Unit-3
Combinational Logic Design

Unit-3
Combinational Logic Design

Logic Gates

- A logic gate is an electronic device that produces **a result** based on one or more input values.
- In reality, gates consist of one to six **transistors**, but digital designers think of them as a single unit.
- Integrated circuits contain collections of gates suited to a particular purpose.

Logic gate can be categories as follows:

1. Basic Gate

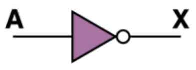
a. NOT gate

A NOT gate accepts one input value and produces one output value.

Boolean Expression

$X = A'$

Logic Diagram Symbol



Truth Table

A	X
0	1
1	0

Fig: Various representations of a NOT gate

By definition, if the input value for a NOT gate is 0, the output value is 1, and if the input value is 1, the output is 0.

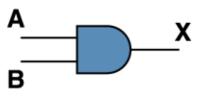
b. AND gate

An AND gate accepts two input signals. If the two input values for an AND gate are both 1, the output is 1; otherwise, the output is 0.

Boolean Expression

$X = A \cdot B$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Fig: Various representations of a AND gate

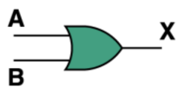
c. OR gate

If the two input values are both 0, the output value is 0; otherwise, the output is 1.

Boolean Expression

$X = A + B$

Logic Diagram Symbol



Truth Table

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Fig: Various representations of a OR gate

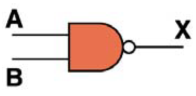
2. Universal Gate

a. ***NAND gate***

NAND gate is the combination of NOT gate and AND gate. If the two input values for an NAND gate are both 1, the output is 0; otherwise, the output is 1.

Boolean Expression **Logic Diagram Symbol** **Truth Table**

$X = (A \cdot B)'$



A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

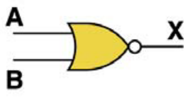
Fig: Various representations of a NAND gate

b. ***NOR gate***

NOR gate is the combination of NOT gate and OR gate. If the two input values for NOR gate are both 0, the output value is 1; otherwise, the output is 0.

Boolean Expression **Logic Diagram Symbol** **Truth Table**

$X = (A + B)'$



A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Fig: Various representations of a NOR gate

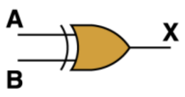
3. Derived/Extended Gate

a. ***Exclusive-OR gate (XOR)***

An XOR gate produces 0 if its two inputs are the same, and a 1 otherwise.

Boolean Expression **Logic Diagram Symbol** **Truth Table**

$$X = A \oplus B$$
$$= AB' + A'B$$

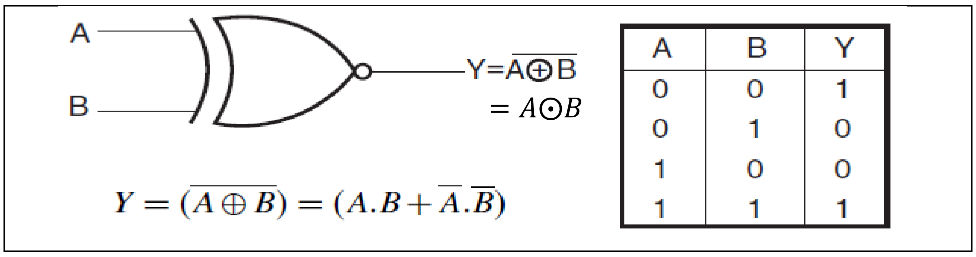


A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Fig: Various representations of a XOR gate

b. ***Exclusive-NOR gate (X-NOR)***

X-NOR is the complement of X-OR. An X-NOR gate produces 1 if its two inputs are the same, and a 0 otherwise.



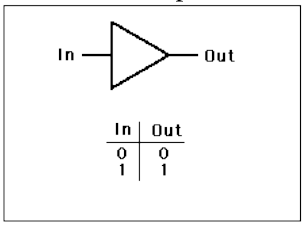
$$Y = (\overline{A \oplus B}) = (A.B + \overline{A}.\overline{B})$$

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

Fig: Various representations of a X-NOR gate

Buffer Gate

The buffer gate returns the same output as same as that of input.



In	Out
0	0
1	1

Universal Gate Realization

A universal gate is a gate which can implement any Boolean function without need to use any other gate type. The **NAND** and **NOR** gates are universal gates.

1. ***NAND gate as a Universal Gate***
   To prove that any Boolean function can be implemented using only NAND gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

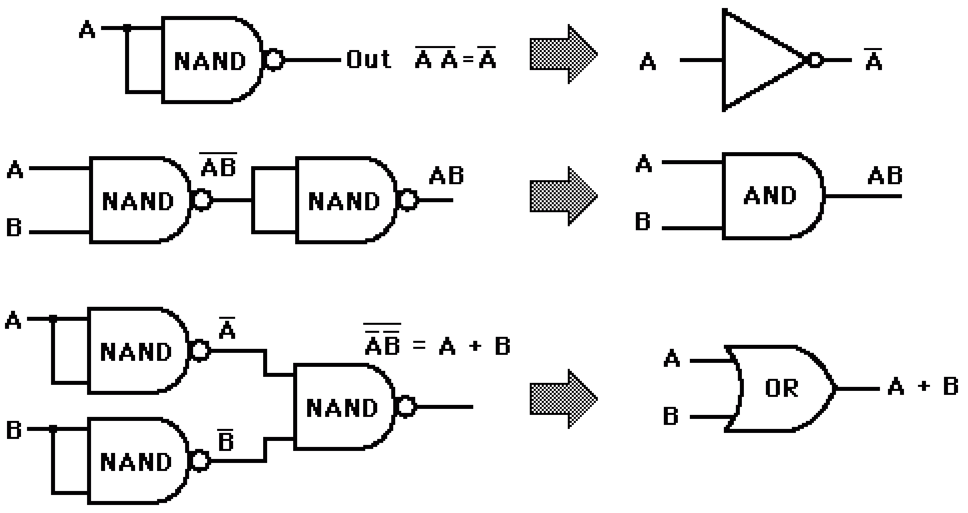


Fig: Three Circuits Constructed Using Only NAND Gates

Thus, the **NAND gate** is a universal gate since it can implement the AND, OR and NOT functions.

2. ***NOR gate as a Universal Gate***
   To prove that any Boolean function can be implemented using only NOR gates, we will show that the AND, OR, and NOT operations can be performed using only these gates.

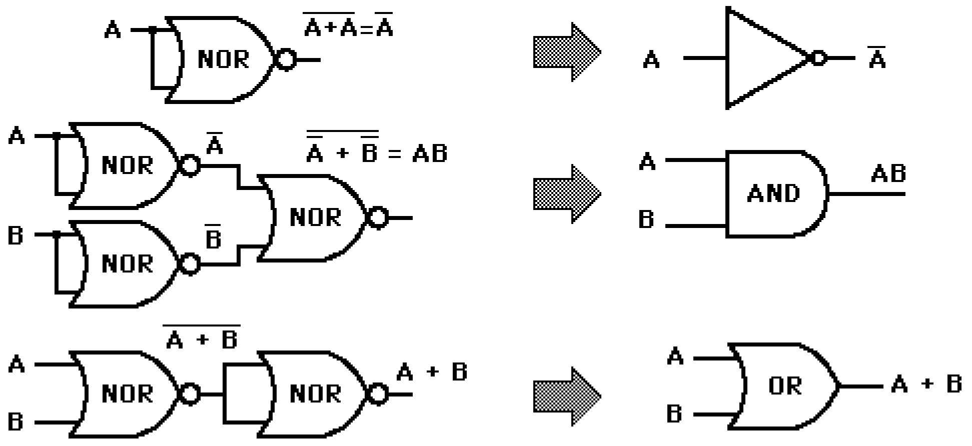
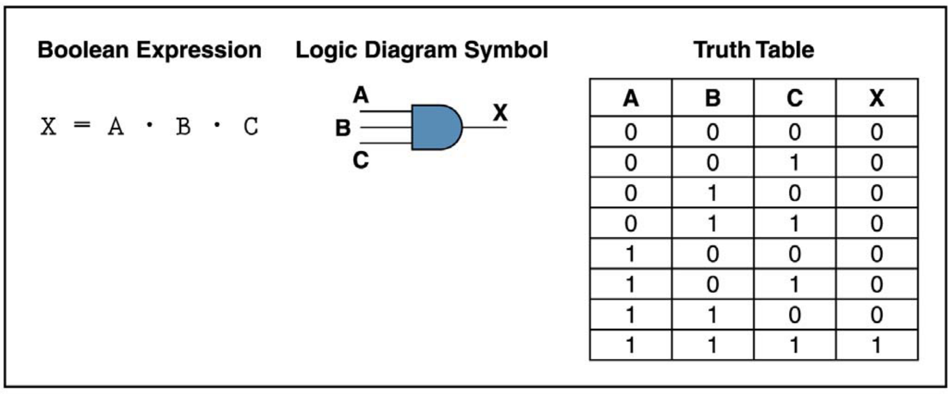


Fig: Three Circuits Constructed Using Only NOR Gates

Thus, the **NOR gate** is a universal gate since it can implement the AND, OR and NOT functions.

Gates with More Inputs

- Gates can be designed to accept three or more input values
- A three-input AND gate, for example, produces an output of 1 only if all input values are 1.



$X = A \cdot B \cdot C$

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

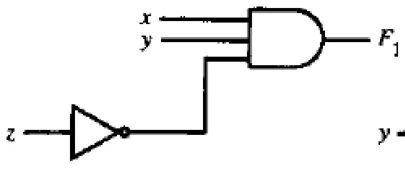
Fig: Various representations of a three input AND gate

Implementation of Boolean function using gates

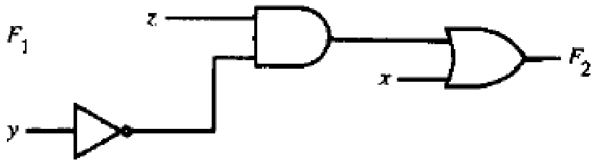
$F_1 = xyz'$

$F_2 = x + y'z$

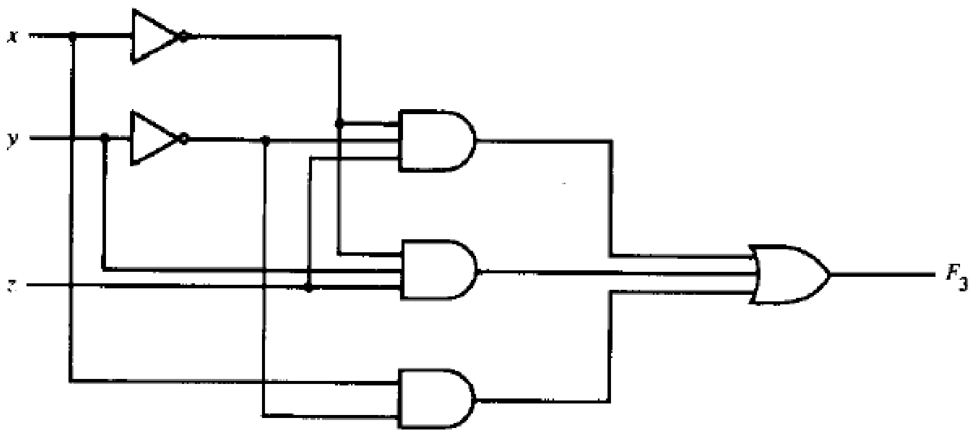
$F_3 = x'y'z + x'yz + xy'$



(a) $F_1 = xyz'$



(b) $F_2 = x + y'z$



(c) $F_3 = x'y'z + x'yz + xy'$

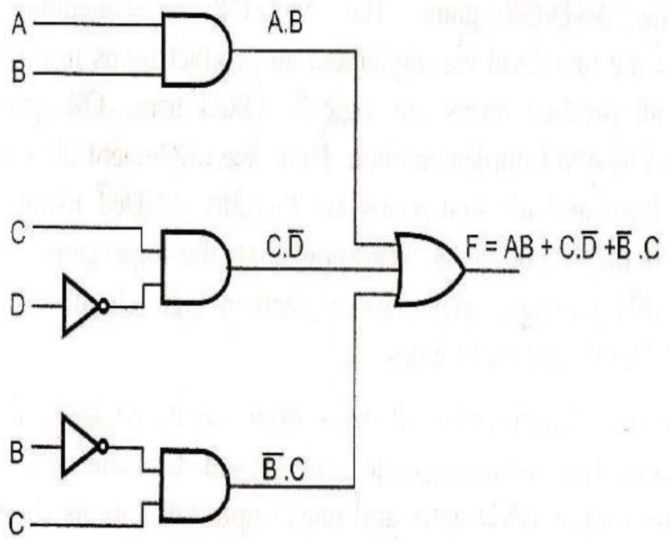
Q. Draw a logic gates that implements the following:

a. F = AB + C D' + B' C

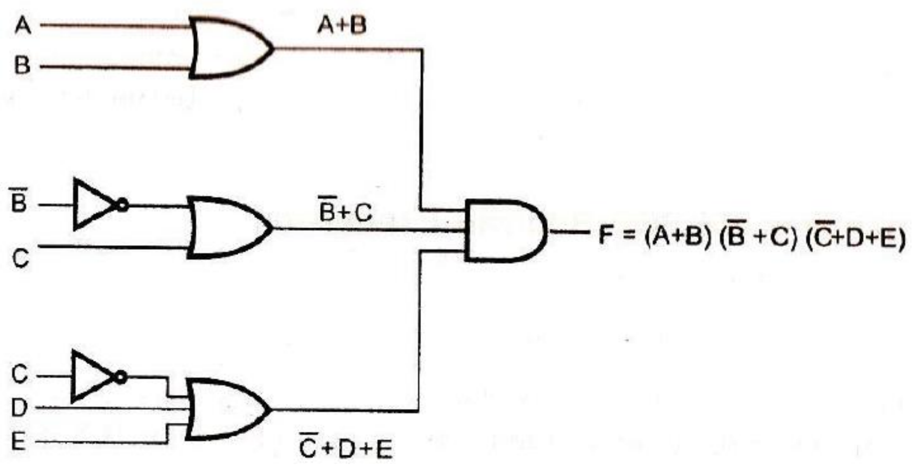
b. F = (A + B) (B' + C) (C' + D + E)

Sol^n:

a.



b.



Boolean Algebra

- Boolean algebra is algebra for the manipulation of objects that can take on only **two** values, typically true and false.
- It is common to interpret the digital value **0** as false and the digital value **1** as true.
- A two-valued Boolean algebra is defined on a set of 2 elements B = {0, 1} with 3 binary operators OR (+), AND (•), and NOT (').

Basic Theorem and Properties of Boolean Algebra

PROPERTIES	**Commutative**	$x+y = y+x$	$x\cdot y = y\cdot x$
	Neutral element	$0+x = x$	$1\cdot x = x$
	Distributive	$x\cdot(y+z) = (x\cdot y)+(y\cdot z)$	$x+(y\cdot z) = (x+y)(x+z)$
	Associative	$x\cdot(y\cdot z) = (x\cdot y)\cdot z$	$x+(y+z) = (x+y)+z$
	Complement	$x+\overline{x} = 1$	$x\cdot\overline{x} = 0$
THEOREMS	**Idempotence**	$x+x = x$	$x\cdot x = x$
	Identity	$x+1 = 1$	$x\cdot 0 = 0$
	Absorption	$x+x\cdot y = x$	$x\cdot(x+y) = x$
	DeMorgan	$\overline{(x+y)} = \overline{x}\cdot\overline{y}$	$\overline{(x\cdot y)} = \overline{x}+\overline{y}$

Duality Principle

It states that "***Every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged***". In a two-valued Boolean algebra, the identity elements and the elements of the set B are the same: 1 and 0. 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.

E.g. $X \cdot Y + Z' = (X' + Y') \cdot Z$

De-Morgans Theorem

De Morgan's theorem is used to convert OR type of expression into AND type and vice-versa.It is further divided into two different types;

1st law:
It state that the total complement of sum is equal to the product of individual complement.
i.e. (A+B)'=A' ⋅B'

Proof:

Input		Output	
A	B	(A+B)'	A' ⋅B'
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

2nd law:
It state that the total complement of the product is equal to the sum of individual complement.
i.e. (A⋅B)' =A'+B'

Proof:

Input		Output	
A	B	(A⋅B)'	A'+B'
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Boolean Function

A binary variable can take a value of 0 or, 1. A Boolean function is an expression formed with binary variables, the two binary operators OR and AND, unary operator NOT, parenthesis and an equal sign. For a given value of variables, the function either can 0 or 1.

E.g.
$F_1 = xyz'$

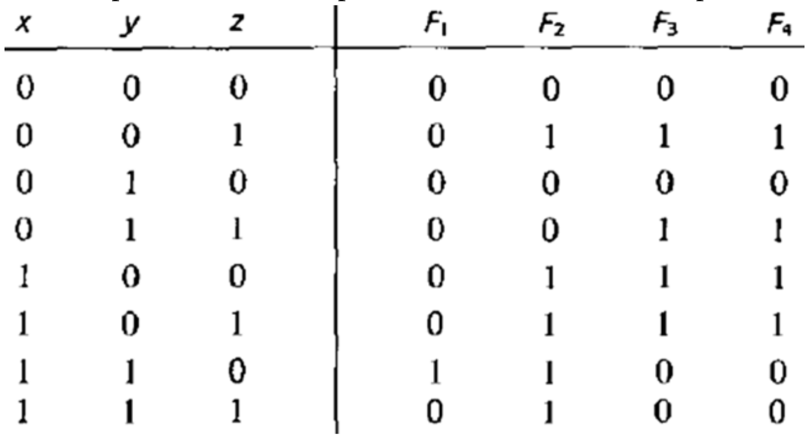
$F_2 = x + y'z$

$F_3 = x'y'z + x'yz + xy'$

$F_4 = xy' + x'z$

Truth Table of Boolean functions:

A **truth table** shows the **relationship**, in tabular form, between the input values and the result of a specific Boolean operator or function on the input variables.



x	y	z	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0

Here, F_3 and F_4 are same. Two functions of n binary variables are said to be equal if they have same values for all possible 2^n combinations of the n variables.

Algaberic Manipulation

When a Boolean function is implemented with logic gates, each literal in the function designates an input to a gate and each term is implemented with a gate. The minimization of the number of literals and the number of terms results is a circuit with less equipment.

Q. Simplify the following Boolean functions to a minimum number of literals.

1. $x(x' + y)$
   $= xx' + xy$
   $= 0 + xy$
   $= xy$

2. $x + x'y$
   $= (x + x')(x + y)$
   $= 1(x + y)$
   $= x + y$

3. $(x + y)(x + y')$
   $= x + xy + xy' + yy'$
   $= x(1 + y + y')$
   $= x$

4. $xy + x'z + yz$
$= xy + x'z + yz(x + x')$
$= xy + x'z + xyz + x'yz$
$= xy(1 + z) + x'z(1 + y)$
$= xy + x'z$

***Q. Prove that:* $(x + y)(\bar{x} + y) = y$**
Sol^n^:

Proof	Identity Name
$(x+y)(\bar{x}+y) = x\bar{x}+xy+y\bar{x}+yy$	Distributive Law
$= 0+xy+y\bar{x}+yy$	Inverse Law
$= 0+xy+y\bar{x}+y$	Idempotent Law
$= xy+y\bar{x}+y$	Identity Law
$= y(x+\bar{x})+y$	Distributive Law (and Commutative Law)
$= y(1)+y$	Inverse Law
$= y+y$	Identity Law
$= y$	Idempotent Law

Note: To prove the equality of two Boolean expressions, you can also create the truth tables for each and compare. If the truth tables are **identical**, the expressions are **equal**.

