Unit 3

Simplification of Boolean functions

We will get four Boolean product terms by combining two variables x and y with logical AND operation. These Boolean product terms are called as min terms or standard product terms. The min terms are x'y', x'y, xy' and xy.
Similarly, we will get four Boolean sum terms by combining two variables x and y with logical OR operation. These Boolean sum terms are called as Max terms or standard sum terms. The Max terms are x+y, x+y', x'+y and x'+y'.
The following table shows the representation of min terms and MAX terms for 2 variables.

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

If the binary variable is '0', then it is represented as complement of variable in min term and as the variable itself in Max term. Similarly, if the binary variable is '1', then it is represented as complement of variable in Max term and as the variable itself in min term.
From the above table, we can easily notice that min terms and Max terms are complement of each other. If there are 'n' Boolean variables, then there will be 2n min terms and 2n Max terms.

Canonical and standard forms

We can write Boolean expressions in many ways, but some ways are more useful than others. We will look first at the "term" types, made up of "literals".

Minterms

- A **minterm** is a special product (ANDing of terms) of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n minterms (since each variable can appear complemented or not)
- A three-variable function, such as f(x, y, z), has $2^3 = 8$ minterms:
  x'y'z' x'y'z x'yz' x'yz
  xy'z' xy'z xyz' xyz
- Each minterm is **true** for exactly one combination of inputs:

Maxterms

- A maxterm is a sum (or ORing of terms) of literals, in which each input variable appears exactly once.
- A function with n variables has 2^n maxterms
- The maxterms for a three-variable function f(x, y, z):
  x' + y' + z' x' + y' + z x' + y + z' x'+ y + z x + y' + z' x + y' + z x + y + z'
  x + y + z
- Each maxterm is **false** for exactly one combination of inputs:

			Minterms		Maxterms	
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

Table: Minterms and Maxterms for 3 Binary Variables with their symbolic shorthand

NOTE: Each maxterm is the complement of its corresponding minterm and vice versa (viz. $m_0 = M_0'$, $M_4 = m_4'$ etc.)
A Boolean function may be expressed algebraically (SOP or POS form) from a given truth table by:

- Forming a minterm for each combination of the variables that produces a 1 in the function, and then taking the OR of all those terms.
- Forming a maxterm for each combination of the variables that produces a 0 in the function, and then taking the AND of all those maxterms.

Canonical forms (Canonical SoP and PoS forms)

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

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

1. Canonical SoP form
2. Canonical PoS form

Boolean functions expressed as a sum of min terms or product of maxterms are said to be in ***canonical form***. These complementary techniques are describes below. Canonical form is not efficient representation but sometimes useful in analysis and design. In an expression in canonical form, every variable appears in every term.

Sum of Minterms (Sum of Products or SoP)

We have seen, one can obtain 2" distinct minterms form n binary input variables and that any Boolean function can be expressed as a sum of minterms. The minterms whose sum defines the Boolean function are those that give the 1's of the function in a truth table. It is sometimes convenient to express the Boolean function in its **sum of minterms form**.

- If not in this form, it can be made so by first expanding the expression into a sum of AND terms.
- 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.

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

Solution: The function has three variables *A, B,* and C.

- The first term A is missing two variables; therefore:
  A = A (B + B ') = AB + AB' [B is missing variable]
- This is still missing one variable C, so A = AB(C + C') + AB'(C + C') = ABC + ABC' + AB'C + AB'C'
- The second term B'C is missing one variable: B'C = B'C(A + A') = AB'C + A'B'C
- Combing all terms, we have
  F = A+B'C = ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C
- But AB'C appears twice, and according to THEOREM 1 of Boolean algebra x + x = x, it is possible to remove one of them. Rearranging the minterms in ascending order, we finally obtain:
  F = A'B'C + AB'C' + AB'C + ABC' + ABC
  m1 + m4 + m5 + m6 + m7

Shorthand notation,

F(A,B,C) =∑(1, 4, 5, 6, 7)

The summation symbol ∑ stands for the ORing of terms: the numbers following it are the minterms of the function.

An **alternate procedure** for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table.

Truth Table for $F = A + B'C$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Truth table for $F = A + B'C$, from the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Example

Consider the following truth table.

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

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

Therefore, the Boolean function of output is, f=p'qr + pq'r + pqr' + pqr. This is the canonical SoP form of output, f.

We can also represent this function in following two notations.

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

f=∑m(3,5,6,7)f=∑m(3,5,6,7)

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

Product of Maxterms (Product of Sums or PoS)

Each of the 22 functions of n binary variables can be also expressed as a product of maxterms.

- To express the Boolean function as a product of maxterms, it must first be brought into a form of OR terms.
- 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'.
- This procedure is clarified by the following example:

Question: Express the Boolean function *F = xy + x'z* in a product of maxterm form.

Solution:

First, convert the function into OR terms using the distributive law: F = xy + x' z = (xy + x ')(xy + z)

= (x + x')(y + x')(x + z)(y + z)

= (x' + y)(x + z)(y + z)

The function has three variables: *x, y,* and z. Each OR term is missing one variable; therefore:

x' + y = x' + y + zz' = (x' + y + z)(x' + y + z ')

y + z = x + z + yy' = (x + y + z)(x + y' + z)

z + z = y + z + xx' = (x + y + z)(x' + y + z)

Combing all maxterms and removing repeated terms:

$= M_0M_2M_4M_5$

Shorthand notation:

F(x,y,z)=∏(0,2,4,5)

The product symbol denotes the ANDing of maxterms; the numbers are the maxterms of the function.

Example

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

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

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

f=∏M(0,1,2,4)f=∏M(0,1,2,4)

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

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

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

Conversion between canonical forms

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function.

For example: Consider the function, F (A,B,C) = ∑(1, 4, 5, 6, 7)
Its complement can be expressed as: F(A,B,C) = ∑(0, 2, 3) = m0+m2+m3
Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:
F (A,B,C) = (m0+m2+m3)' = m0'.m2'.m3' =M0.M2.M3=) = ∏(0,2,3)

The last conversion follows from the definition of min terms and maxterms that $m_j' = M_j$

General Procedure: To convert from one canonical form to another, interchange the symbols and and list those numbers missing from the original form. In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2^n (numbered as 0 to 2^{n-1}), where *n* is the number of binary variables in the function.

Consider a function, *F = xy + x'z.* First, we derive the truth table of the function

- The minterms of the function are read from the truth table to be 1, 3, 6, and 7. The function expressed in sum of minterms is

  F(x,y,z) = ∑(1, 3, 6, 7)
- Since there are a total of eight minterms or maxterms in a function of three variables, we determine the missing terms to be 0, 2, 4, and 5. The function expressed in product of maxterm is

  F(x,y,z) = ∏(0, 2, 4, 5)

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Standard Forms

- This is another way to express Boolean functions. In this configuration, the terms that form the function may contain one, two, or any number of literals. There are two types of standard forms: the sum of products and product of sums.
- The sum of products is a Boolean expression containing AND terms, called product terms, of one or more literals each. The sum denotes the ORing of these terms.
- Example: F_1 = y' + xy + x'yz', the expression has three product terms of one, two, and three literals each, respectively. Their sum is in effect an OR operation.
- A product of sums is a Boolean expression containing OR terms, called sum terms. Each term may have any number of literals. The product denotes the ANDing of these terms. An example of a function expressed in product of sums is F_1 = x(y' + z) (x' + y + z' + w), this expression has three sum terms of one, two, and four literals each, respectively. The product is an AND operation.
- Function can also be in non-standard form: F_3 = (AB + CD) (A'B' + CD') is neither in SOP nor in POS forms. It can be changed to a standard form by using the distributive law as F3 = A'B'CD + ABC'D'.

Standard SoP form

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

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

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

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical SoP form. In that case, both canonical and standard SoP forms are same.

Example

Convert the following Boolean function into Standard SoP form.

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

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

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

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

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

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

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

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

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

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

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

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

Standard PoS form

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

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

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

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical PoS form. In that case, both canonical and standard PoS forms are same.

