UNIT – II

Combinational Logic Circuits

- **Boolean Laws and Theorems**
- **Sum of Product Method**
- **Karnaugh Map**
- **Pair, Quad and Octet**
- **Don’t Care Condition**
- **Product of Sum Method**
- **Product of Sum Simplification**

Data Processing Circuit

- **Multiplexer**
- **De-Multiplexer**
- **1-of-16 Decoder**
- **BCD to Decimal Decoder**
- **Seven Segment Decoders**
- **Encoders**
- **Exclusive OR Gates**
- **Parity Generator and checker**

COMBINATIONAL LOGIC CIRCUIT

Boolean Laws and Theorems

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

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

Postulates and Basic Laws of Boolean Algebra

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

Boolean Postulates

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

x + 0 = x

x + 1 = 1

x + x = x

x + x’ = 1

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

x.1 = x

x.0 = 0

x.x = x

x.x’ = 0

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

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

Basic Laws of Boolean Algebra

Following are the three basic laws of Boolean Algebra.

- Commutative law
- Associative law
- Distributive law

Commutative Law

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

x + y = y + x

x.y = y.x

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

Associative Law

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

$$\mathbf{x + (y + z) = (x + y) + z}$$

$$\mathbf{x.(y.z) = (x.y).z}$$

Associative law obeys for logical OR & logical AND operations.

Distributive Law

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

$$\mathbf{x.(y + z) = x.y + x.z}$$

$$\mathbf{x + (y.z) = (x + y).(x + z)}$$

Distributive law obeys for logical OR and logical AND operations.

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

Theorems of Boolean Algebra

The following two theorems are used in Boolean algebra.

- Duality theorem
- DeMorgan’s theorem

Duality Theorem

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

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

Group1	Group2
x + 0 = x	x.1 = x
x + 1 = 1	x.0 = 0
x + x = x	x.x = x
x + x’ = 1	x.x’ = 0

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

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

DeMorgan's Theorem

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

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

$$\mathbf{(x + y)' = x'.y'}$$

The dual of the above Boolean function is

$$\mathbf{(x.y)' = x' + y'}$$

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

Simplification of Boolean Functions

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

Eg

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

We can simplify this function in two methods.

Method 1

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

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

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

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

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

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

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

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

Method 2

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

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

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

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

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

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

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

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

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

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

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

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

Eg

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

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

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

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

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

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

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

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

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

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

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

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

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

Sum of product Method(SOP)

A canonical sum of products is a boolean expression that entirely consists of minterms. The Boolean function F is defined on two variables X and Y. The X and Y are the inputs of the boolean function F whose output is true when any one of the inputs is set to true. The truth table for Boolean expression F is as follows:

Inputs		Output
X	**Y**	**F**
0	0	0
0	1	1
1	0	1
1	1	1

we can form the minterm from the variable's value. Now, a column will be added for the minterm in the above table. The complement of the variables is taken whose value is 0, and the variables whose value is 1 will remain the same.

Inputs		Output	Minterm
X	**Y**	**F**	**M**
0	0	0	X'Y'
0	1	1	X'Y
1	0	1	XY'
1	1	1	XY

Now, we will add all the minterms for which the output is true to find the desired canonical SOP(Sum of Product) expression.

F=X' Y+XY'+XY

Converting Sum of Products (SOP) to shorthand notation

The process of converting SOP form to shorthand notation is the same as the process of finding shorthand notation for minterms. There are the following steps to find the shorthand notation of the given SOP expression.

- Write the given SOP expression.
- Find the shorthand notation of all the minterms.
- Replace the minterms with their shorthand notations in the given expression.

Example: F = X'Y+XY'+XY

1. Firstly, we write the SOP expression:

F = X'Y+XY'+XY

2. Now, we find the shorthand notations of the minterms X'Y, XY', and XY.

X'Y = $(01)_2$ = m_1

XY' = $(10)_2$ = m_2

XY = $(11)_2$ = m_3

3. In the end, we replace all the minterms with their shorthand notations:

F=m1+m2+m3

Converting shorthand notation to SOP expression

The process of converting shorthand notation to SOP is the reverse process of converting SOP expression to shorthand notation. Let's see an example to understand this conversion.

Example:

Let us assume that we have a boolean function F, which defined on two variables X and Y. The minterms for the function F are expressed as shorthand notation is as follows:

F=∑(1,2,3)

Now, from this expression, we will find the SOP expression. The Boolean function F has two input variables X and y and the output of F=1 for m1, m2, and m3, i.e., 1st, 2nd, and 3rd combinations. So,

F=∑(1,2,3)

F= m1 + m2 + m3

F= 01 + 10 + 11

Now, we replace zeros with either X' or Y' and ones with either X or Y. Simply, the complement variable is used when the variable value is 1 otherwise the non-complement variable is used.

F = ∑(1,2,3)

F=01+10+11

F= A'B + AB' + AB

Karnaugh Map(K-Map) method

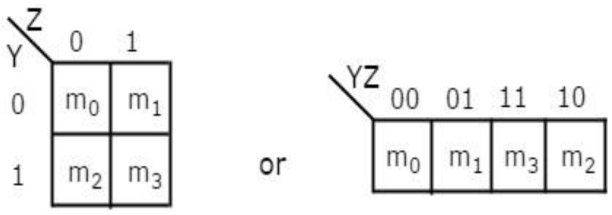
Karnaugh introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 5 Variables

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

2 Variable K-Map

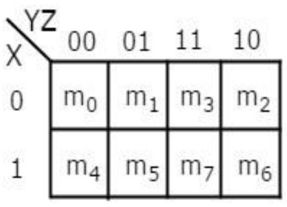
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



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

3 Variable K-Map

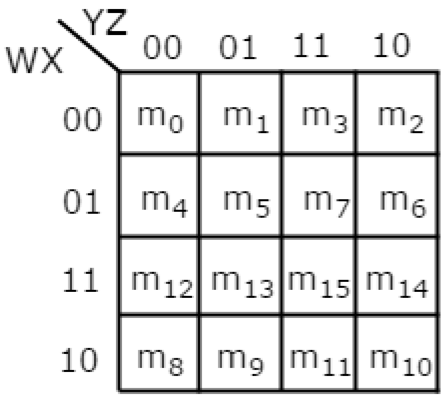
The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