***Q. Prove the Boolean expression:* $AB + AB'C + A'BC = AB + AC + BC$**
Sol^n^:

$AB + AB'C + A'BC$
$= A(B + B'C) + A'BC$
$= A(B + C) + A'BC$ $[\therefore A + A'B = A + B]$
$= AB + AC + A'BC$
$= AB + (A + A'B)C$
$= AB + (A + B)C$
$= AB + AC + BC$

Q. Minimize the expression:* $AB + A\overline{C} + BC$ *using theorem and properties of Boolean Algebra.
Sol^n^:

$$AB + A\overline{C} + BC = AB(C + \overline{C}) + A\overline{C} + BC$$
$$= ABC + AB\overline{C} + A\overline{C} + BC$$
$$= BC(1 + A) + A\overline{C}(1 + B)$$
$$= BC + A\overline{C}$$

Q. Prove that: $\overline{\overline{A\overline{B}C} + \overline{ACD} + B\overline{C}} = A\overline{B}CD$

Sol^n:

$$\overline{\overline{A\overline{B}C}+\overline{ACD}+B\overline{C}} = \overline{(\overline{A}+B+\overline{C})+(\overline{A}+\overline{C}+\overline{D})+B\overline{C}}$$
$$= \overline{(\overline{A}+B+\overline{C}+\overline{D})+B\overline{C}}$$
$$= \overline{(\overline{A}+B+\overline{C}+\overline{D})}$$
$$= A\overline{B}CD$$

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 De Morgan's theorem.

$(A + B + C)' =$	$(A + X)'$	let $B + C = X$
	$= A'X'$	by theorem 5(a) (DeMorgan)
	$= A' \cdot (B + C)'$	substitute $B + C = X$
	$= A' \cdot (B'C')$	by theorem 5(a) (DeMorgan)
	$= A'B'C'$	by theorem 4(b) (associative)

Generalized theorems for finding complement:

$$(A + B + C + D + \cdots + F)' = A'B'C'D' \cdots F'$$
$$(ABCD \cdots F)' = A' + B' + C' + D' + \cdots + F'$$

Q. Find the Complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$.

Sol^n:

By applying DeMorgan's theorem as many times as necessary, the complements are obtained as follows:

$$F_1' = (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z')$$
$$F_2' = [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')' \cdot (yz)'$$
$$= x' + (y + z)(y' + z')$$

Different forms of Boolean Algebra

1. **Sum of Product (SOP)**
   This is an expression in which each term is a product term and all the product terms are summed together.
   a) Minimal SOP
      An expression in which each product term consist of minimum numbers of variables.
      E.g. XY+X'Y+XY'

   b) Expanded SOP
      An expression in which each product term consist of maximum numbers of variables.
      E.g. XYZ+X'YZ+WXYZ

2. **Product of Sum (POS)**
   This is an expression in which several sum terms are multiplied together.
   a) Minimal POS
      In this an expression in which there are minimum number of sum terms.
      E.g. (X+Y) (X'+Y) (X+Y')

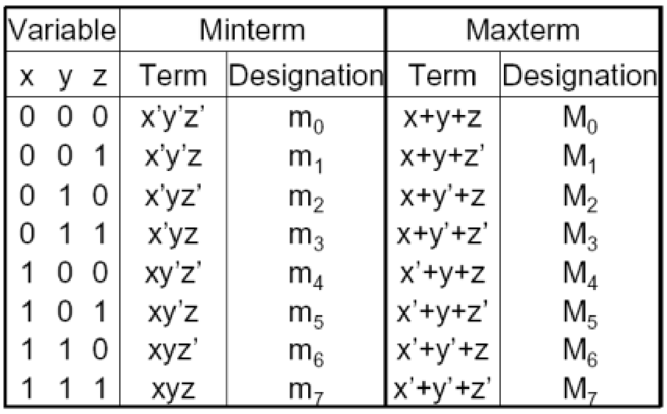
   b) Expanded POS
      In this an expression in which there are maximum number of variables.
      E.g. (X+Y+Z)$\cdot$(X'+Y'+Z')$\cdot$(W+X+Y+Z)

Minterms or standard product

- Each row of a truth table can be associated with a *minterm*, which is a product (AND) of all variables in the function, in direct or complemented form.
- A function with n variables has 2^n minterms.
- A minterm has the property that it is equal to 1 on exactly one row of the truth table.

Maxterms or standard sums

- Each row of a truth table is also associated with a *maxterm*, which is a sum (OR) of all the variables in the function, in direct or complemented form.
- A function with n variables has 2^n maxterms
- A maxterm has the property that it is equal to 0 on exactly one row of the truth table.



Variable	Minterm		Maxterm	
x y z	Term	Designation	Term	Designation
0 0 0	x'y'z'	m_0	x+y+z	M_0
0 0 1	x'y'z	m_1	x+y+z'	M_1
0 1 0	x'yz'	m_2	x+y'+z	M_2
0 1 1	x'yz	m_3	x+y'+z'	M_3
1 0 0	xy'z'	m_4	x'+y+z	M_4
1 0 1	xy'z	m_5	x'+y+z'	M_5
1 1 0	xyz'	m_6	x'+y'+z	M_6
1 1 1	xyz	m_7	x'+y'+z'	M_7

Note: Each maxterm is the complement of its corresponding minterm and vice versa.

Expressing Boolean function by sum of minterms

A boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of the variables which produces a 1 in the function and then taking the OR of all those terms.

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	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

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

Finding Complement of Boolean function by sum of minterms

The complement of a Boolean function can be obtained from the truth table by forming a minterm of each combination that produces a 0 in the function and then ORing those terms.

Functions of Three Variables

x	y	z	Function f_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The Complement of f_1 is written as:

$$f_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

Expressing Boolean Function by product of maxterms

A boolean function may be expressed algebraically from a given truth table by forming a maxterm for each combination of the variables which produces a 0 in the function and then taking the AND of all those terms.

x	y	z	Function f_1	Function f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	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

$$f_1 = (x + y + z)(x + y' + z)(x + y' + z')(x' + y + z')(x' + y' + z)$$
$$= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$$
$$f_2 = (x + y + z)(x + y + z')(x + y' + z)(x' + y + z)$$
$$= M_0 M_1 M_2 M_4$$

Canonical and Standard forms

- Boolean functions expressed as a sum of minterms or product of maxterms are said to be in **canonical form**.
- In an expression in canonical form, every variable appears in every term.
- The two canonical forms of Boolean algebra are basic forms that one obtain from reading a function from the truth table. These forms are very seldom the ones with least number of literals, because **each minterm or maxterm must contain, by definition, all the variables either complemented or uncomplemented**.

- Another way to express Boolean functions is in **standard form**. In this configuration, the terms that form the function may contain one, two or any number of literal.

- There are two types of standard forms: the sum of products and product of sums.
  Sum of products: $F_1 = y' + xy + x'yz'$
  Product of sums: $F_2 = x(y' + z)(x' + y + z' + w)$

Expressing Boolean function as a sum of minterms and product of maxterms using Boolean algebra:

- To express the Boolean function as a sum of minterms, expand the Boolean function into a sum of products. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables.

- To express the Boolean function as a product of maxterms, expand the Boolean function into a product of sums. This may be done by using the distributive law, $x + yz = (x + y)(x + z)$. Then any missing variable x in each OR term is ORed with xx'.

Q. Express the Boolean function $F = A + B'C$ in a sum of minterms.

Sol^n:

The function has three variables, the first term A is missing two variables B and C.
Inclusion of variable B: $A = A(B + B') = AB + AB'$
Inclusion of variable of C: $A = AB(C + C') + AB'(C + C')$
$= ABC + ABC' + AB'C + AB'C'$

The second term $B'C$ is missing one variable A.
Inclusion of variable A: $B'C = B'C(A + A') = AB'C + A'B'C$

Combining all terms
$F = A + B'C$
$= ABC + ABC' + \mathbf{AB'C} + AB'C' + \mathbf{AB'C} + A'B'C$
$F = A'B'C + AB'C' + AB'C + ABC' + ABC$
$= m_1 + m_4 + m_5 + m_6 + m_7$

$F(A, B, C) = \sum(1, 4, 5,6 ,7)$

Q. Express the Boolean function $F = xy + x'z$ in a product of maxterm form.

Sol^n:

Converting the function into OR terms using distributive law:

$$F = xy + x'z = (xy + x')(xy + z)$$
$$= (x + x')(y + x')(x + z)(y + z)$$
$$= (x' + y)(x + z)(y + z)$$

Including missing with each term:

$$x' + y = x' + y + zz' = (x' + y + z)(x' + y + z')$$
$$x + z = x + z + yy' = (x + y + z)(x + y' + z)$$
$$y + z = y + z + xx' = (x + y + z)(x' + y + z)$$

Combining and avoiding the repeated terms:

$$F = (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z')$$
$$= M_0M_2M_4M_5$$
$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

<u>Conversion between canonical forms</u>

$$\boxed{\boldsymbol{m_j' = M_j}}$$

Consider a function
$F(A, B, C) = \sum(1, 4, 5, 6,7)$
Taking the complement of F
$F'(A, B, C) = \sum(0, 2, 3) = m_0 + m_2 + m_3$
Taking the complement of F'
$F = (m_0 + m_2 + m_3\,)' = m_0' \cdot m_2' \cdot m_3' = M_0M_2M_3 = \prod(0, 2, 3)$
Similarly,
$F(x, y, z) = \sum(1, 3, 6,7) \leftrightarrow F(x, y, z) = \prod(0, 2, 4,5)$

Karnaugh Map (K-Map)

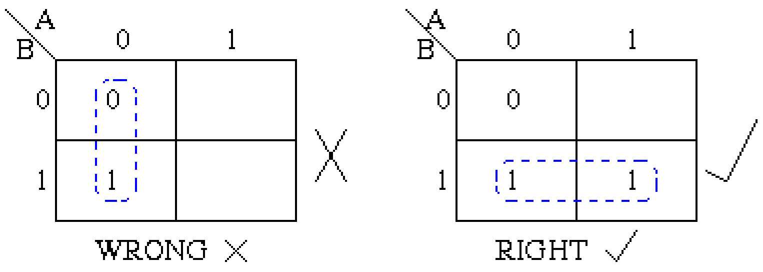
- **Karnaugh map** or **K-map** is a map of a function used in a technique used for minimization or **simplification of a Boolean expression**.
- Booleans expression can be simplified using Boolean algebraic theorems but there are no specific rules to make the most simplified expression. However, K-map can easily minimize the terms of a Boolean function.
- Unlike an algebraic method, K-map is a pictorial method and it does not need any Boolean algebraic theorems.
- K-map is basically a diagram made up of squares. Each of these squares represents a min-term of the variables. If n = number of variables then the number of squares in its K-map will be 2^n. K-map is made using the truth table.
- Karnaugh map can produce **Sum of product (SOP)** or **product of Sum(POS)** expression considering which of the two (0,1) outputs are being grouped in it. The grouping of 0's result in Product of Sum expression & the grouping of 1's result in Sum of Product expression.

Rules of minimization in K-Map

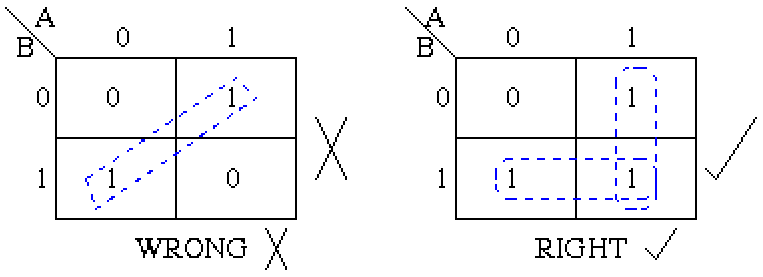
1. While grouping, you can make groups of 2^n number where n=0, 1, 2, 3…
2. You can either make groups of 1's or 0's but not both.
3. Grouping of 1's lead to Sum of Product form and Grouping of 0's lead to Product of Sum form.
4. While grouping, the groups of 1's should not contain any 0 and the group of 0's should not contain any 1.
5. The function output for 0's grouping should be complemented as F'.

Rules for grouping 1's (Sum of Product form):

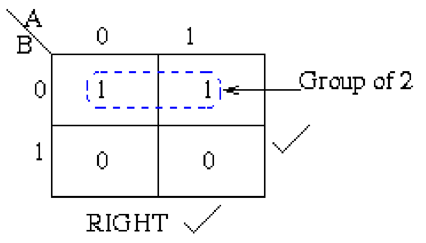
- Groups may not include any cell containing a zero.

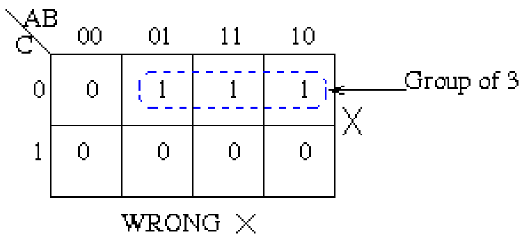


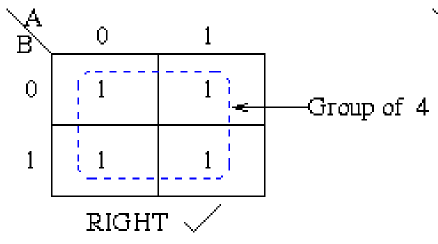
- Groups may be horizontal or vertical, but not diagonal.

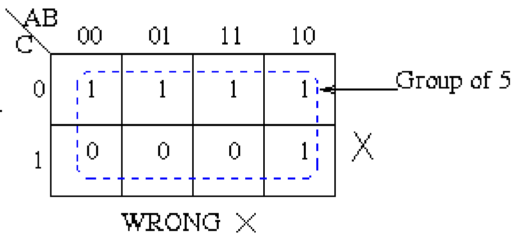


- Groups must contain 1, 2, 4, 8, or in general 2^n cells. That is if n = 1, a group will contain two 1's since 2^1 = 2. If n = 2, a group will contain four 1's since 2^2 = 4.

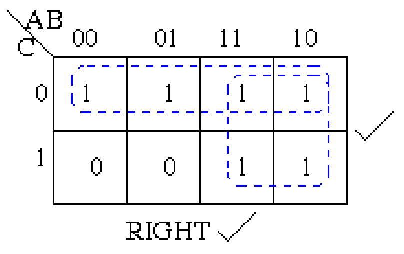


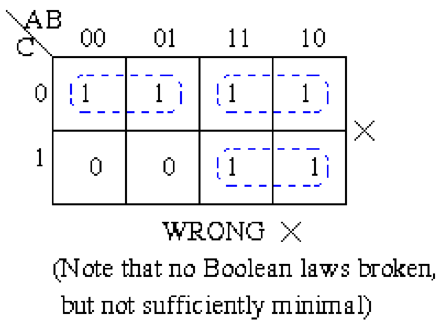




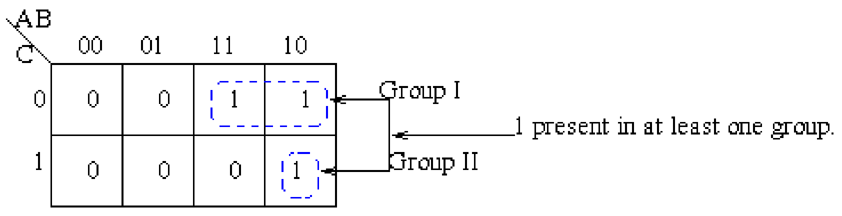


- Each group should be as large as possible.

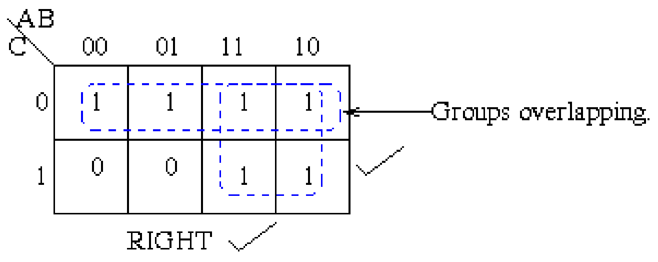


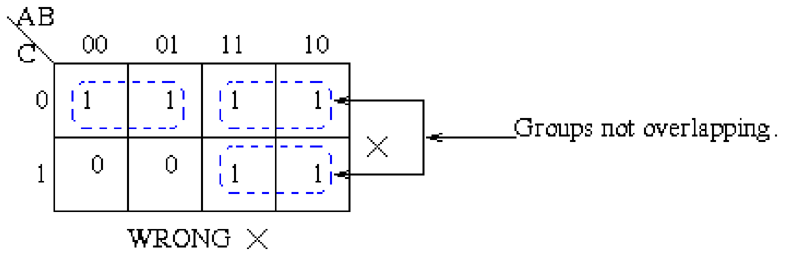

WRONG ✗
(Note that no Boolean laws broken, but not sufficiently minimal)

- Each cell containing a *one* must be in at least one group.

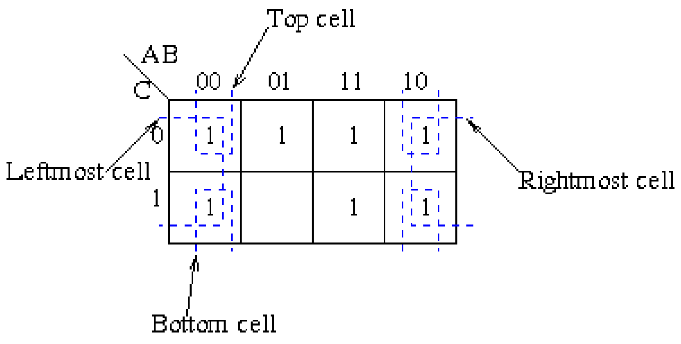


- Groups may overlap.

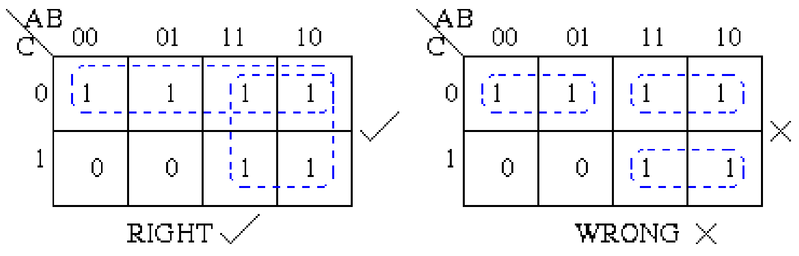




- Groups may wrap around the table. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.



- There should be as few groups as possible, as long as this does not contradict any of the previous rules.



Two Variable Map

There are four minterms for two variables; hence the map consists of fours squares, one of each minterm.

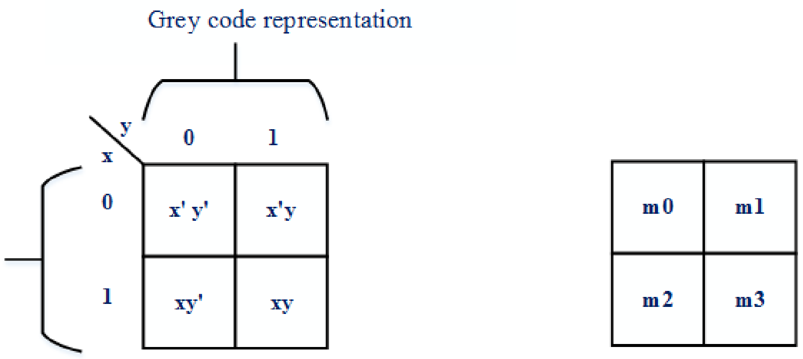
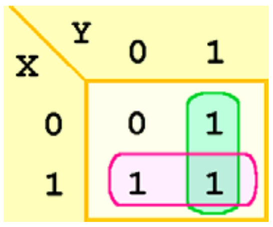


Fig: Two variable map

Q. Simplify the Boolean function $F = x'y + xy' + xy$ using K-Map.

Sol^n:

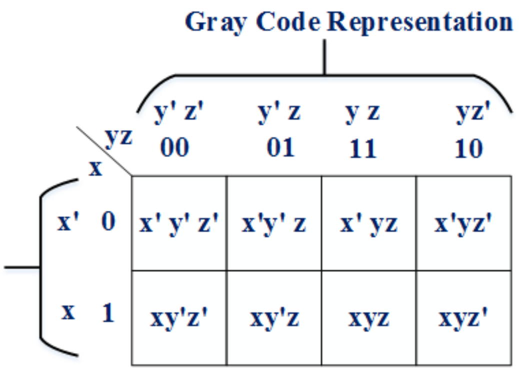
K-map for given function:



So, after simplification, $F = X + Y$

Three Variable Map

- There are eight minterms for three variables, i.e. the map consist of eight squares.
- Minterms are not arranged in binary sequence but in a sequence similar to gray code/reflected code.