Example

Convert the following Boolean function into Standard PoS form. f=(p+q+r).(p+q+r').(p+q'+r).(p'+q+r)

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

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

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

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

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

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

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

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

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

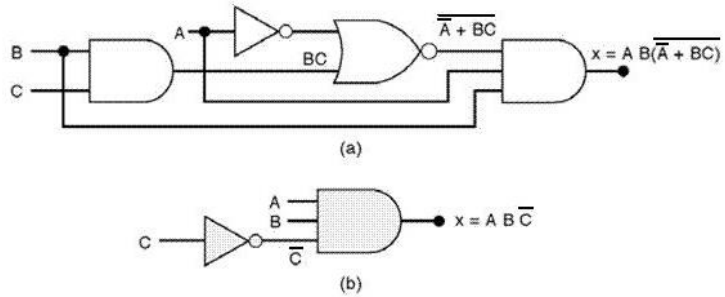
$$\Rightarrow f=(p+q).(q+r).(p+r)$$

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

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

Simplifying Logic Circuits (Boolean functions): Two methods
First obtain one expression for the circuit, then try to simplify.
Example: In diagram below, (a) can be simplified to (b) using one of following two methods:

1. Algebraic method (use Boolean algebra theorems)
2. Karnaugh mapping method (systematic, step-by-step approach)



METHOD 1: Minimization by Boolean algebra

- Make use of relationships and theorems to simplify Boolean Expressions
- Perform algebraic manipulation resulting in a complexity reduction.
- This method relies on your algebraic skill
- things to try:
  - Grouping

A + AB + BC
A (1+ B) + BC
A + BC [since 1+ B =1]

- Multiplication by redundant variables
  - Multiplying by terms of the form A + A' does not alter the logic
  - Such multiplications by a variable missing from a term may enable minimization

Example:

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

- Application of DeMorgan's theorem
  - Expressions containing several inversions stacked one upon the other often are simplified by using DeMorgan's law which **unwraps** multiple inversions.

Example:

$$\begin{aligned} \overline{\overline{A}\overline{B}\overline{C} + \overline{A}\overline{C}\overline{D} + B\overline{C}} &= \overline{(\overline{A} + B + \overline{C}) + (\overline{A} + \overline{C} + \overline{D}) + B\overline{C}} \\ &= \overline{(\overline{A} + B + \overline{C} + \overline{D}) + B\overline{C}} \\ &= \overline{(\overline{A} + B + \overline{C} + \overline{D})} \\ &= A\overline{B}CD \end{aligned}$$

Question (Logic Design): Design a logic circuit having 3 inputs, A, B, C will have its output HIGH only when a majority of the inputs are HIGH.

Solution:

Step 1 Set up the truth table:
Step 2 Write minterm (AND term) for each case where the output is 1.
Step 3 Write the SOP from the output.

$x = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

Step 4 Simplify the output expression

$x = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

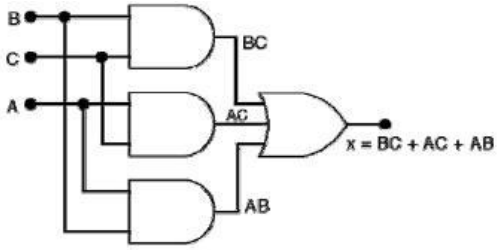
$x = \overline{A}BC + ABC + A\overline{B}C + \boxed{ABC} + AB\overline{C} + \boxed{ABC}$

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

$= BC + AC + AB$

Step 5 Implement the circuit.

A	B	C	x	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\rightarrow \overline{A}BC$
1	0	0	0	
1	0	1	1	$\rightarrow A\overline{B}C$
1	1	0	1	$\rightarrow AB\overline{C}$
1	1	1	1	$\rightarrow ABC$



METHOD 2: Minimization by K-map (Karnaugh map)

Algebraic minimization of Boolean functions is rather awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method provides a simple straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method, first proposed by Veitch and modified by Karnaugh, is also known as the "Veitch diagram" or the "Karnaugh map."

- The k-map is a diagram made up of grid of squares.
- Each square represents one minterm.
- The minterms are ordered according to Gray code (only one variable changes between adjacent squares).
- Squares on edges are considered adjacent to squares on opposite edges.
- Karnaugh maps become clumsier to use with more than 4 variables.

In fact, the map presents a visual diagram of all possible ways a function may be expressed in a standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which he can select the simplest one. We shall assume that the simplest algebraic expression is anyone in a sum of products or product of sums that has a minimum number of literals. (This expression is not necessarily unique)

Two variable maps

There are four minterms for a Boolean function with two variables. Hence, the two-variable map consists of four squares, one for each minterm, as shown in Figure:

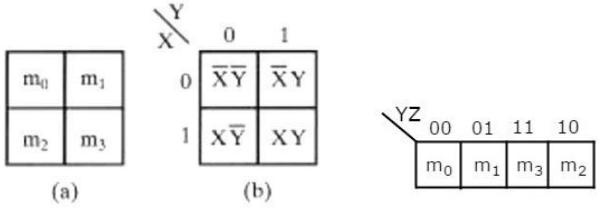


Fig: Two-variable map

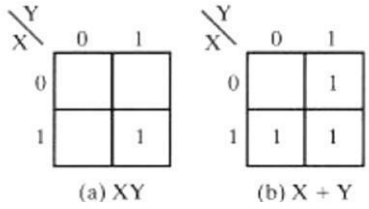


Fig: Representation of functions in the map

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

Three variable maps

There are eight minterms for three binary variables. Therefore, a three-variable map consists of eight squares, as shown in Figure. The map drawn in part (b) is marked with binary numbers for each row and each column to show the binary values of the minterms.

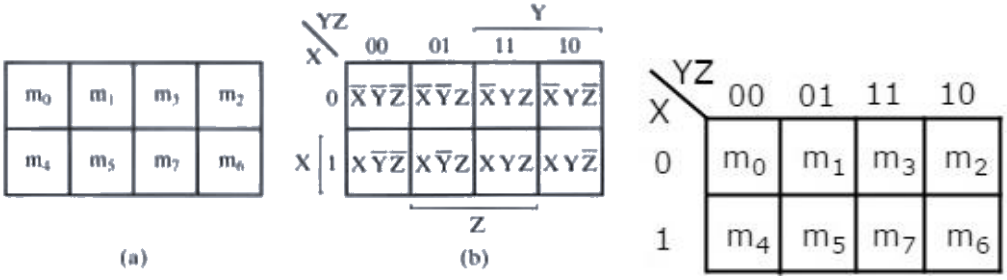


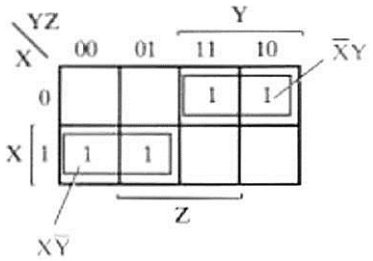
Fig: Three-variable map

There is only one possibility of grouping 8 adjacent min terms.

- The possible combinations of grouping 4 adjacent min terms are {(m0, m1, m3, m2), (m4, m5, m7, m6), (m0, m1, m4, m5), (m1, m3, m5, m7), (m3, m2, m7, m6) and (m2, m0, m6, m4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m1, m3), (m3, m2), (m2, m0), (m4, m5), (m5, m7), (m7, m6), (m6, m4), (m0, m4), (m1, m5), (m3, m7) and (m2, m6)}.
- If x=0, then 3 variable K-map becomes 2 variable K-map.

Question: Simplify the Boolean function
F(X,Y,Z) = ∑(2, 3, 4, 5).
Solution:

Step 1: First, a 1 is marked in each minterm that represents the function. This is shown in Figure, where the squares for minterms 010, 011, 100, and 101 are marked with 1's. For convenience, all of the remaining squares for which the function has value 0 are left blank rather than entering the 0's.



Step 2: Explore collections of squares on the map representing product terms to be considered for the simplified expression. We call such objects ***rectangles***. Rectangles that correspond to product terms are restricted to contain numbers of squares that are powers of 2, such as 1, 2(pair), 4(quad), 8(octet) …..

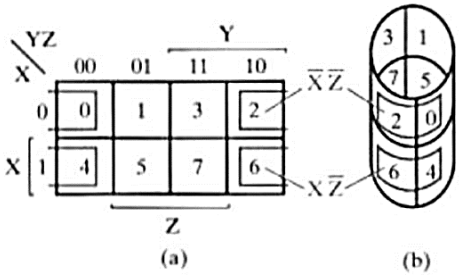
Goal is to find the fewest such rectangles that include all of the minterms marked with 1's. This will give the fewest product terms.

Step 3: Sum up each rectangles (it may be pair, quad etc representing term) eliminating the variable that changes in value (or keeping intact the variables which have same value) throughout the rectangle.