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

4 Variable K-Map

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

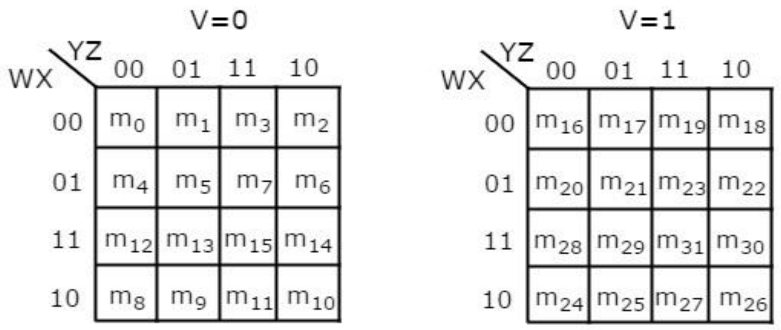


WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)}.
- If w=0, then 4 variable K-map becomes 3 variable K-map.

<u>5 Variable K-Map</u>

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



V=0

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

V=1

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

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

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

<u>Minimization of Boolean Functions using K-Maps</u>

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

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

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

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

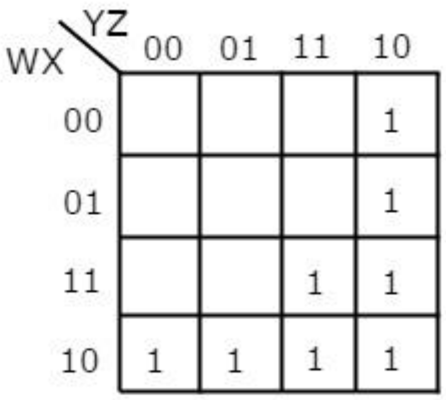
Note 1 − If outputs are not defined for some combination of inputs, then those output values will be represented with **don’t care symbol ‘x’**. That means, we can consider them as either ‘0’ or ‘1’.

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

Eg

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

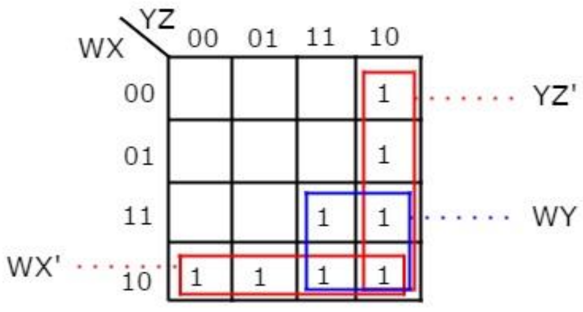
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.



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

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

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



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

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

Therefore, the **simplified Boolean function** is

f = WX’ + WY + YZ’

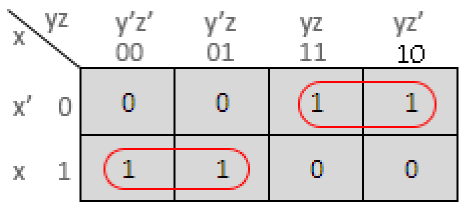
Follow these **rules for simplifying K-maps** in order to get standard product of sums form.

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

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

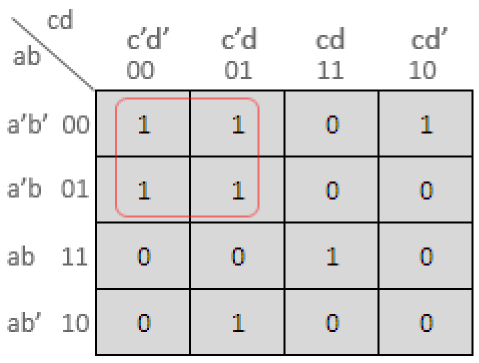
Pair, Quad and Octet

Pair Reduction Rule : Remove the variable which changes its state from complemented to uncomplemented or vice versa.Pair removes one variable only.



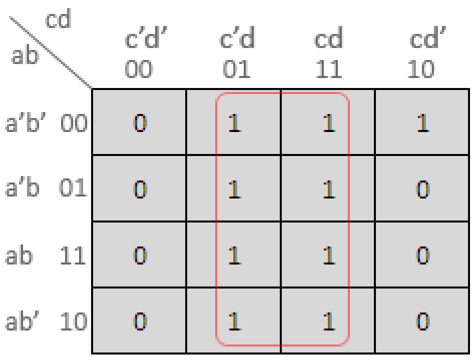
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	0	0	1	1
x 1	1	1	0	0

Quad Reduction Rule : Remove the two variables which change their states.A quad removes two variables.



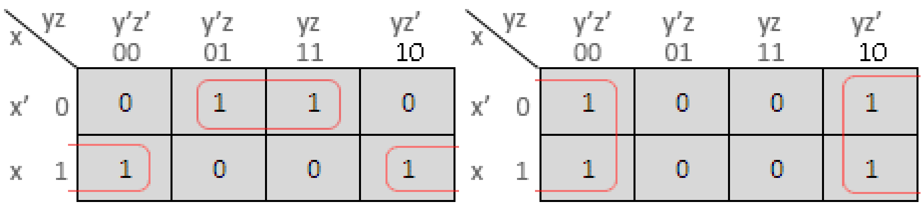
ab \ cd	c'd' 00	c'd 01	cd 11	cd' 10
a'b' 00	1	1	0	1
a'b 01	1	1	0	0
ab 11	0	0	1	0
ab' 10	0	1	0	0

Octet Reduction Rule : Remove the three variables which changes their state.Octet removes three variables.



ab \ cd	c'd' 00	c'd 01	cd 11	cd' 10
a'b' 00	0	1	1	1
a'b 01	0	1	1	0
ab 11	0	1	1	0
ab' 10	0	1	1	0

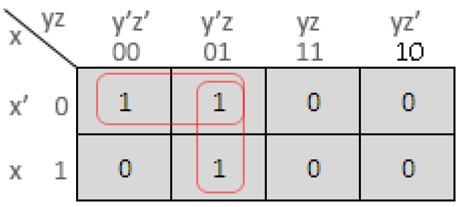
Map Rolling : Map rolling means roll the map considering the map as if its left edges are touching the right edges and top edges are touching bottom edges.While marking the pairs quads and octet, map must be rolled.