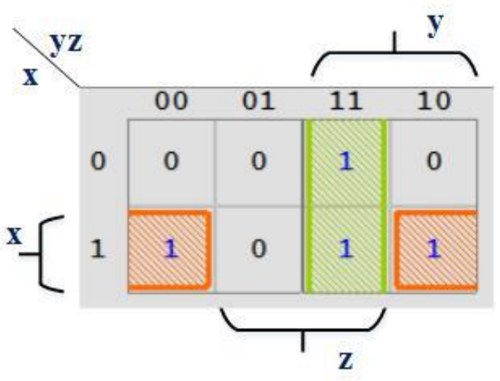
m0	m1	m3	m2
m4	m5	m7	m6

Fig: Three variable map

Q. Simplify the Boolean function, $F = X'YZ + XY'Z' + XYZ + XYZ'$ using K-Map.

Sol^n:

K-map for given function:



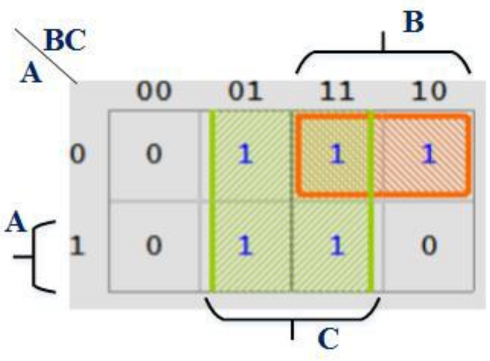
After simplification, $F = YZ + XZ'$

Q. Simplify the Boolean function, $F = A'C + A'B + AB'C + BC$ using K-Map.

Sol^n:

$F = A'C + A'B + AB'C + BC = \boldsymbol{A'BC} + A'B'C + \boldsymbol{A'BC} + A'BC' + AB'C + ABC + \boldsymbol{A'BC}$

K-map for given function:

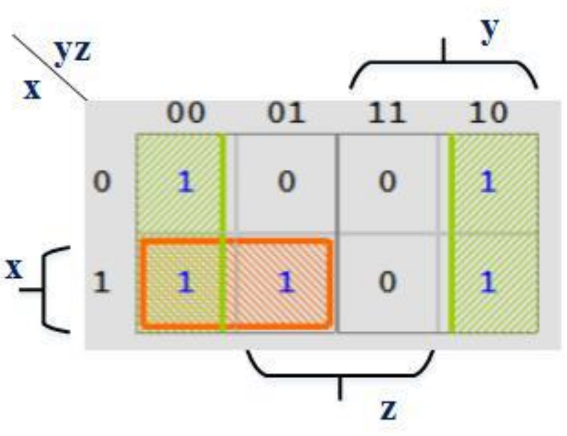


After simplification, $F = C + A'B$

Q. Simplify the Boolean function: $F(X,Y,Z) = \sum(0,2,4,5,6)$ using three variable K-map.

Solⁿ:

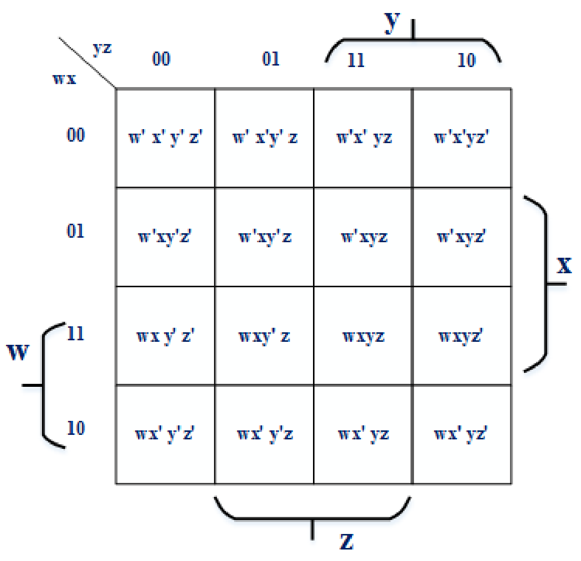
K-map for given function:

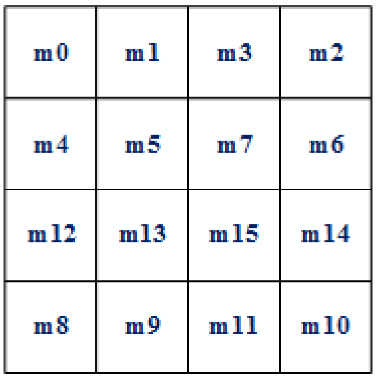


After simplification, $F = Z' + XY'$

Four Variable Map

The map for Boolean function of four binary variables has 16 minterms and the squares assigned to each.





m0	m1	m3	m2
m4	m5	m7	m6
m12	m13	m15	m14
m8	m9	m11	m10

Fig: Four variable map

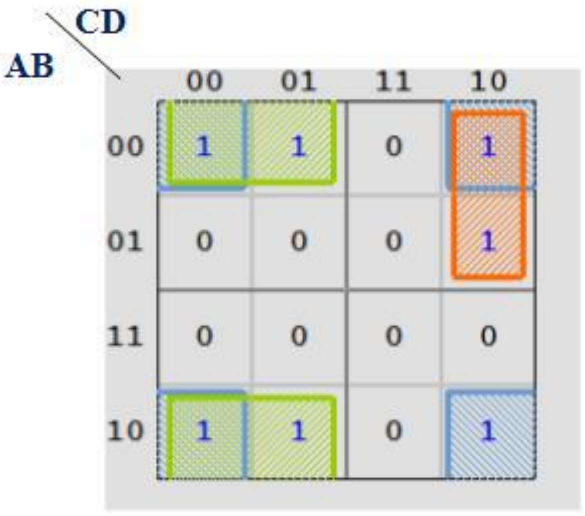
- The rows and columns are numbered in a reflected code sequence, with only one digit changing value between two adjacent rows and columns.
- The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number.

Q. Simplify the Boolean function $F = A'B'C' + B'CD' + A'BCD' + AB'C'$ using K-map.

Solⁿ:

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

K-map for given function:

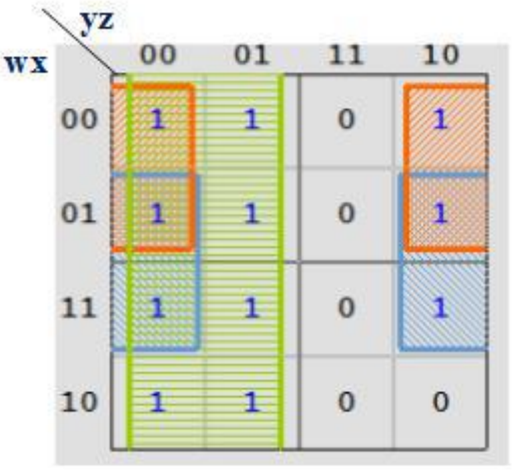


After simplification, $F = B'D' + B'C' + A'CD'$

Q. Simplify the Boolean function $\boldsymbol{F(W,X,Y,Z) = \sum(0,1,2,4,5,6,8,9,12,13,14)}$.

Sol

K-map for given function:



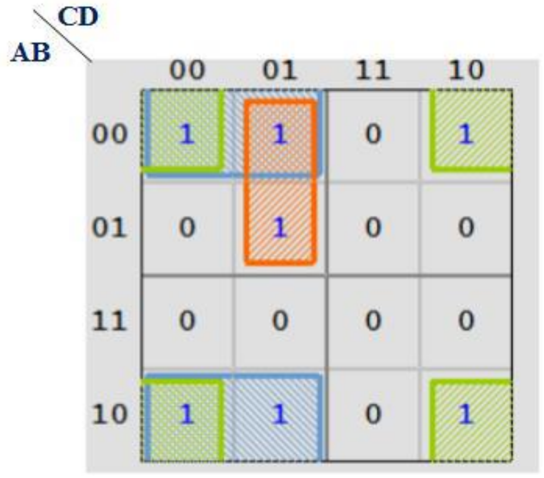
After simplification, $F = Y' + W'Z' + XZ'$

<u>Product of sum simplification</u>

- All previous examples are in sum-of-products form.
- To obtain the product-of-sum form:
  - Simplify F' in the form of sum of products. [If we mark the empty squares by 0's and combine them into valid rectangles, we obtained a simplified expression of the complement of the function, i.e. of F']
  - Apply DeMorgan's theorem $F = (F')'$
  - $\boldsymbol{F'}$**:** sum of products $\Rightarrow$ $\boldsymbol{F}$ **:** product of sums

Q. Simplify the Boolean function: $F = \sum(0,1,2,5,8,9,10)$ in (a) sum of product (SOP) and (b) product of sum (POS).
Sol^n:

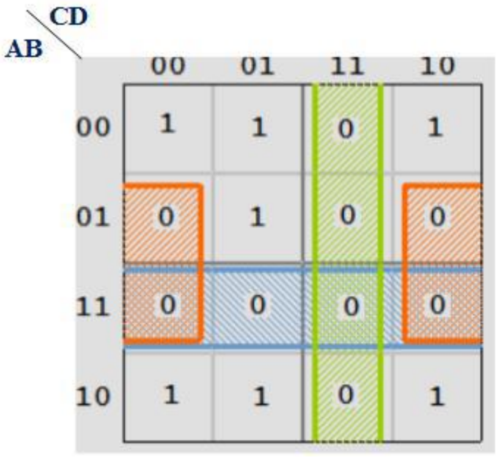
(a) Sum of products simplification



AB \ CD	00	01	11	10
00	1	1	0	1
01	0	1	0	0
11	0	0	0	0
10	1	1	0	1

$$F = B'D' + B'C' + A'C'D$$

(b) Product of sums simplification

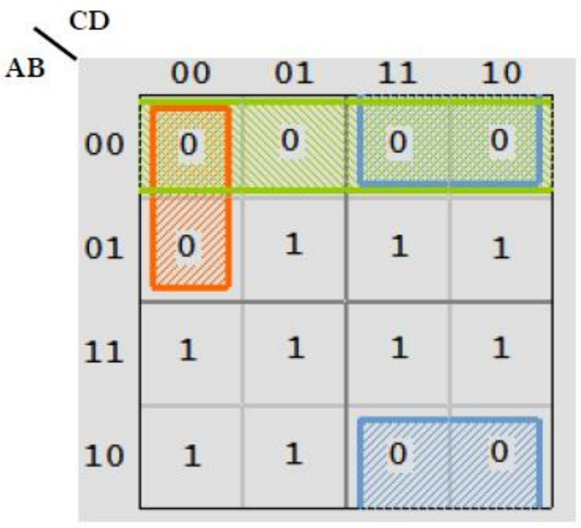


AB \ CD	00	01	11	10
00	1	1	0	1
01	0	1	0	0
11	0	0	0	0
10	1	1	0	1

$$F' = AB + CD + BD'$$

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

Q. Simply the Boolean function $F(A,B,C,D) = \prod(0,1,2,3,4,10,11)$ in POS.
Sol^n:
K-map for given function:



AB \ CD	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	1	1	1	1
10	1	1	0	0

From Map,
$F' = A'B' + B'C + A'C'D'$

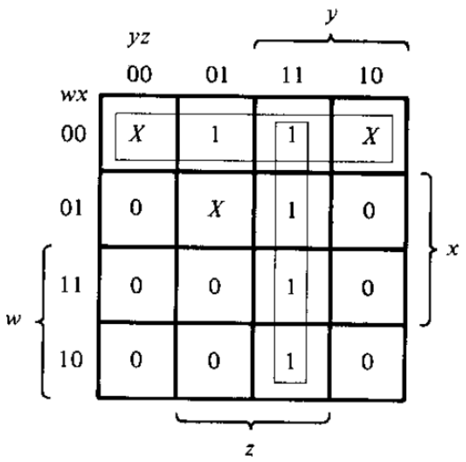
So, $F = (A + B)(B + C')(A + C + D)$

<u>Don't Care Conditions</u>

- There are certain situations where a function may not be completely specified, meaning there may be some input combination that are **undefined** for the function.
- **Real circuits don't** always need to have an **output** defined for every possible input.
- The unused combinations is known as don't care conditions and can be used on the map to provide further simplification of the boolean expression.
- It should be realized that a don't care minterm is a combination of variables whose logical value is not specified. It cannot be marked with a 1 or, 0 in the map as it is not specified

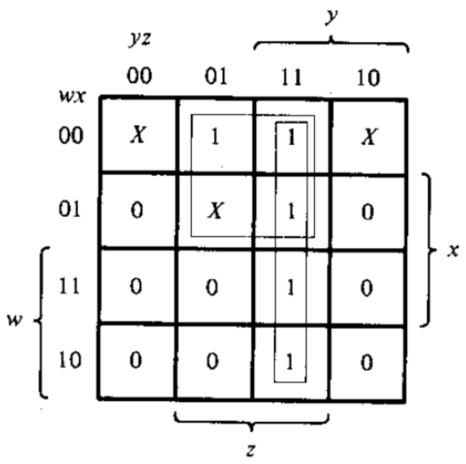
as 0 or 1. To distinguish the don't care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterms.

Q. Simplify $F(w,x,y,z) = \sum(1,3,7,11,15)$ which has the don't care conditions $d(w,x,y,z) = \sum(0,2,5)$.
Sol^n:



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

(a) $F = yz + w'x'$

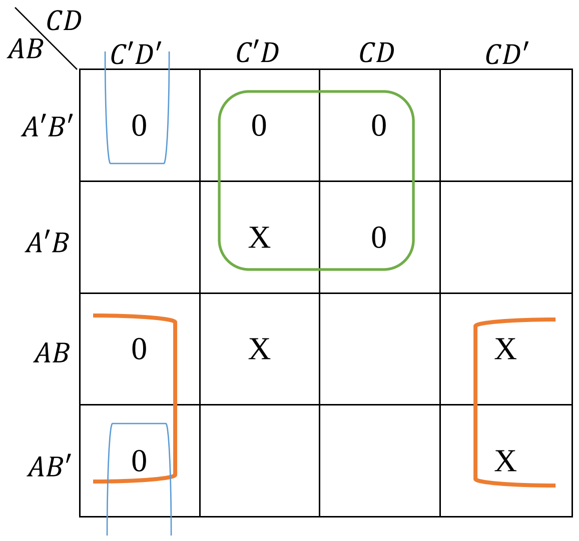


wx \ yz	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

(b) $F = yz + w'z$

Q. $F(A,B,C,D) = \prod(0,1,3,7,8,12)$ and $\prod d(5,10,13,14)$. Obtain SOP and POS.
Sol^n:

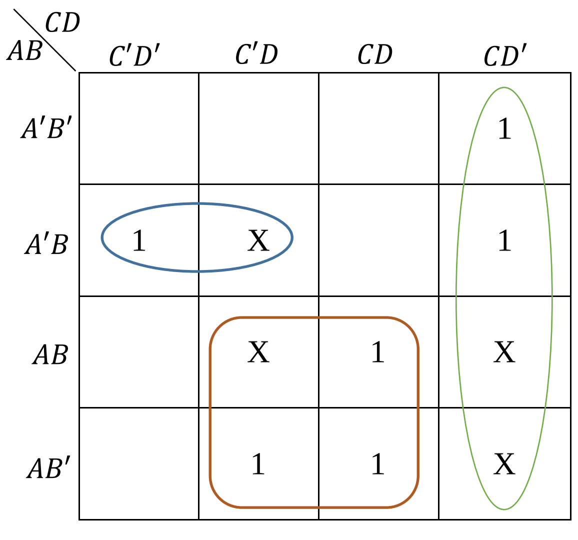
For POS:



AB \ CD	$C'D'$	$C'D$	CD	CD'
$A'B'$	0	0	0	
$A'B$		X	0	
AB	0	X		X
AB'	0			X

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

For SOP:



AB \ CD	$C'D'$	$C'D$	CD	CD'
$A'B'$				1
$A'B$	1	X		1
AB		X	1	X
AB'		1	1	X

$$F = CD' + AD + A'BC'$$

Combinational Logic

Combinational circuit is a circuit which consist of logic gates whose outputs at any instant of time are determined directly from the present combination of inputs without regard to previous input. The combinational circuit do not use any memory.

- There will be 2^n combination of input variable for n inputs.
- A combinational circuit can have n number of inputs and m number of outputs.
- For e.g. adders, subtractors, decoders, encoders etc.

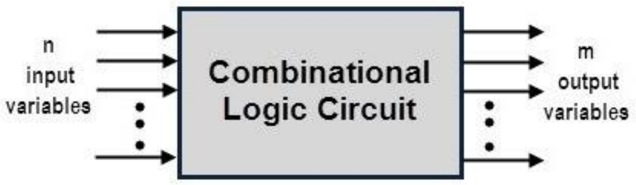


Fig: Block diagram of combinational circuit

Combinational logic circuit design procedure:

1. The problem is stated.
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letter symbols.
4. The truth table that defines the required relationships between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.
6. The logic diagram is drawn.

Adders

Adders are the combinational circuits which is used to add two or more than two bits at a time.

Types of adders:
- Half Adder
- Full Adder

1. Half Adder:

A combinational circuit that performs the addition of bits is called half adder. This circuit needs two binary inputs and two binary outputs. The input variables designate the augend(A) and addend(B) bits; the output variables produce the sum(S) and carry(C).