From figure, logical sum of the corresponding two product terms gives the optimized expression for *F:*

F = X'Y+XY'

Point to understand



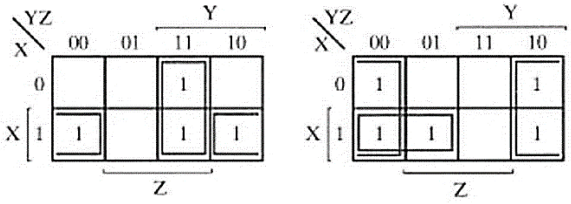
Minterm adjacencies are circular in nature. This figure shows Three-Variable Map in Flat and on a Cylinder to show adjacent squares.

Question: Simplify the following two Boolean functions:

F(X,Y,Z) = ∑(3, 4, 6, 7)

G(X,Y,Z) = ∑ (0, 2, 4, 5, 6)

Solution: The map for F and G are given below:



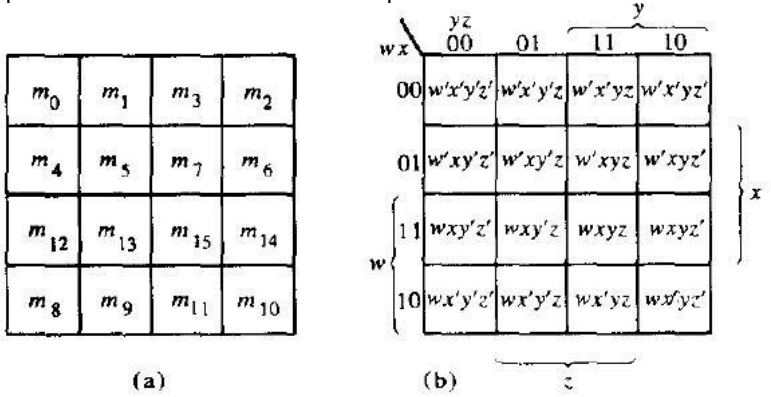
Writing the simplified expression for both functions:

F = YZ +XZ' and G = Z'+XY'

NOTE On occasion, there are alternative ways of combining squares to produce equally optimized expressions. It's upon your skill to use the easy and efficient strategy.

Four variable maps

The map for Boolean functions of four binary variables is shown in Fig below. In (a) are listed the 16 minterms and the squares assigned to each. In (b) the map is redrawn to show the relationship with the four variables.



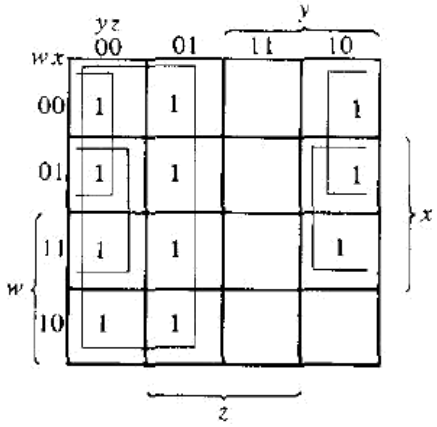
WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R1, R2, R3 and R4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C1, C2, C3 and C4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R1, R2), (R2, R3), (R3, R4), (R4, R1), (C1, C2), (C2, C3), (C3, C4), (C4, C1)}.
- If w=0, then 4 variable K-map becomes 3 variable K-map.

The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example, m_0 and m_2 form adjacent squares, as do m_3 and m_{11}.

Question: Simplify the Boolean function
$F(w,x,y,z) = \sum(0,1,2,4,5,6,8,9,12,13,14)$

Solution:
Since the function has four variables, a four-variable map must be used. Map representation is shown below:



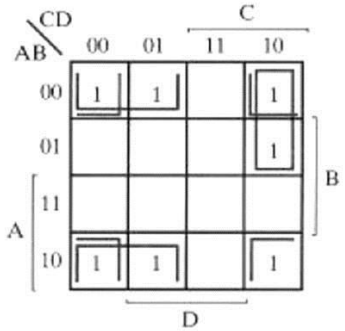
The simplified function is: F = y' + w'z' + xz'

Question: Simplify the Boolean function
$F = A'B'C' + B'CD' + A'BCD' + AB'C'$

Solution:
First try just to reduce the standard form function into SOP form and then mark 1 for each minterm in the map.

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

This function also has 4 variables, so the area in the map covered by this function consists of the squares marked with 1's in following Fig.



Optimized function thus is: $F = B'D' + B'C' + A'CD'$

Minimization of Boolean Functions using K-Maps

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

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

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

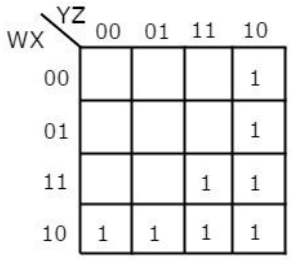
- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one product term. It is known as prime implicant. The prime implicant is said to be essential prime implicant, if atleast single '1' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

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

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

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

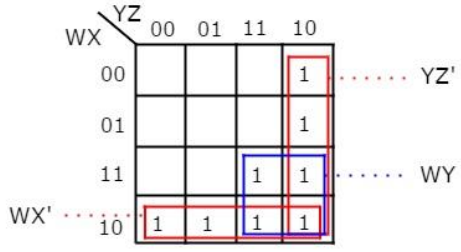
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require 4 variable K-map. The 4 variable K-map with ones corresponding to the given product terms is shown in the following figure.



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

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

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



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

- Two ones (m8 & m9) of fourth row grouping are not covered by any other groupings. Only fourth row grouping covers those two ones.
- Single one (m15) of square shape grouping is not covered by any other groupings. Only the square shape grouping covers that one.
- Two ones (m2 & m6) of fourth column grouping are not covered by any other groupings. Only fourth column grouping covers those two ones.

Therefore, the simplified Boolean function is

f= WX' + WY + YZ'

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

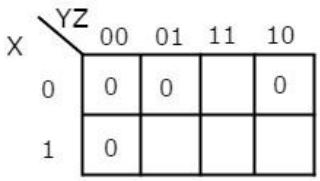
- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as product of Max terms form, then place the zeroes at respective Max term cells in the K-map. If the Boolean function is given as product of sums form, then place the zeroes in all possible cells of K-map for which the given sum terms are valid.
- Check for the possibilities of grouping maximum number of adjacent zeroes. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one sum term. It is known as prime implicant. The prime implicant is said to be essential prime implicant, if atleast single '0' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

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

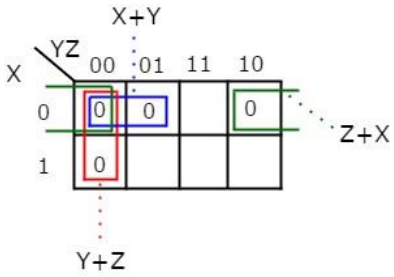
Example

Let us simplify the following Boolean function, f(X,Y,Z)=∏M(0,1,2,4)f(X,Y,Z)=∏M(0,1,2,4) using K-map.

The given Boolean function is in product of Max terms form. It is having 3 variables X, Y & Z. So, we require 3 variable K-map. The given Max terms are M0, M1, M2 & M4. The 3 variable K-map with zeroes corresponding to the given Max terms is shown in the following figure.



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



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

Therefore, the simplified Boolean function is

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

In this way, we can easily simplify the Boolean functions up to 5 variables using K-map method. For more than 5 variables, it is difficult to simplify the functions using K-Maps. Because, the number of cells in K-map gets doubled by including a new variable. Due to this checking and grouping of adjacent ones (min terms) or adjacent zeros (Max terms) will be complicated.

Don't care Conditions

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the min terms. This assumes that all the combinations of the values for the variables of the function are valid. In practice, there are some applications where the function is not specified for certain combinations of the variables.

Example: four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered as unspecified.

In most applications, we simply **don't care what value is assumed by the function** for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function **don't-care conditions**. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

Don't-care minterm is a combination of variables whose logical value is not specified. 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 min term.

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.

Question: Simplify the Boolean function

$$F(w,x,y,z) = \sum(1, 3, 7, 11, 15)$$

that has the don't-care conditions

$$d(w,x,y,z) = \sum(0, 2, 5)$$

Solution:
The map simplification is shown below. The minterms of *F* are marked by 1's, those of *d* are marked by X's, and the remaining squares are filled with 0's.

