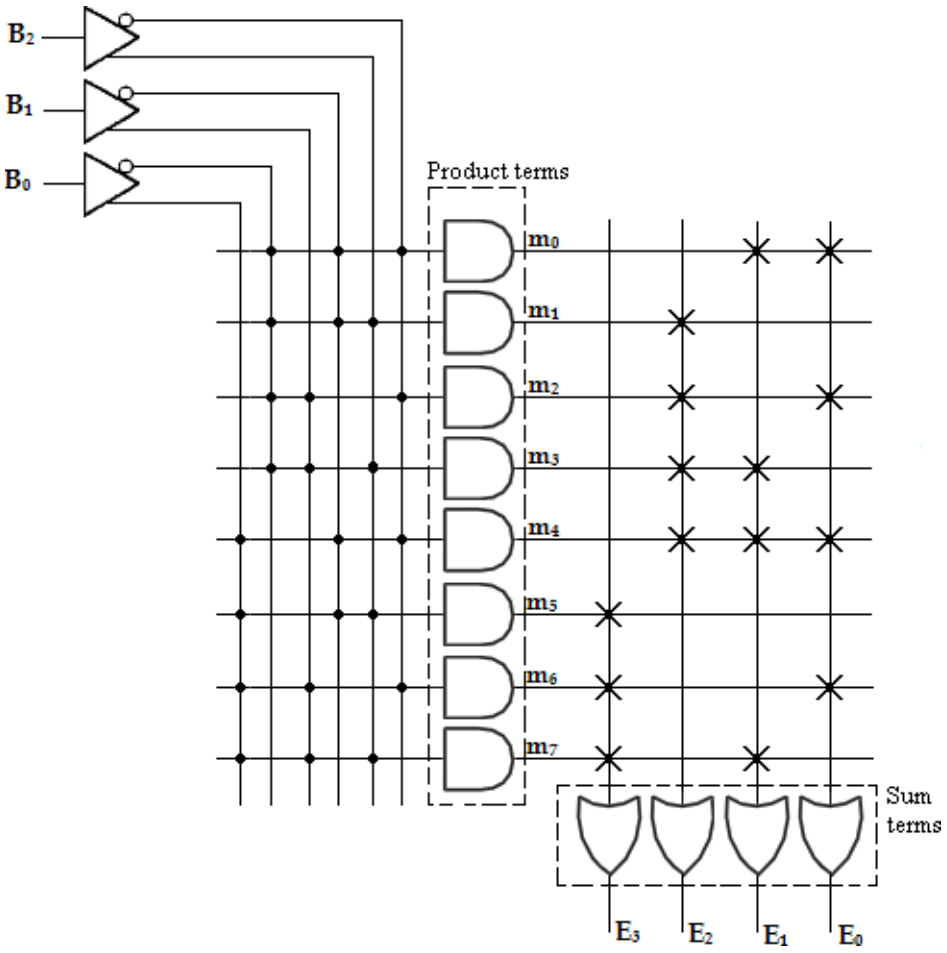
A	B	C	F_1	F_2
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0



2. Design a combinational circuit using PROM. The circuit accepts 3-bit binary and generates its equivalent Excess-3 code.

Truth table for the given function

B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	0



PROGRAMMABLE LOGIC ARRAY: (PLA)

The PLA is similar to the PROM in concept except that the PLA does not provide full coding of the variables and does not generate all the minterms.

The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product term are then connected to OR gates to provide the sum of products for the required Boolean functions. The AND gates and OR gates inside the PLA are initially fabricated with fuses among them. The specific boolean functions are implemented in sum of products form by blowing the appropriate fuses and leaving the desired connections.