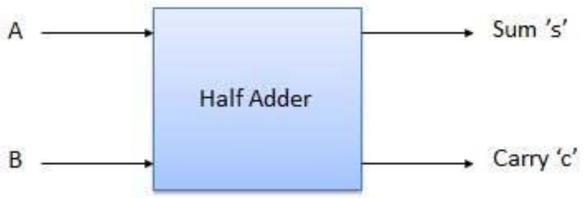


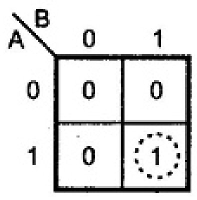
Fig: Block diagram

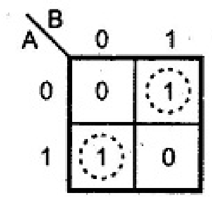
Truth table to identify the function of half adder:

Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

K-map:

For Carry **For Sum**



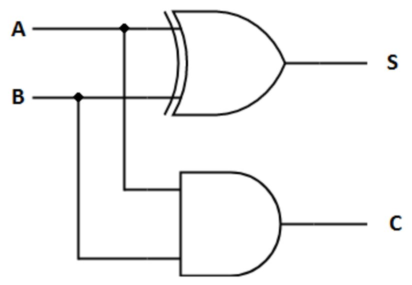


From k-map the logical expression for sum and carry is:

$C = AB$

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

Logic diagram:



Q. Design a half adder using only NAND gates.

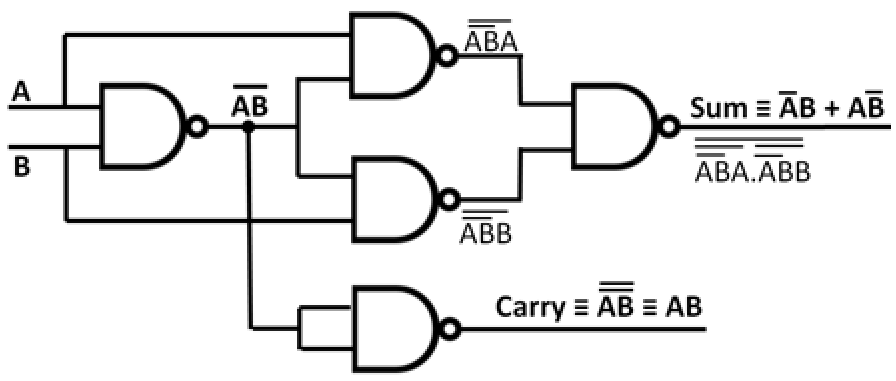
Sol^n^:

Input variables: A & B, Output variables: sum(S) and carry(C)

$S = \bar{A}B + A\bar{B}$

$C = AB$

Logic diagram of half adder using NAND gates only:


A
B
$\overline{AB}$
$\overline{\overline{AB}A}$
$\overline{\overline{AB}B}$
Sum ≡ $\bar{A}B + A\bar{B}$
$\overline{\overline{\overline{AB}A}.\overline{\overline{AB}B}}$
Carry ≡ $\overline{\overline{AB}}$ ≡ AB

Q. Design a half adder logic circuit using NOR gates only.

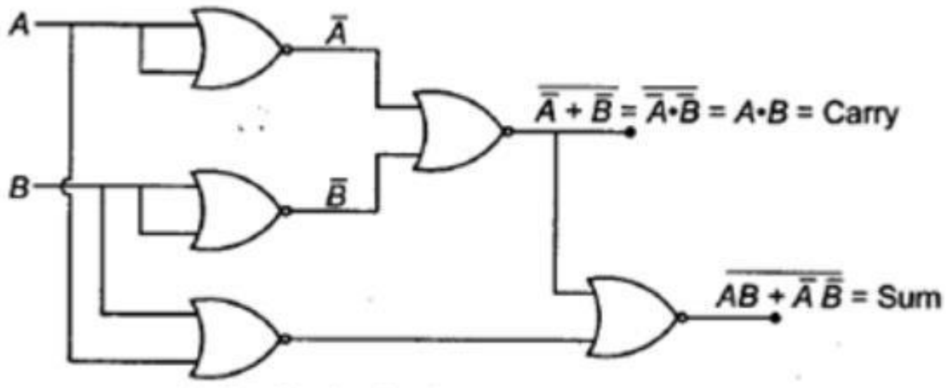
Sol^n:

Input variables: A & B, Output variables: sum(S) and carry(C)

$S = \bar{A}B + A\bar{B}$

$C = AB$

Logic diagram of half adder using NOR gates only:


A
$\bar{A}$
$\overline{\bar{A}+\bar{B}} = \overline{\bar{A}}\cdot\overline{\bar{B}} = A\cdot B = \text{Carry}$
B
$\bar{B}$
$\overline{AB+\bar{A}\bar{B}} = \text{Sum}$

Half adder logic diagram using NOR gate

2. **Full Adder**:

A combinational circuit that performs the addition of three bits at a time is called full adder. It consists of three inputs and two outputs, two inputs are the bits to be added, the third input represents the carry from the previous position.

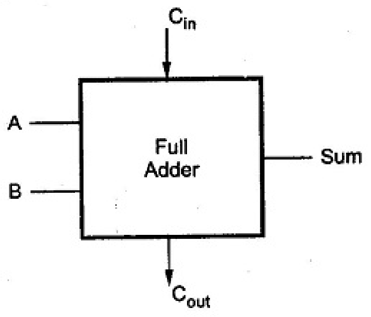


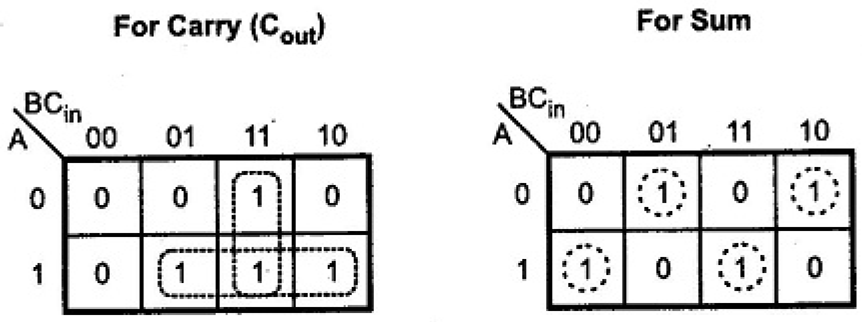
Fig: Block diagram

Truth table for full adder:

Input			Output	
A	B	Cin	Sum	Carry
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

- The sum(S) output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1.
- The carry output (C_{out}) has a carry 1 if two or three inputs are equal to 1.

Simplified expression using k-map in SOP can be obtained as;

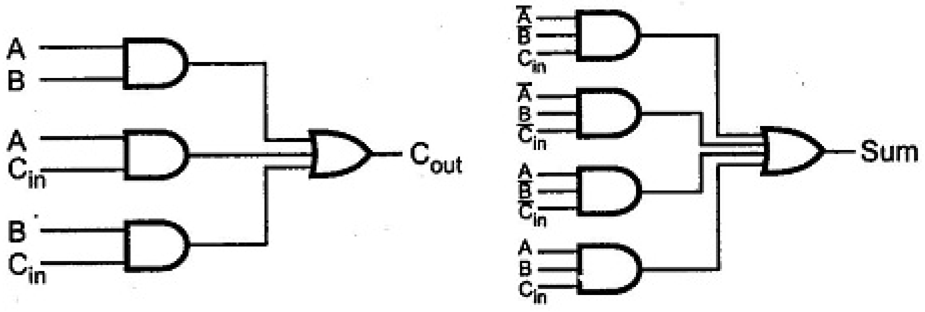


A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

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

$Sum(S) = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Logic Diagram:

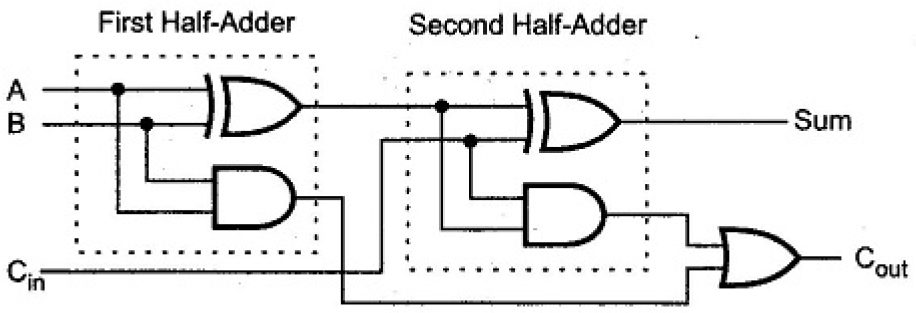


<u>For carry</u> <u>For sum</u>

Fig: SOP implementation of full-adder

Note: *It can also be implemented in POS form. (Try yourself)*

<u>Implementation of a full-adder with two half-adders and an OR gate:</u>



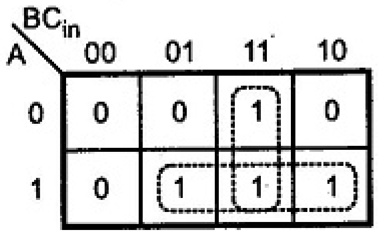
Logic expression for sum:

$(A \oplus B) \oplus C_{in} = (\bar{A}B + A\bar{B}) \oplus C_{in} = \left(\overline{\bar{A}B + A\bar{B}}\right)C_{in} + (\bar{A}B + A\bar{B})\bar{C}_{in}$
$= (A + \bar{B})(\bar{A} + B)C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in}$
$= \bar{A}\bar{B}C_{in} + ABC_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in}$

Logic expression for carry:

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

Simplification of carry:



$\therefore C_{out} = AB + AC_{in} + BC_{in}$

Subtractors

Subtractor is a combinational logic circuit which is used to subtract two or more than two bits at a time, and provides difference and borrow as an output.

Types of Subtractors:

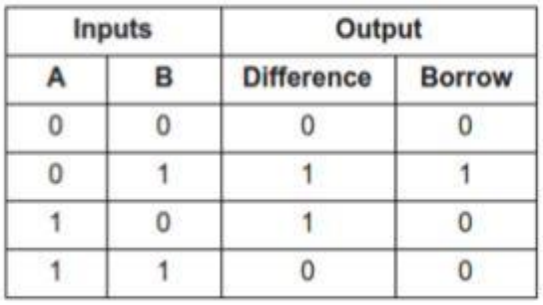
- Half subtractor
- Full subtractor

1. Half Subtractor:

A half-subtractor is a combinational logic circuit that subtract two bits at a time and produces their difference.

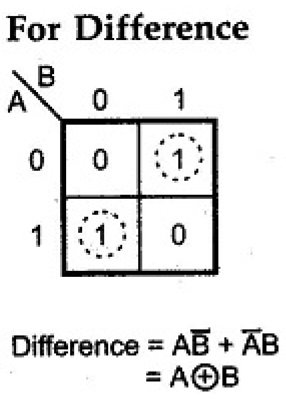
It has two inputs minuend (A) & subtrahend (B) and two outputs difference and borrow. The difference is a result of subtraction and borrow is used to indicate borrow from next most significant bit. The borrow bit is present only when $A < B$.

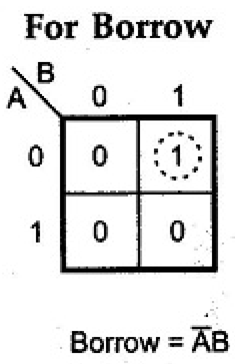
Truth table:



Inputs		Output	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map:





Logic Diagram:

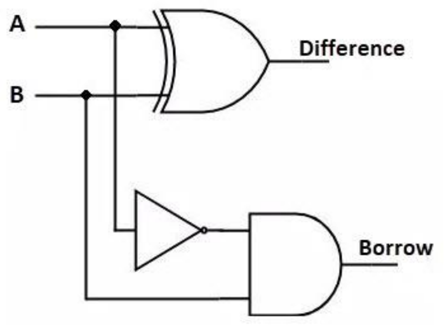


Fig: Implementation of half-subtractor

2. <u>Full Subtractor</u>:

A combinational logic circuit used to subtract three binary digits at a time is called full subtractor.

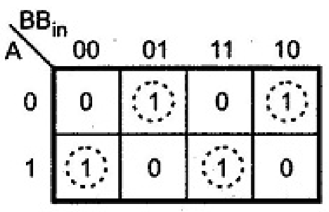
This circuit has three input and two outputs. The three inputs are $A, B \text{ and } B_{in}$, denote the minuend, subtrahend and previous borrow respectively. The two outputs, $D \text{ and } B_{out}$ represent the difference and output borrow, respectively.

Truth table for full subtractor:

Inputs			Outputs	
A	B	B_{in}	D	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

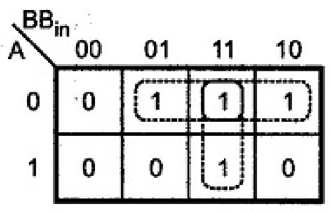
Simplified expression of D and B_{out} using k-map in SOP can be obtained as;

For D



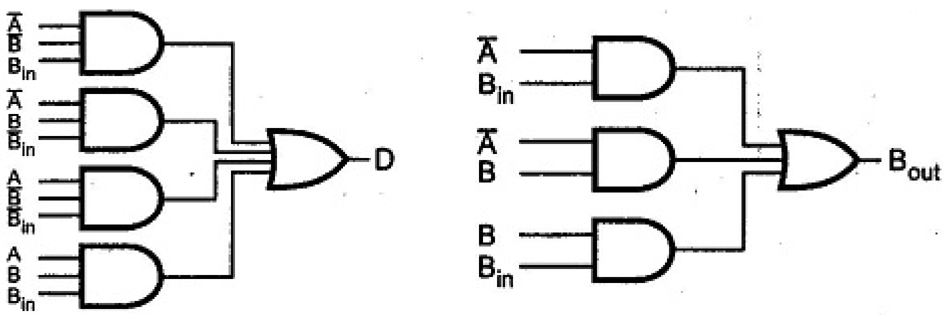
$$D = \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in} + ABB_{in}$$

For B_{out}



$$B_{out} = \bar{A}B_{in} + \bar{A}B + BB_{in}$$

Logic circuit for full subtractor:



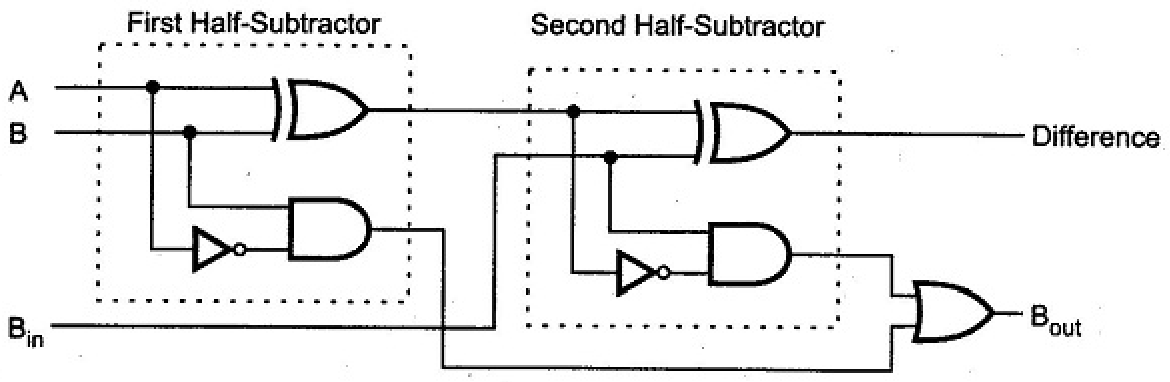
$\bar{A}$ $\bar{B}$ B_{in}
$\bar{A}$ B $\bar{B}_{in}$
A $\bar{B}$ $\bar{B}_{in}$
A B B_{in}
D
$\bar{A}$ B_{in}
$\bar{A}$ B
B B_{in}
B_{out}

For D For B_{out}

Fig: SOP implementation of full-subtractor

Note: *It can also be implemented in POS form. (Try yourself)*

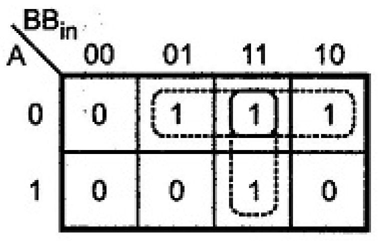
Implementation of full subtractor using two half subtractor and one OR gate:



$$D = (A\oplus B)\oplus B_{in} = (\bar{A}B + A\bar{B})\oplus B_{in} = \left(\overline{\bar{A}B + A\bar{B}}\right)B_{in} + (\bar{A}B + A\bar{B})\bar{B}_{in}$$
$$= \{(A+\bar{B})(\bar{A}+B)\}B_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in}$$
$$= \bar{A}\bar{B}B_{in} + ABB_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in}$$

$$B_{out} = (\overline{A\oplus B})B_{in} + \bar{A}B = \left(\overline{\bar{A}B + A\bar{B}}\right)B_{in} + \bar{A}B = \{(A+\bar{B})(\bar{A}+B)\}B_{in} + \bar{A}B$$
$$= \bar{A}\bar{B}B_{in} + ABB_{in} + \bar{A}B$$

Using k-map:



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

$$\therefore B_{out} = \bar{A}B_{in} + \bar{A}B + BB_{in}$$

Code Conversion

- The availability of large variety of codes for the same discrete elements of information results in the use of different codes by the different system.
- A conversion circuit must be inserted between the two systems if each use different codes for same information.
- Thus a code converter is a circuit that makes the two systems compatible even though each uses a different binary information.

BCD to excess-3 Code Conversion:

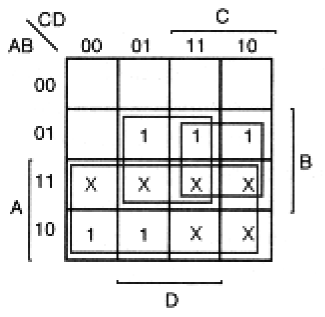
BCD Excess-3 circuit will convert numbers from their binary representation to their excess-3 representation. Since each code uses four bits to represent a decimal digit, there must be four input variables and four outputs variables. Let the input four binary variables are A, B, C & D and the four output variables are W, X, Y & Z.

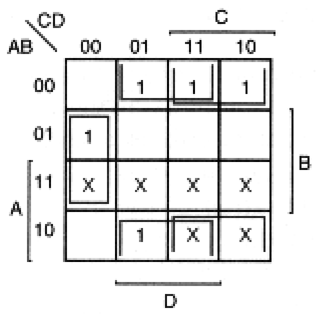
Truth table:

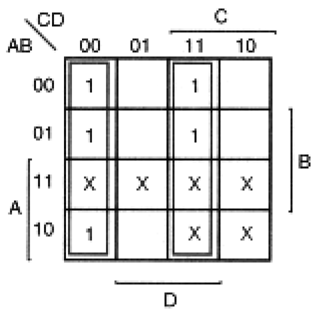
BCD Inputs				excess-3 Outputs			
A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

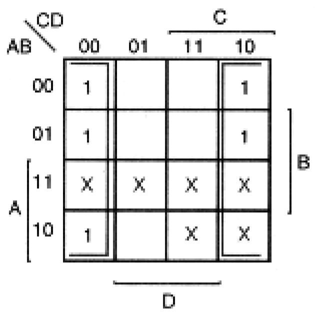
Note: Four binary variables may have 16 bit combinations, and only 10 of which are listed in truth table i.e. from 0 to 9. The rest six bit combinations not listed for input variables are don't care combinations.