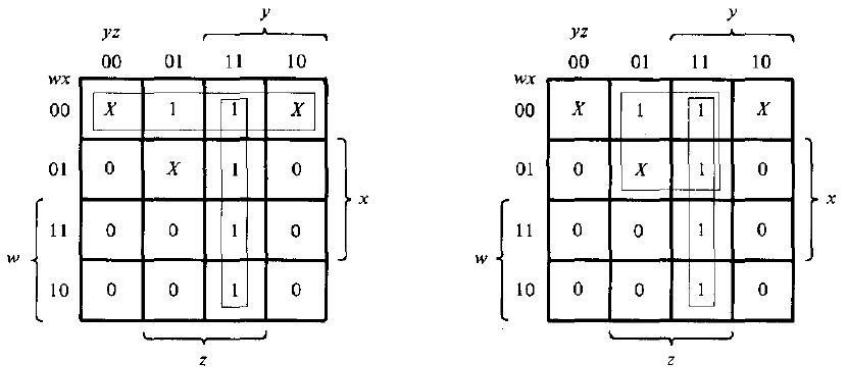


Fig: Map simplification with don't care conditions
In part (a) of the diagram, don't-care minterms 0 and 2 are included with the 1's, which results in the simplified function
F = yz + w'x'
In part (b), don't-care minterm 5 is included with the 1's and the simplified function now is
F = yz + w'z
Either one of the above expressions satisfies the conditions stated for this example.

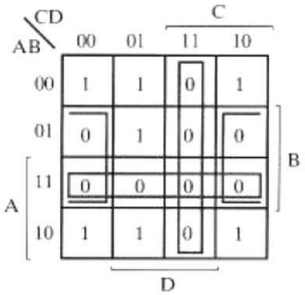
Product of sum simplification
The optimized Boolean functions derived from the maps in all of the previous examples were expressed in sum-of-products (SOP) form. With only minor modification, the product-of-sums form can be obtained.mplification of

Procedure:
The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the function belong to the complement of the function. From this, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares with 0's and combine them into valid rectangles, we obtain an optimized expression of the complement of the function (F'). We then take the complement of F to obtain the function *F* as a product of sums.
Question: Simplify the following Boolean function F(A,B,C,D) = $\sum(0, 1, 2, 5, 8, 9, 10)$ in

(a) Sum of products (SOP) and
(b) Product of sums (POS).

Solution:
The 1's marked in the map below represent all the minterms of the function. The squares marked with 0's represent the minterms not included in *F* and, therefore, denote *F'*.

a. Combining the squares with 1's gives the simplified function in sum of products:
F = B'D' + B'C' + A'C'D

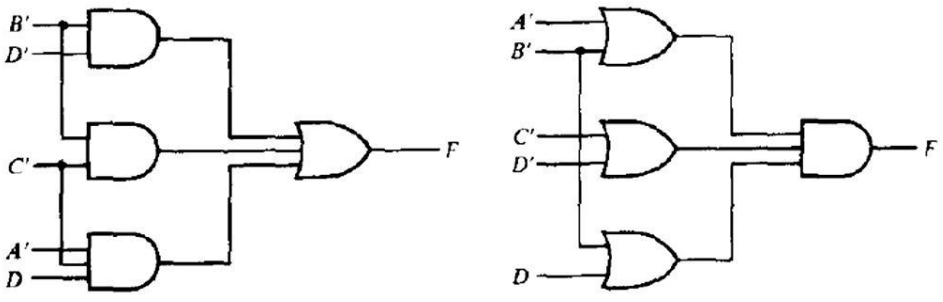


b. If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

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

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in unit2), we obtain the simplified function in product of sums:
F = (A' + B')(C' + D')(B' + D)

The Gate implementation of the simplified expressions obtained above in (a) and (b):



(a) $F = B'D' + B'C' + A'C'D$ (b) $F = (A' - B')(C' + D')(B' + D)$

NAND and NOR implementation

Digital circuits are more frequently constructed with NAND or NOR gates than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. The procedure for **two-level implementation** is presented in this section.

NAND and NOR conversions (from AND, OR and NOT implemented Boolean functions)

Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams. To facilitate the conversion to NAND and NOR logic, there are two other graphic symbols for these gates.

(a) NAND gate

Two equivalent symbols for the NAND gate are shown in diagram below:

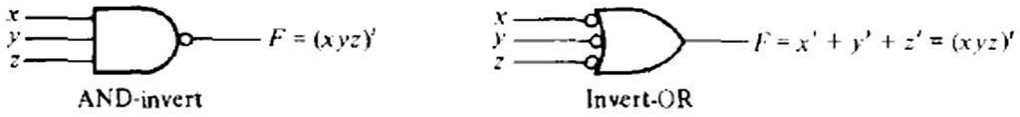


Fig: Two graphic symbols for NAND gate

(b) NOR gate

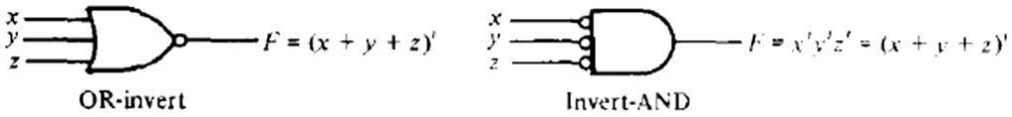


Fig: Two graphic symbols for NOR gate

(c) Inverter

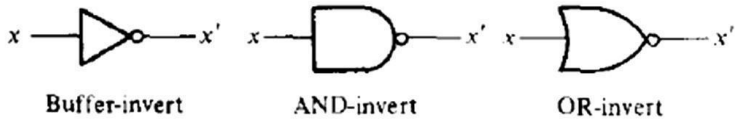
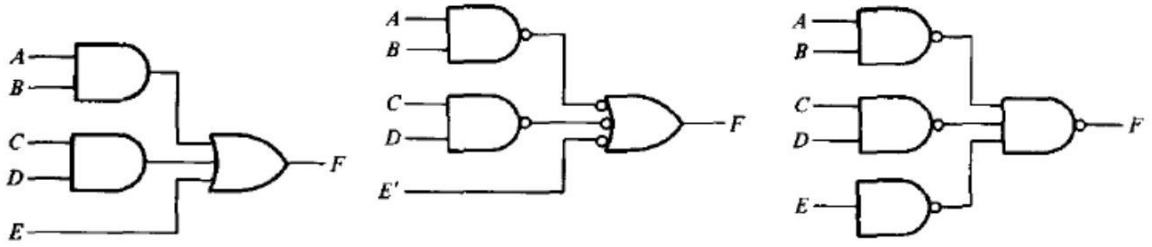


Fig: Three graphic symbols for NOT gate

NAND implementation

The implementation of a Boolean function with NAND gates requires that the function be simplified in the sum of products form. To see the relationship between a sum of products expression and its equivalent NAND implementation, consider the logic diagrams of Fig below. All three diagrams are equivalent and implement the function: $F=AB + CD + E$



(a) AND-OR (b) NAND-NAND (c) NAND-NAND

The rule for obtaining the NAND logic diagram from a Boolean function is as follows:

First method:

- **(a)** Simplify the function and express it in **sum of products**.
- **(b)** Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of the term. This constitutes a group of **first-level gates.**
- **(c)** Draw a single NAND gate (using the AND-invert or invert-OR graphic symbol) in the second level, with inputs coming from outputs of first-level gates.
- **(d)** A term with a single literal requires an inverter in the first level or may be complemented and applied as an input to the **second-level NAND gate**.

Second method:
If we combine the 0's in a map, we obtain the simplified expression of the *complement* of the function in sum of products. The complement of the function can then be implemented with two levels of NAND gates using the rules stated above. If the normal output is desired, it would be necessary to **insert a one-input NAND or inverter gate**. There are occasions where the designer may want to generate the complement of the function; so this second method may be preferable.

Question: Implement the following function with NAND gates:
F(x,y,z) = ∑(0, 6)
Solution:
The first step is to simplify the function in sum of products form. This is attempted with the map. There are only two 1's in the map, and they can't be combined.