x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	0	1	1	0
x 1	1	0	0	1

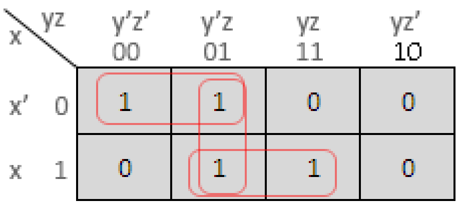
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	0	0	1
x 1	1	0	0	1

Overlapping Groups : Overlapping means same 1 can be encircled more than once. Overlapping always leads to simpler expressions.



x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	1	0	0
x 1	0	1	0	0

Redundant Group : It is a group whose all 1's are overlapped by other groups. Redundant groups must be removed. Removal of redundant group leads to much simpler expression.



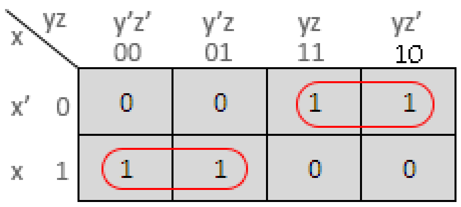
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	1	0	0
x 1	0	1	1	0

Eg : Represent the following boolean expression in a K-map and simplify.

F = x'yz + x'yz' + xy'z' + xy'z

Solution :

The K-map is as follows :



x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	0	0	1	1
x 1	1	1	0	0

Hence the simplified expression is

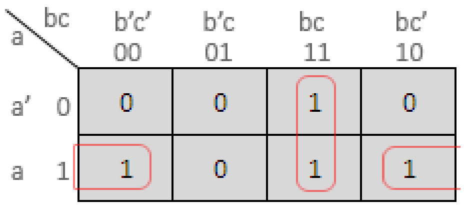
F = x'y + xy'

<u>Ex. 2</u> **:**Simplify the following boolean expression using K-map.

F = a'bc + ab'c' + abc + abc'

<u>Solution</u> :

The K-map is as follows :



a \ bc	b'c' 00	b'c 01	bc 11	bc' 10
a' 0	0	0	1	0
a 1	1	0	1	1

Hence the simplified expression is

F = bc + ac'

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

The "Don't Care" conditions allow us to replace the empty cell of a K-Map to form a grouping of the variables. While forming groups of cells, we can consider a "Don't Care" cell as either 1 or 0 or we can simply ignore that cell. Therefore, "Don't Care" condition can help us to form a larger group of cells.

A Don't Care cell can be represented by a cross(X) in K-Maps representing a invalid combination. For example, in Excess-3 code system, the states 0000, 0001, 0010, 1101, 1110 and 1111 are invalid or unspecified. These are called don't cares. Also, in design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don't cares.

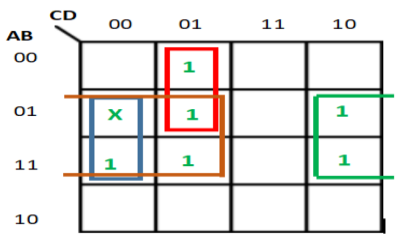
A standard SOP function having don't cares can be converted into a POS expression by keeping don't cares as they are, and writing the missing minterms of the SOP form as the maxterm of POS form. Similarly, a POS function having don't cares can be converted to SOP form keeping the don't cares as they are and write the missing maxterms of the POS expression as the minterms of SOP expression.

<u>Eg</u>

Minimise the following function in SOP minimal form using K-Maps:

f = m(1, 5, 6, 12, 13, 14) + d(4)

Explanation
The SOP K-map for the given expression is:



Therefore, SOP minimal is,

f = BC' + BD' + A'C'D

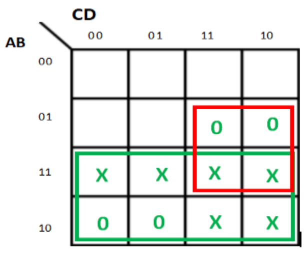
Eg
Minimise the following function in SOP minimal form using K-Maps:

F(A, B, C, D) = m(0, 1, 2, 3, 4, 5) + d(10, 11, 12, 13, 14, 15)

Explanation:
Writing the given expression in POS form:

F(A, B, C, D) = M(6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)

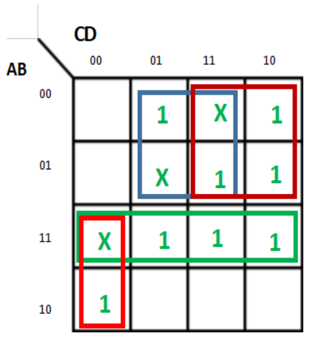
The POS K-map for the given expression is:



Therefore, POS minimal is,

F = A'(B' + C')

Eg
Minimise the following function in SOP minimal form using K-Maps: F(A, B, C, D) = m(1, 2, 6, 7, 8, 13, 14, 15) + d(3, 5, 12)
Explanation:
The SOP K-map for the given expression is:



Therefore,

f = AC'D' + A'D + A'C + AB

Product of Sum Method(POS)

A canonical product of sum is a boolean expression that entirely consists of maxterms. The Boolean function F is defined on two variables X and Y. The X and Y are the inputs of the boolean function F whose output is true when only one of the inputs is set to true. The truth table for Boolean expression F is as follows:

Inputs		**Output**
X	**Y**	**F**
0	0	0
0	1	1
1	0	1
1	1	0

In our minterm and maxterm section, we learned about how we can form the maxterm from the variable's value. A column will be added for the maxterm in the above table. The complement of the variables is taken whose value is 0, and the variables whose value is 1 will remain the same.

Inputs		**Output**	**Minterm**
X	**Y**	**F**	**M**
0	0	0	X'+Y'
0	1	1	X'+Y
1	0	1	X+Y'
1	1	1	X+Y

Now, we will multiply all the minterms for which the output is false to find the desired canonical POS(Product of sum) expression.