K-maps for BCD to excess-3 code converter:


$W = A + BC + BD$


$X = \bar{B}C + \bar{B}D + B\bar{C}\bar{D}$


$Y = CD + \bar{C}\bar{D}$


$Z = \bar{D}$

The Boolean functions for the outputs lines of the circuit are derived from k-maps which are:

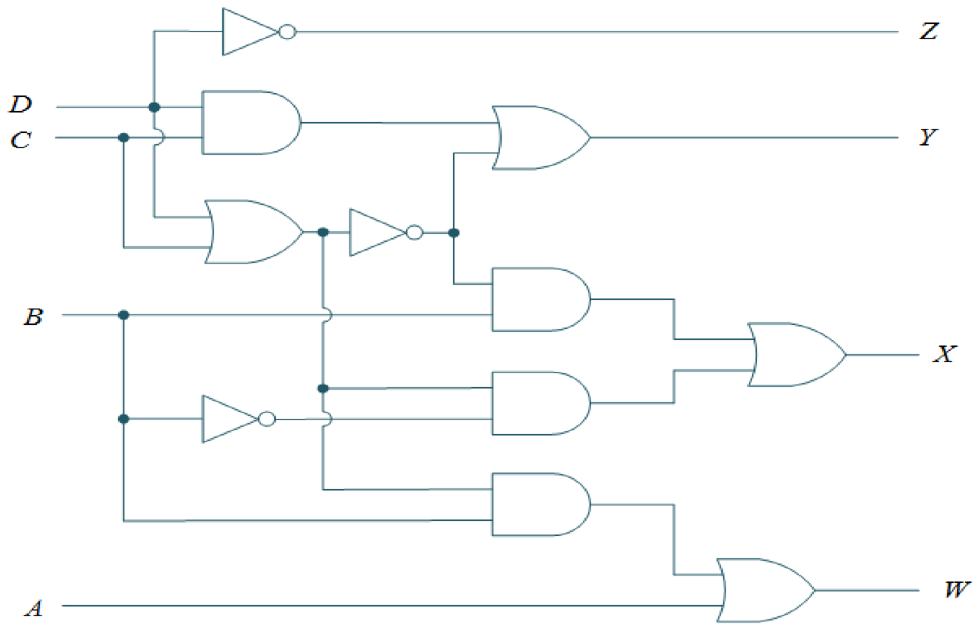
$W = A + BC + BD = A + B(C + D)$

$X = B'C + B'D + BC'D' = B'(C + D) + B(C + D)'$

$Y = CD + C'D' = CD + (C + D)'$

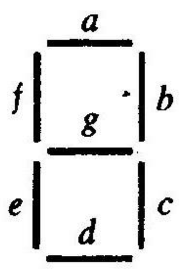
$Z = D'$

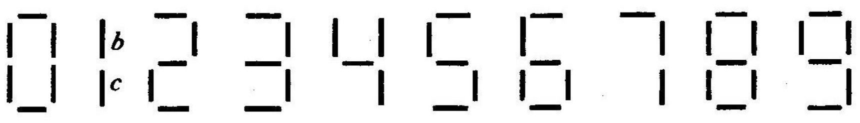
Logic diagram for BCD to excess-3 converter:



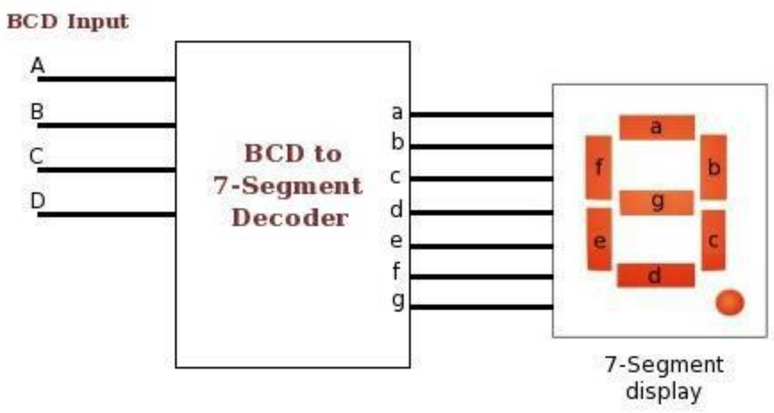
<u>BCD to Seven-segment Decoder</u>

A BCD to seven-segment decoder is a combinational circuit that accepts a decimal digit in BCD and generates the appropriate outputs for the selection of segments in a display indicator used for displaying the decimal digit. The seven output of the decoder (a,b,c,d,e,f,g) select the corresponding segments in the display as shown in figure a. The numeric designation chosen to represent the decimal digit is shown in figure b.



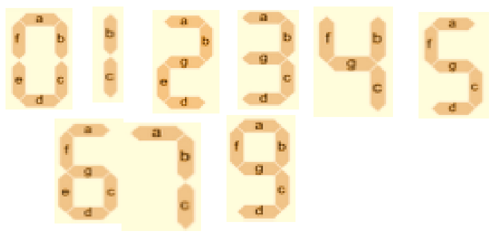


Fig(a):Segment designation Fig(b):Numerical designation for display



Truth table:

BCD				Segment O/P							
A	B	C	D	a	b	c	d	e	f	g	Displayed
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	x	x	x	x	x	x	x	Don't Care
1	0	1	1	x	x	x	x	x	x	x	Don't Care
1	1	0	0	x	x	x	x	x	x	x	Don't Care
1	1	0	1	x	x	x	x	x	x	x	Don't Care
1	1	1	0	x	x	x	x	x	x	x	Don't Care
1	1	1	1	x	x	x	x	x	x	x	Don't Care



From truth table we obtained-

a= $\sum$ (0,2,3,5,6,7,8,9)

b= $\sum$ (0,1,2,3,4,7,8,9)

c= $\sum$ (0,1,3,4,5,6,7,8,9)

d= $\sum$ (0,2,3,5,6,8,9)

e= $\sum$ (0,2,6,8)

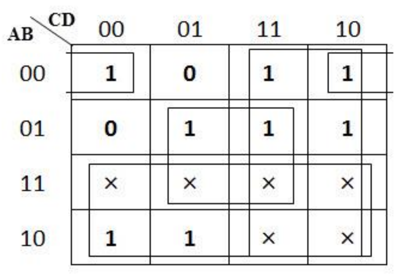
f= $\sum$ (0,4,5,6,8,9)

g= $\sum$ (2,3,4,5,6,8,9)

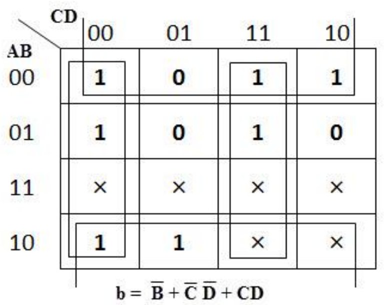
Don't care conditions,

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

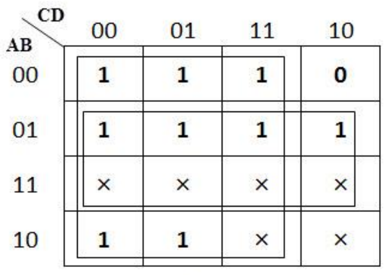
K-map:



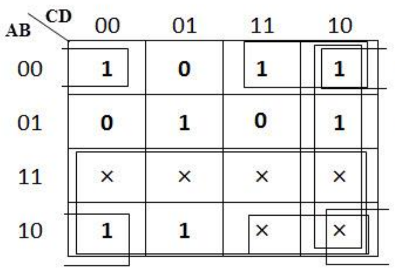
$a = A + C + BD + \overline{B}\,\overline{D}$



$b = \overline{B} + \overline{C}\,\overline{D} + CD$

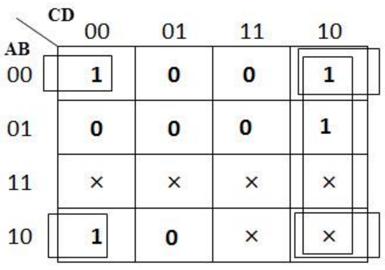


$c = B + \overline{C} + D$



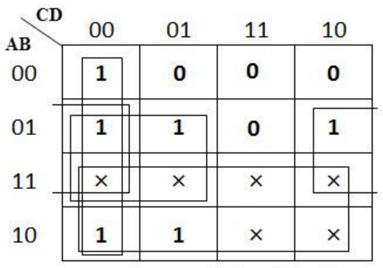
AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	×	×	×	×
10	1	1	×	×

$d = \overline{B}\,\overline{D} + C\overline{D} + B\overline{C}D + \overline{B}C + A$



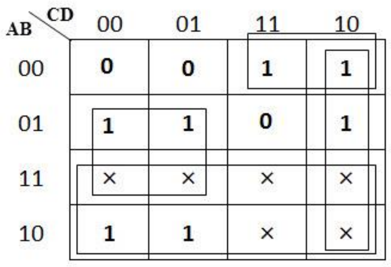
AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	×	×	×	×
10	1	0	×	×

$e = \overline{B}\,\overline{D} + C\overline{D}$



AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	×	×	×	×
10	1	1	×	×

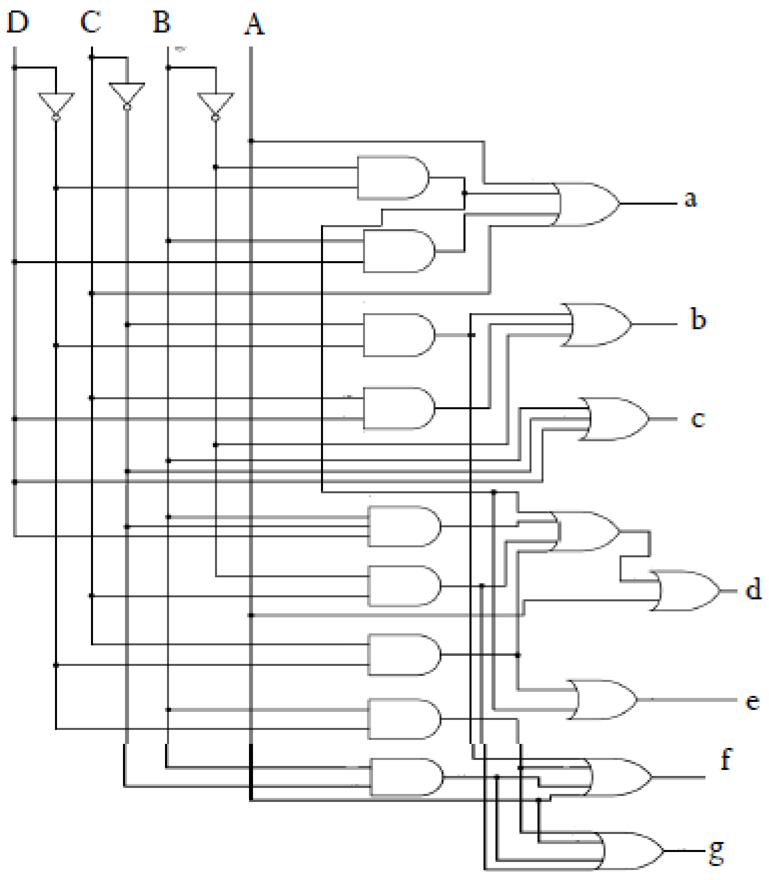
$f = A + \overline{C}\,\overline{D} + B\overline{C} + B\overline{D}$



AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	×	×	×	×
10	1	1	×	×

$g = \overline{B}C + C\overline{D} + B\overline{C} + B\overline{C} + A$

Logic Diagram:



Parity Generator and Checker

Parity Generator:

A parity generator is a combinational logic circuit that generates the parity bit in the transmitter.

- A parity bit is used for the purpose of detecting errors during transmission of binary information. It is an extra bit included with a binary message to make the number of 1's either odd or even.
- Types of parity: Even parity & Odd parity.
- In Even parity, added parity bit will make the total number of 1's an even amount.
- In Odd parity, added parity bit will make the total number of 1's an odd amount.

3-bit even parity generator truth table:

3-bit message			Even parity bit generator (P)
A	**B**	**C**	**Y**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Solving the truth table for all the cases where *P* is 1 using SOP method:

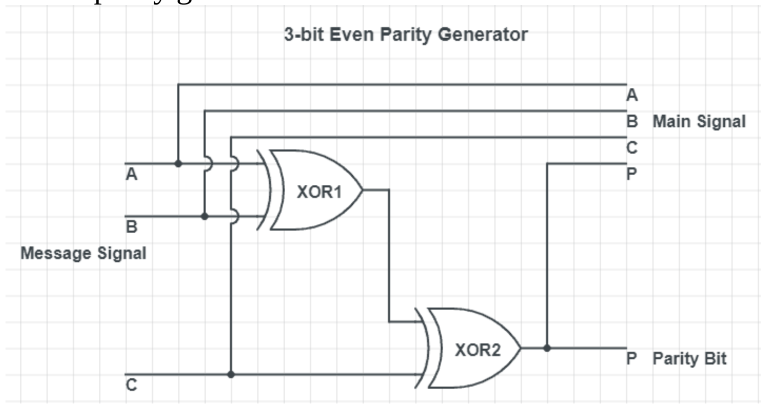
$$P = \bar{A}\,\bar{B}\,C + \bar{A}\,B\,\bar{C} + A\,\bar{B}\,\bar{C} + A\,B\,C$$

$$= \bar{A}\,(\bar{B}\,C + B\,\bar{C}) + A\,(\bar{B}\,\bar{C} + B\,C)$$

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

$$P = A \oplus B \oplus C$$

3-bit even parity generator circuit:



Parity checker:

A circuit that checks the parity in the receiver is called parity checker. The parity checker circuit checks for possible errors in the transmission.

- Since the information transmitted with even parity, the received must have an even number of 1's. If it has odd number of 1's, it indicates that there is an error occurred during transmission.

3-bit even parity checker truth table;

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

The output of the parity checker is denoted by PEC (Parity Error Checker). If there is error, that is, if it has odd number of 1's, it will indicate 1. If no then PEC will indicate 0.

$$PEC = \bar{A}\,\bar{B}(\bar{C}D + C\bar{D}) + \bar{A}B(\bar{C}\bar{D} + CD) + AB(\bar{C}D + C\bar{D}) + A\bar{B}(\bar{C}\bar{D} + CD)$$

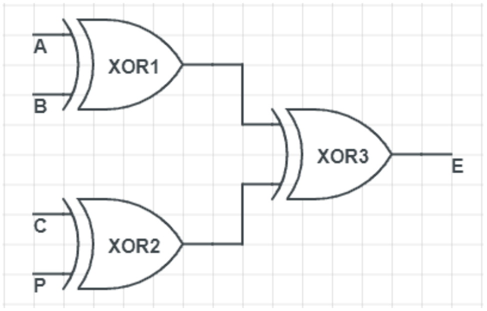
$$= \bar{A}\,\bar{B}(C \oplus D) + \bar{A}B(\overline{C \oplus D}) + AB(C \oplus D) + A\bar{B}(\overline{C \oplus D})$$

(Here truth table's $P = D$)

$$= (\bar{A}\,\bar{B} + AB)(C \oplus D) + (\bar{A}B + A\bar{B})(\overline{C \oplus D})$$

$$= (A \oplus B) \oplus (C \oplus D)$$

3-bit even parity checker circuit:



Combinational Logic with MSI and LSI

Binary Adder

This circuit sums up two binary numbers A and B of n-bits using full-adders to add each bit pair and carry from previous bit position.

Binary Parallel Adder:

A binary parallel adder is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full adders connected in cascade, with the output carry from one full adder connected to the input carry of the next full adder. An n bit parallel adder requires n full adders.

4-bit binary parallel adder:

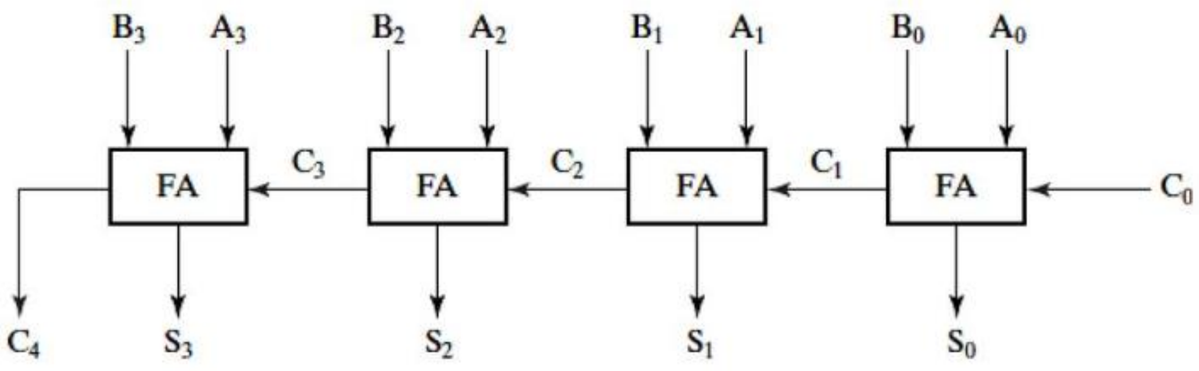


Fig: 4-bit binary parallel adder

A 4-bit binary parallel adder consists of 4-full adder. The augend bits are A_4, A_3, A_2, A_1 and addend bits are B_1, B_2, B_3, B_4. This parallel adder produces their sum as $C_4S_3S_2S_1S_0$ where C_4 is the final carry. The carries are connected in chain through the full-adders. The input carry to the first full adder is C_1 and the output carry from MSB position of full adder is C_4.

Q. Design a BCD-to-excess-3 code converter using a 4-bit full adders MSI circuit.

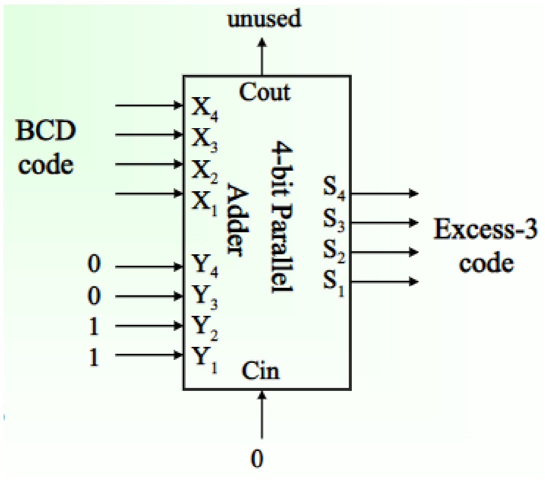
Solⁿ:

$Excess - 3\ code = BCD\ code + (0011)_2$

Augend bits = $X_4X_3X_2X_1$ *(Input bits)*

Addend bits = $Y_4Y_3Y_2Y_1 = 0011$

Excess-3 code = $S_4S_3S_2S_1$ *(output)*



Decimal adder/BCD adder:

BCD adder is a combinational digital circuit that adds two BCD digits in parallel and produces sum which is also BCD.

- In BCD adder, each input digit does not exceed 9, so the output sum can't be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input carry.
- Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result which may range from 0 to 19.

Truth table for BCD adder is:

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 content 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 1001, we obtain a non- valid BCD representation. The addition of binary 0110 (6 in decimal) to the binary sum converts it to the correct BCD representation and also produces an output carry.
- It is obvious from the table that a correction is needed when the binary sum has an output carry $k = 1$.
- The other six combination from 1010 to 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have 1.
- The condition for a correction and an output carry can be expressed by the Boolean function: $\boldsymbol{C = K + Z_8Z_4 + Z_8Z_2}$
- When output carry $\mathrm{C} = 0$, nothing is added to the binary sum.

- When output carry $C = 1$, binary 0110 is added to the binary sum through the bottom 4-bit binary adder to convert the binary sum into BCD sum. (*In fig. below*)

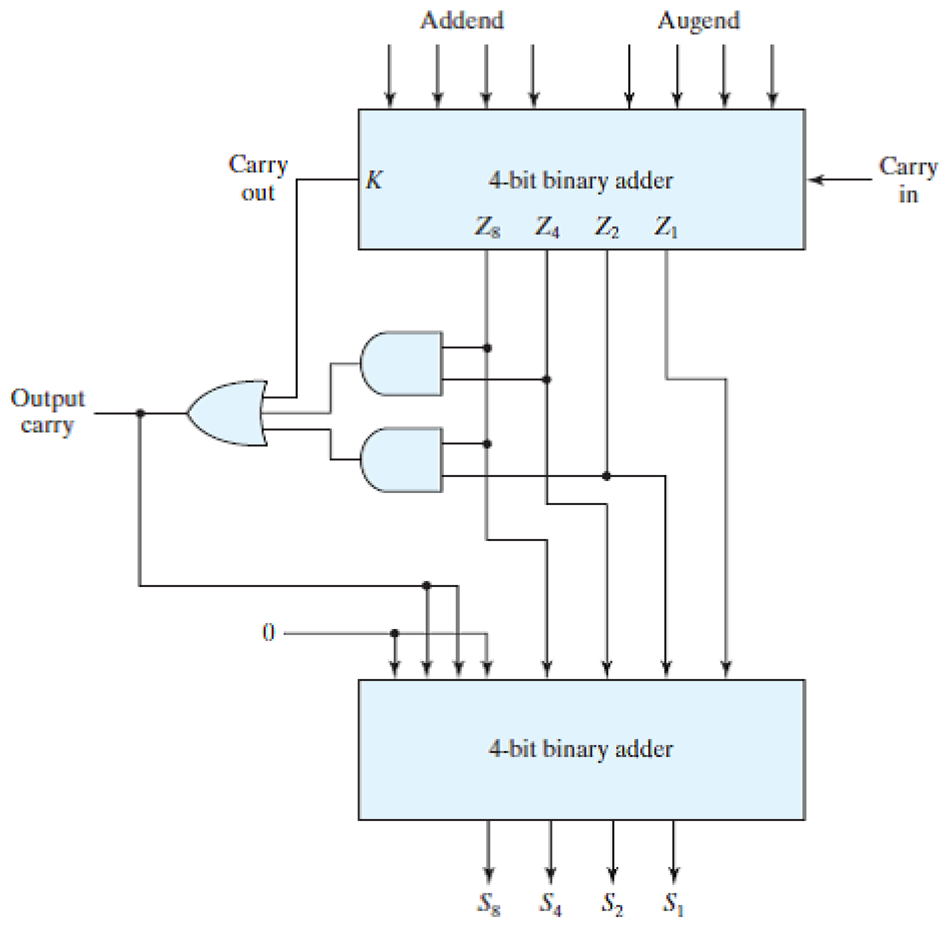
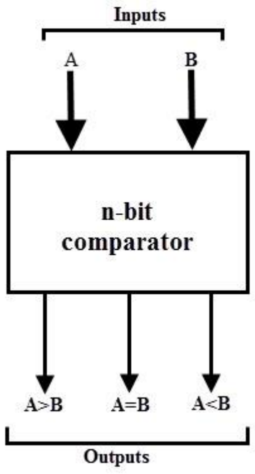


Fig: BCD adder

Magnitude Comparator

A magnitude comparator is a combinational circuit that compares two numbers A & B and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether $A > B, A = B, or\ A < B$.