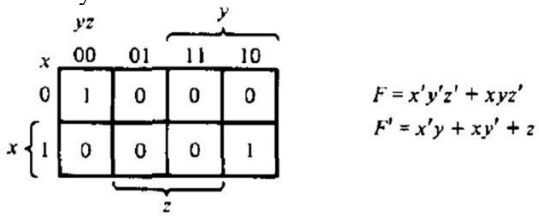


Fig: Map simplification in SOP

METHOD1:
Two-level NAND implementation is shown below:

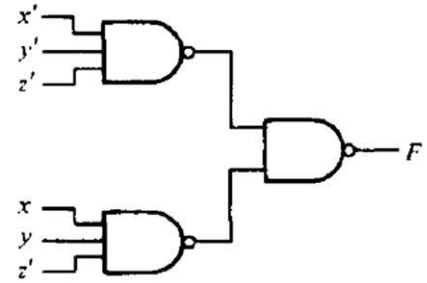


Fig: *F = x'y'z' +xyz'*

METHOD2:
Next we try to simplify the complement of the function in sum of products. This is done by combining the 0's in the map:
F' = x'y + xy' + z
The two-level NAND gate for generating *F'* is shown below:

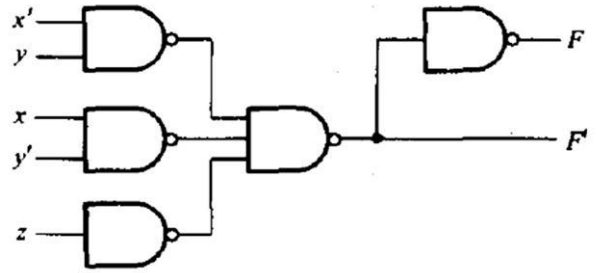
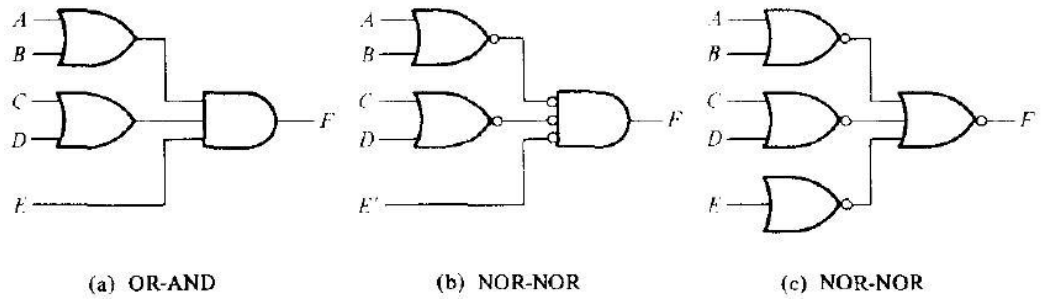


Fig: *F' = x'y + xy'* + z
If output *F* is required, it is necessary to add a one· input NAND gate to invert the function. This gives a three-level implementation.

NOR Implementation
The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The implementation of a Boolean function with NOR gates requires that the function be simplified in product of sums form. A product of sums expression specifies a group of OR gates for the sum terms, followed by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is depicted in Fig below. It is **similar to the NAND transformation discussed previously, except that now we use** the product of sums expression.



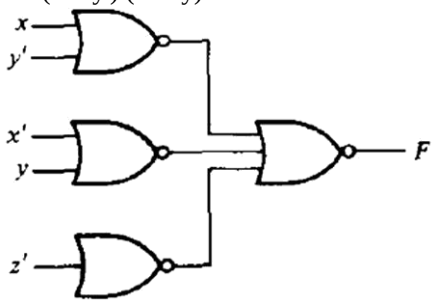
(a) OR-AND (b) NOR-NOR (c) NOR-NOR

Fig: Three ways to implement $F = (A + B)(C + D)E$
All the rules for NOR implementation are similar to NAND except that these are duals, so I won't describe them here.

Question: Implement the following function with NOR gates:
F(x,y,z) = $\sum(0, 6)$
Solution:
Map is drawn in previous question.

METHOD1
First, combine the 0's in the map to obtain
$F'=x'y+xy'+z$ this is the complement of the function in sum of products. Complement F' to obtain the simplified function in product of sums as required for NOR implementation:

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

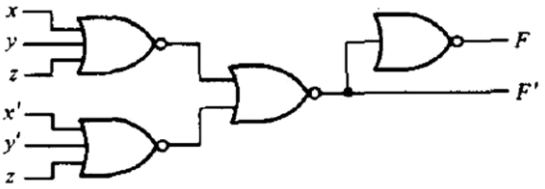


METHOD2
A second implementation is possible from the complement of the function in product of sums. For this case, first combine the 1's in the map to obtain

$F = x'y'z' + xyz'$

Complement this function to obtain the complement of the function in product of sums as required for NOR implementation:

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



Summary of NAND and NOR implementation

Case	Function to simplify	Standard form to use	How to derive	Implement with	Number of levels to F
(a)	F	Sum of products	Combine 1's in map	NAND	2
(b)	F'	Sum of products	Combine 0's in map	NAND	3
(c)	F	Product of sums	Complement F' in (b)	NOR	2
(d)	F'	Product of sums	Complement F in (a)	NOR	3

Unit 4
Combinational Logic

Introduction

In digital circuit theory, **combinational logic** is a *type of digital logic which is implemented by Boolean circuits, where the output is a pure function of the present input only*. This is in contrast to sequential logic, in which the output depends not only on the present input but also on the history of the input. In other words, sequential logic has *memory* while combinational logic does not.
Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −

- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

Combinational Circuit

- These are the circuit gates employing combinational logic.
- A combinational circuit consists of *n* input variables, logic gates, and *m* output variables. The logic gates accept signals from the inputs and generate signals to the outputs.
- For *n* input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by *m* Boolean functions, one for each output variable. Each output function is expressed in terms of the *n* input variables.

Obviously, both input and output data are represented by binary signals, i.e., logic-1 and the other logic- 0. The n input binary variables come from an external source; the m output variables go to an external destination. A block diagram of a combinational circuit is shown in Fig:

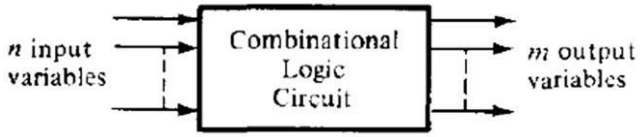


Fig: Block diagram of combinational circuit

Design procedure

The design of combinational circuits starts from the verbal outline of the problem and ends in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be easily obtained. The procedure involves the following steps:

1. **Specification**
   - Write a specification for the circuit if one is not already available
2. **Formulation**
   - Derive a truth table or initial Boolean equations that define the required relationships between the inputs and outputs, if not in the specification.
   - Apply hierarchical design if appropriate
3. **Optimization**
   - Apply 2-level and multiple-level optimization
   - Draw a logic diagram for the resulting circuit using ANDs, ORs, and inverters
4. **Technology Mapping**
   - Map the logic diagram to the implementation technology selected
5. **Verification**
   - Verify the correctness of the final design manually or using simulation

In simple words, we can list out the design procedure of combinational circuits as:

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

Digital computers perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations. The most basic arithmetic operation, no doubt, is the addition of two binary digits.

Half-Adder

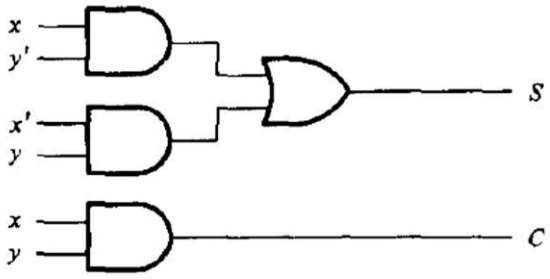
- A combinational circuit that performs the addition of two bits is called a *half-adder.*
- Circuit needs **two inputs** and **two outputs**. The input variables designate the augend (x) and addend (y) bits; the output variables produce the sum (S) and carry (C).
- Now we **formulate a Truth table** to exactly identify the function of half-adder.

