Unit – 4
Combinational Logic

Combinational Logic circuit Implementation:

Combinational Logic Circuit: A combinational circuit is one in which the state of the output at any instant is entirely determined by the states of the inputs at that time. The output occurs immediately after a slight propagation delay once the input is given.

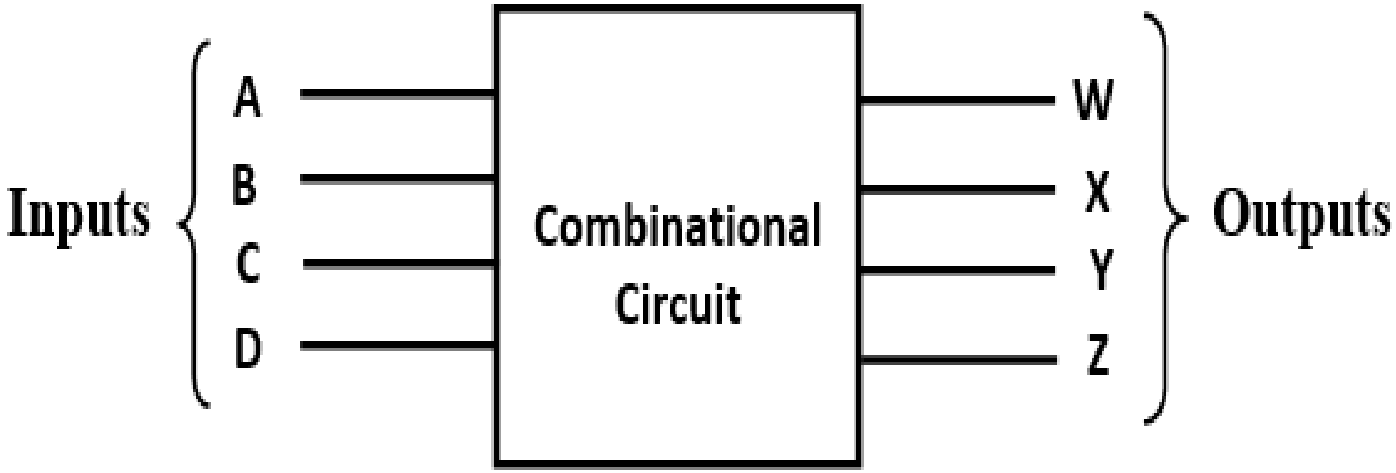
There is no memory in combinational circuits. There will be 2^n combinations of input variable for n inputs. The output will be different for each input combination. Adders, subtractors, decoders, encoders, XOR, XNOR gates, Multiplexer, De-multiplexer etc.

Combinational circuit design procedure:

1) Problem definition. That is problem is stated.
2) Determine the number of input and output variables.
3) Assign each input and output variable with letter symbol.
4) Find the relationship between input and output variables using truth table and logic functions.
5) Simplify the logic function using K-map or Boolean algebra.
6) Draw logic diagram for the simplified logic expression.

Example: Design a combinational circuit with four input lines that represent a decimal digit in BCD and four output lines that generate the 9's compliment of the input digit.

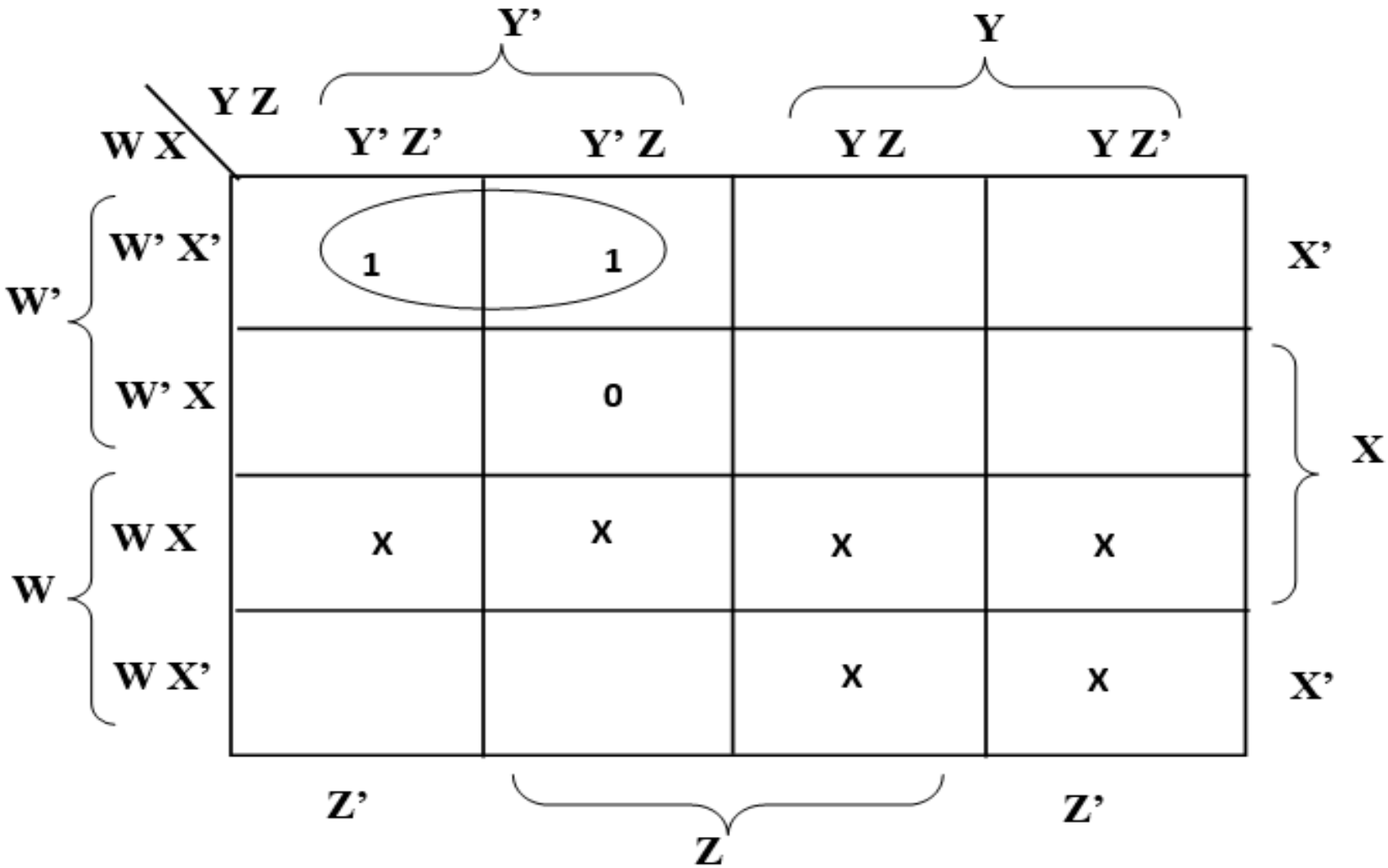
Solution:



Truth table:

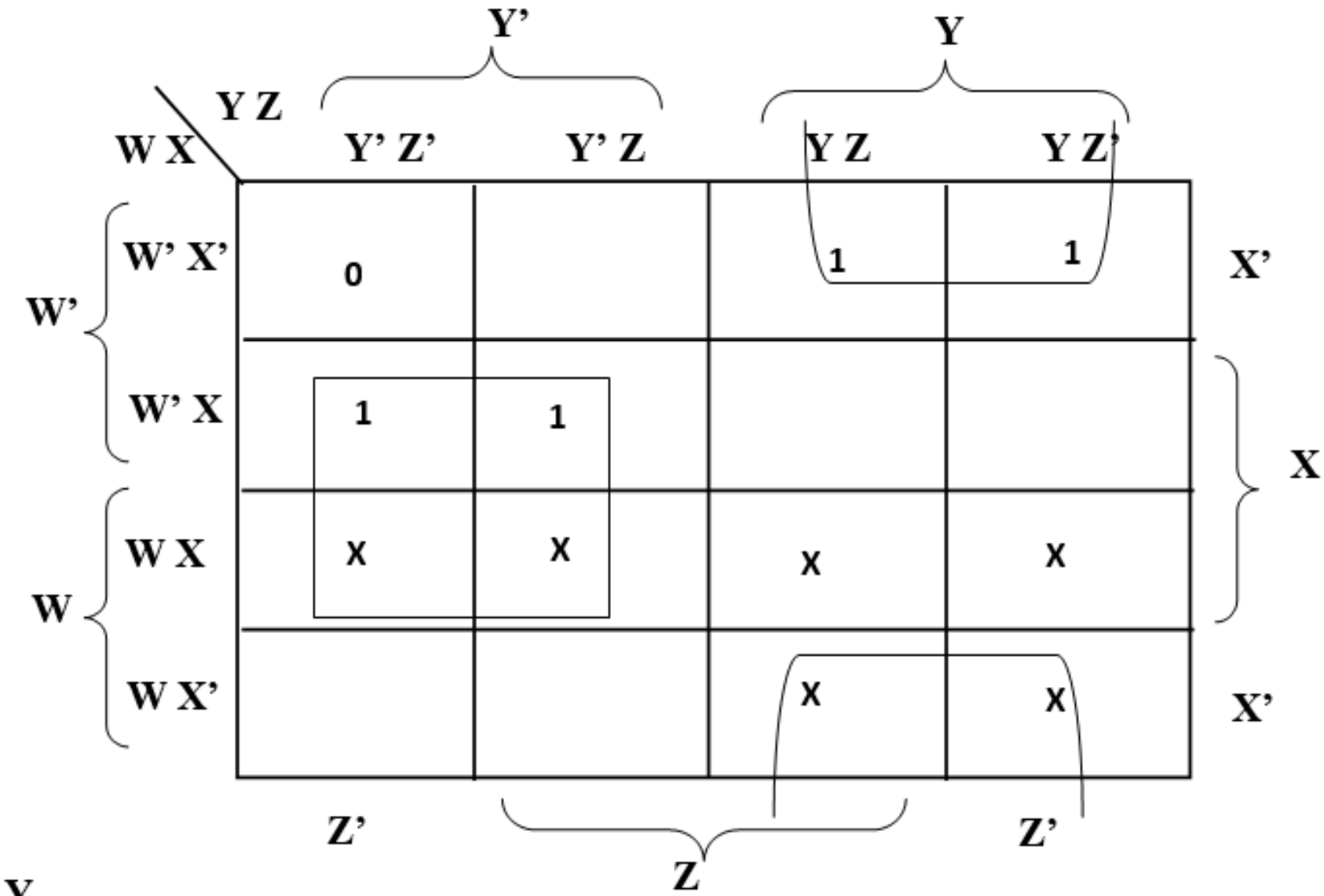
Inputs		Outputs
Decimal	BCD	9's complement
	W X Y Z	A B C D
0	0 0 0 0	1 0 0 1
1	0 0 0 1	1 0 0 0
2	0 0 1 0	0 1 1 1
3	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 0 1
5	0 1 0 1	0 1 0 0
6	0 1 1 0	0 0 1 1
7	0 1 1 1	0 0 1 0
8	1 0 0 0	0 0 0 1
9	1 0 0 1	0 0 0 0

K-Map for A:



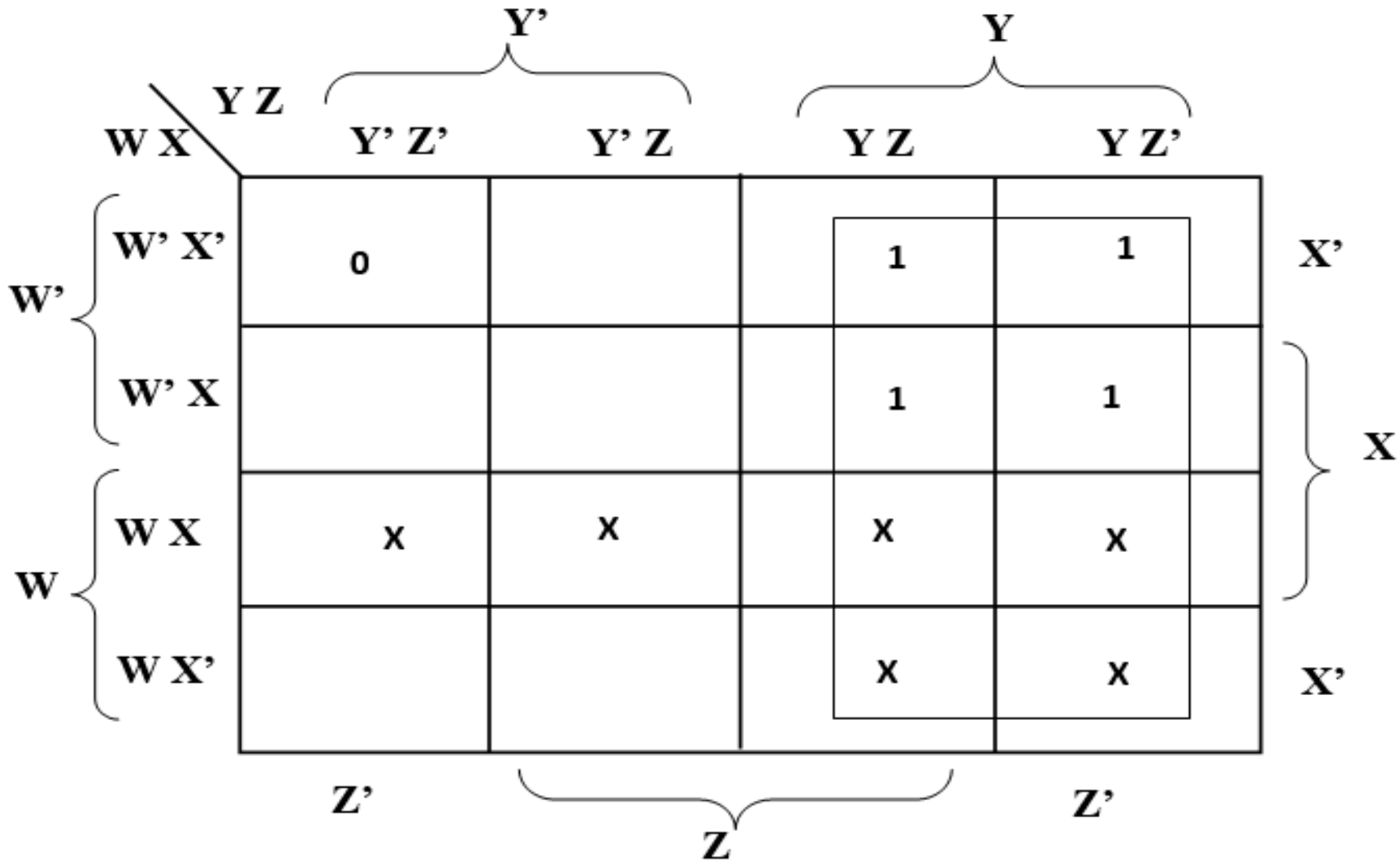
A = W' X' Y'