Note: Out of these three outputs only one output will be 1 and other two outputs will be 0 at a time.

4-bit magnitude comparator:

4-bit magnitude comparator is a combinational logic circuit that compares two binary numbers each of 4-bits.
Consider two numbers A & B with four digits each.

$$A = A_3A_2A_1A_0$$
$$B = B_3B_2B_1B_0$$

Verification of $(\boldsymbol{A} = \boldsymbol{B})$***:***

- The equality relation of each pair of bits can be expressed:
$$x_i = A_iB_i + \bar{A}_i\bar{B}_i\ ,\ i = 0, 1, 2, 3$$
Where $x_i = 1$ only if $A_i = B_i$ and $x_i = 0$ only if $A_i \neq B_i$.
- For equality condition to exist, all x_i variables must be equal to 1. A & B will be equal if $x_3x_2x_1x_0 = 1$.

$$\boxed{\therefore (\boldsymbol{A} = \boldsymbol{B}) = \boldsymbol{x_3x_2x_1x_0}}$$

Verification of $(\boldsymbol{A} > \boldsymbol{B})$***:***

- If $A_3 > B_3$ then $A > B$, it means $A_3 = 1$ & $B_3 = 0$. Therefore A is greater than B if $\boldsymbol{A_3\bar{B}_3 = 1}$.
- If $A_3 = B_3 (i.e\ x_3 = 1)$ and $A_2 > B_2$ then $A > B$. Therefore A is greater than B if $\boldsymbol{x_3A_2\bar{B}_2 = 1}$.
- If $A_3 = B_3 (i.e\ x_3 = 1)$ & $A_2 = B_2 (i.e\ x_2 = 1)$ and $A_1 > B_1$ then $A > B$. Therefore A is greater than B if $\boldsymbol{x_3x_2A_1\bar{B}_1 = 1}$.
- If $A_3 = B_3 (i.e\ x_3 = 1)$ & $A_2 = B_2 (i.e\ x_2 = 1)$ & $A_1 = B_1 (i.e\ x_1 = 1)$ and $A_0 > B_0$ then $A > B$. Therefore A is greater than B if $\boldsymbol{x_3x_2x_1A_0\bar{B}_0 = 1}$.

$$\boxed{\therefore (\boldsymbol{A} > \boldsymbol{B}) = \boldsymbol{A_3\bar{B}_3 + x_3A_2\bar{B}_2 + x_3x_2A_1\bar{B}_1 + x_3x_2x_1A_0\bar{B}_0}}$$

In the same manner we can derive the expression for $(\boldsymbol{A} < \boldsymbol{B})$.

$$\boxed{\therefore (\boldsymbol{A} > \boldsymbol{B}) = \boldsymbol{\bar{A}_3B_3 + x_3\bar{A}_2B_2 + x_3x_2\bar{A}_1B_1 + x_3x_2x_1\bar{A}_0B_0}}$$

Logic Diagram:

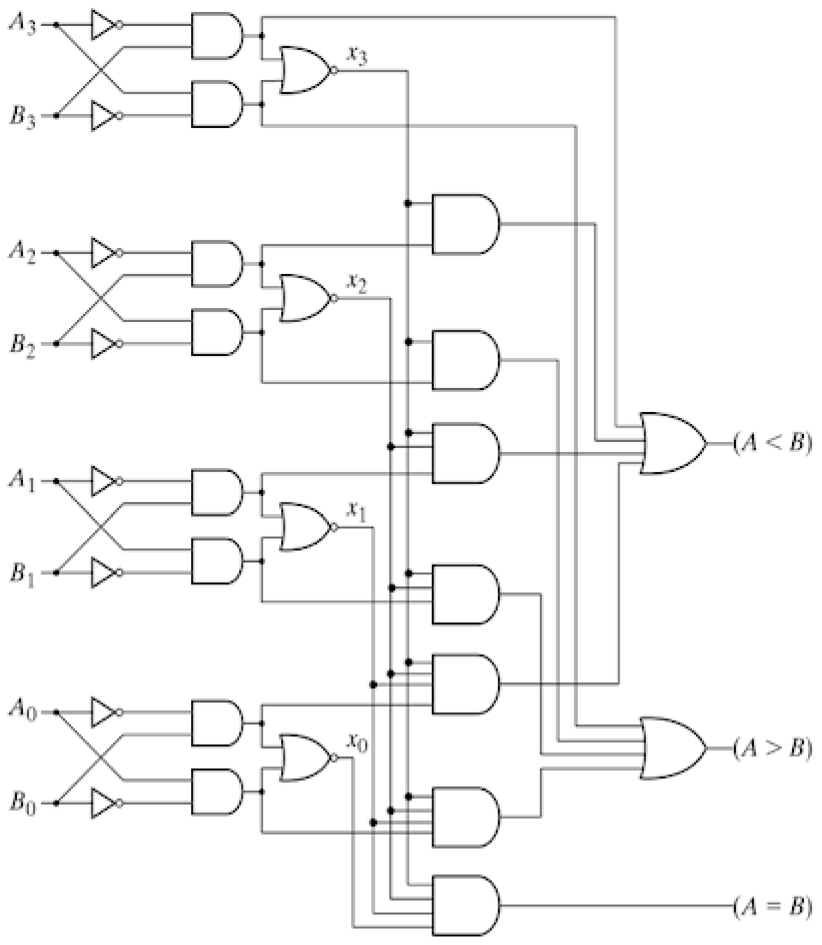
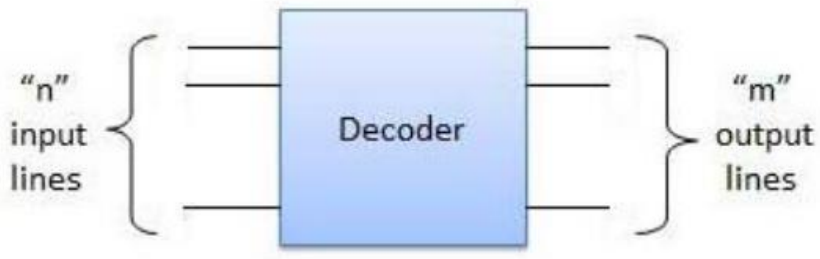


Fig. 4-17 4-Bit Magnitude Comparator

Decoders

A decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.

- If n-bit decoded information has unused or don't care combinations, the decoder output will have less than 2^n outputs.
- The decoders presented here are called $n - to - m$ line decoders where $m \leq 2^n$. Their purpose is to generate the 2^n (or less) minterms of n input variables.



3-to-8 line decoder:

The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables.
A particular application of this decoder would be a binary-to-octal conversion. The input variable may represent a binary number and the outputs will then represent the eight digits in the octal number system

Three inputs: X, Y & Z
Eight outputs: $D_0 - D_7$

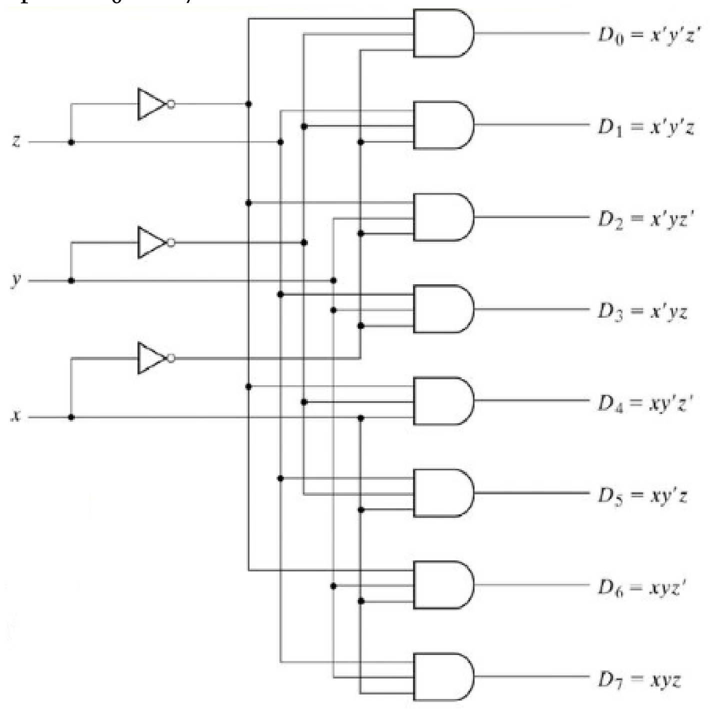


Fig: 3-to-8 line decoder

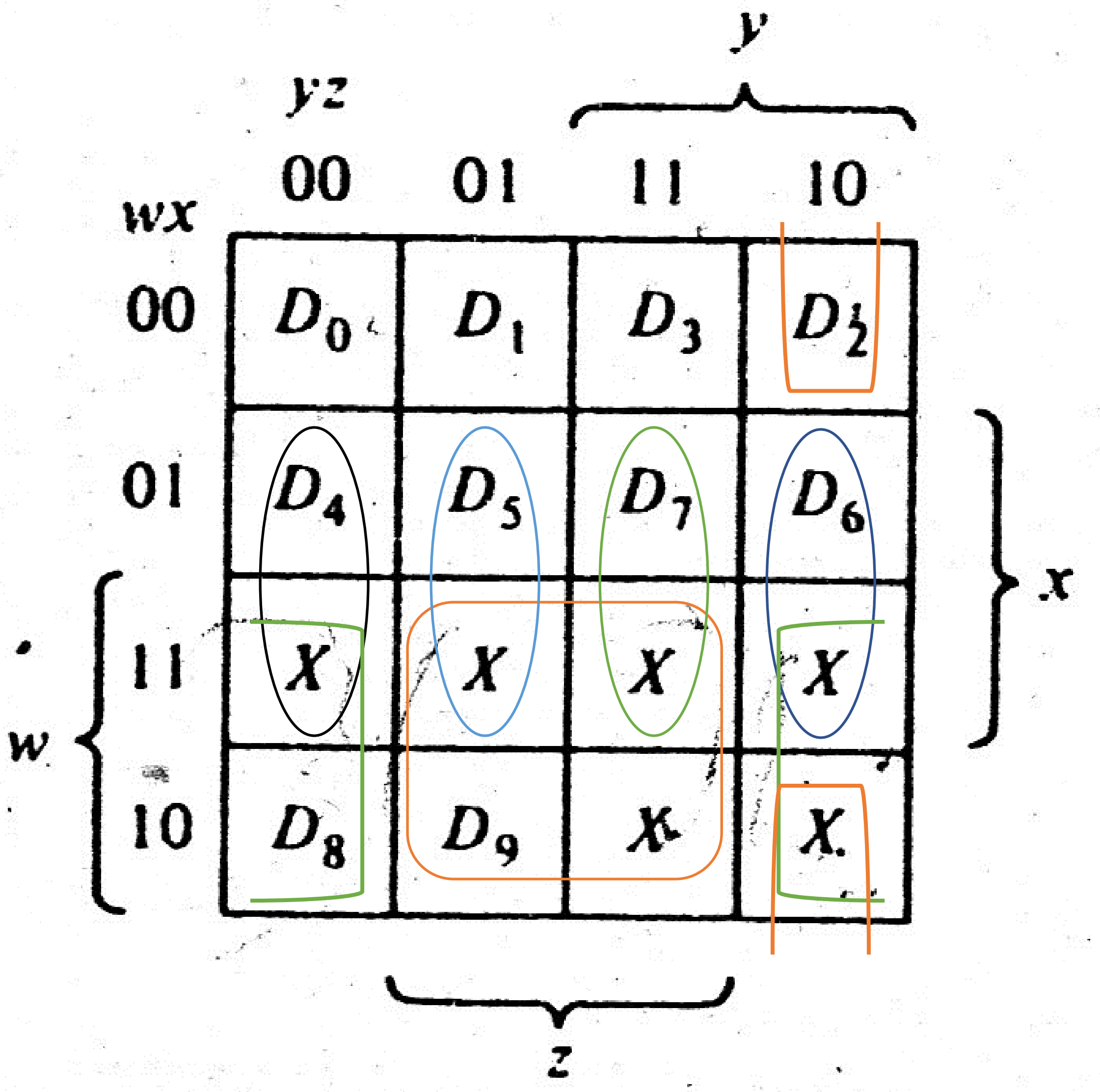
Truth table:

Inputs			Outputs							
X	Y	Z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_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**

From the truth table it is observed that 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.

BCD-to-Decimal Decoder

The decoder which convert binary decimal code into decimal values is called BCD to decimal decoder. The BCD code uses 4-bits and therefore there can be 2^4 input combinations. This produces 16-different output signals but the decimal digits are from 0 to 9. Hence six input combinations are not used in decimal system. Therefore we can use don't care to represent 10 to 15 and use K-map to simplify the circuit.



Simplified expression for different outputs are:

$D_0 = w'x'y'z'$ $D_1 = w'x'y'z$ $D_2 = x'yz'$ $D_3 = x'yz$ $D_4 = xy'z'$

$D_5 = xy'z$ $D_6 = xyz'$ $D_7 = xyz$ $D_8 = wz'$ $D_9 = wz$

Logic diagram of BCD-to-decimal decoder:

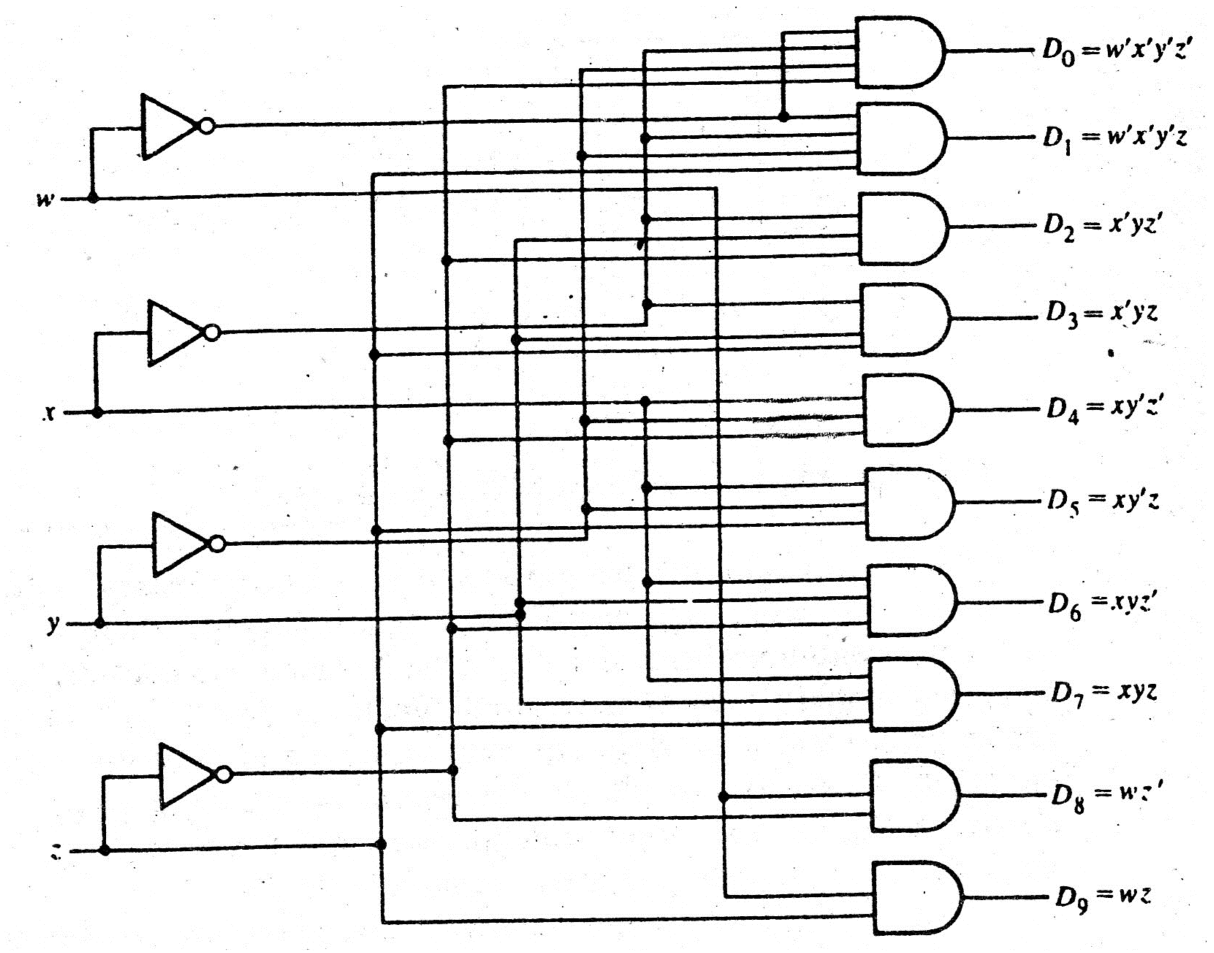


Fig: BCD to Decimal decoder

Q. Implement a full-adder circuit with a decoder and two OR gates.

Sol^n:

The truth table for full adder:

Input			Output	
A	B	Cin	Sum	Carry
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

From the truth table

$S(A, B, C_{in}) = \sum(1, 2, 4, 7)$

$C(A, B, C_{in}) = \sum(3, 5, 6, 7)$

Since there are three inputs and a total of eight minterms. So we need 3-to-8 line decoder. The decoder generates the eight minterms for A, B & C_{in}. The OR gate for output sum (S) forms the sum of minterms 1, 2, 4 & 7. The OR gate for the output carry (C) forms the sum of minterms 3, 5, 6 & 7.

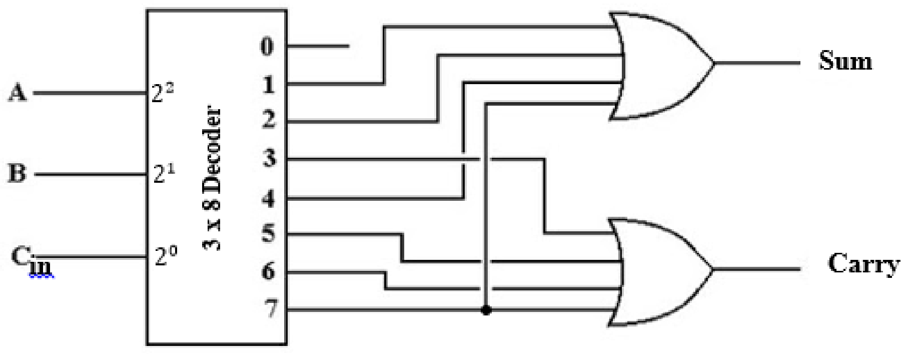


Fig: Full adder implementation with decoder

Encoder

An encoder is a combinational circuit that performs the inverse operation from that of decoder. It has 2^n input lines and n output lines.