F=(X'+Y').(X+Y)

Converting Product of Sum (POS) to shorthand notation

The process of converting POS form to shorthand notation is the same as the process of finding shorthand notation for maxterms. There are the following steps used to find the shorthand notation of the given POS expression.

- Write the given POS expression.
- Find the shorthand notation of all the maxterms.
- Replace the minterms with their shorthand notations in the given expression.

Eg

F = (X'+Y').(X+Y)

1. Firstly, we will write the POS expression:

F = (X'+Y').(X+Y)

2. Now, we will find the shorthand notations of the maxterms X'+Y' and X+Y.

X'+Y' = $(00)_2 = M_0$

X+Y = $(11)_2 = M_3$

3. In the end, we will replace all the minterms with their shorthand notations:

$F=M_0.M_3$

Converting shorthand notation to POS expression

The process of converting shorthand notation to POS is the reverse process of converting POS expression to shorthand notation. Let's see an example to understand this conversion.

Eg

Let us assume that we have a boolean function F, defined on two variables X and Y. The maxterms for the function F are expressed as shorthand notation is as follows:

$$F=\prod(1,2,3)$$

Now, from this expression, we find the POS expression. The Boolean function F has two input variables X and Y and the output of F=0 for M1, M2, and M3, i.e., 1st, 2nd, and 3rd combinations. So,

$$F=\prod(1,2,3)$$

$$F= M1.M2.M3$$

$$F= 01.10.11$$

Next, we replace zeros with either X or Y and ones with either X' or Y'. Simply, if the value of the variable is 1, then we take the complement of that variable, and if the value of the variable is 0, then we take the variable "as is".

F = ∑(1,2,3)

F=01.10.11

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

Product of Sum Simplification

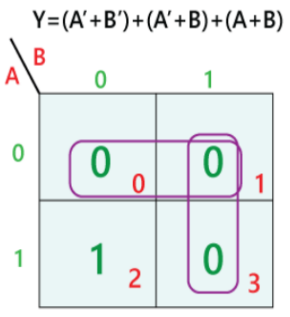
To find the simplified maxterm solution using K-map is the same as to find for the minterm solution. There are some minor changes in the maxterm solution, which are as follows:

1. We will populate the K-map by entering the value of 0 to each sum-term into the K-map cell and fill the remaining cells with one's.
2. We will make the groups of 'zeros' not for 'ones'.
3. Now, we will define the boolean expressions for each group as sum-terms.
4. At last, to find the simplified boolean expression in the POS form, we will combine the sum-terms of all individual groups.

Let's take some example of 2-variable, 3-variable, 4-variable and 5-variable K-map examples

Eg

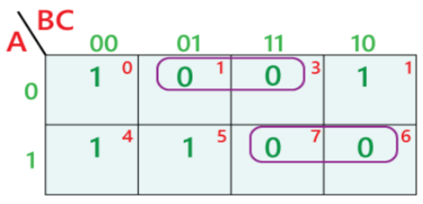
Y=(A'+B')+(A'+B)+(A+B)



Simplified expression: A'B

Eg

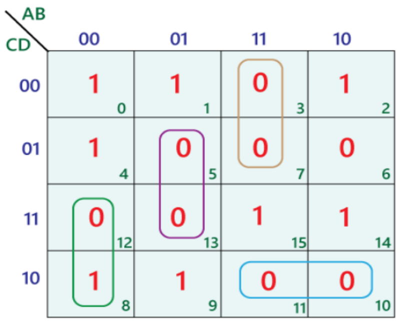
Y=(A + B + C') + (A + B' + C') + (A' + B' + C) + (A' + B' + C')



Simplified expression: Y=(A + C') .(A' + B')

Eg

F(A,B,C,D)=π(3,5,7,8,10,11,12,13)



Simplified expression: Y=(A + C') .(A' + B')

Data Processing Circuit

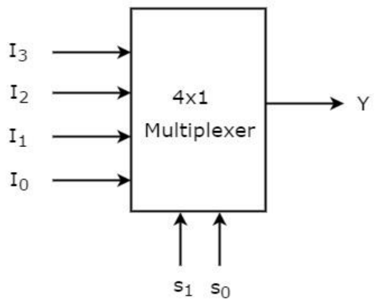
Multiplexer

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

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

4x1 Multiplexer

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



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

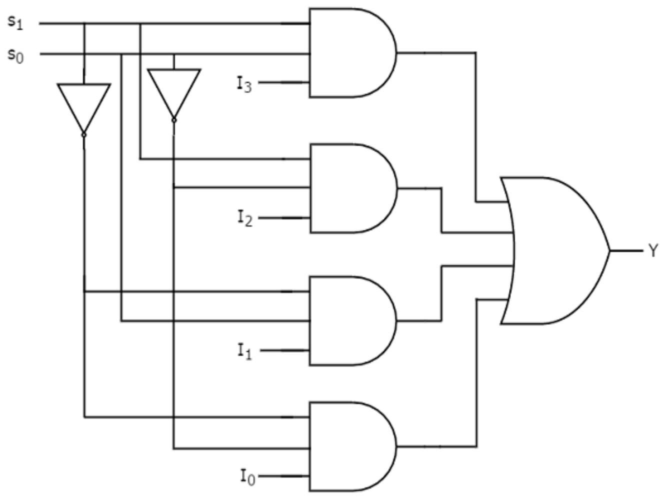
Selection Lines		**Output**
S_1	**S_0**	**Y**
0	0	I_0
0	1	I_1

1	0	I_2
1	1	I_3

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

Y={S{1}}'{S{0}}'I{0}+{S{1}}'S{0}I{1}+S{1}{S{0}}'I{2}+S{1}S{0}I{3}

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



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

Implementation of Higher-order Multiplexers.