K-Map for B:



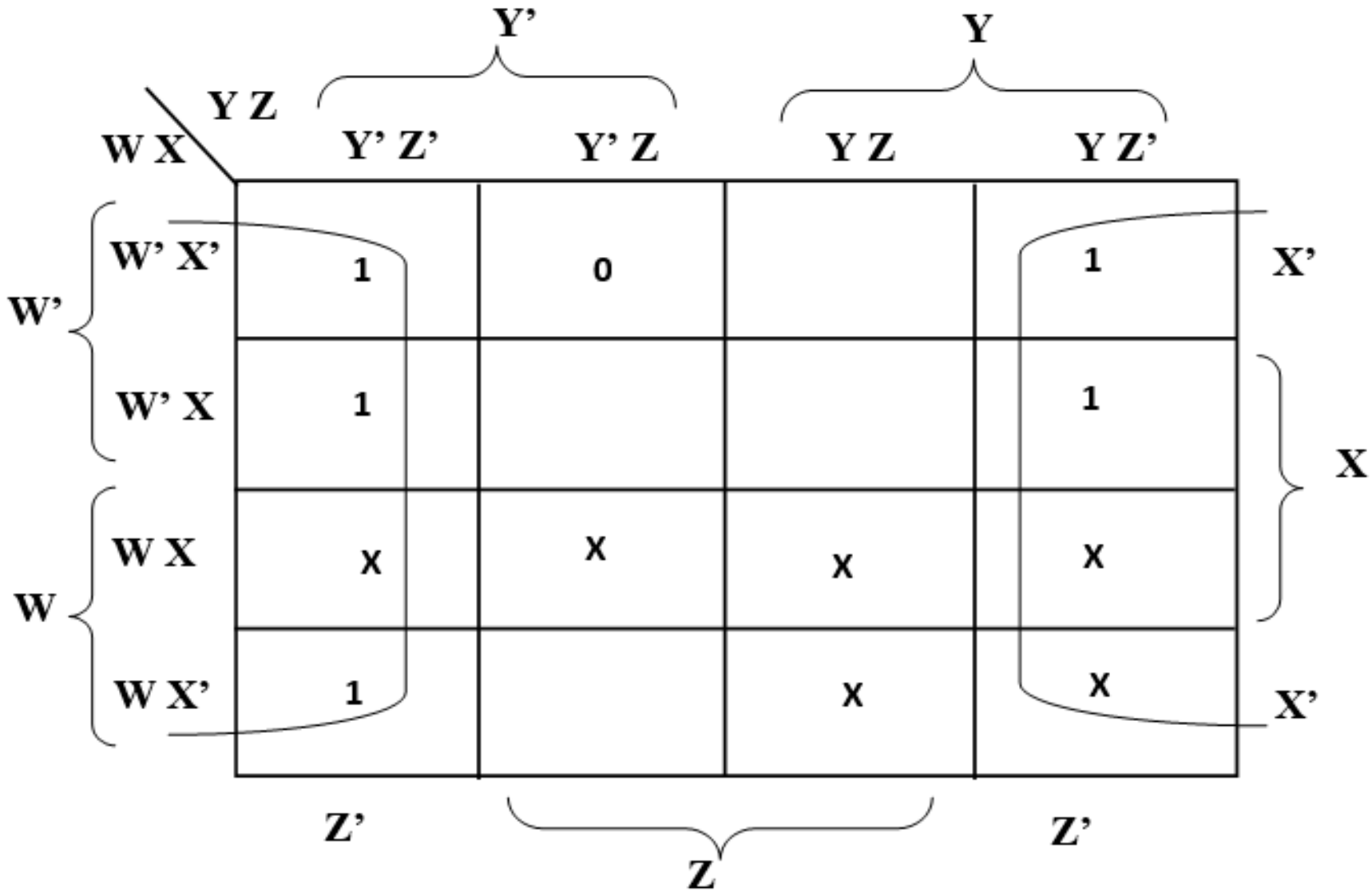
B = X Y' + X' Y

K-Map for C:



C = Y

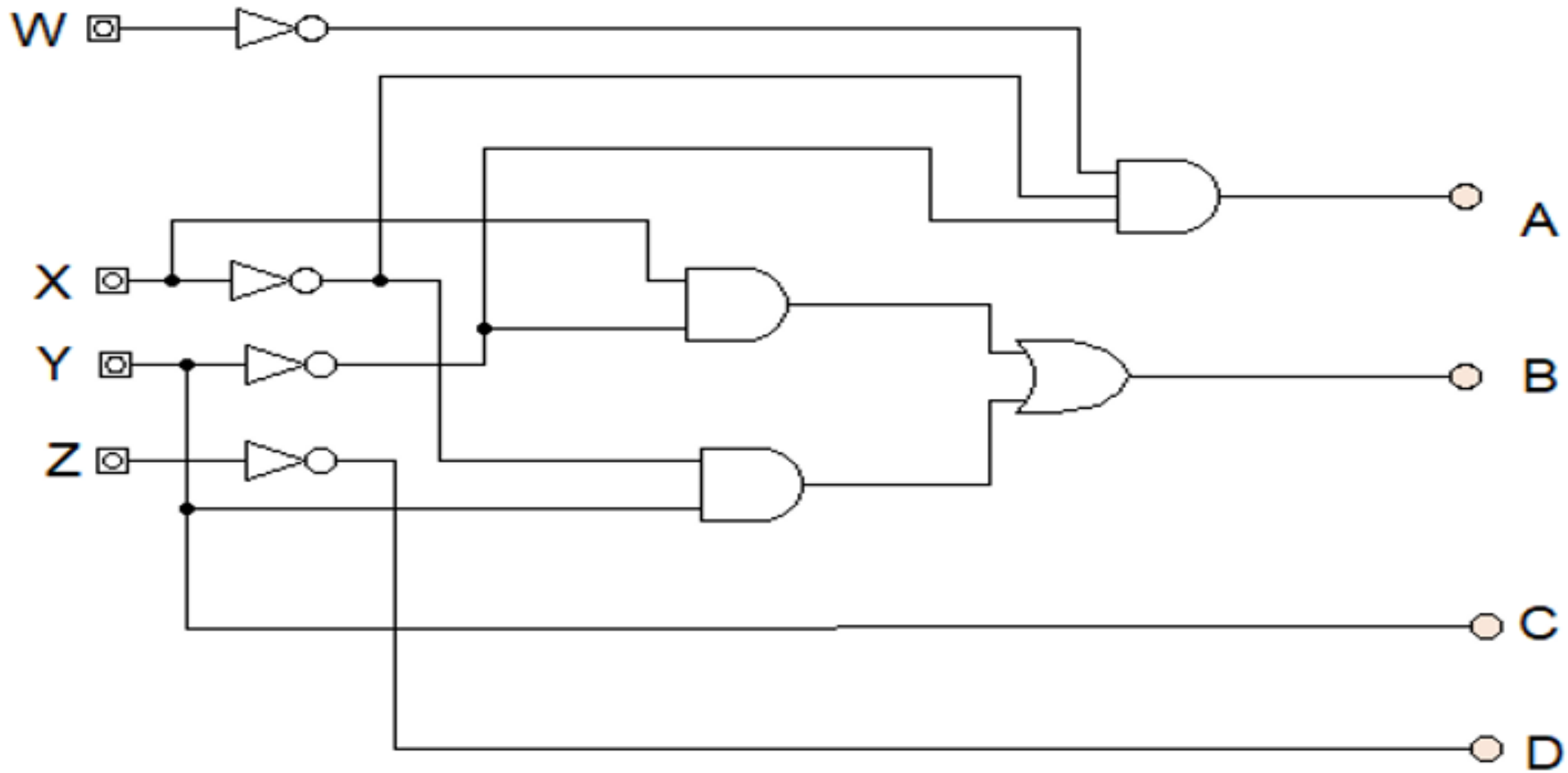
K-Map for D:



D = Z'

Logic diagram implementation:

$A = W' X' Y' \quad B = X Y' + X'Y \quad C = Y \quad D = Z'$



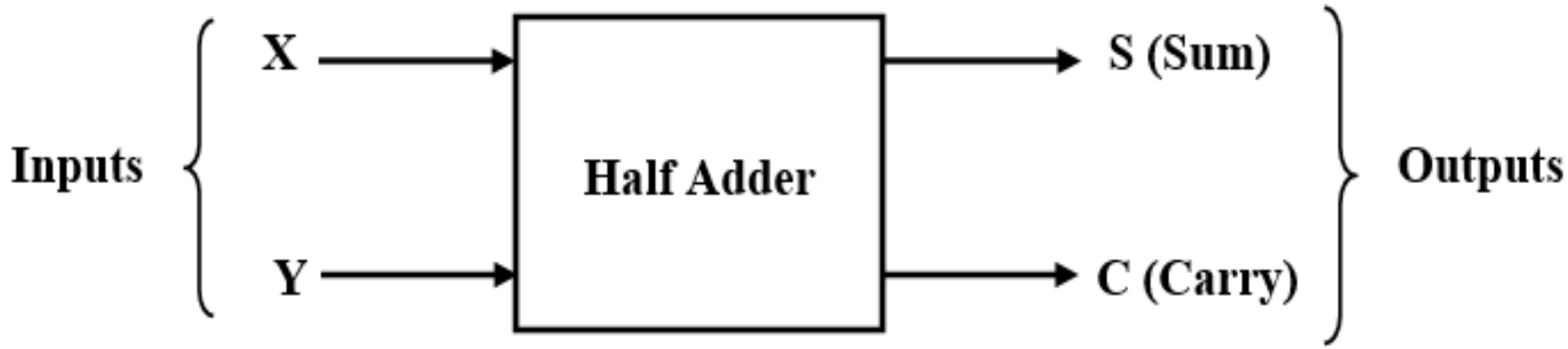
Adders: Adders are the combinational logic circuit, which is used to add two or more than two bits at a time.

Types of adders:

1) **Half Adder**
2) **Full Adder**

Half-Adder: It is a combinational logic circuit, which is used to find the sum of two binary digits at a time. 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).

Block diagram of Half-Adder:



Now we formulate a Truth table to 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

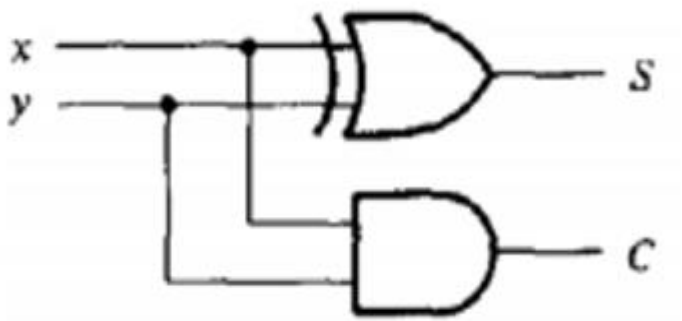
The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified **sum of products** expressions are:

$$S = x'y + xy' \quad \text{or} \quad S = x \oplus y$$

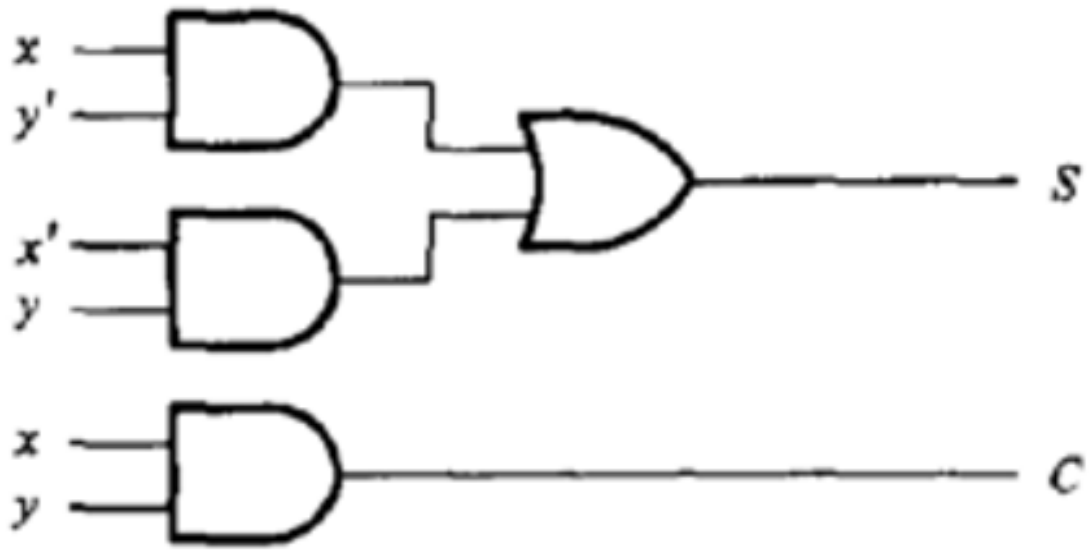
$$C = xy$$

Logic Diagram of Half Adder:

Or



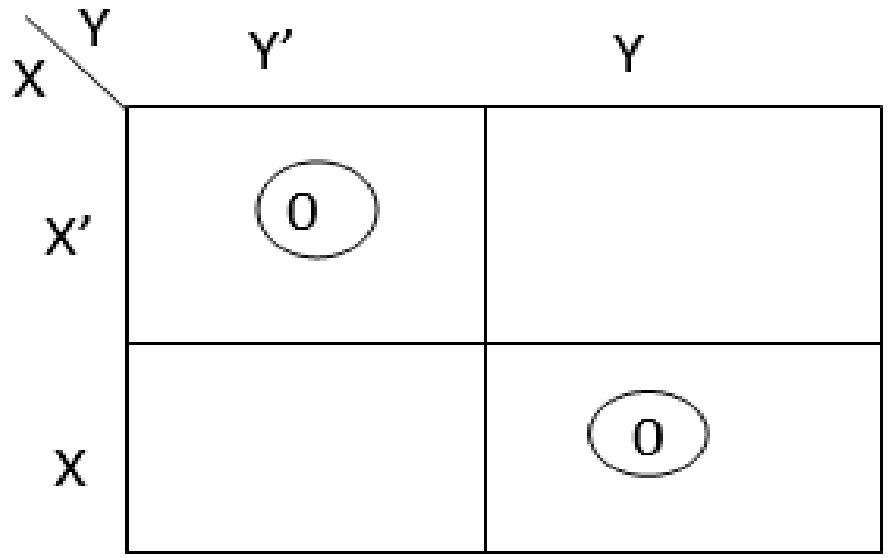
(c) $S = x \oplus y$
$C = xy$



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

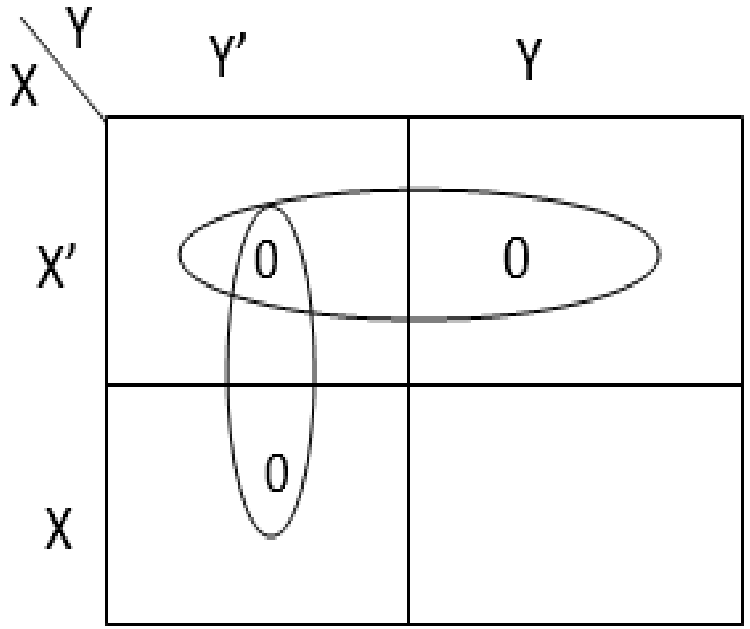
The Sum and Carry can be realized in Product of Sums form.

K-map for simplified expression in POS for Sum:



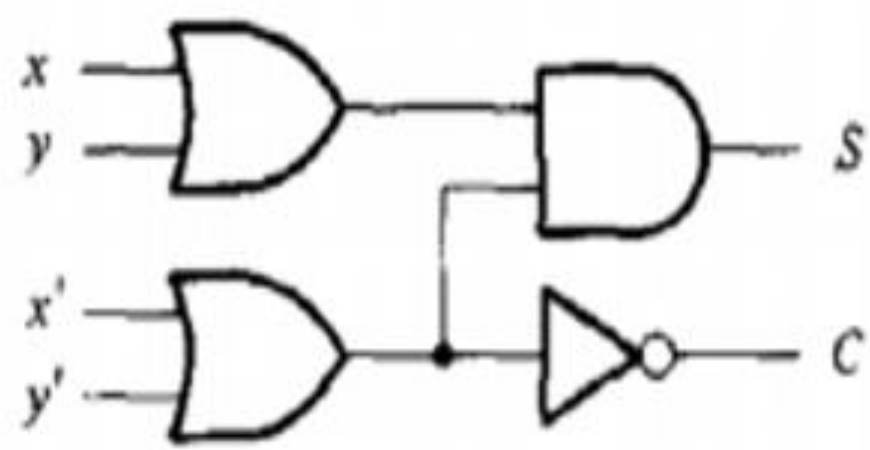
Sum (S) = (x + y) (x' + y')

K-Map for Carry:



Carry (C) = (x' + y')'

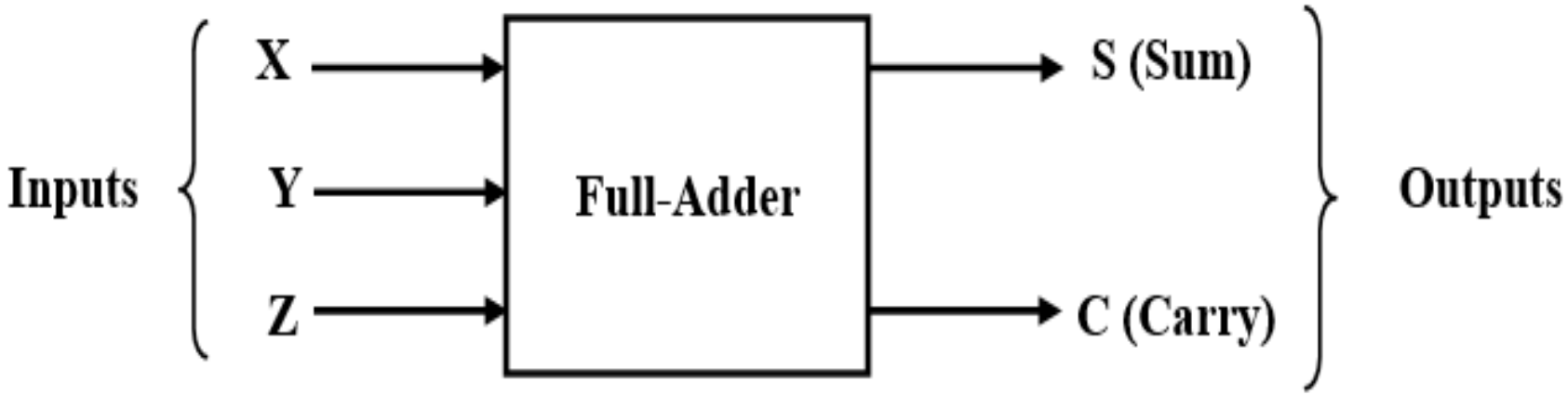
Logic diagram of Half-adder for sum and carry in POS:



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

Full Adder: 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.

Block diagram of Full-Adder:

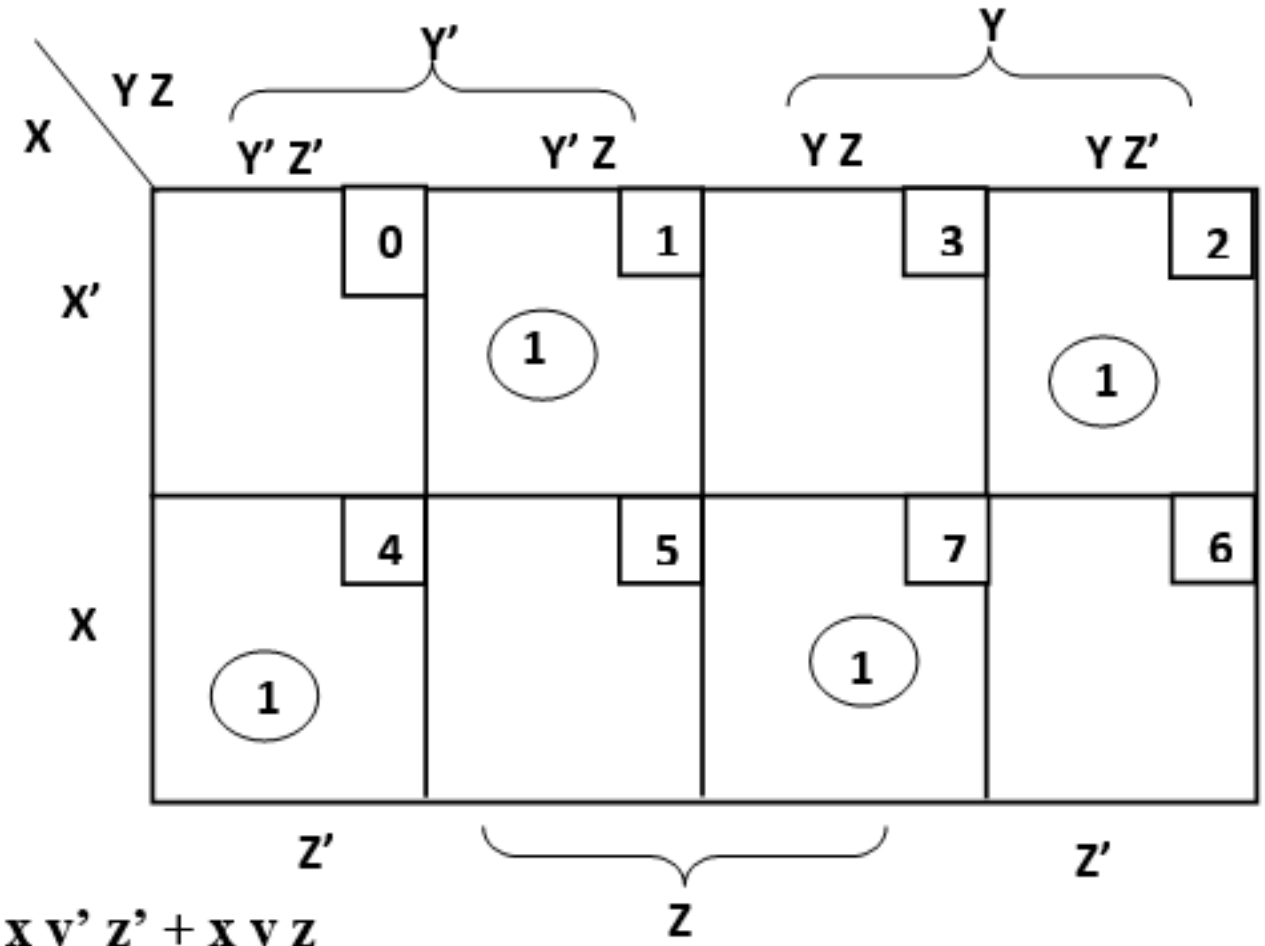


Now we formulate a Truth table to identify the function of Full-adder.

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

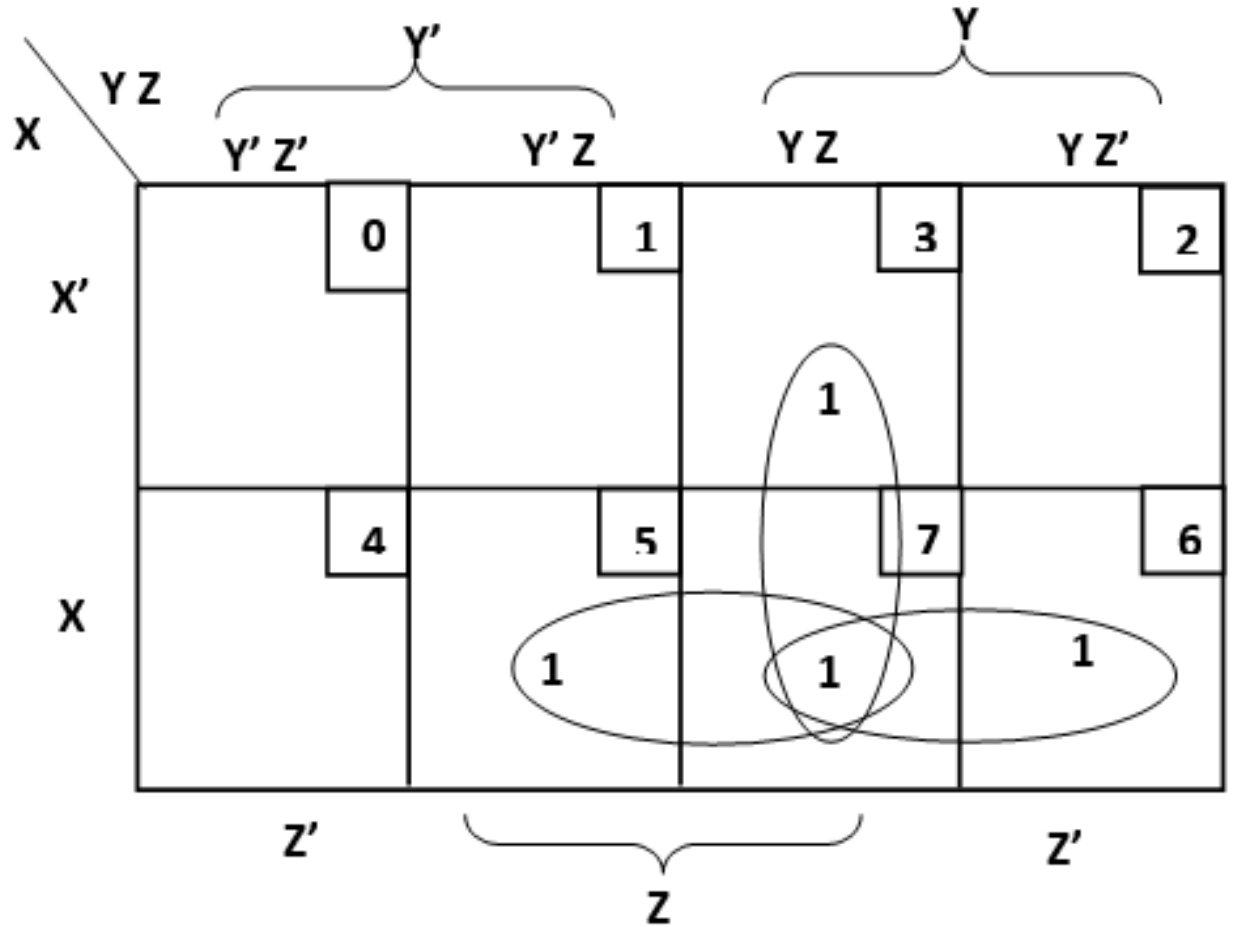
K-Map for simplified expression in SOP for full- adder:

For Sum:



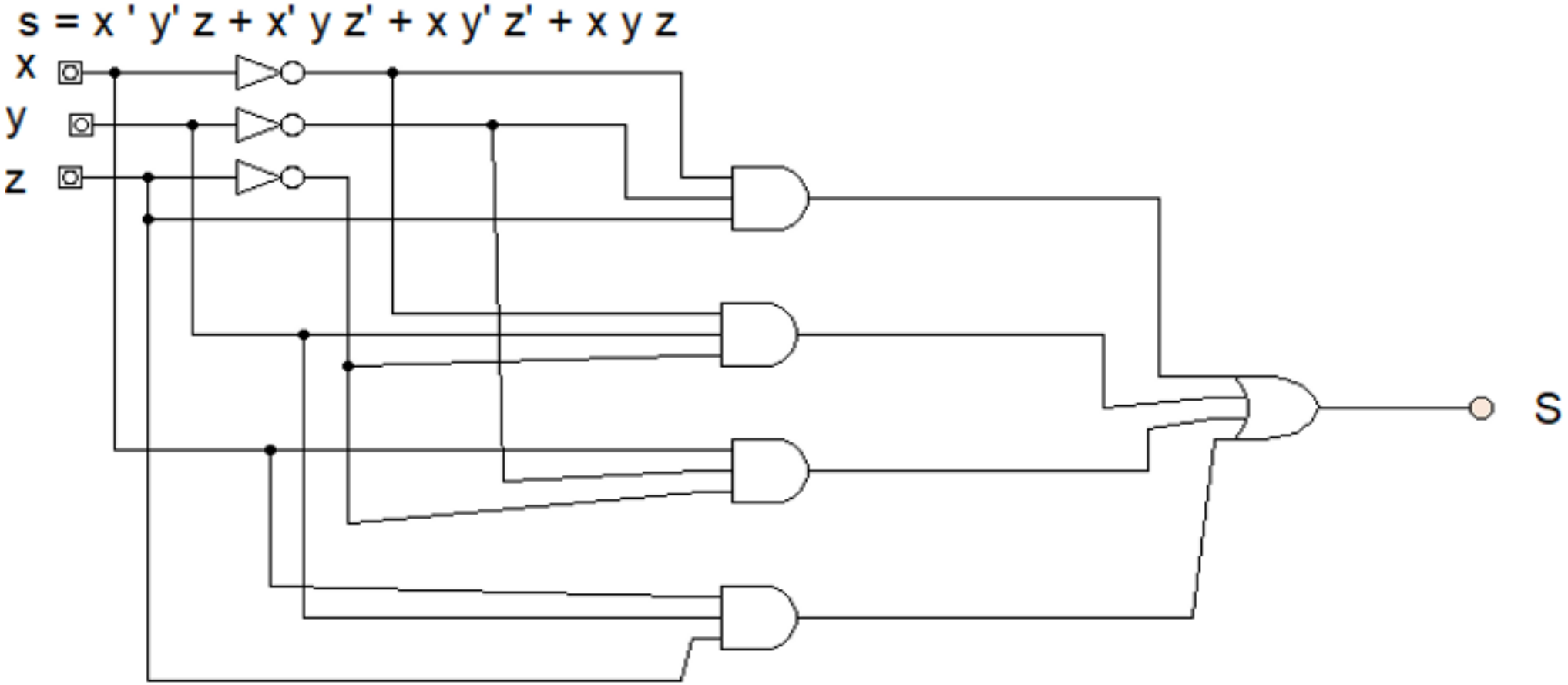
$S = x' y' z + x' y z' + x y' z' + x y z$

For Carry:



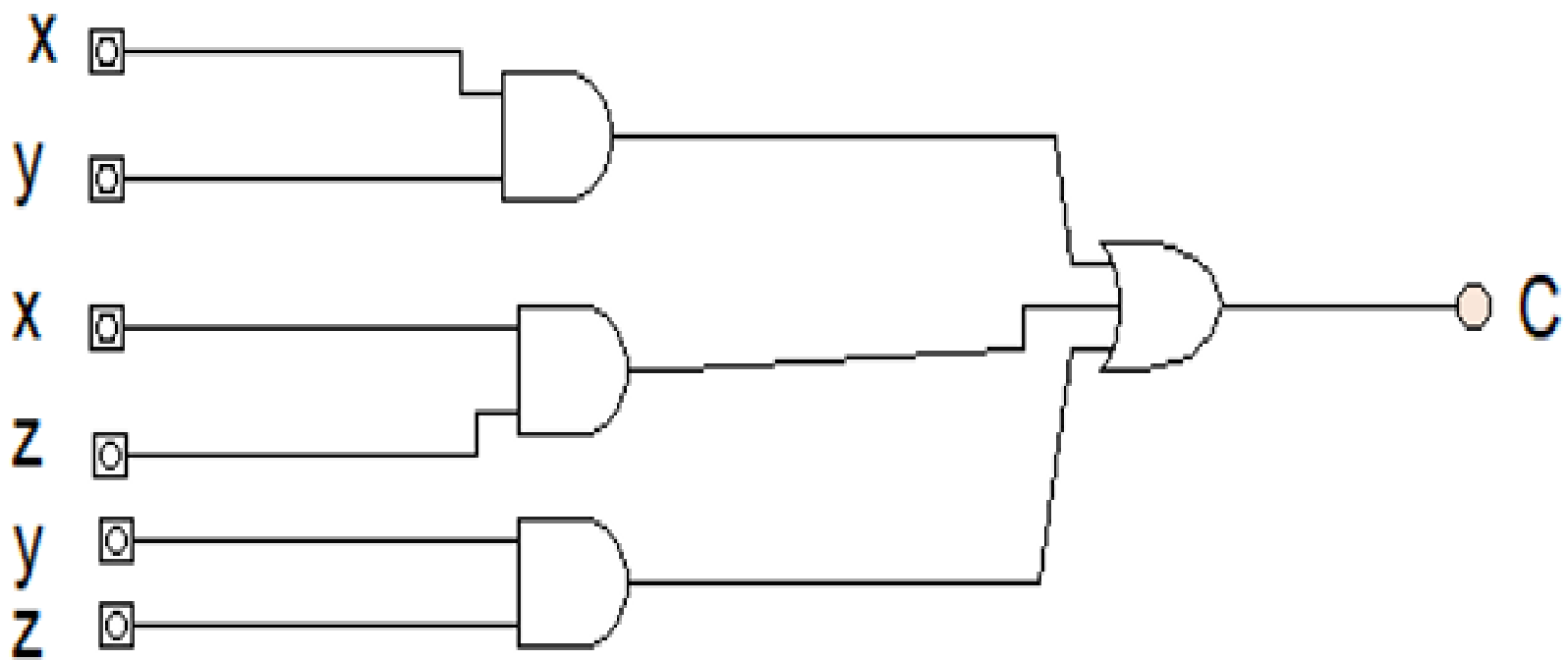
$$C = x\,z + y\,z + x\,y$$

Logic Diagram implementation for Full-Adder:



$$C = x\,y + x\,z + y\,z$$

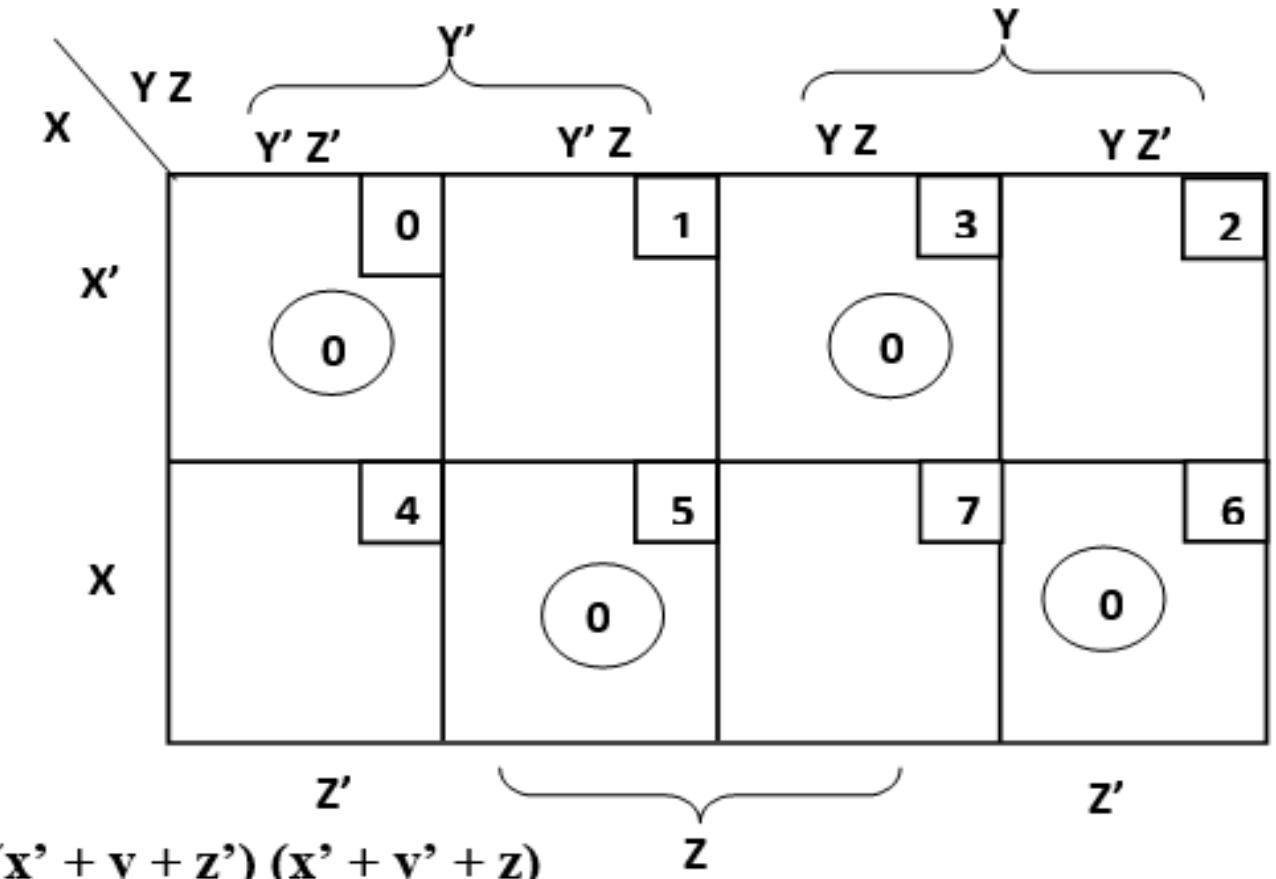
$$C = x\,y + x\,z + y\,z$$



Implementation of Full-adder in product of sums form:

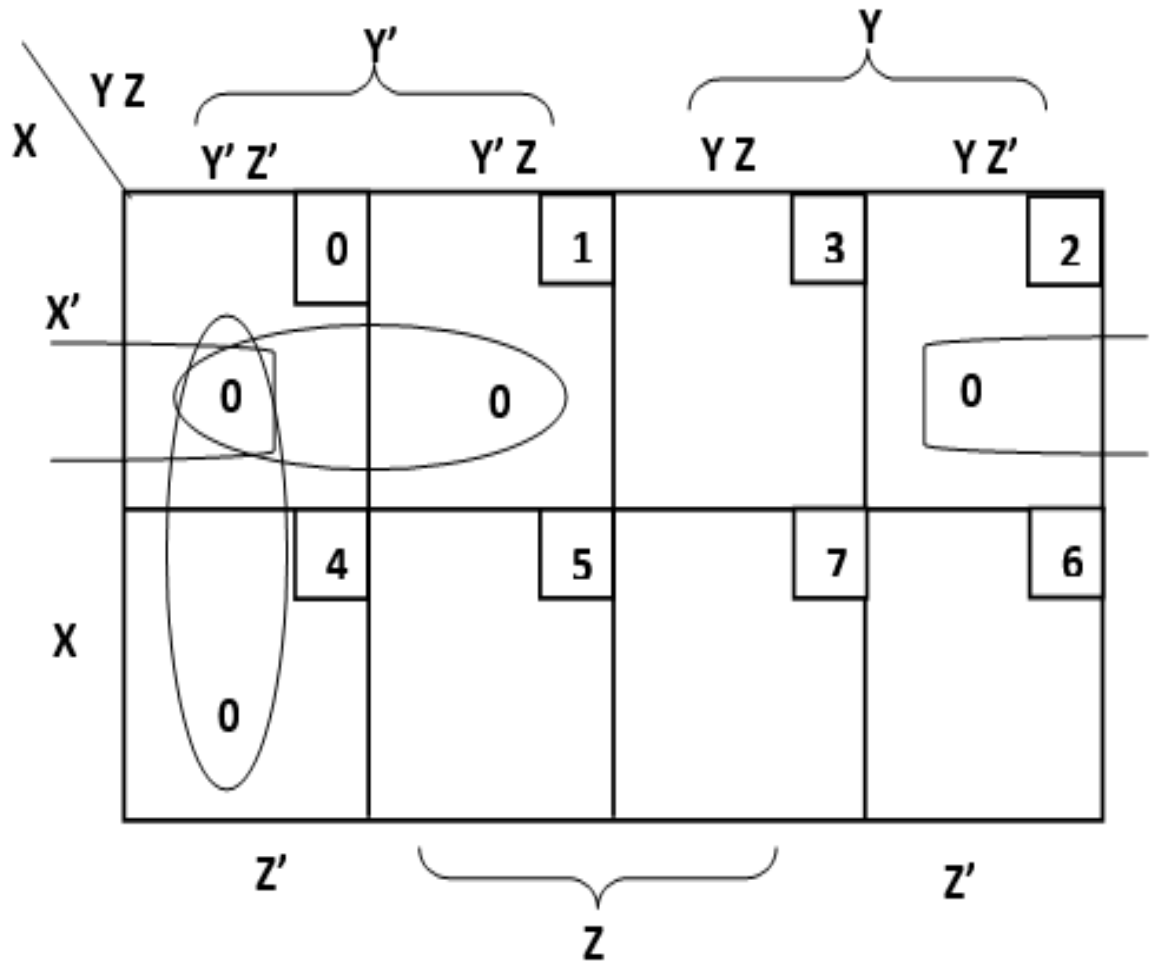
K-Map for simplified expression in POS for full adder:

For Sum:



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

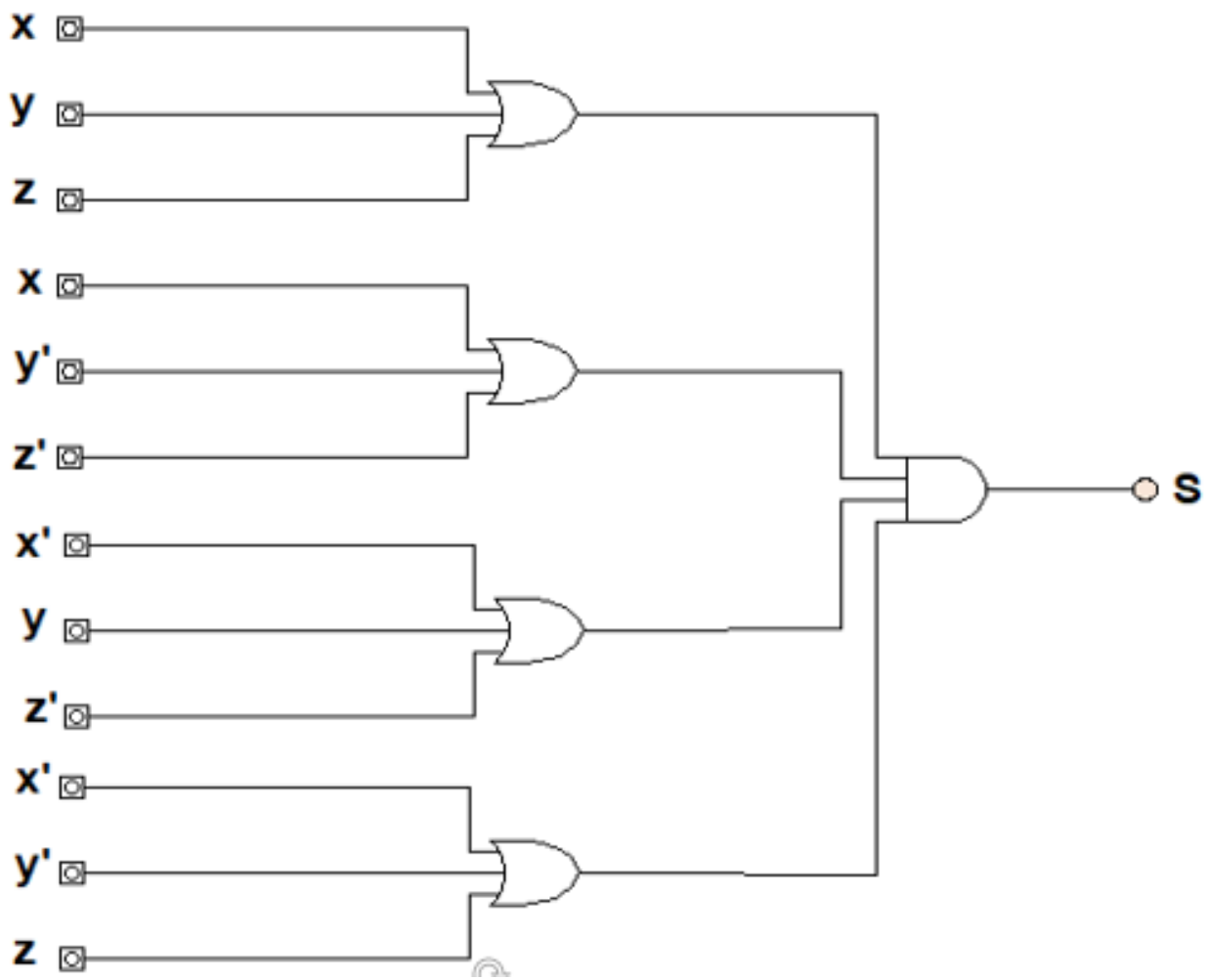
For Carry:



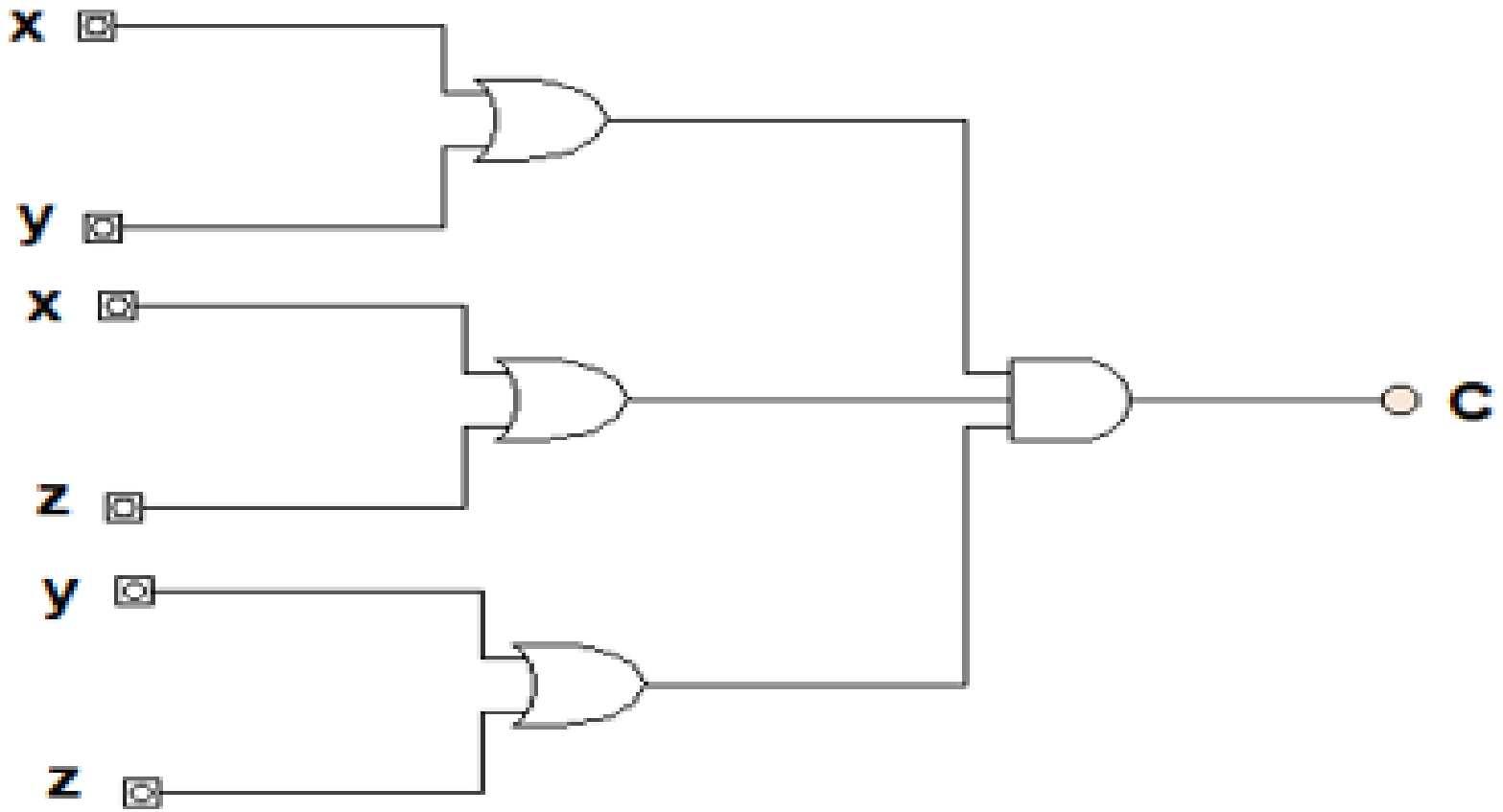
$C = (x + y)(x + z)(y + z)$

Implementation of full adder in POS with gates:

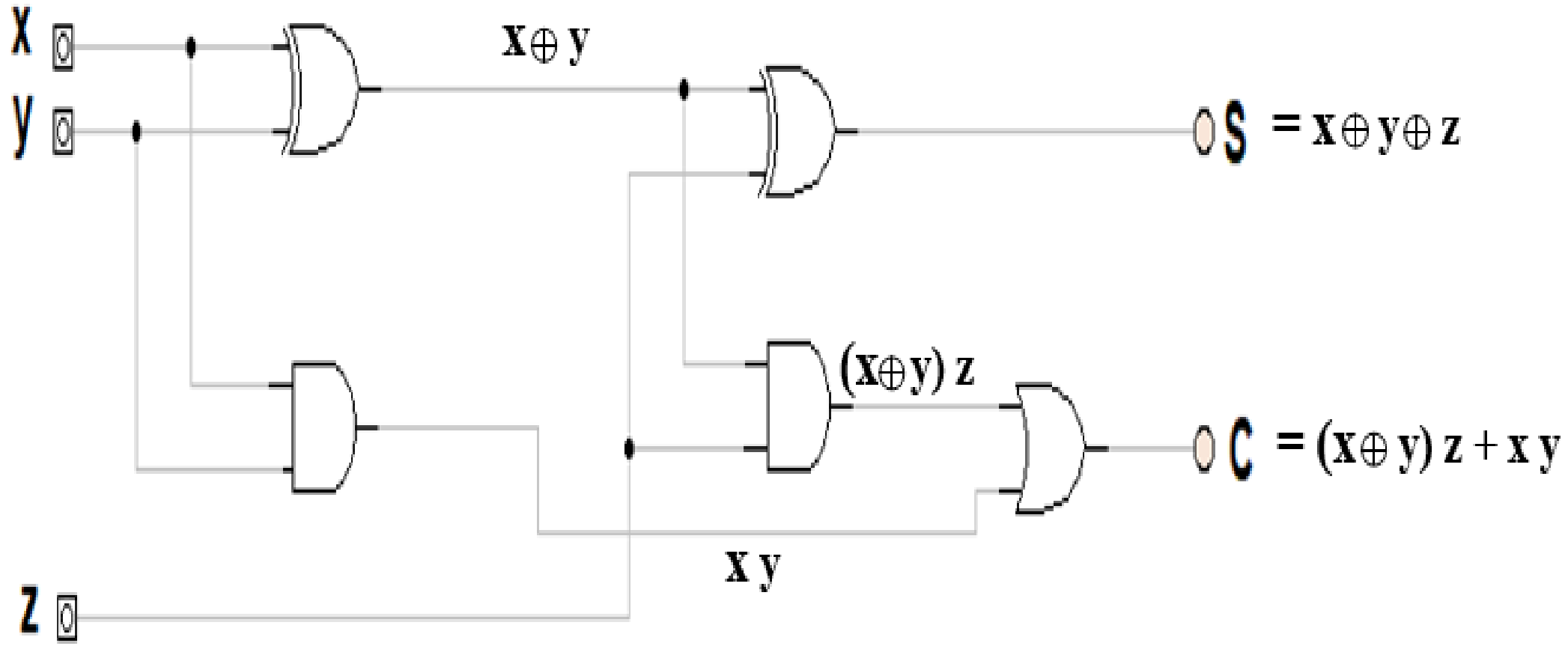
S = (x + y + z) (x + y' + z') (x' + y + z') (x' + y' + z)



$$C = (x + y)\,(x + z)\,(y + z)$$



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



$S = x \oplus y \oplus z$

$= (x' y + x y') \oplus z$

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

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

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

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

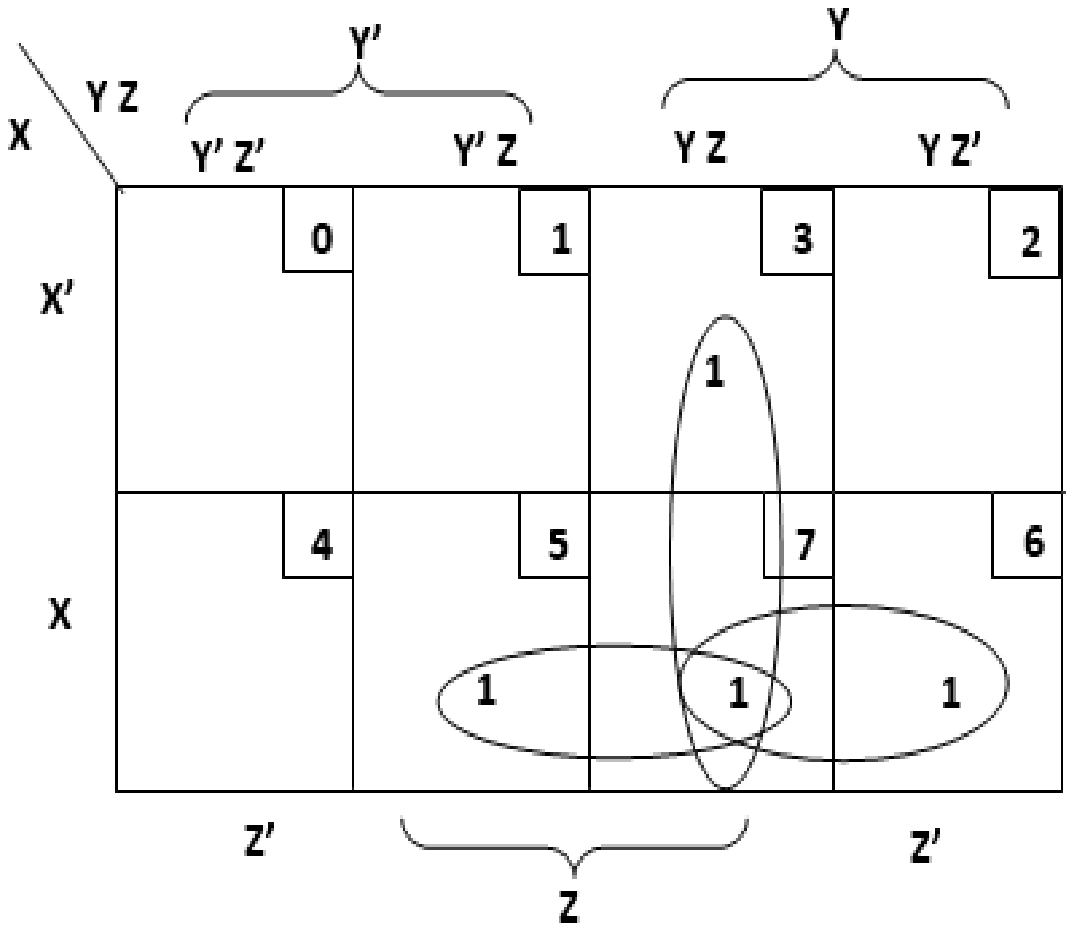
$= x y z + x' y' z + x' y z' + x y' z'$ **this expression cannot be simplified further**

$C = (x \oplus y) z + x y$

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

$= x' y z + x y' z + x y$

K – Map for carry



C = x z + x y + y z

Subtractor: Subtractor is the combinational circuit, which is used to subtract two or more than two binary digits at a time.