The output lines generate the binary code corresponding to the input value.

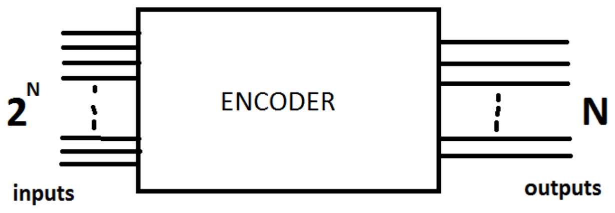


Fig: Block diagram of encoder

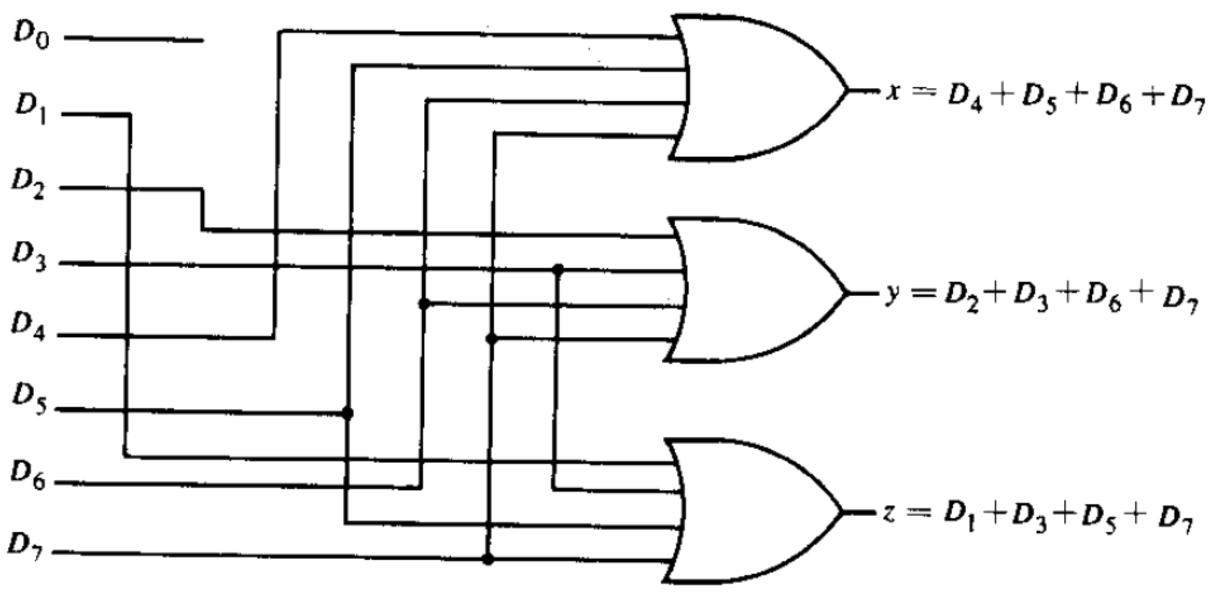
E.g. **Octal to binary encoder** which has 8 inputs and 3 outputs.

Truth table for octal to binary encoder:

INPUT								OUTPUT		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	X	Y	Z
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

Boolean function of output variables:
$X = D_4 + D_5 + D_6 + D_7$
$X = D_2 + D_3 + D_6 + D_7$
$X = D_1 + D_3 + D_5 + D_7$

Logic circuit:


D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
$x = D_4 + D_5 + D_6 + D_7$
$y = D_2 + D_3 + D_6 + D_7$
$z = D_1 + D_3 + D_5 + D_7$

Limitation: Only one input can be enabled at a time. If two inputs are enabled at the same time, then output is undefined.

Q. Design a 3 to 8 line decoder using two 2 to 4 line decoder and explain it.
Sol^n:

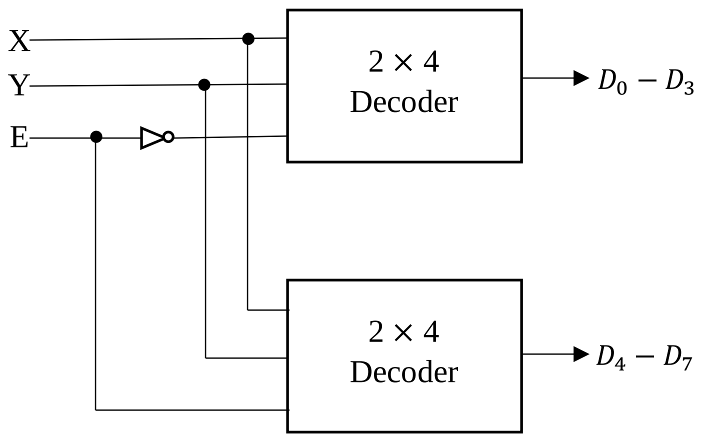
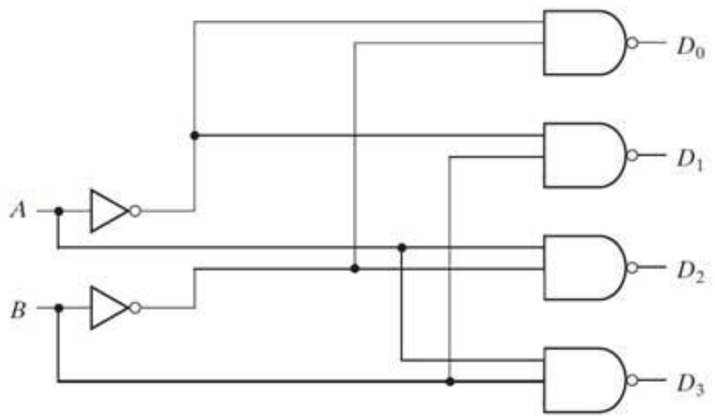


Fig: 3 to 8 decoder using two 2 to 4 decoder

The figure shows two 2×4 decoder with enable input (E) connected to form a 3×8 decoder. When $E = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's and the top four outputs generate minterms 000 to 001. When $E = 1$, the enable conditions are reversed. The bottom decoder outputs generate minterms 100 to 111 while the outputs of the top decoder are all 0's.

Q. Design a 2-to-4 line decoder using NAND gates.
Sol^n:



Truth table:

Inputs		Outputs			
A	B	D_0	D_1	D_2	D_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

For the NAND decoder only one output can be LOW and equal to logic '0' at any given time with all other outputs being HIGH at logic '1'.

Note: *Similar method for 3-to-8 line decoder in which 3-lines of input are present and 8 output lines.*

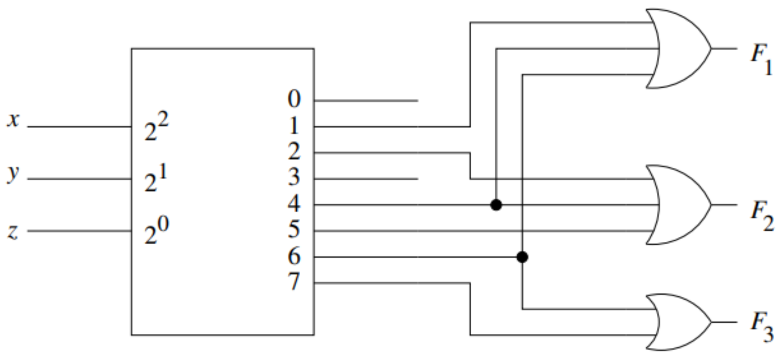
Q. Using a decoder and external gates, design the combinational circuit defined by the following three Boolean functions:

$$F_1 = x'y'z + xz'$$
$$F_2 = x'yz' + xy'$$
$$F_3 = xyz' + xy$$

Sol^n^:

Truth table:

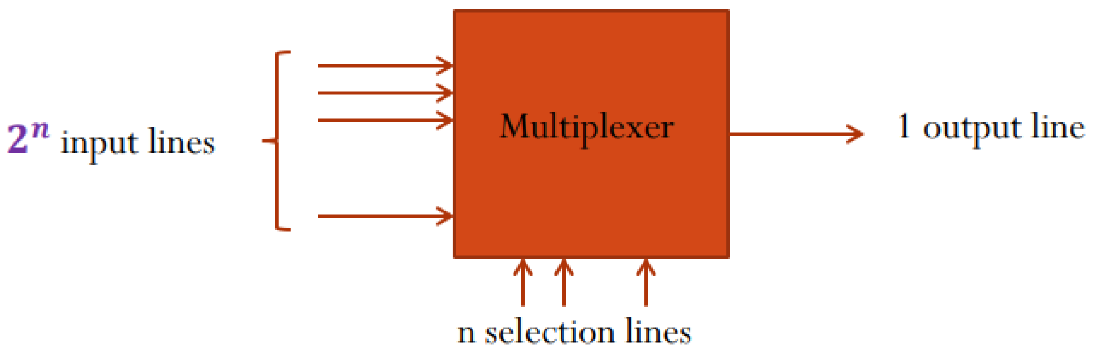
x	y	z	F_1	F_2	F_3
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1



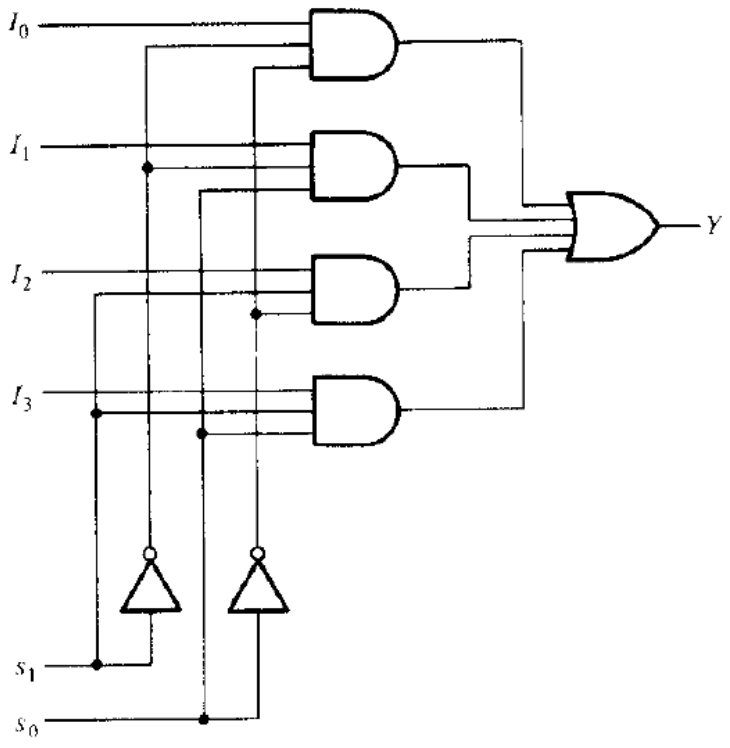
Logic Diagram

Multiplexer (MUX)

- A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line.
- Multiplexing is the process of transmitting a large number of information over a single line.
- The selection of a particular input lines is controlled by a set of selection lines. Normally there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.
- A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output line.



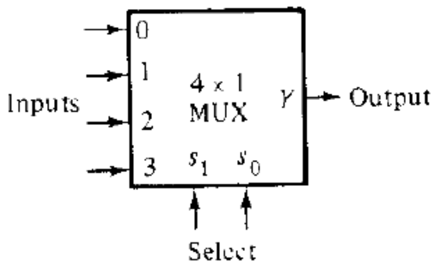
4-to-1 line Multiplexer:



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

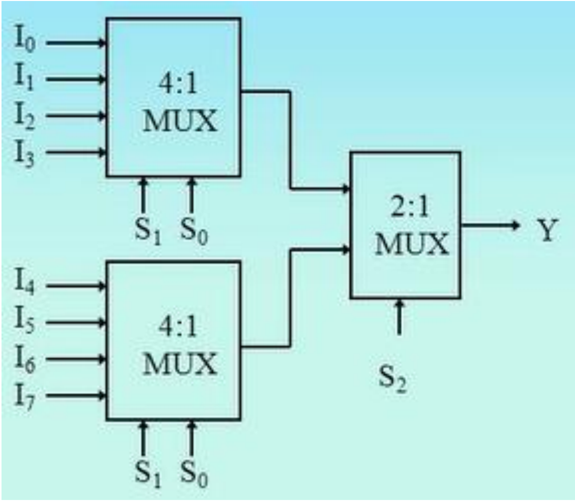
(b) Function table



(c) Block diagram

Q. *Design a 8-to-1 line multiplexer using lower order multiplexers and explain it.*

Sol^n:



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

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

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

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

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

Q. Implement the Boolean function $F(A,B,C) = \sum(1,3,5,6)$ with multiplexer.

Sol^n:
The multiplexer can be implemented with 4 to 1 multiplexer.

Note: *It is possible to generate n+1 variables with 2^n to 1 mutiplexer.*

Now, truth table for the given function is:

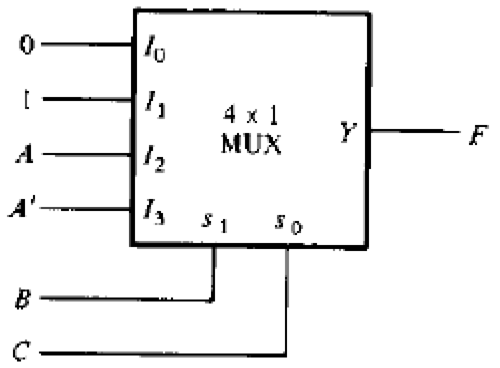
Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Now the implementation table is

	I_0	I_1	I_2	I_3
A'	0	$\textcircled{1}$	2	$\textcircled{3}$
A	4	$\textcircled{5}$	$\textcircled{6}$	7
	0	1	A	A'

- If the minterms in a column are not circled, then apply 0 to the corresponding multiplexer unit.
- If the 2 minterms are circled, then apply 1 to the corresponding multiplexer unit.
- If the bottom minterm is circled, and top is not circled then apply A to the corresponding multiplexer unit.
- If the top minterm is circled, and bottom is not circled then apply A' to the corresponding multiplexer unit.

Multiplexer implementation:



Q. Implement the Boolean function $F(A, B, C, D) = \sum(0, 1, 3, 4, 8, 9, 15)$ by multiplexer.

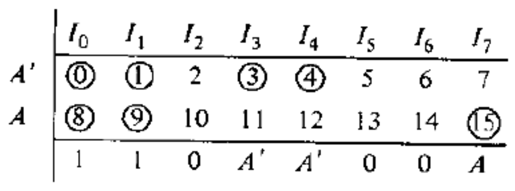
Sol^n:

This function can be implemented with 8 to 1 MUX.

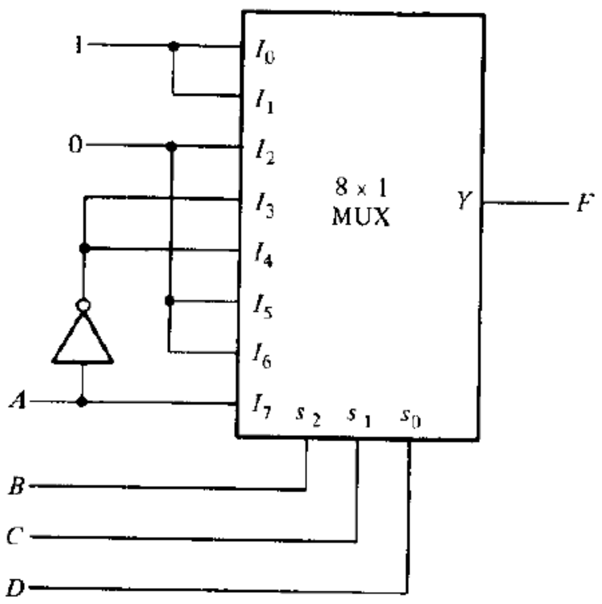
The truth table for the function is

Minterm	A	B	C	D	F
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	1

Now the implementation table and multiplexer implementation are given below:

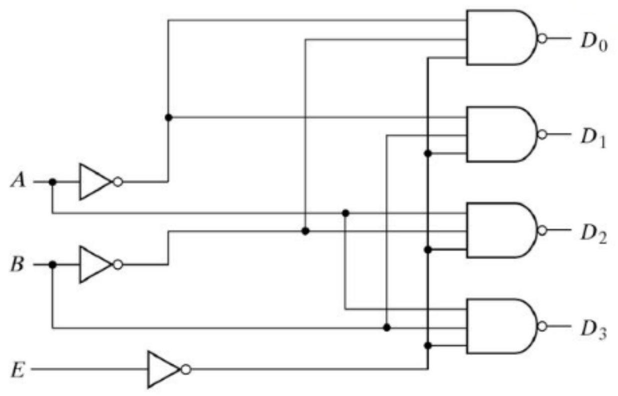


	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	$\textcircled{0}$	$\textcircled{1}$	2	$\textcircled{3}$	$\textcircled{4}$	5	6	7
A	$\textcircled{8}$	$\textcircled{9}$	10	11	12	13	14	$\textcircled{15}$
	1	1	0	A'	A'	0	0	A

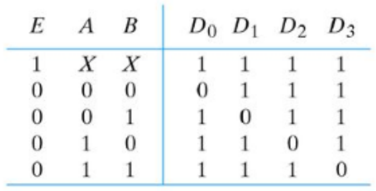


Demultiplexer (DEMUX)

- A decoder with an enable input can function as a de-multiplexer.
- A de-multiplexer is a circuit that **receives information on a single line** and transmit this information on one of 2^n **possible output lines**. The selection for particular output line is controlled by the bit values of n selection lines.



(a) Logic diagram

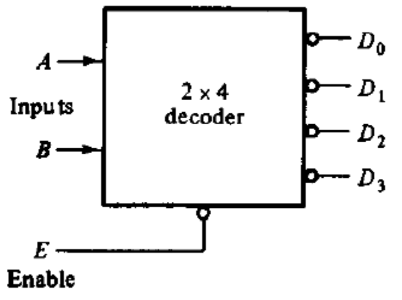


E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

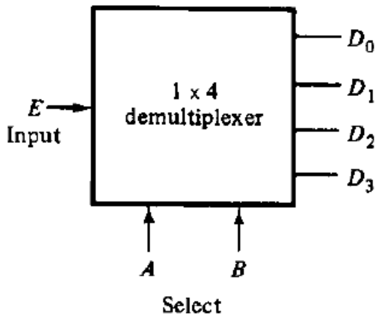
(b) Truth table

Fig: A 2-to-4 line decoder with enable (E) input

The decoder of fig can function as a de-multiplexer if the E line is taken as a data input line and lines A and B are taken as the selection lines.