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

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer

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

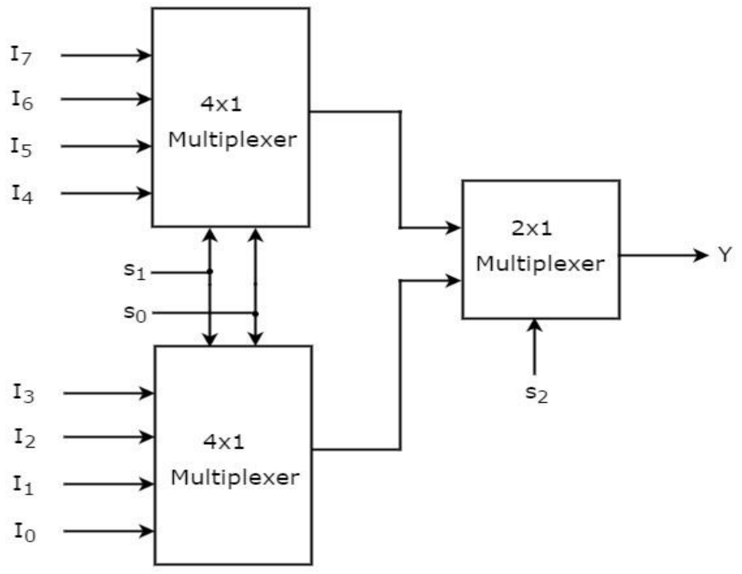
So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

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

Selection Inputs			**Output**
$\mathbf{S_2}$	$\mathbf{S_1}$	$\mathbf{S_0}$	**Y**

0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

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



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

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

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

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

<u>16x1 Multiplexer</u>

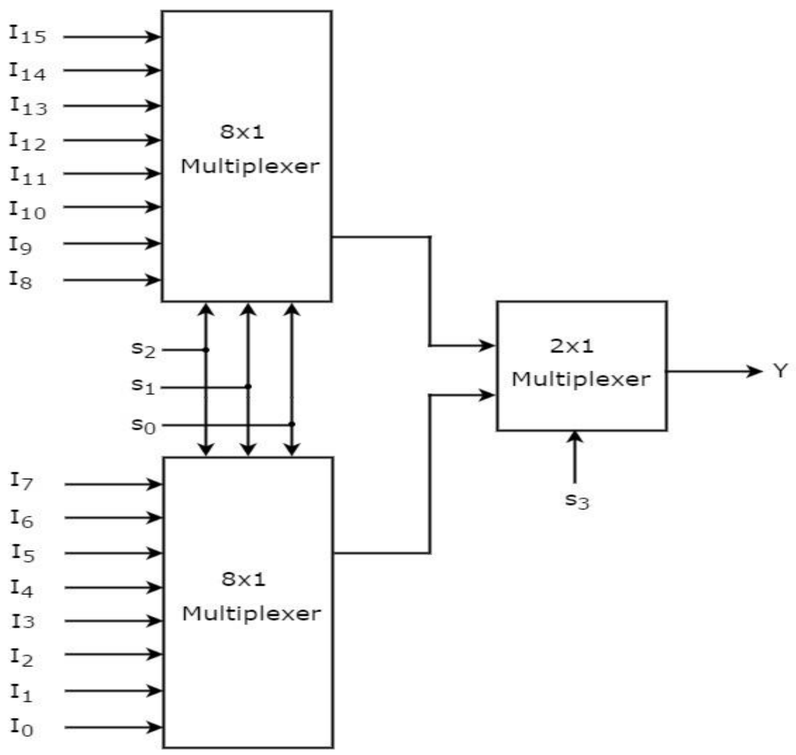
In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

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

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

Selection Inputs				**Output**
S_3	**S_2**	**S_1**	**S_0**	**Y**
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9
1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}
1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

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



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

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

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

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

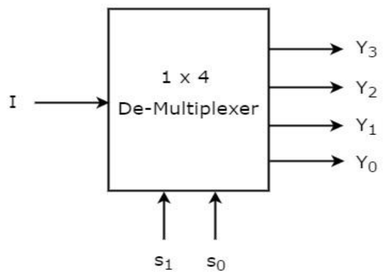
De-Multiplexer

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

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

1x4 De-Multiplexer

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



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

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

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

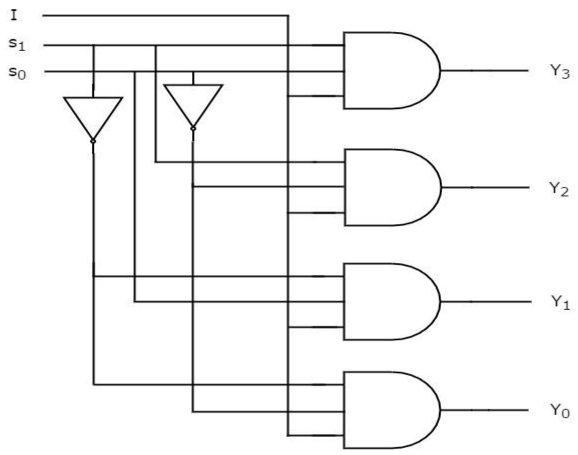
Y{3}=s{1}s{0}I

Y{2}=s{1}{s{0}}'I

Y{1}={s{1}}'s{0}I

Y{0}={s1}'{s{0}}'I

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



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

Implementation of Higher-order De-Multiplexers

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

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

1x8 De-Multiplexer

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

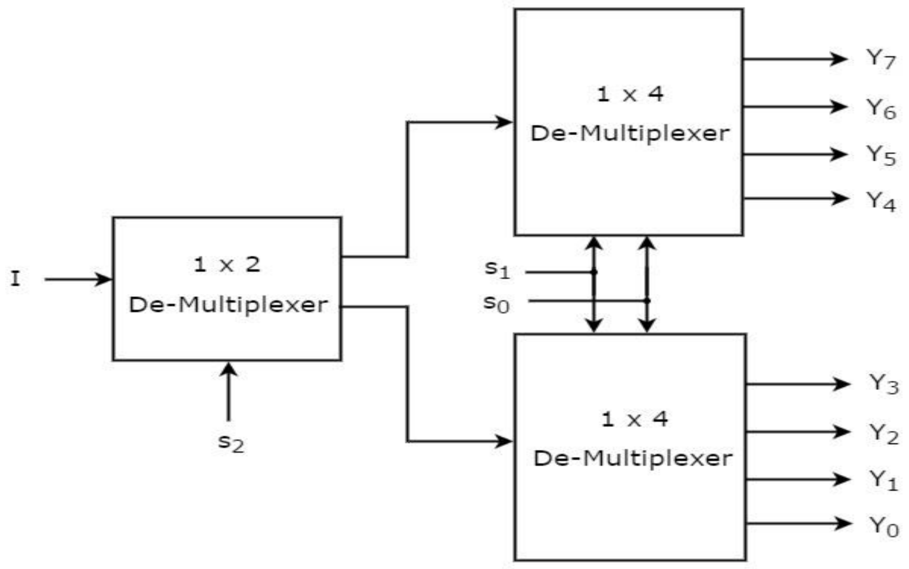
So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer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