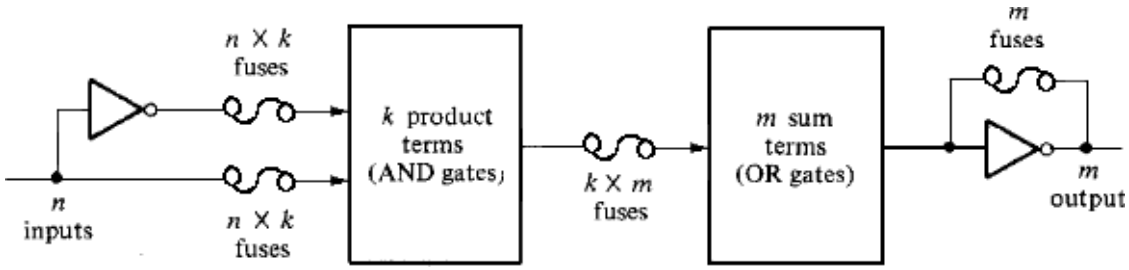


Fig. PLA block diagram

The block diagram of the PLA is shown above. It consists of _n' inputs, _m' outputs, _k' product terms and _m' sum terms. The product terms constitute a group of _k' AND gates and the sum terms constitute a group of _m' OR gates. Fuses are inserted between all _n' inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gate and the inputs of the OR gates. Another set of fuses in the output inverters allow the output function to be generated either in the AND-OR form or in the AND-OR-INVERT form. With the inverter fuse in place, the inverter is bypassed, giving an AND-OR implementation. With the fuse blown, the inverter becomes part of the circuit and the function is implemented in the AND-OR- INVERT form.

Implementation of Combinational Logic Circuit using PLA

1.Implement the combinational circuit with a PLA having 3 inputs, 4product terms and 2 outputs for the functions.

F_1 (A, B, C) = ∑m (0, 1, 2,4)

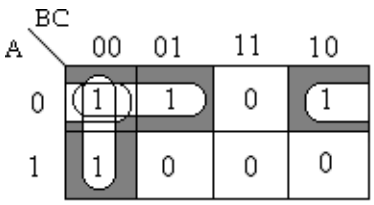
F_2 (A, B, C) = ∑m (0, 5, 6, 7)

Solution:

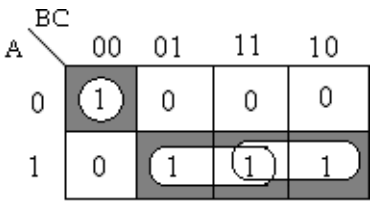
Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Step 2: K-map Simplification

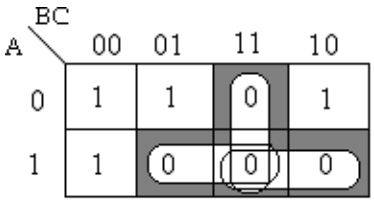


F_1= A′B′+ A′C′+ B′C′

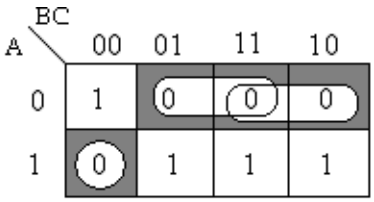


F_2= AC+ AB+ A′B′C′

With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F_1‘ and F2‘.