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum of products expressions are:

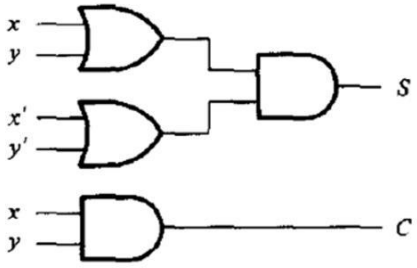
$$S = x'y + xy'$$
$$C = xy$$

- Implementation**:**

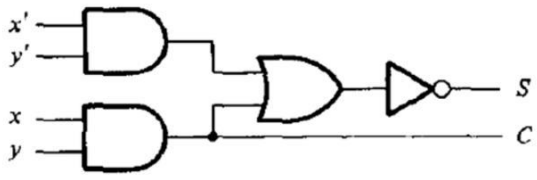


(a) $S = xy' + x'y$
$C = xy$

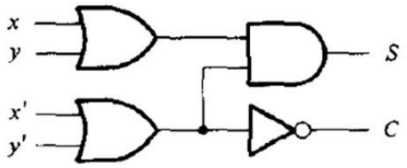
- Other realizations and implementations of Half-adders are:



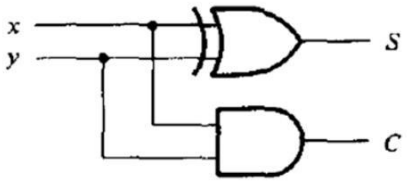
(b) $S = (x + y)(x' + y')$
$C = xy$



(c) $S = (C + x'y')'$
$C = xy$



(d) $S = (x + y)(x' + y')$
$C = (x' + y')'$



(e) $S = x \oplus y$
$C = xy$

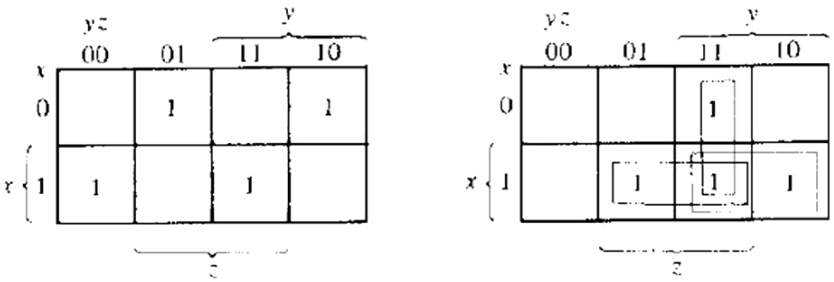
Full-Adder

- A *full-adder* is a combinational circuit that forms the arithmetic sum of three input bits.
- It consists of **three inputs** and **two outputs**. Two of the input variables, denoted by *x* and *y*, represent the **two significant bits** to be added. The third input, z, represents the **carry** from the previous lower significant position.
- **Truth table formulation:**

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The input-output logical relationship of the full-adder circuit may be expressed in two Boolean functions, one for each output variable. Each output Boolean function requires a unique map for its simplification (maps are not necessary; you guys can use algebraic method for simplification). Simplified expression in sum of products can be obtained as:



S = x'y'z + x'yz' + xy'z' + xyz C = xy + xz + yz

- **Implementation:**

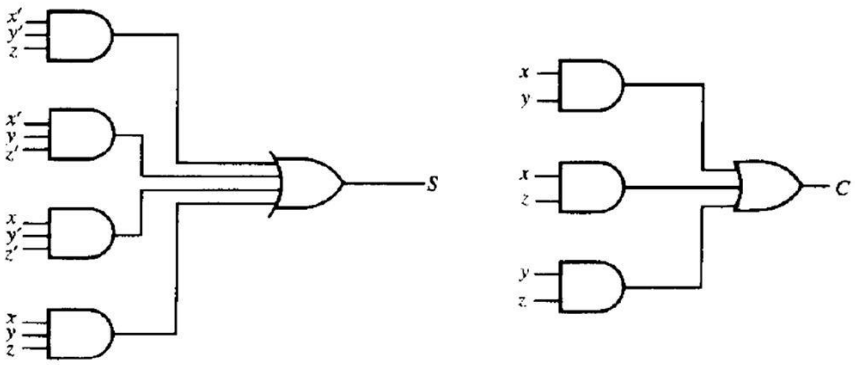


Fig: Implementation of a full-adder in sum of products.

- A full-adder can be implemented with **two half-adders** and one OR gate.

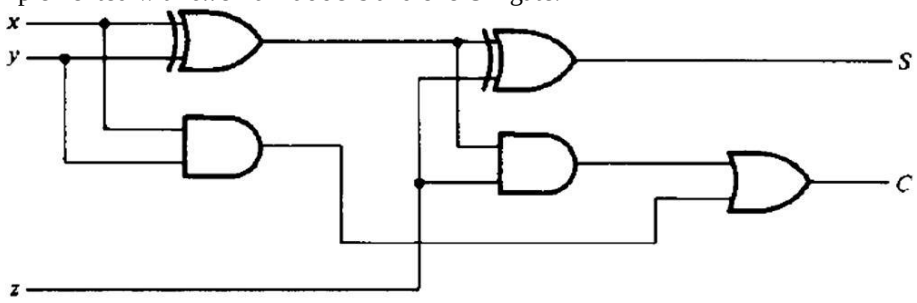


Fig: Implementation of a full-adder with two half-adders and an OR gate

Here, The S output from the second half-adder is the exclusive-OR of z and the output of the first half-adder, giving:

$S = z \oplus (x \oplus y)$
$= z'(xy' + x'y) + z(xy' + x'y)'$
$= z'(xy' + x'y) + z(xy + x'y')$
$= xy'z' + x'yz' + xyz + x'y'z$

$C = z(x \oplus y) + xy$
$= z(xy' + x'y) + xy$
$= xy'z + x'yz + xy$

Subtractors

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes anaddition operation requiring full-adders for its machine implementation. It is possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding **significant minuend bit** to form a **difference bit**. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. Just as there are half- and full-adders, there are half- and full-subtractors.

Half-Subtractor

- A half-subtractor is a combinational circuit that subtracts two bits and produces their difference bit.
- Denoting minuend bit by x and the subtrahend bit by *y*. To perform *x* - *y*, we have to check the relative magnitudes of *x* and *y*:
  - If $x \geq y$, we have three possibilities: 0 - 0 = 0, 1 - 0 = 1, and 1 - 1 = 0.
  - If $x < y$, we have 0 - 1, and it is necessary to borrow a 1 from the next higher stage.
- The half-subtractor needs **two outputs**, difference (D) and borrow (B).
- The **truth table** for the input-output relationships of a half-subtractor can now be derived as follows:

x	y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

The output borrow *B* is a 0 as long as $x \geq y$. It is a 1 for $x = 0$ and $y = 1$. The *D* output is the result of the arithmetic operation $2B + x - y$.
The Boolean functions for the two outputs of the half-subtractor are derived directly from the truth table:

D = x'y + xy'
B = x'y

Implementation for Half-subtractor is similar to Half-adder except the fact that x input of B is inverted. (Here, D is analogous to S and B is similar to C of half-adder circuit). Try it out, I don't like redundancy…

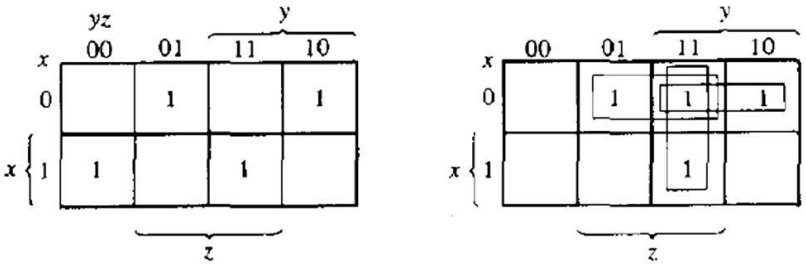
Full-Subtractor

- A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 **may have been borrowed** by a lower significant stage.
- This circuit has **three inputs** and **two outputs**. The three inputs, *x*, *y*, and z, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, *D* and *B*, represent the difference and output-borrow, respectively.
- **Truth-table and output-function formulation**:

x	y	z	B	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

  - The 1's and 0's for the output variables are determined from the subtraction of x - y - z.
  - The combinations having input borrow z = 0 reduce to the same four conditions of the half-adder.
  - For x = 0, y = 0, and z = 1, we have to borrow a 1 from the next stage, which makes B = 1 and adds 2 to x. Since 2 - 0 -1= 1, D = 1.
  - For x = 0 and yz = 11, we need to borrow again, making B =1 and x = 2. Since 2 - 1 - 1 = 0, D = 0.
  - For x = 1 and yz = 01, we have x - y - z = 0, which makes B =0 and D = 0.
  - Finally, for x = 1, y = 1, z =1I, we have to borrow 1, making B = 1 and x = 3, and 3 - 1 - 1 = 1, making D=1.