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

Selection Inputs			**Outputs**							
s_2	**s_1**	**s_0**	**Y_7**	**Y_6**	**Y_5**	**Y_4**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	0	0	0	0	0	0	0	0	0	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0

1	1	1	**I**	0	0	0	0	0	0	0

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



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

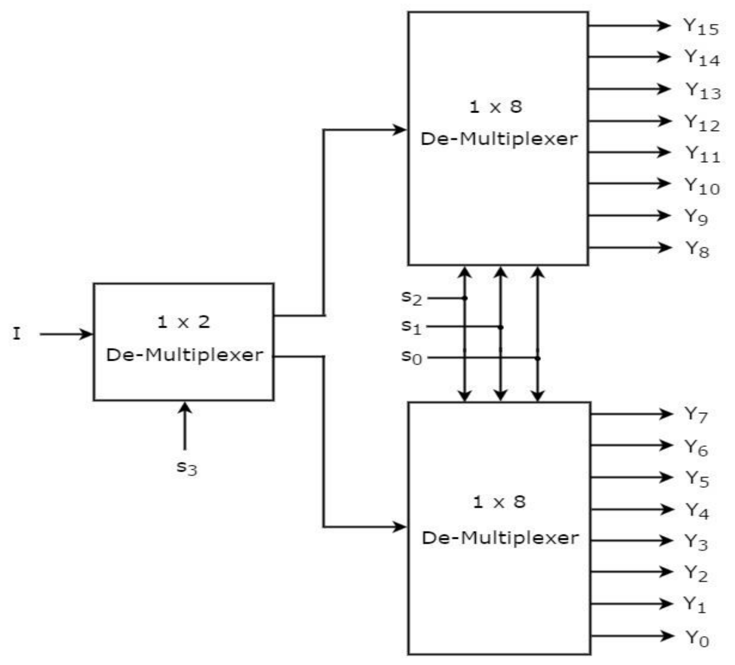
The other **selection line, s_2** is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0. Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will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

<u>1x16 De-Multiplexer</u>

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

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

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



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

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

Decoder

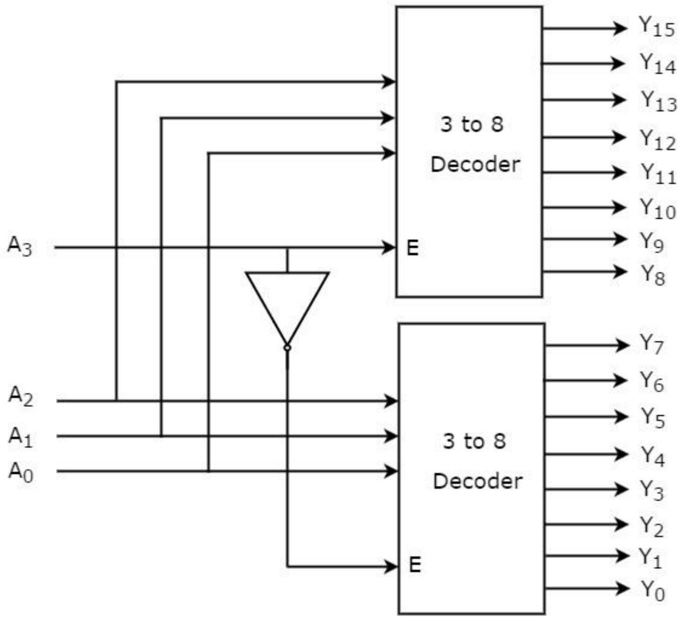
Decoder is a combinational circuit that has ‘n’ input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min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

1 of 16 Decoder

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

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

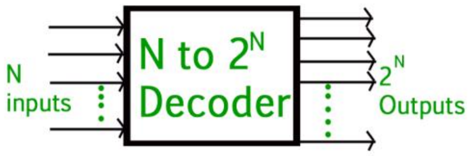
Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The **block diagram** of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.



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

<u>BCD to Decimal Decoder</u>

In Digital Electronics, discrete quantities of information are represented by binary codes. A binary code of **n bits** is capable of representing up to **2^n distinct elements** of coded information. The name **"Decoder"** means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output. A **decoder** is a **combinational circuit** that converts binary information from **n input lines** to a maximum of **2^n unique output lines**.



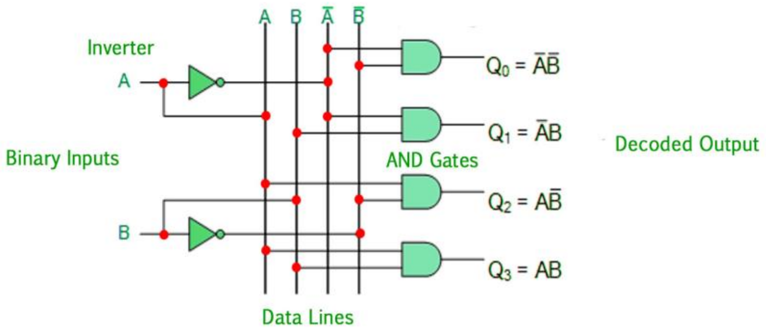
<u>Binary Decoder</u>

- Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an n-bit code, and therefore it will be possible to represent 2^n possible values.
- If a binary decoder receives n inputs it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

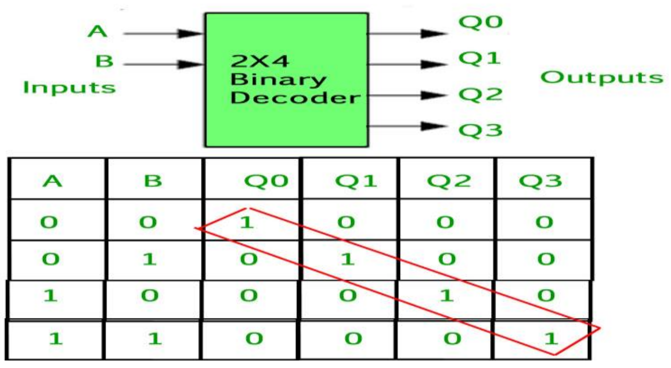
- Example, an inverter (NOT-gate) can be classified as a 1-to-2 binary decoder as 1-input and 2-outputs is possible. i.e an input A can give either A or A complement as the output.
- Then we can say that a standard combinational logic decoder is an n-to-m decoder, where m <= 2^n, and whose output, Q is dependent only on its present input states.
- Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output.