$F_1' = AC + BC + AB$



$F_2' = A'C + A'B + A'B'C'$

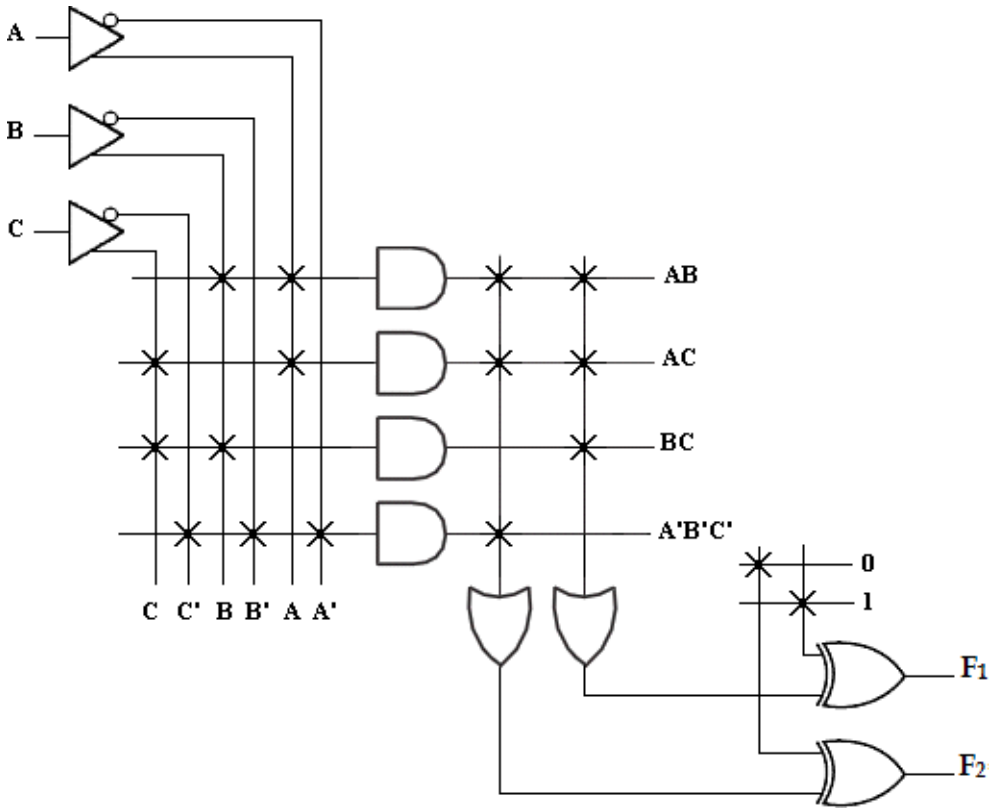
Now select, F_1‘ and F2, the product terms are AC, AB, BC and A‘B‘C‘

Step 3: PLA Program table:

	Product term	**Inputs**			**Outputs**	
		A	**B**	**C**	**F_1 (C)**	**F_2 (T)**
AB	1	1	1	-	1	1
AC	2	1	-	1	1	1
BC	3	-	1	1	1	-
A‘B‘C‘	4	0	0	0	-	1

In the PLA program table, first column lists the product terms numerically as 1, 2, 3, and 5. The second column (Inputs) specifies the required paths between the AND gates and the inputs. For each product term, the inputs are marked with 1, 0, or - (dash). If a variable in the product form appears in its normal form, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash (-). The third column (output) specifies the path between the AND gates and the OR gates. The output variables are marked with 1‘s for all those product terms that formulate the required function.

Step 4: PLA Diagram



The PLA diagram uses the array logic symbols for complex symbols. Each input and its complement is connected to the inputs of each AND gate as indicated by the intersections between the vertical and horizontal lines. The output of the AND gate are connected to the inputs of each OR gate. The output of the OR gate goes to an EX-OR gate where the other input can be programmed to receive a signal equal to either logic 1 or 0.

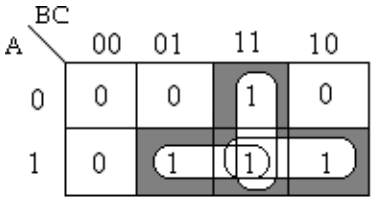
The output is inverted when the EX-OR input is connected to 1 ie., **(x** Ex-OR **1= x')**. The output does not change when the EX-OR input is connected to 0 ie., **(x** Ex – OR **0=**

2.Implement the combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions. $F_1 (A, B, C) = \sum m (3, 5, 6, 7)$, $F_2 (A, B, C) = \sum m (0, 2, 4, 7)$

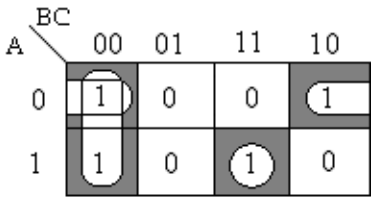
Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	0	1
0	0	1	0	0
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