The simplified Boolean functions for the two outputs of the full-subtractor are derived in the maps:



D = x'y'z + x'yz' + xy'z' + xyz B = x'y + x'z + yz

- **Circuit implementations** are same as Full-adder except B output (analogous to C) is little different. (Don't worry! ladies and gentlemen, we will discuss it in class…)

Code Conversion

- The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a **code converter** is a circuit that makes the two systems compatible even though each uses a different binary code.
- To convert from binary code A to binary code B, code converter has **input lines** supplying the bit combination of elements as specified by code A and the output lines of the converter generating the corresponding bit combination of code B. A Code converter (combinational circuit) performs this transformation by means of logic gates.
- The design procedure of code converters will be illustrated by means of a *specific example* of conversion from the BCD to the excess-3 code. I will describe 5-step design procedure of this code converter so that you guys will be able to understand how practical combinational circuits are designed.

Design example: <u>BCD to Excess-3 code converter</u>

1. Specification

- Transforms BCD code for the decimal digits to Excess-3 code for the decimal digits
- BCD code words for digits 0 through 9: 4-bit patterns 0000 to 1001, respectively.
- Excess-3 code words for digits 0 through 9: 4-bit patterns consisting of 3 (binary 0011) added to each BCD code word
- Implementation:
  - multiple-level circuit

2. Formulation

- Conversion of 4-bit codes can be most easily formulated by a truth table
- Variables- BCD: A, B, C, D
- Variables- Excess-3: W, X, Y, Z
- Don't Cares: BCD 1010 to 1111

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

Table: Truth table for code converter example

Note that the four BCD input variables may have 16 bit combinations, but only 10 are listed in the truth table. Others designate "don't care conditions".

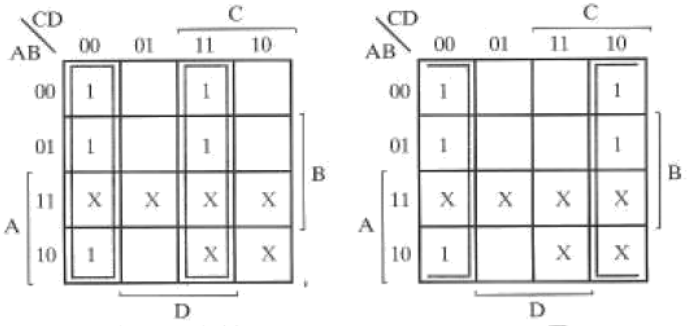
3. Optimization

a. 2-level optimization

The k-maps are plotted to obtain simplified sum-of-products Boolean expressions for the outputs. Each of the four maps represents one of the outputs of the circuit as a function of the four inputs.

b. Multiple-level optimization

This second optimization step reduces the number of gate inputs (and hence the no. gates). The following manipulation illustrates optimization with multiple-output circuits implemented with three levels of gates:



$T_1 = C + D$

$W = A + BC + BD = A + BT_1$

$X = \bar{B}C + \bar{B}D + B\bar{C}\bar{D} = \bar{B}T_1 + B\bar{C}\bar{D}$

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

$Z = \bar{D}$

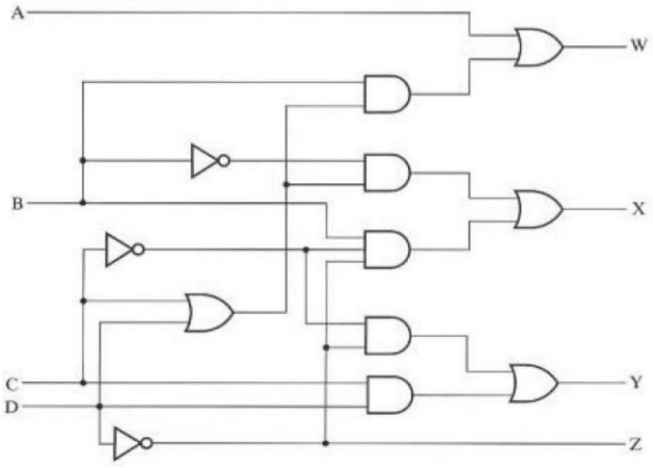


Fig: Logic Diagram of BCD- to-Excess-3 Code Converter

4. Technology mapping

This is concerned with the act of mapping of basic circuit (using AND, OR and NOT gates) to a specific circuit technology (such as NAND, NOR gate tech.)

NOTE: This is advanced topic, and I won't discuss here. For exam point of view, if you are asked for BCD-to-Excess-3 code converter, you will finish up your answer by drawing basic circuit shown above.

5. Verification

Here we need to test our designed circuit, whether it works correctly.

Analysis Procedure

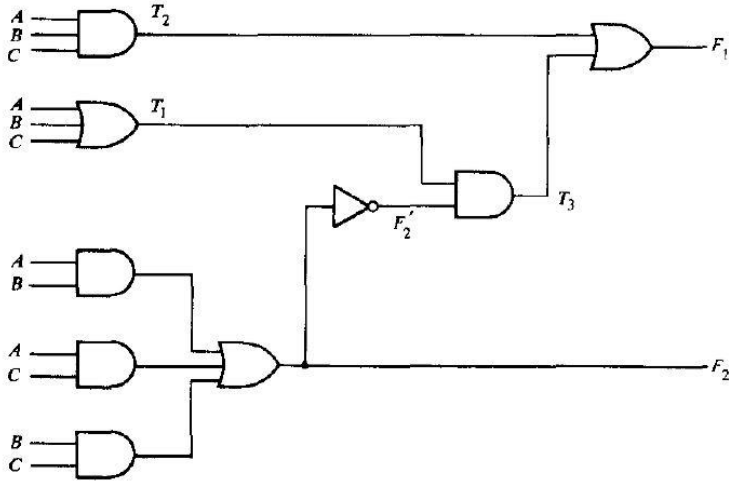
The design of a combinational circuit starts from the verbal specifications of a required function and ends with a set of output Boolean functions or a logic diagram. The **analysis of a combinational circuit** is somewhat the reverse process. It starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or a verbal explanation of the circuit operation.

Obtaining Boolean functions from logic diagram

Steps in analysis:

1. The first step in the analysis is to make sure that the given circuit is combinational and not sequential.
2. Assign symbols to all gate outputs that are a function of the input variables. Obtain the Boolean functions for each gate.
3. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates.
4. Repeat step 3 until the outputs of the circuit are obtained.
5. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables only.

Analysis of the combinational circuit below illustrates the proposed procedure:



We note that the circuit has three binary inputs, *A, B,* and C, and two binary outputs, F_1 and F_2. The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function of input variables only are F_2, T_1 *and* T_2. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$
$$T_1 = A+B+C$$
$$T_2 = ABC$$

Next we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2'T_1$$
$$F_1 = T_3 + T_2$$

The output Boolean function F_2 just expressed is already given as a function of the in puts only. To obtain F_1 as a function of *A, B,* and C, form a series of substitutions as follows:

$$F_1 = T_3 + T_2 = F_2'T_1 + ABC = (AB + AC + BC)'(A + B + C) + ABC$$
$$= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC$$
$$= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC$$
$$= A'BC' + A'B'C + AB'C' + ABC$$

If you want to determine the information-transformation task achieved by this circuit, you can derive the truth table directly from the Boolean functions and try to recognize a familiar operation. For this example, we note that the circuit is a **full-adder**, with *F,* being the sum output and *F,* the carry output. *A, B,* and C are the three inputs added arithmetically.

Obtaining truth-table from logic diagram

The derivation of the truth table for the circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, proceed as follows:

Steps in analysis:

1. Determine the number of input variables to the circuit. For *n* inputs, form the 2^n possible input combinations of 1's and 0's by listing the binary numbers from 0 to $2^n - 1$.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

This process can be illustrated using the circuit above:

We form the eight possible combinations for the three input variables. The truth table for F_2 is determined directly from the values of A, B, and C, with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F_2' is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T_3 are derived

from T_1 and F_2'. T_3 is equal to 1 when both T_1 and F_2' are equal to 1, and to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1.

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Inspection of the truth-table combinations for *A, B,* C, F_1 and F_2 of table above shows that it is identical to the truth-table of the full-adder.

NOTE: When a circuit with don't-care combinations is being analyzed, the situation is entirely different.
We assume here that the don't-care input combinations will never occur.

NAND, NOR and Ex-OR circuits

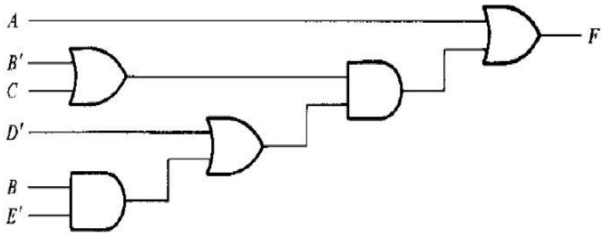
In unit 3, SOP and POS form of Boolean functions are studied. Also we got to know, Such Boolean functions can be implemented with 2-level circuits using universal gates (look at NAND and NOR implementation of Boolean function, unit 3). Here we will look at the multiple level circuits employing universal gates i.e we will treat the functions which are in standard form.