A Binary Decoder converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to "decode" either a Binary or BCD (8421 code) input pattern to typically a Decimal output code.
Practical "binary decoder" circuits include 2-to-4, 3-to-8 and 4-to-16 line configurations.

<u>2-to-4 Binary Decoder</u>



The 2-to-4 line binary decoder depicted above consists of an array of four AND gates. The 2 binary inputs labeled A and B are decoded into one of 4 outputs, hence the description of a 2-to-4 binary decoder. Each output represents one of the minterms of the 2 input variables, (each output = a minterm).



A	B	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The output values will be:

Qo=A'B'
Q1=A'B
Q2=AB'
Q3=AB

The binary inputs A and B determine which output line from Q0 to Q3 is “HIGH” at logic level “1” while the remaining outputs are held “LOW” at logic “0” so only one output can be active (HIGH) at any one time. Therefore, whichever output line is “HIGH” identifies the binary code present at the input, in other words, it “decodes” the binary input.

Some binary decoders have an additional input pin labeled “Enable” that controls the outputs from the device. This extra input allows the outputs of the decoder to be turned “ON” or “OFF” as required. The output is only generated when the Enable input has value 1; otherwise, all outputs are 0. Only a small change in the implementation is required: the Enable input is fed into the AND gates which produce the outputs.

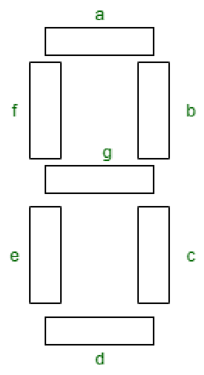
If Enable is 0, all AND gates are supplied with one of the inputs as 0 and hence no output is produced. When Enable is 1, the AND gates get one of the inputs as 1, and now the output depends upon the remaining inputs. Hence the output of the decoder is dependent on whether the Enable is high or low.

Seven Segment Decoder

Light Emitting Diode (LED) is the most widely used semiconductor which emits either visible light or invisible infrared light when forward biased. Remote controls generate invisible light. A Light emitting diode (LED) is an optical electrical energy into light energy when voltage is applied.

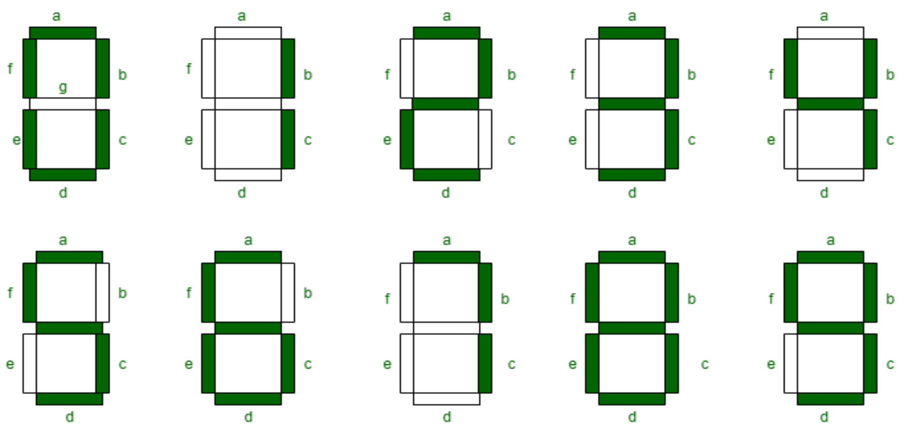
Seven Segment Displays

Seven segment displays are the output display device that provide a way to display information in the form of image or text or decimal numbers which is an alternative to the more complex dot matrix displays. It is widely used in digital clocks, basic calculators, electronic meters, and other electronic devices that display numerical information. It consists of seven segments of light emitting diodes (LEDs) which is assembled like numerical 8.



Working of Seven Segment Displays

The number 8 is displayed when the power is given to all the segments and if you disconnect the power for ‘g’, then it displays number 0. In a seven segment display, power (or voltage) at different pins can be applied at the same time, so we can form combinations of display numerical from 0 to 9. Since seven segment displays can not form alphabet like X and Z, so it can not be used for alphabet and it can be used only for displaying decimal numerical magnitudes. However, seven segment displays can form alphabets A, B, C, D, E, and F, so they can also used for representing hexadecimal digits.



We can produce a truth table for each decimal digit

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Therefore, Boolean expressions for each decimal digit which requires respective light emitting diodes (LEDs) are ON or OFF. The number of segments used by digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 are 6, 2, 5, 5, 4, 5, 6, 3, 7, and 6 respectively. Seven segment displays must be controlled by other external devices where different types of microcontrollers are useful to communicate with these external devices, like switches, keypads, and memory.

<u>Types of Seven Segment Displays</u>

According to the type of application, there are two types of configurations of seven segment displays: common anode display and common cathode display.

1. In common cathode seven segment displays, all the cathode connections of LED segments are connected together to logic 0 or ground. We use logic 1 through a current limiting resistor to forward bias the individual anode terminals a to g.

2. Whereas all the anode connections of the LED segments are connected together to logic 1 in common anode seven segment display. We use logic 0 through a current limiting resistor to the cathode of a particular segment a to g.

Common anode seven segment displays are more popular than cathode seven segment displays, because logic circuits can sink more current than they can source and it is the same as connecting LEDs in reverse.