Step 2: K-map Simplification

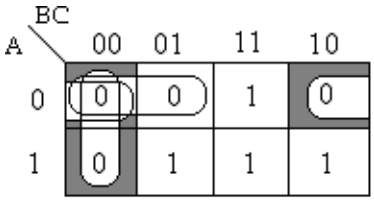


$F_1 = AC + AB + BC$

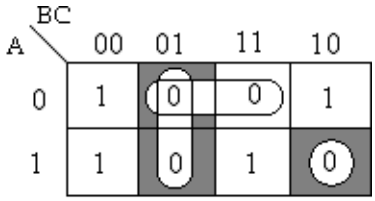


$F_2 = B'C' + A'C' + ABC$

With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F_1‘ and F2‘.



$F_1' = B'C' + A'C' + A'B'$



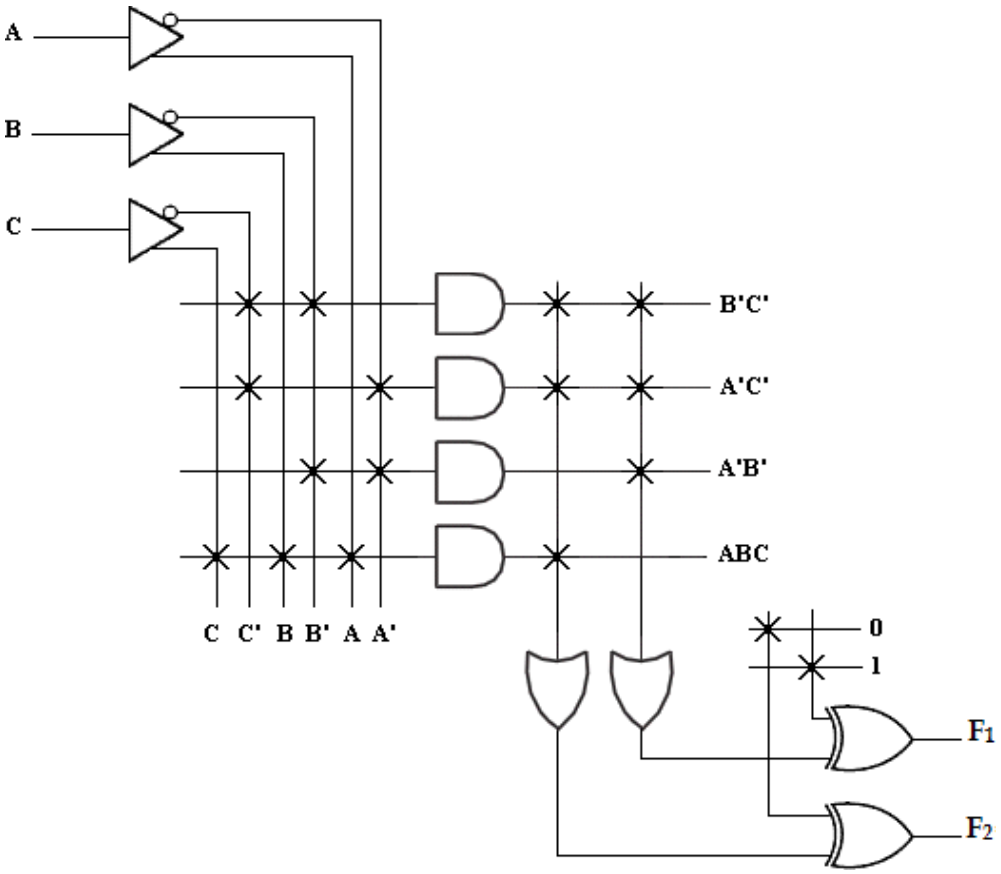
$F_2' = A'C + B'C + ABC'$

Now select, F_1‘ and F2, the product terms are **B’C’, A’C’, A’B’ and ABC**.