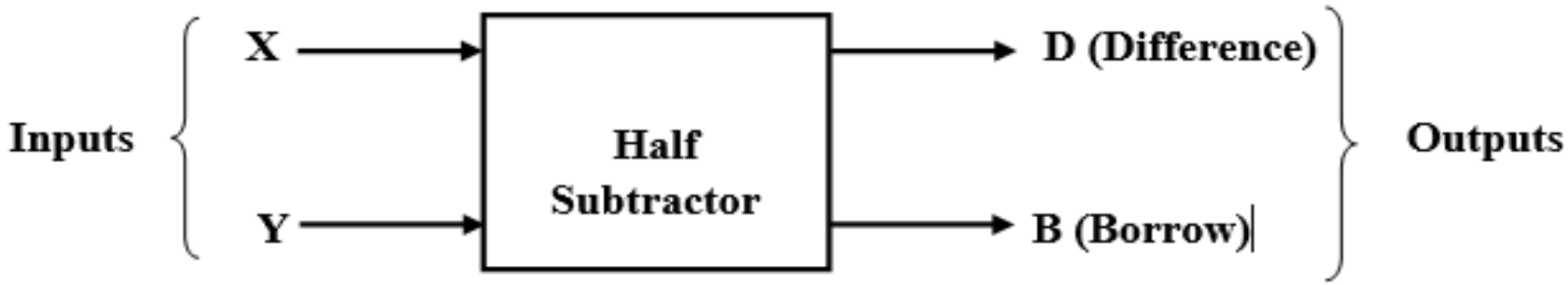
Types of subtractors:

1) **Half subtractor**
2) **Full subtractor**

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).

Block diagram of Half-Subtractor:

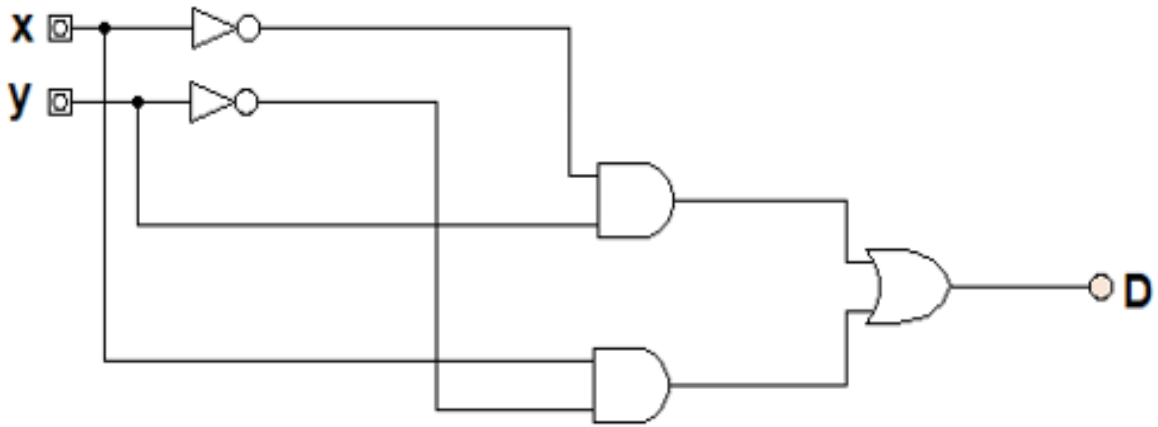


Truth table to identify the function of half-subtractor:

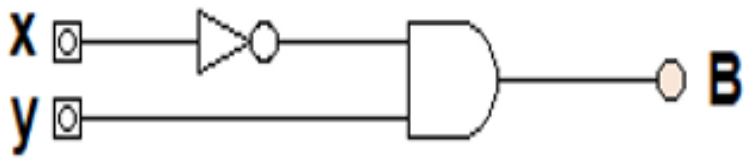
x	y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

Logic Diagram of Half Subtractor:

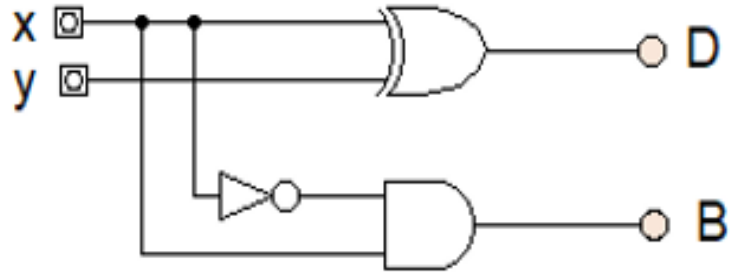
$D = x'y + xy'$



$B = x'y$



or

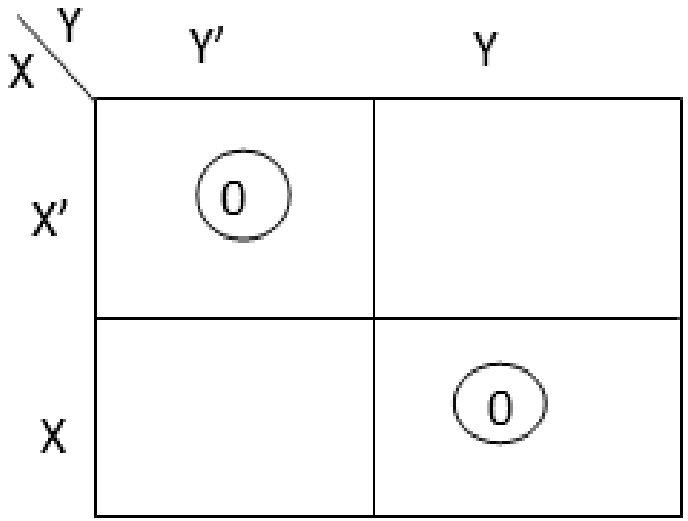


$D = x \oplus y$

$B = x' y$

Difference and borrow of Half- Subtractor can be implemented in POS form:

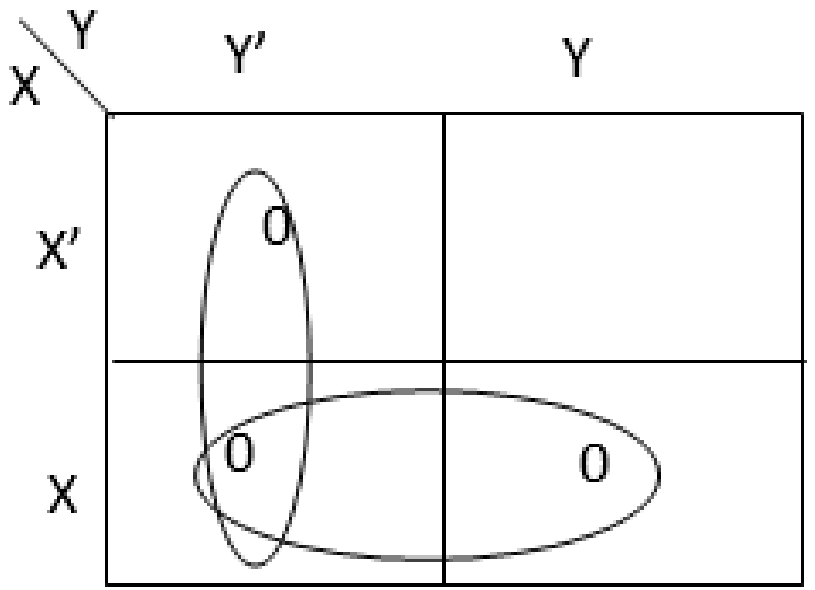
K-map for simplified expression for Difference in POS form:



Difference (D) = (x + y) (x' + y')

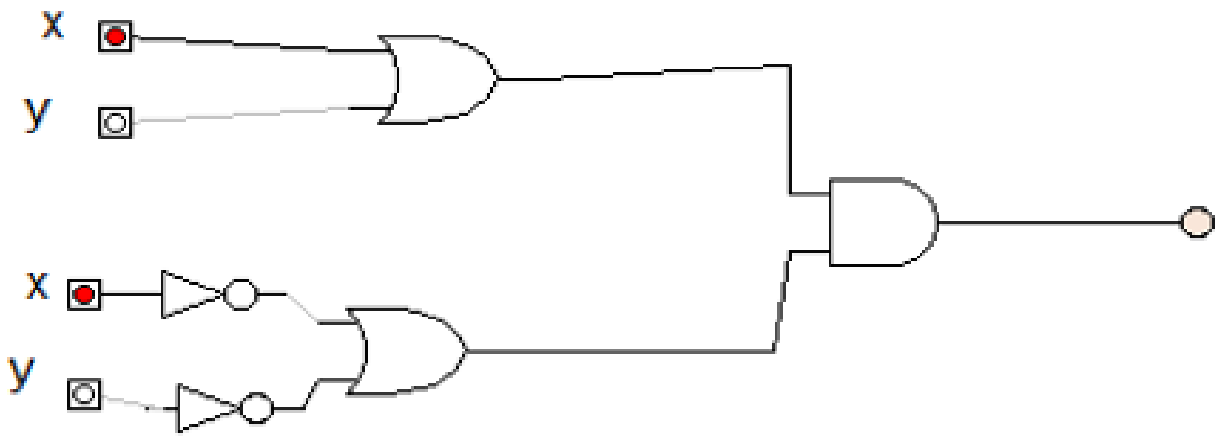
K-map for simplified expression for Borrow in POS for Sum:

B = (x + y')'



Logic diagram implementation of Half-subtractor in POS form:

D = (x + y) (x' + y')



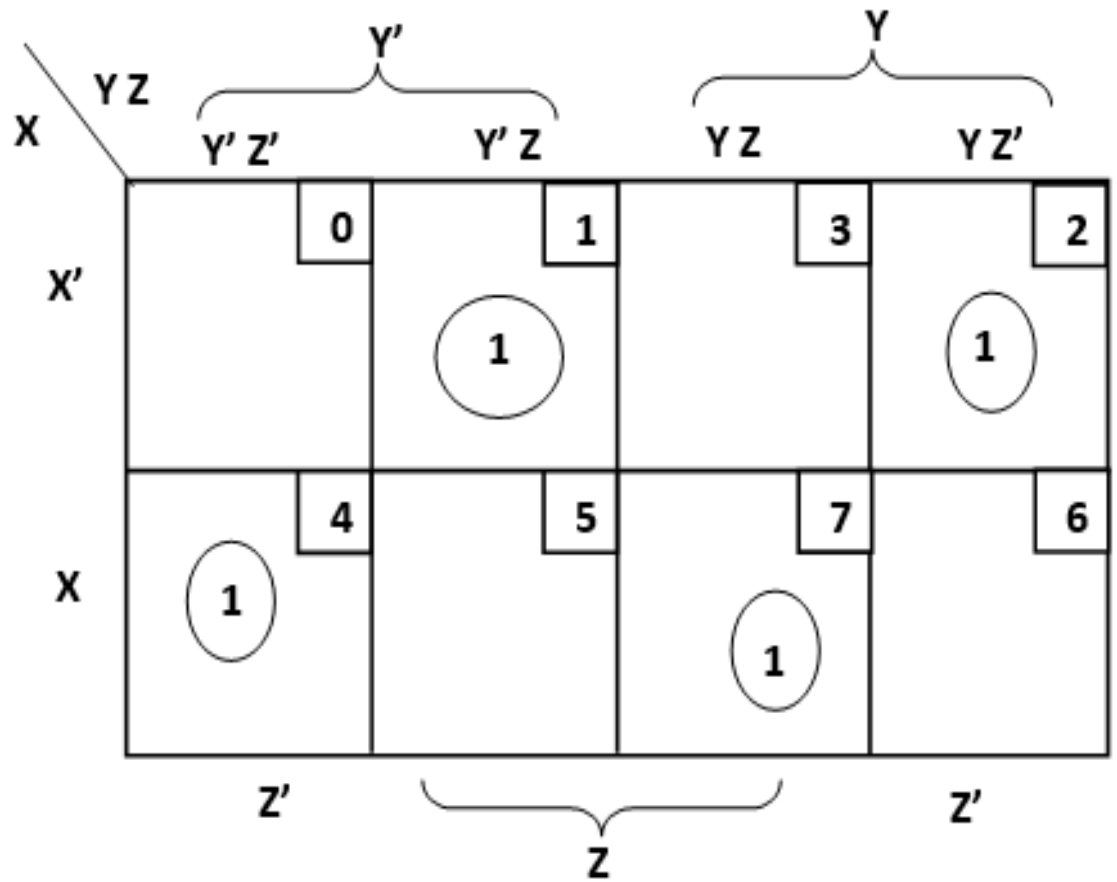
B = (x + y')'

Full Subtractor: Used to subtract three binary digits at a time. Inputs x, y and z, outputs Difference (D) and Borrow (B).