Applications of Seven Segment Displays

Common applications of seven segment displays are in:

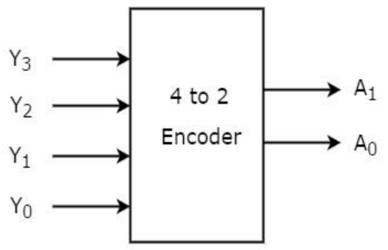
1. Digital clocks
2. Clock radios
3. Calculators
4. Wristwatchers
5. Speedometers
6. Motor-vehicle odometers
7. Radio frequency indicators

Encoder

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

4 to 2 Encoder

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



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

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

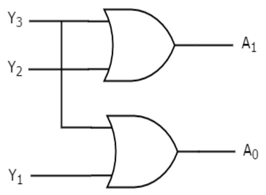
1	0	0	0	1	1

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

$$A\{1\}=Y\{3\}+Y\{2\}$$

$$A\{0\}=Y\{3\}+Y\{1\}$$

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



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

<u>Exclusive-OR gate</u>

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

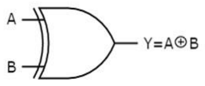
The following table shows the **<u>truth table</u>** of 2-input Ex-OR gate.

A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

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

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

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



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

Parity Bit Generator

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

Even Parity Generator

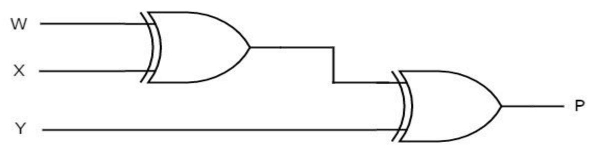
Now, let us implement an even parity generator for a 3-bit binary input, WXY. It generates an even parity bit, P. If odd number of ones present in the input, then even parity bit, P should be '1' so that the resultant word contains even number of ones. For other combinations of input, even parity bit,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

Binary Input WXY	Even Parity bit P
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

From the above Truth table, we can write the **Boolean function** for even parity bit as

P={W}'{X}'Y+{W}'X{Y}'+W{X}'{Y}'+WXY

The following figure shows the **circuit diagram** of even parity generator.

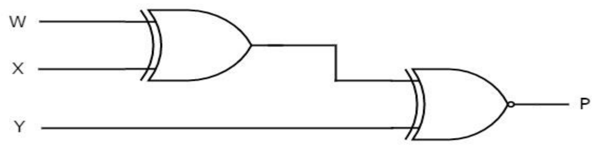


This circuit consists of two **Exclusive-OR gates** having two inputs each. First ExclusiveOR gate having two inputs W & X and produces an output $W \oplus X$. This output is given as one input of second Exclusive-OR gate. The other input of this second Exclusive-OR gate is Y and produces an output of $W \oplus X \oplus Y$.

Odd Parity Generator

If even number of ones present in the input, then odd parity bit, P should be '1' so that the resultant word contains odd number of ones. For other combinations of input, odd parity bit, P should be '0'.

Follow the same procedure of even parity generator for implementing odd parity generator. The **circuit diagram** of odd parity generator is shown in the following figure.



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

Parity Checker

There are two types of parity checkers based on the type of parity has to be checked. **Even parity checker** checks error in the transmitted data, which contains message bits along with even parity. Similarly, **odd parity checker** checks error in the transmitted data, which contains message bits along with odd parity.

Even parity checker

Now, let us implement an even parity checker circuit. Assume a 3-bit binary input, WXY is transmitted along with an even parity bit, P. So, the resultant word (data) contains 4 bits, which will be received as the input of even parity checker.

It generates an **even parity check bit, E**. This bit will be zero, if the received data contains an even number of ones. That means, there is no error in the received data. This even parity check bit will be one, if the received data contains an odd number of ones. That means, there is an error in the received data.

The following table shows the **Truth table** of an even parity checker.

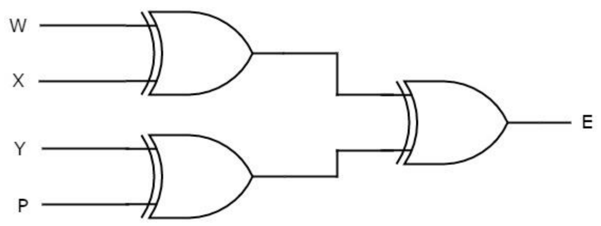
4-bit Received Data WXYP	Even Parity Check bit E
0000	0
0001	1
0010	1
0011	0
0100	1
0101	0
0110	0
0111	1
1000	1
1001	0

1010	0
1011	1
1100	0
1101	1
1110	1
1111	0

From the above Truth table, we can observe that the even parity check bit value is ‘1’, when odd number of ones present in the received data. That means the Boolean function of even parity check bit is an **odd function**. Exclusive-OR function satisfies this condition. Hence, we can directly write the **Boolean function** of even parity check bit as

$$E=W + X + Y + P$$

The following figure shows the **circuit diagram** of even parity checker.



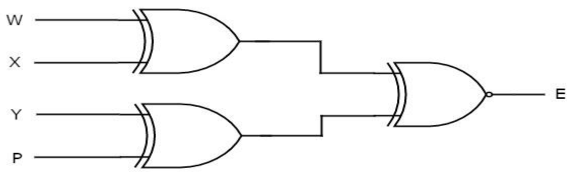
This circuit consists of three **Exclusive-OR gates** having two inputs each. The first level gates produce outputs of W + X & Y + P. The Exclusive-OR gate, which is in second level produces an output of W + X + Y + P.

Odd Parity Checker

Assume a 3-bit binary input, WXY is transmitted along with odd parity bit, P. So, the resultant word (data) contains 4 bits, which will be received as the input of odd parity checker.

It generates an **odd parity check bit, E**. This bit will be zero, if the received data contains an odd number of ones. That means, there is no error in the received data. This odd parity check bit will be one, if the received data contains even number of ones. That means, there is an error in the received data.

Follow the same procedure of an even parity checker for implementing an odd parity checker. The **circuit diagram** of odd parity checker is shown in the following figure.



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

REVIEW QUESTIONS

1) Write associate and commutative law.
2) Define multiplexer.
3) Explain in detail about seven segment decoder.
4) Define karnaugh map.
5) How to fine pair, quad and octet in k-map.
6) Explain about don’t care condition.
7) Explain about parity checker.
8) Discuss about XOR gates.