Multi-level NAND circuits

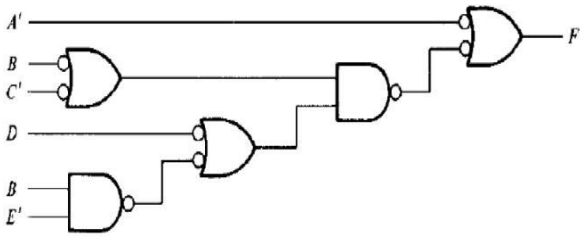
To implement a Boolean function with NAND gates we need to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit-manipulation techniques that change AND-OR diagrams to NAND diagrams.
To obtain a multilevel NAND diagram from a Boolean expression, proceed as follows:

1. From the given Boolean expression, draw the logic diagram with AND, OR, and inverter gates. Assume that both the normal and complement inputs are available.
2. Convert all AND gates to NAND gates with AND-invert graphic symbols.
3. Convert all OR gates to NAND gates with invert-OR graphic symbols.
4. Check all small circles in the diagram. For every small circle that is not compensated by another small circle along the same line, insert an inverter (one-input NAND gate) or complement the input variable.

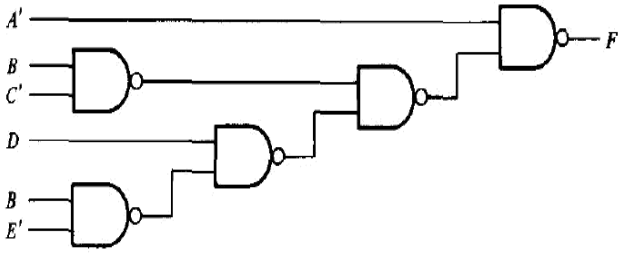
Example: F = A + (B' + C) (D' + BE ')



AND-OR diagram



NAND diagram using two graphic symbols



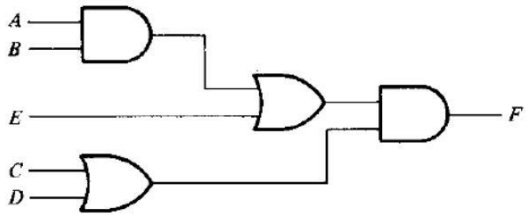
NAND diagram using one graphic symbol

Multi-level NOR circuits

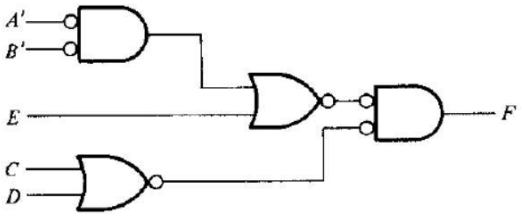
The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic form a dual of the corresponding procedures and rules developed for NAND logic. Similar to NAND, NOR has also two graphic symbols: OR-invert and invert-AND symbol.
The procedure for implementing a Boolean function with NOR gates is similar to the procedure outlined in the previous section for NAND gates:

1. Draw the AND-OR logic diagram from the given algebraic expression. Assume that both the normal and complement inputs are available.
2. Convert all OR gates to NOR gates with OR-invert graphic symbols.
3. Convert all AND gates to NOR gates with invert-AND graphic symbols.
4. Any small circle that is not compensated by another small circle along the same line needs an inverter or the complementation of the input variable.

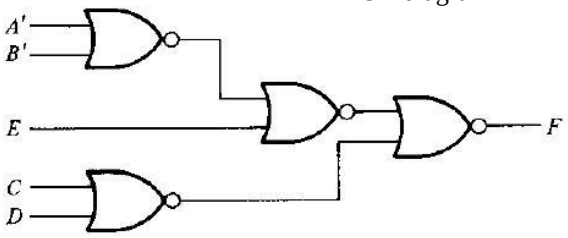
Example: F = (AB + E) (C + D)



AND-OR diagram



NOR diagram



Alternate NOR diagram

Ex-OR function

The exclusive-OR (XOR) denoted by the symbol ⊕, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

It is equal to 1 if only *x* is equal to 1 or if only y is equal to 1 but not when both are equal.

Realization of XOR using Basic gates and universal gates

A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate and next firgure shows the implementation of the exclusive-OR with four NAND gates.

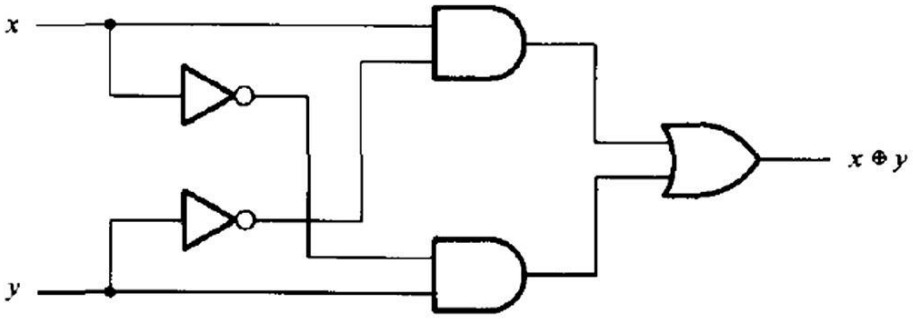


Fig: Implementation XOR with AND-OR-NOT gates

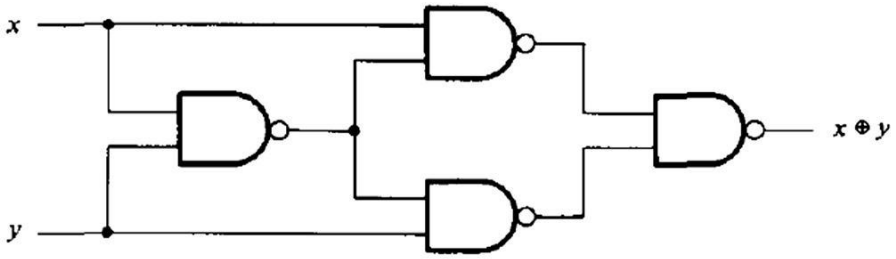


Fig: Realization of XOR with NAND gates

In second diagram, first NAND gate performs the operation (xy)' = (x' + y'). The other two-level NAND circuit produces the sum of products of its inputs:

(x' + y')x + (x' + y')y = xy' + x'y = x ⊕ y

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error-detection and correction circuits.

Parity generator and Checker

Exclusive-OR functions are very useful in systems requiring error-detection and correction codes. As discussed before, a **parity bit** is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.

- The circuit that generates the parity bit in the transmitter is called a ***parity generator***.
- The circuit that checks the parity in the receiver is called a ***parity checker***.

Example: Consider a 3-bit message to be transmitted together with an even parity bit.

The three bits, *x, y,* and z, constitute the message and are the inputs to the circuit. The parity bit *P* is the output. For even parity, the bit *P* must be generated to make the total number of 1's even (including *P).*

Three-Bit Message			Parity Bit
x	y	z	P
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

Table: Even parity generator truth table

From the truth table, we see that *P* constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, *P* can be expressed as a three-variable exclusive-OR function: $P = x \oplus y \oplus z$.

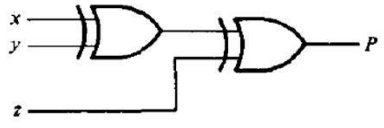
The three bits in the message together with the parity bit are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission.

Four Bits Received				Parity Error Chec
x	y	z	P	C
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

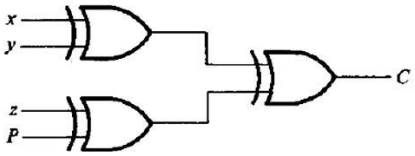
Table: Even parity checker truth table

- Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission.
- The output of the parity checker, denoted by C, will be equal to 1 if an error occurs, that is, if the four bits received have an odd number of 1's.
- The parity checker can be implemented with exclusive-OR gates: $C = x \oplus y \oplus z \oplus P$.

Logic diagrams for parity generator and Parity checker are shown below:



(a) 3-bit even parity generator



(b) 4-bit even parity checker