Step 3: PLA Program table

	Product term	**Inputs**			**Outputs**	
		A	**B**	**C**	**F_1 (C)**	**F_2 (T)**
B‘C‘	1	-	0	0	1	1
A‘C‘	2	0	-	0	1	1
A‘B‘	3	0	0	-	1	-
ABC	4	1	1	1	-	1

Step 4: PLA Diagram



3. Implement the following functions using PLA.

F_1 (A, B, C) = ∑m (1, 2, 4,6)

F_2 (A, B, C) = ∑m (0, 1, 6, 7)

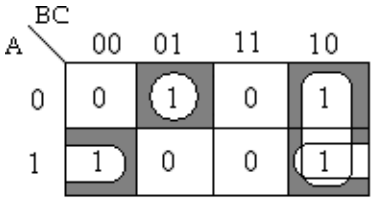
F_3 (A, B, C) = ∑m (2, 6)

Solution:

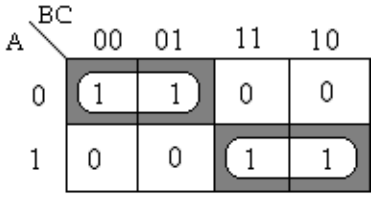
Step 1: Truth table for the given functions

A	B	C	F_1	F_2	F_3
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0