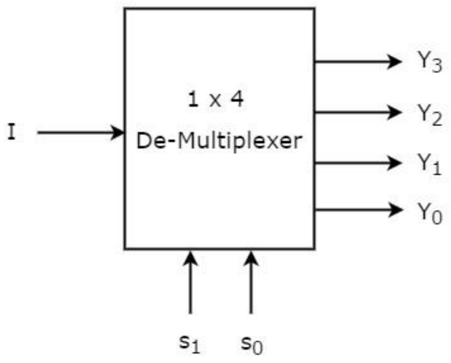
(a) Decoder with enable

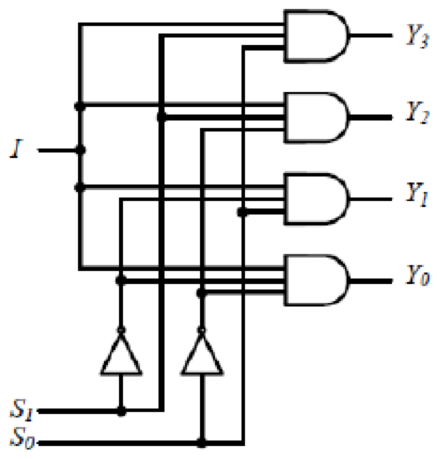


(b) Demultiplexer

1 to 4 DEMUX:

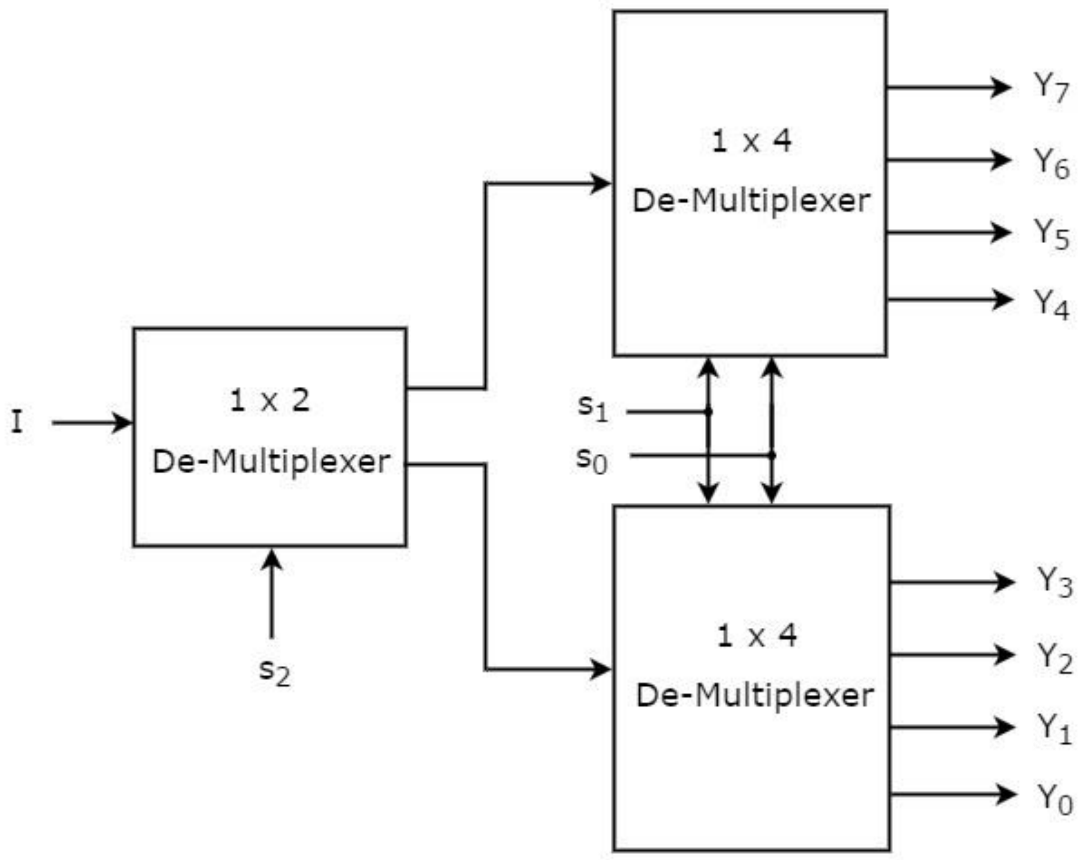
The 1:4 Demux consists of 1 data input bit, 2 control bits and 4 output bits. I is the input bit, Y_0, Y_1, Y_2, Y_3 are the four output bits and S_0 and S_1 are the control bits.





S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

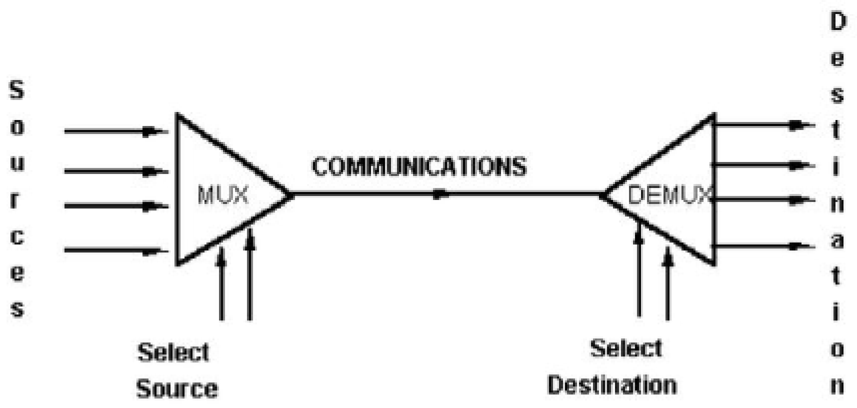
1 to 8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer:



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

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

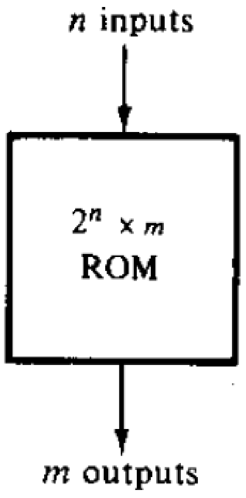
MUX-DEMUX Application Example



- This enables sharing a single communication line among a number of devices.
- At any time, only one source and one destination can use the communication line.

Read Only Memory (ROM)

- **A read-only memory (ROM)** is a device that includes **both the decoder and the OR gates** within a **single IC package**. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration by "programming" the ROM.
- A ROM is essentially a memory (or storage) device in which a fixed set of binary information is stored.
- The binary information must first be specified by the user and is then embedded in the unit to form the required interconnection pattern. ROM's come with special internal links that can be fused or broken. The desired interconnection for a particular application requires that certain links be fused to form the required circuit paths. Once a pattern is established for a ROM, it remain fixed even when power is turned off and then on again.
- A ROM consists of n input lines and m output lines.
- Each bit combination of input variables is called an address.
- Each bit combination that comes out of the output lines is called a word. The number of bits per word is equal to the number of output lines m.
- A ROM with n input lines has 2^ndistinct addresses, so there are 2^n distinct words which are said to be stored in the unit.



- Internally, the ROM is a combinational circuit with AND gates connected as a decoder and a number of OR gates equal to the number of outputs in the unit.

<u>Combinational Logic implementation of ROM:</u>

When a combinational circuit is implemented by means of ROM the function must be expressed in sum of min terms or better yet by a truth table.

Q. Implement the following combinational logic function with a 4X2 ROM.

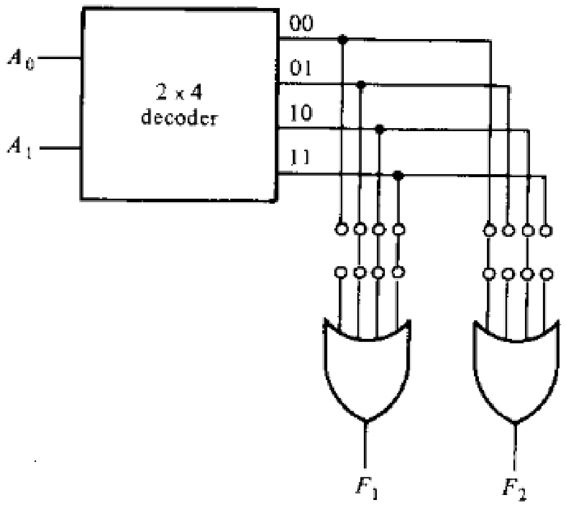
A_1	A_0	F_1	F_2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	1	0

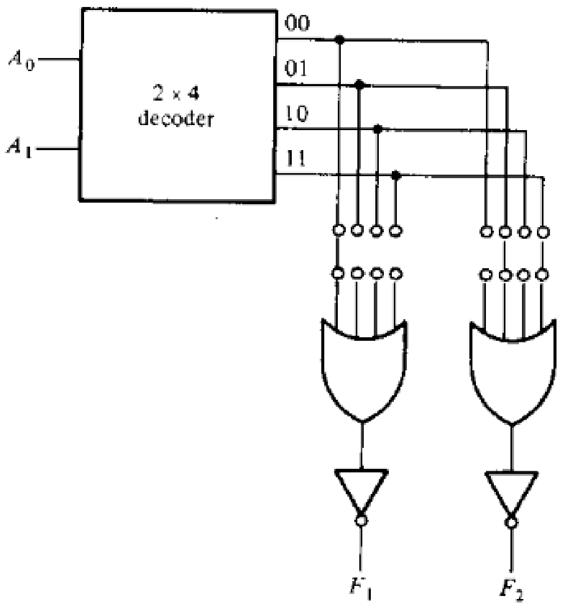
Sol^n:

Truth table specifies a combinational circuit with 2 inputs and 2 outputs. The Boolean function can be represented in SOP as;

$$F_1(A_1, A_0) = \Sigma(1, 2, 3)$$
$$F_2(A_1, A_0) = \Sigma(0, 2)$$

Combinational-circuit implementation with a 4 x 2 ROM:





<u>ROM with AND-OR gates</u> ***<u>ROM with AND-OR-INVERT gates</u>***

Q. Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

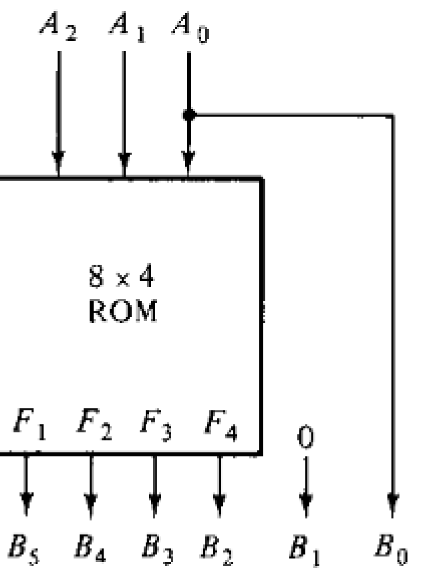
Sol^n:

First step is to derive the truth table for the combinational circuit

Inputs			Outputs						
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	Decimal
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49

Output B_0 is always equal to input A_0; so there is no need to generate B_0 with a ROM since it is equal to an input variable. Moreover, output B1 is always 0, so this outputs is always known.

Implementation by ROM:



(a) Block diagram

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
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	1
1	1	1	1	1	0	0

(b) ROM truth table

Types of ROM:

1. Mask ROM

- Permanent programming done at fabrication time
- Fabrication take place at factory as per customer order
- Very expensive and therefore feasible only for large quantity orders
- Once the memory is programmed during the manufacturing process, the user cannot alter the programs.

2. PROM (Programmable ROM)

- A blank chip which can be programmed only once using a special device called programmer.
- Once it's programmed its content cannot be modified or erased.

3. EPROM (Erasable Programmable ROM)

- Can be programmed multiple times.
- Its content can be erased by using UV (ultra violet) light.
- Exposure to the UV light will erase all contents.

4. EEPROM (Electrically Erasable Programmable ROM)

- Similar to EPROM but its contents can be electrically erased and re-written without having to remove it from the computer.

Programmable Logic Array (PLA)

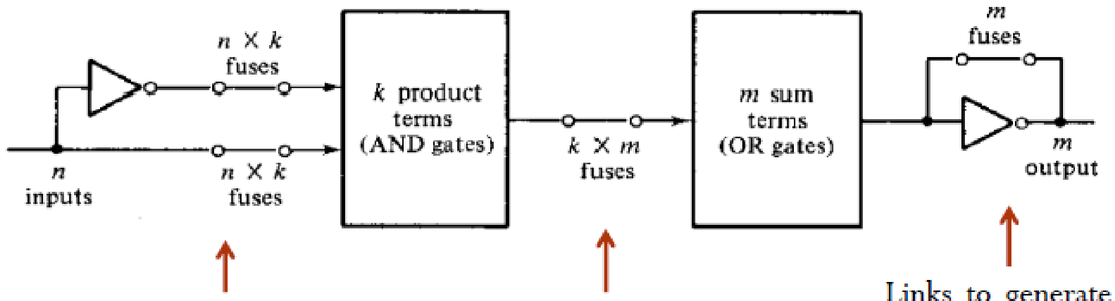
A combinational circuit may occasionally have don't care conditions. When implemented with a ROM, a don't care condition becomes an address input that will never occur. The words at the don't care addresses need not be programmed and may be left in their original state (all 0's or all 1's). The result is that not all the bit patterns available in the ROM are used, which may be considered as waste of available equipment.
For example, a combinational circuit that converts a 12-bit card code to a 6-bit internal alphanumeric code.
* It consists 12 inputs and 6 outputs. The size of the ROM must be 4096 × 6 ($2^{12} \times 6$).
* There are only 47 valid entries for the card code, all other input combinations are don't care. The remaining 4049 words of ROM are not used and are thus wasted.

*So, **Programmable Logic Array*** is a LSI component that can be used in economically as an alternative to ROM where number of don't-care conditions is excessive.

✓ PLA does not provide full decoding of the variables and does not generate all the minterms as in the ROM.

Block diagram of PLA:

A block diagram is shown in fig. It consists 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.



Links between all n inputs and their complement values to each of the AND gates.
2n X k links

Links between outputs of the AND gates and the inputs of the OR gates.
k X m links

Links to generate AND-OR form or, AND-OR-INVERT form
m links

✓ The number of programmed links is $2n \times k + k \times m + m$, whereas that of a ROM is $2^n \times m$.

Implementation of combinational circuit by PLA:

$$F_1 = AB' + AC$$
$$F_2 = AC + BC$$

PLA program table:

	Product term	Inputs A	B	C	Outputs F_1	F_2
AB'	1	1	0	–	1	–
AC	2	1	–	1	1	1
BC	3	–	1	1	–	1
					T	T	T/C

Input side:
1=uncomplemented in term
0=complemented in term
- = does not participate
Output side:
1= term connected to output
- = no connection to output

PLA Logic Circuit:

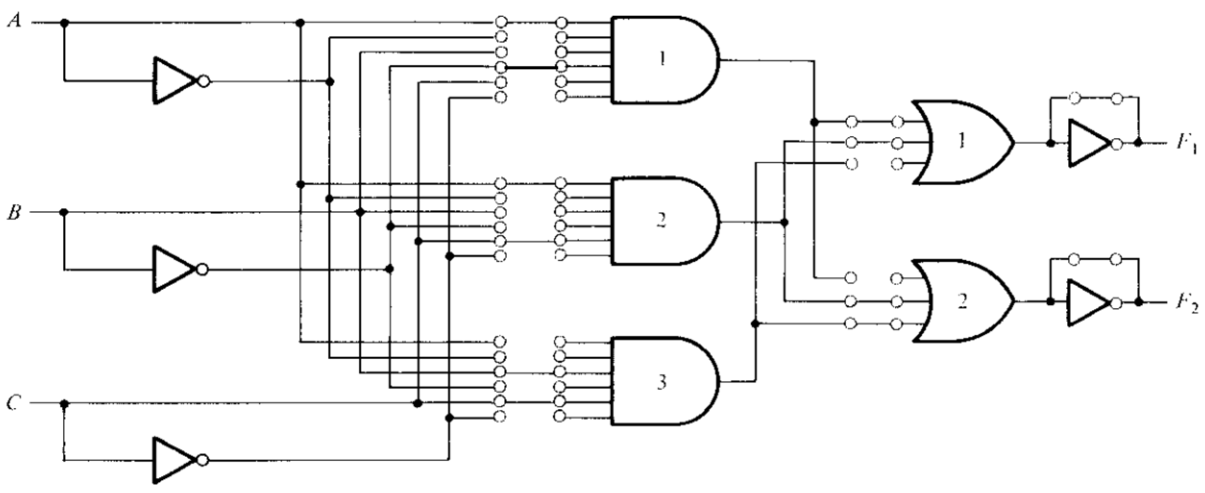


Fig: PLA with 3 inputs, 3 product terms, and 2 outputs

PLA program table consists of three columns:

- ***First column:*** lists the product terms numerically.
- ***Second column:*** specifies the required paths between inputs and AND gates.
- ***Third column:*** specifies the paths between the AND gates and the OR gates.

Under each output variable, we write a *T* (for true) if the output inverter is to be bypassed, and C (for complement) if the function is to be complemented with the output inverter.

Note: PLA implements the functions in their sum of products form (standard form, not necessarily canonical as with ROM). Each product term in the expression requires an AND gate. It is necessary to simplify the function to a minimum number of product terms in order to minimize the number of AND gates used.

Q. A combinational circuit is defined by the functions:

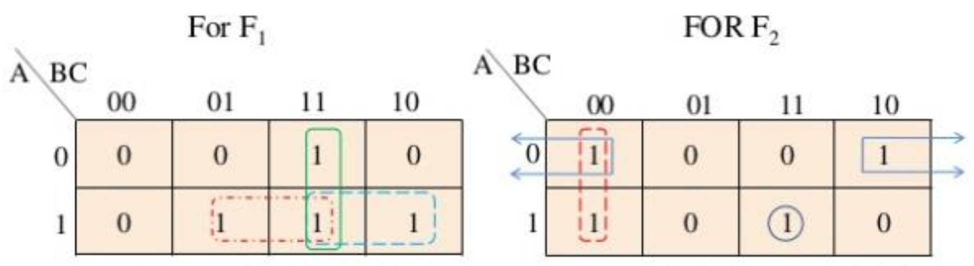
$$F_1(A,B,C) = \sum(3,5,6,7)$$

$$F_2(A,B,C) = \sum(0,2,4,7)$$

Implement the circuit with a PLA having three inputs, four product terms, and two outputs.

Sol^n:

First of all we have to write the function in minimize SOP form:



$F_1 = AC + AB + BC$ $\qquad$ $F_2 = \overline{B}\,\overline{C} + ABC + \overline{A}\,\overline{C}$

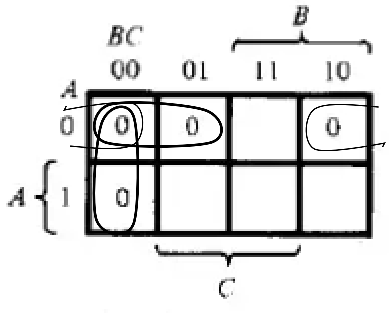
There are six product terms in F_1 and F_2, but only four product terms are allowed to use.

Now implement $F_1'(A,B,C)$

$$F_1'(A,B,C) = \sum(0,1,2,4)$$
$$F_2(A,B,C) = \sum(0,2,4,7)$$

From these equation it is clear that the minterms 0, 2 and 4 are common.

Now obtain the minimized expression by using them

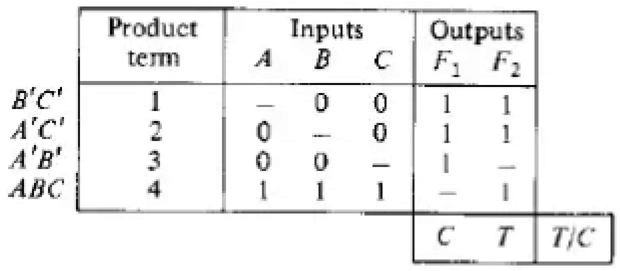


$F_1' = B'C' + A'C' + A'B'$

Now four product terms are $B'C', A'C', A'B'$ *and* ABC.

$F_1 = B'C' + A'C' + A'B'$
$F_2 = B'C' + ABC + A'C'$

Now, PLA program table:



	Product term	Inputs			Outputs	
		A	B	C	F_1	F_2
$B'C'$	1	–	0	0	1	1
$A'C'$	2	0	–	0	1	1
$A'B'$	3	0	0	–	1	–
ABC	4	1	1	1	–	1
				T/C	C	T

Note that output F_1 is the normal (or true) output even though a C is marked under it. This is because F_1' is generated prior to the output inverter. The inverter complements the function to produce F_1 in the
output.
Draw PLA circuit yourself.

<u>References</u>:

M. Morris Mano, "Digital Logic & Computer Design"