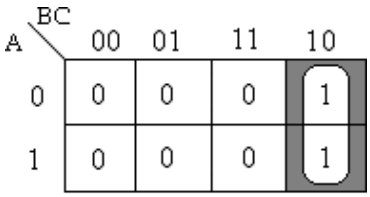
Step 2: K-map Simplification



$F_1 = A'B'C + AC' + BC'$



$F_2 = A'B' + AB$

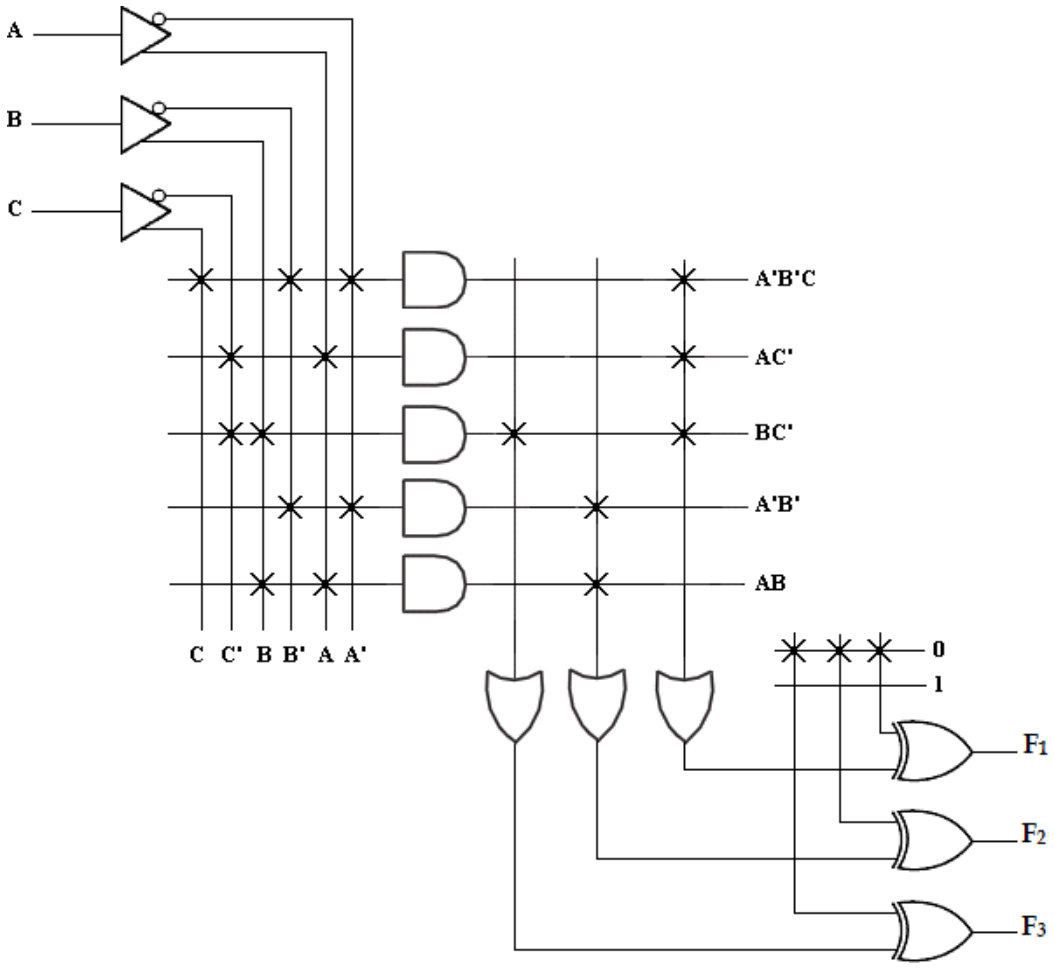


$F_3 = BC'$

Step 3: PLA Program table

	Product Term	Inputs			Outputs		
		A	B	C	F_1 (T)	F_2 (T)	F_3 (T)
A‘B‘C	1	0	0	1	1	-	-
AC‘	2	1	-	0	1	-	-
BC‘							
	3	-	1	0	1	-	1
A‘B‘							
	4	0	0	-	-	1	-
AB							
	5	1	1	-	-	1	-

Step 4: PLA Diagram



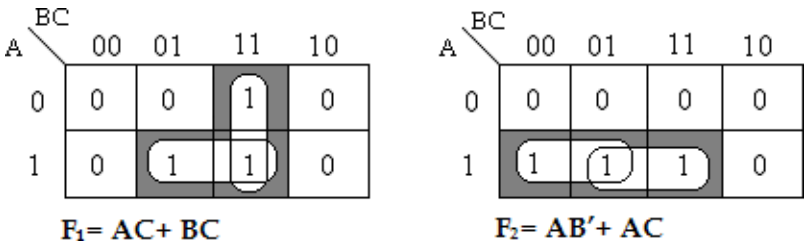
4. A combinational circuit is designed by the function F_1 (A, B, C) = ∑m (3, 5, 7)
F_2 (A, B, C) = ∑m (4, 5, 7)

Solution:

Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

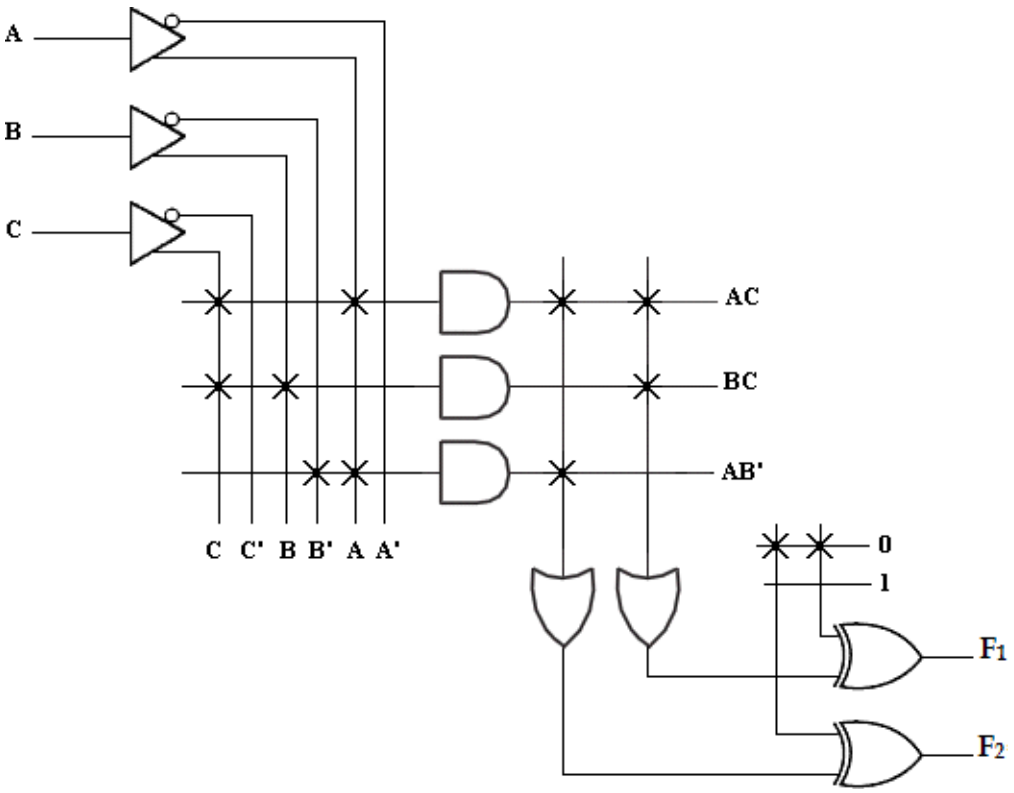
Step 2: K-map Simplification



Step 3: PLA Program table

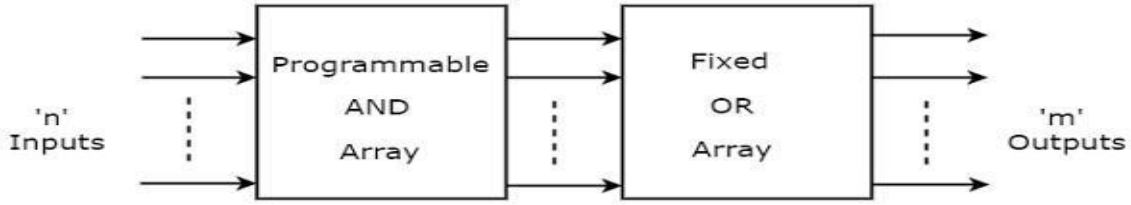
	Product term	Inputs			Outputs	
		A	B	C	F_1 (C)	F_2 (T)
AC BC AB‘	1	1	-	1	1	1
	2	-	1	1	1	-
	3	1	0	-	-	1

Step 4: PLA Diagram



Programmable Array Logic (PAL)

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The block diagram of PAL is shown in the following figure.



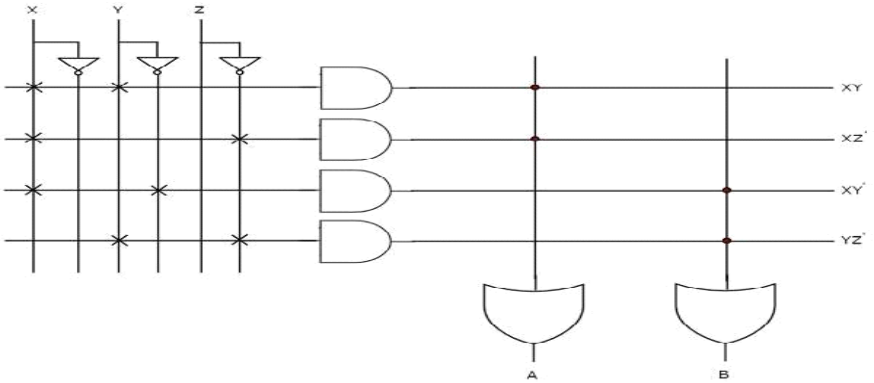
Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required product terms by using these AND gates. Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of sum of products form

Example

Let us implement the following Boolean functions using PAL.

A=XY+XZ′ ; B=XY′+YZ′

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



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

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

Comparison between PROM, PLA, and PAL:

S.No	PROM	PLA	PAL
1	AND array is fixed and OR array is programmable	Both AND and OR arrays are programmable	OR array is fixed and AND array is programmable
2	Cheaper and simpler to use	Costliest and complex	Cheaper and simpler
3	All minterms are decoded	AND array can be programmed to get desired minterms	AND array can be programmed to get desired minterms

Field Programmable Gate Arrays (FPGA)

A more advanced programmable logic than the Complex Programmable Logic Devices (CPLD) is the Field Programmable Gate Array (FPGA). CPLDs are created that consist of multiple PLDs with programmable wiring channels between the PLDs. An FPGA is more flexible than CPLD, allows more complex logic implementations, and can be used for implementation of digital circuits that use equivalent of several Million logic gates. An FPGA is like a CPLD except that its logic blocks that are linked by wiring channels are much smaller than those of a CPLD and there are far more such logic blocks than there are in a CPLD. FPGA logic blocks consist of smaller logic elements. A logic element has only one flip-flop that is individually configured and controlled. Logic complexity of a logic element is only about 10 to 20 equivalent gates. A further enhancement in the structure of FPGAs is the addition of memory blocks that can be configured as a general purpose RAM.

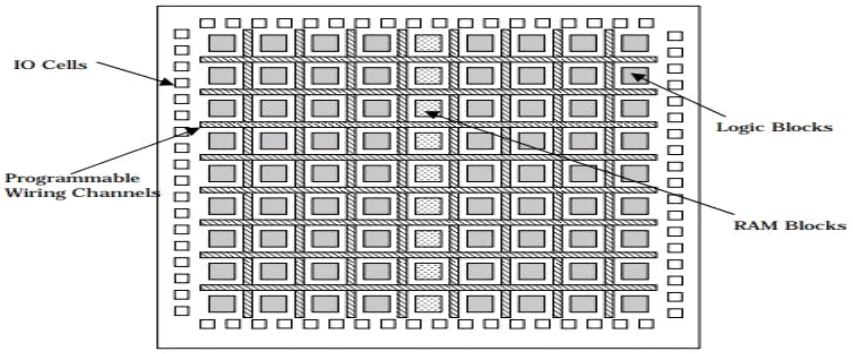


Fig FPGA general structure

An FPGA is an array of many logic blocks that are linked by horizontal and vertical wiring channels. FPGA RAM blocks can also be used for logic implementation or they can

be configured to form memories of various word sizes and address space. Linking of logic blocks with the I/O cells and with the memories are done through wiring channels. Within logic blocks, smaller logic elements are linked by local wires. IO Cells Programmable Wiring Channels Logic Blocks RAM Blocks 129 FPGAs from different manufacturers vary in routing mechanisms, logic blocks, memories and I/O pin capabilities.

TEXT BOOKS:

1. Morris Mano, “Digital design”, 3rd Edition, Prentice Hall of India, 2008.

REFERENCE BOOKS:

1. Milos Ercegovac, Jomas Lang, “Introduction to Digital Systems”, Wiley publications, 1998.
2. John M. Yarbrough, “Digital logic: Applications and Design”, Thomas – Vikas Publishing House, 2002.
3. R.P.Jain, “Modern digital Electronics”,3rd Edition, TMH, 2003.
4. William H. Gothmann, “Digital Electronics”, Prentice Hall, 2001.

QUESTION BANK

PART-A

1. Define fan-out?
2. Illustrate the basic TTL NAND gate.
3. Define CMOS.
4. List the application of ROM.
5. Distinguish between ROM and RAM.
6. Recall flash memory?
7. Name the digital IC families?
8. Discuss EEPROM and its advantages.
9. Distinguish PLA and PAL.
10. List the advantages of ECL logic families
11. Define FPGA.

PART-B

1. Discuss the impact of characteristics of a logic family in choosing a suitable family.
2. RTL,DTL can be used for designing a logic gate, Justify the answer with suitable circuit diagram
3. Explain about MOS and CMOS technologies for the design of logic circuits, Illustrate your answer with suitable diagram.
4. Discuss the TTL and ECL families be used for the design of logic circuits.
5. Explain briefly about the bipolar transistor characteristics.
6. Distinguish the following memories.
    a. RAM and ROM
    b. ROM and EEPROM
    c. EPROM and EEPROM
    d. SRAM and DRAM