Truth table:

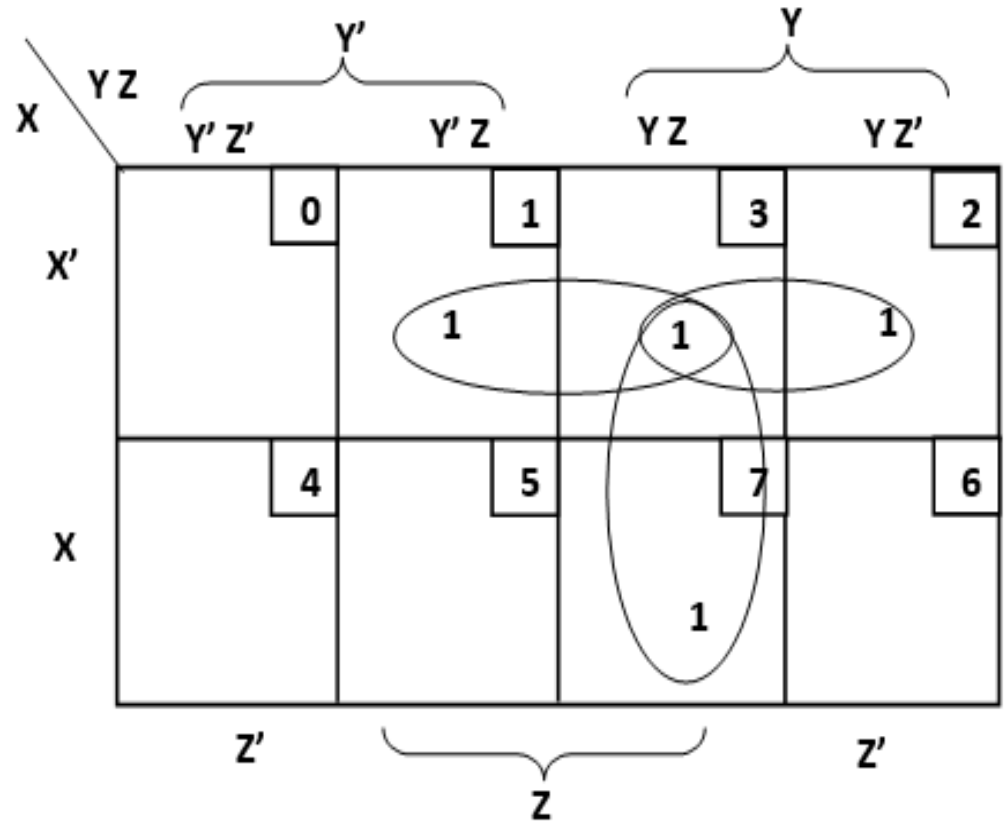
X Y Z	D B
0 0 0	0 0
0 0 1	1 1
0 1 0	1 1
0 1 1	0 1
1 0 0	1 0
1 0 1	0 0
1 1 0	0 0
1 1 1	1 1

K-map for simplified expression in SOP for Difference (D):



D = x' y' z + x' y z' + x y' z' + x y z

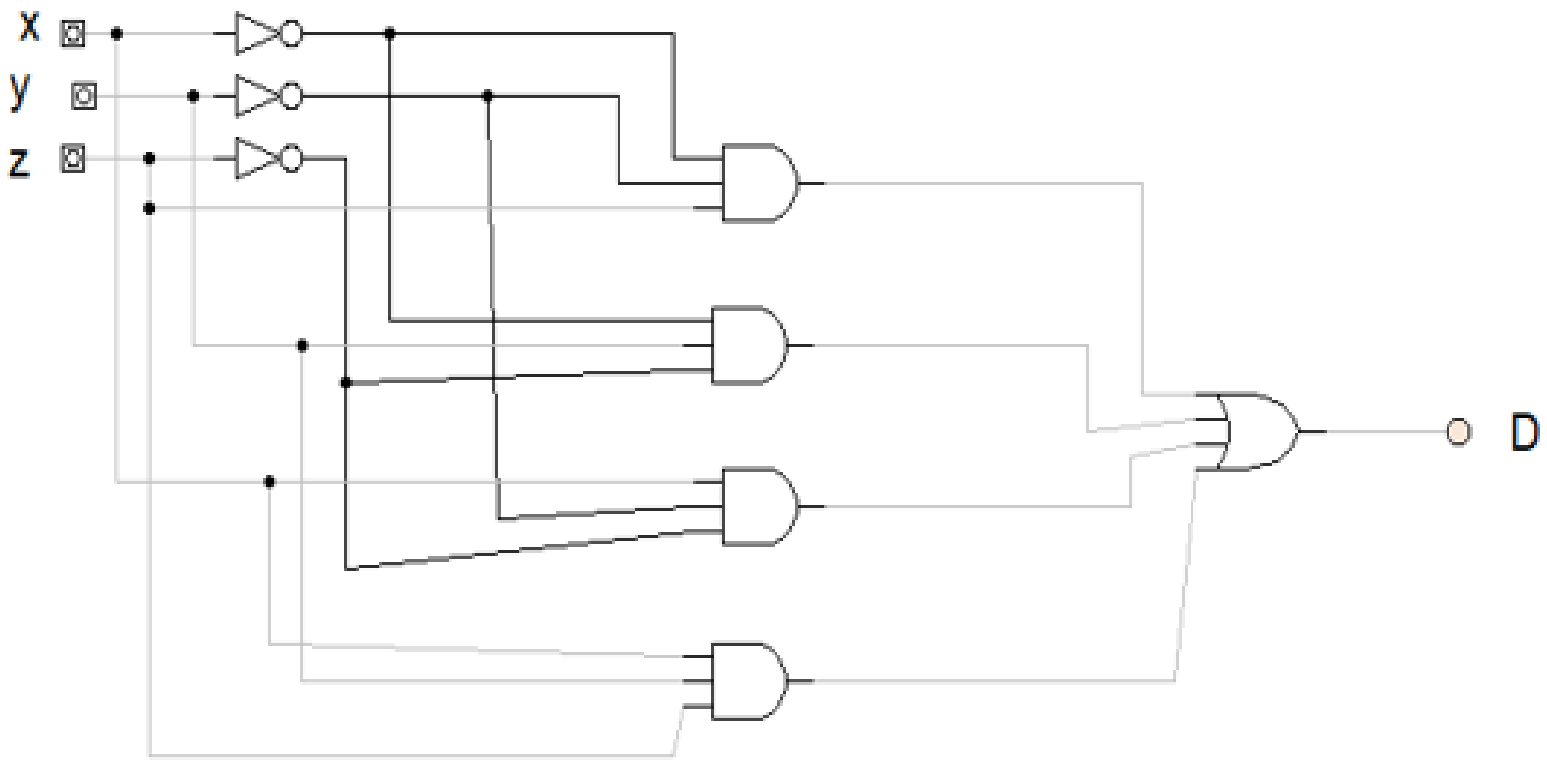
K-map for simplified expression in SOP for Borrow (B):



B= x' z + x' y + y z

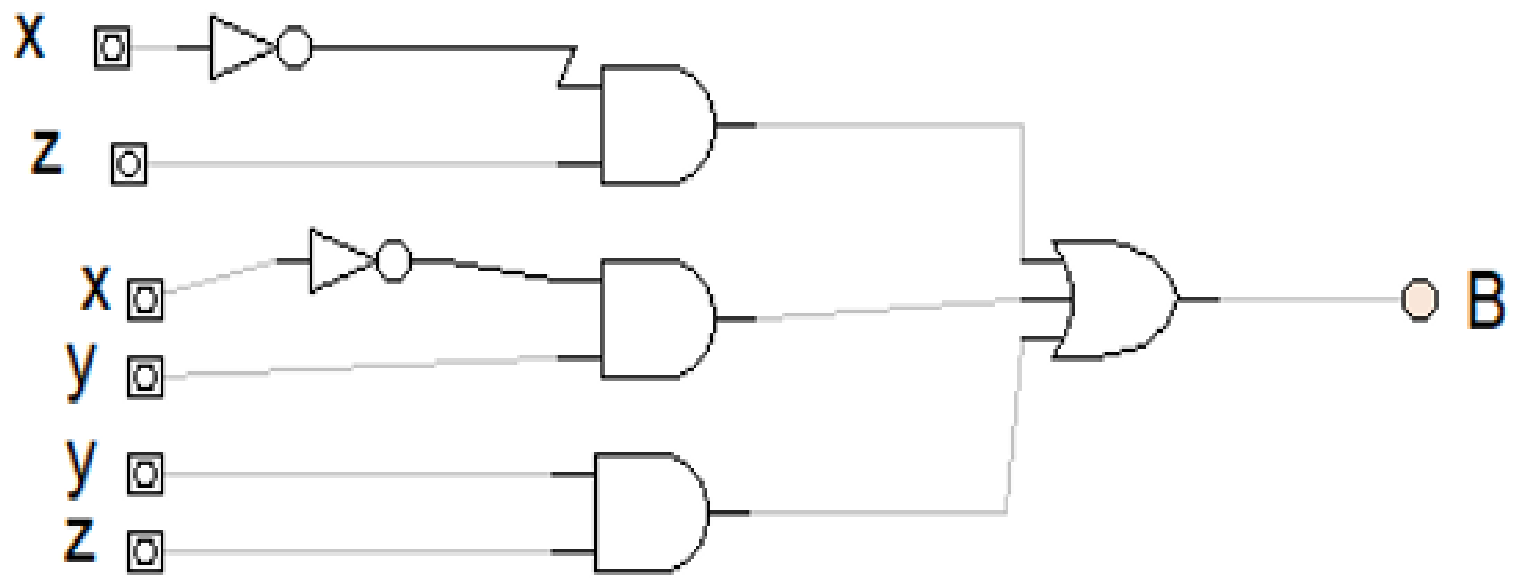
Logic diagram implementation of Full Subtractor in SOP form:

D = x' y' z + x' y z' + x y' z' + x y z



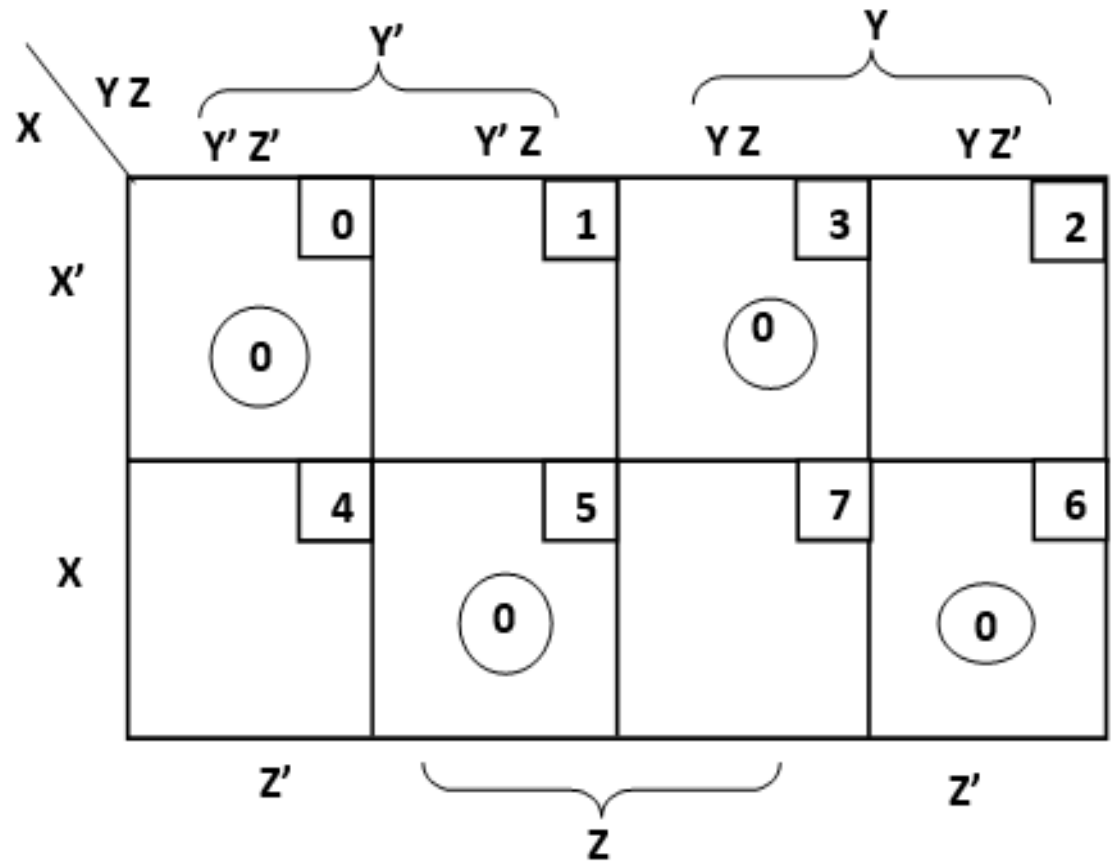
Logic diagram implementation of Full Subtractor in SOP form:

$$B = x' z + x' y + yz$$



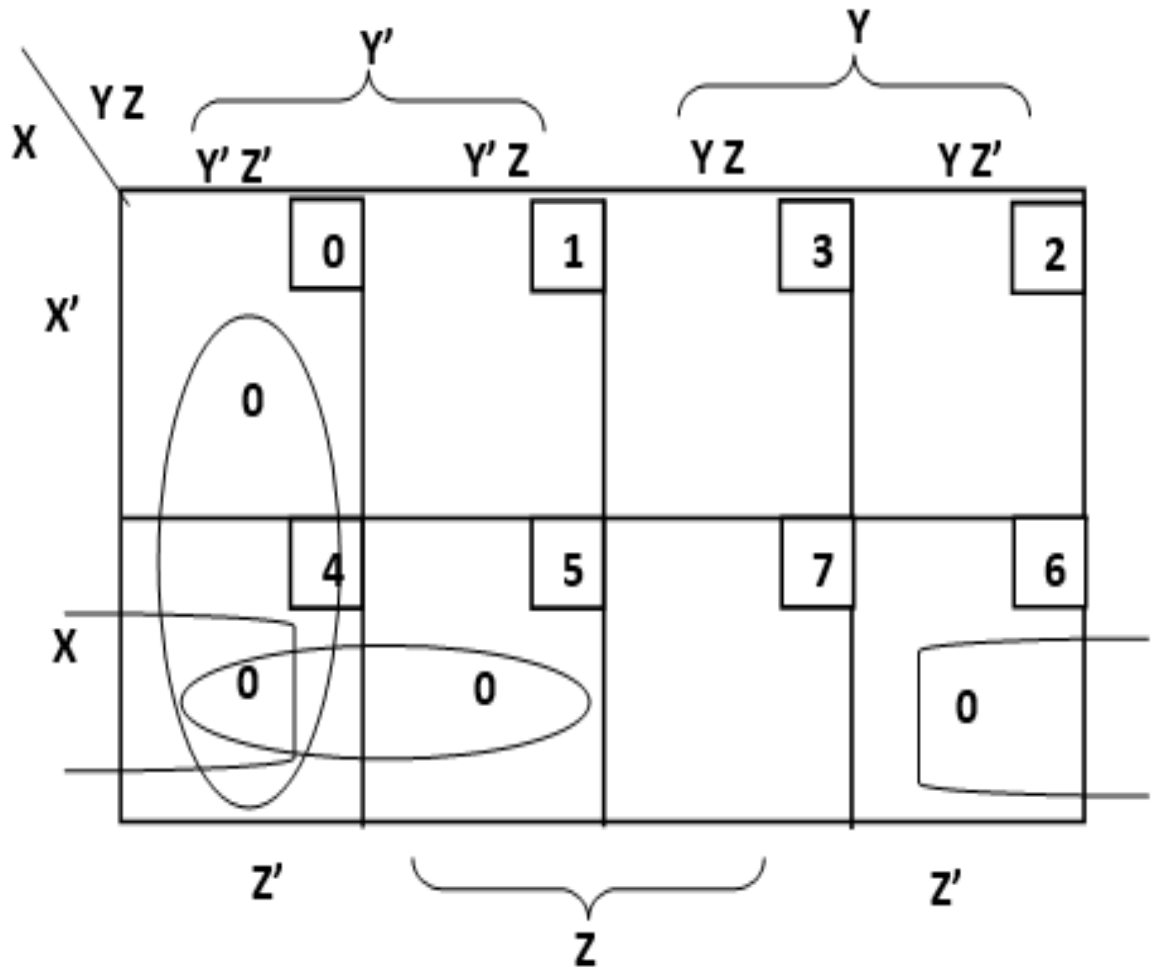
POS implementation of Full-Subtractor:

K-Map for D:



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

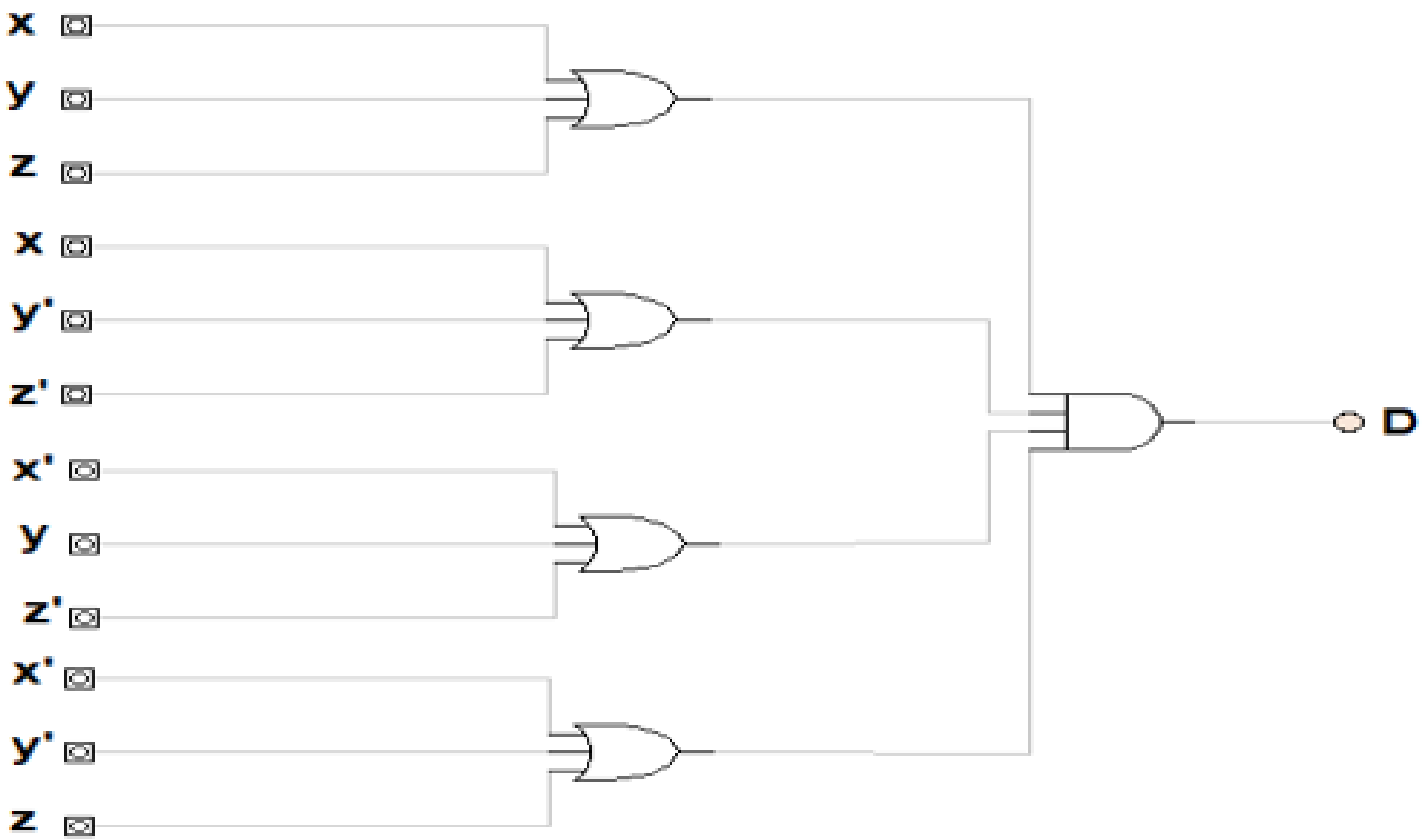
K-Map for Borrow(B)



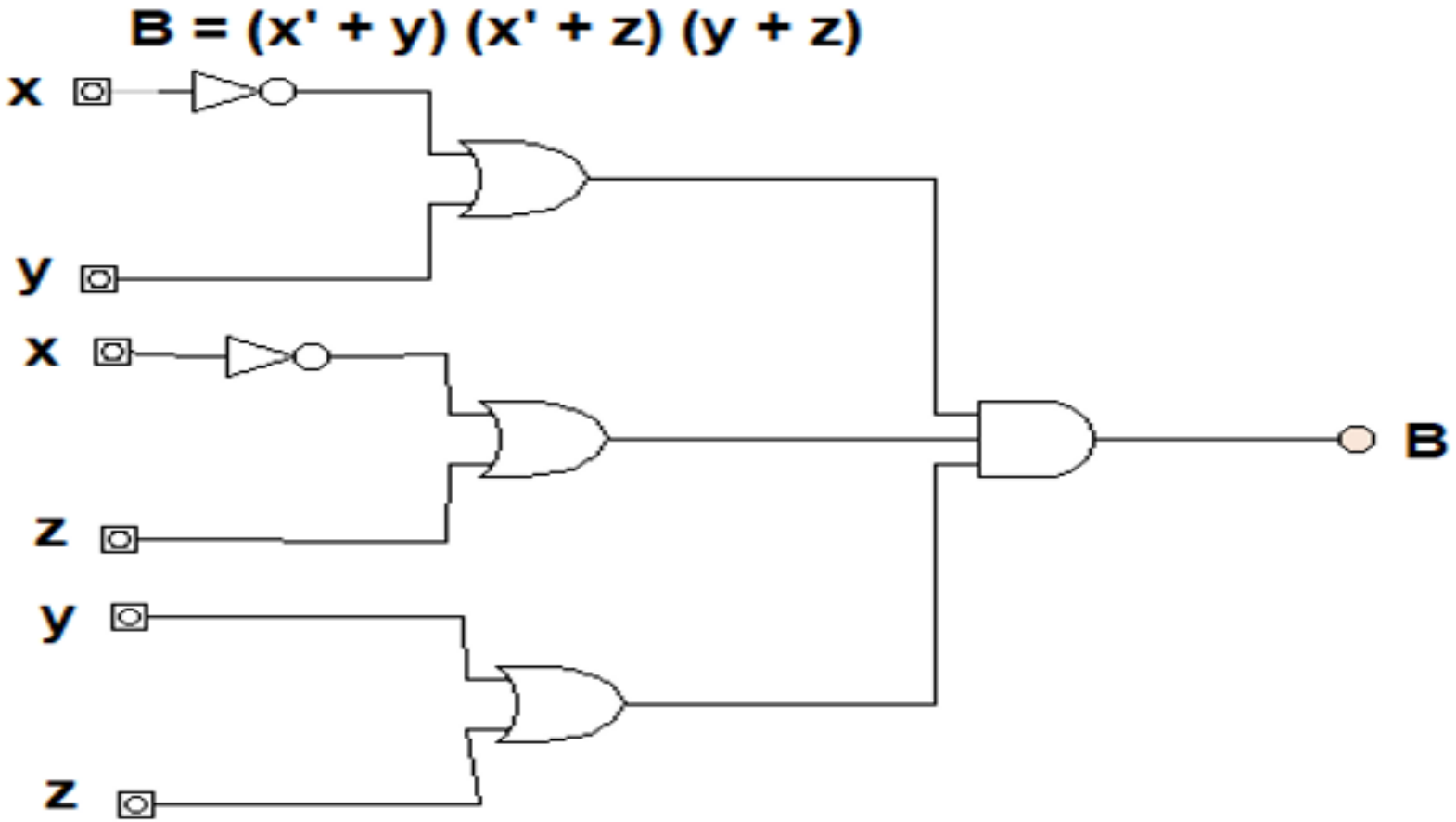
B = (y + z) (x' + y) (x' + z)

Logic diagram implementation of Full- Subtractor in POS form:

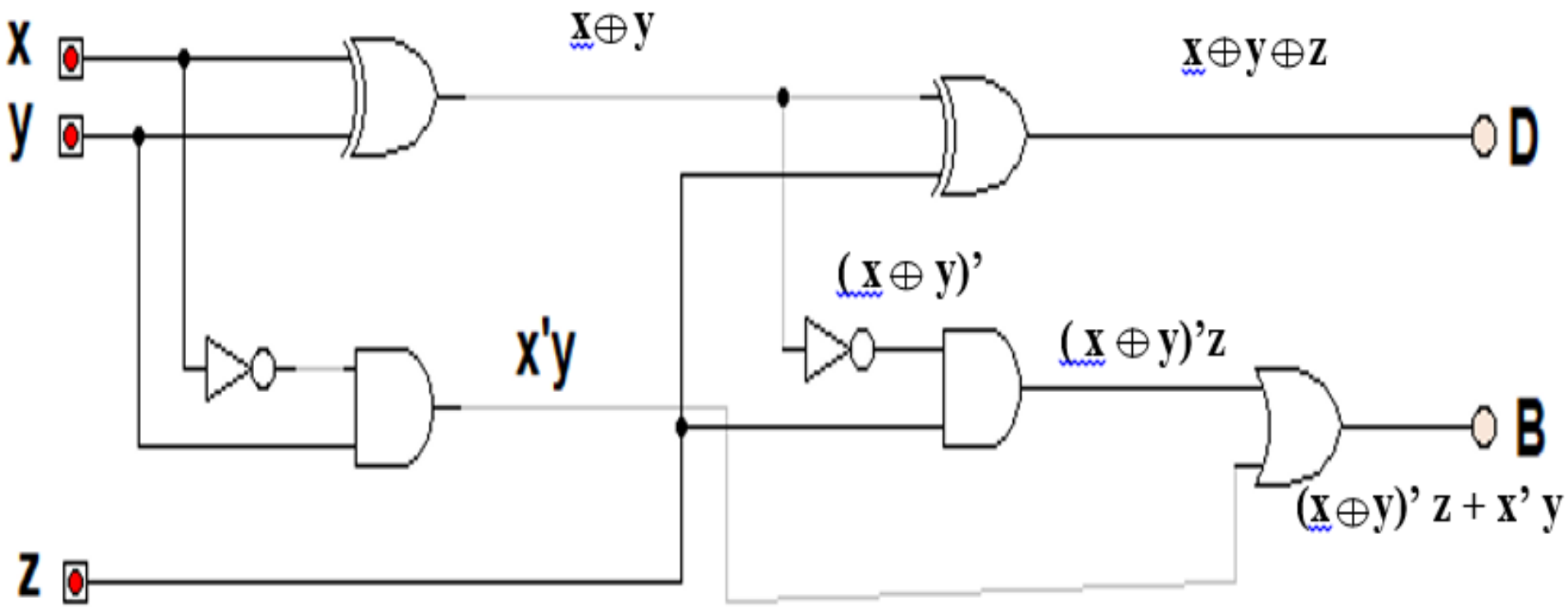
$$D = (x + y + z)\,(x + y' + z')\,(x' + y + z')\,(x' + y' + z)$$



Logic diagram implementation of Full- Subtractor in POS form:



Implementation of Full-Subtractor using two half-subtractor and one OR gate:



$D = S \oplus x \oplus y \oplus z$

$= (x' y + x y') \oplus z$

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

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

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

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

$= x y z + x' y' z + x' y z' + x y' z'$ **This expression cannot be simplified further.**

$B = (x \oplus y)' z + x' y$

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

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

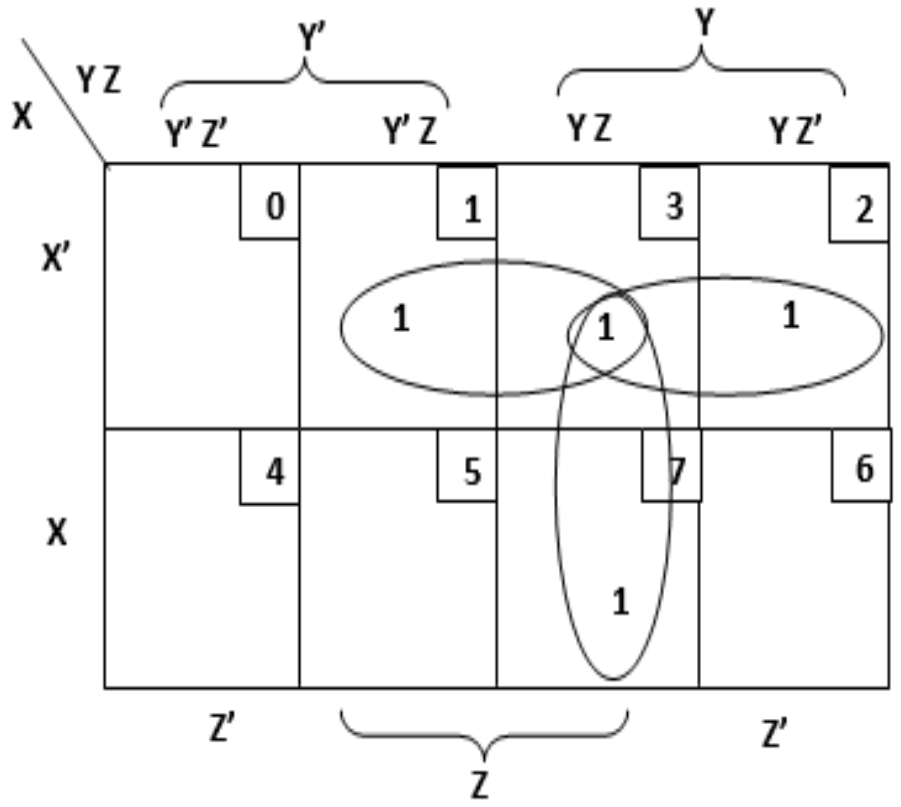
$=\{ (x + y') (x' + y)\} z + x' y$

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

$= x y z + x' y' z + x' y$

$$B = x\,y\,z + x'\,y'\,z + x'\,y$$

K-Map for B:



$$B = x'\,z + x'\,y + y\,z$$

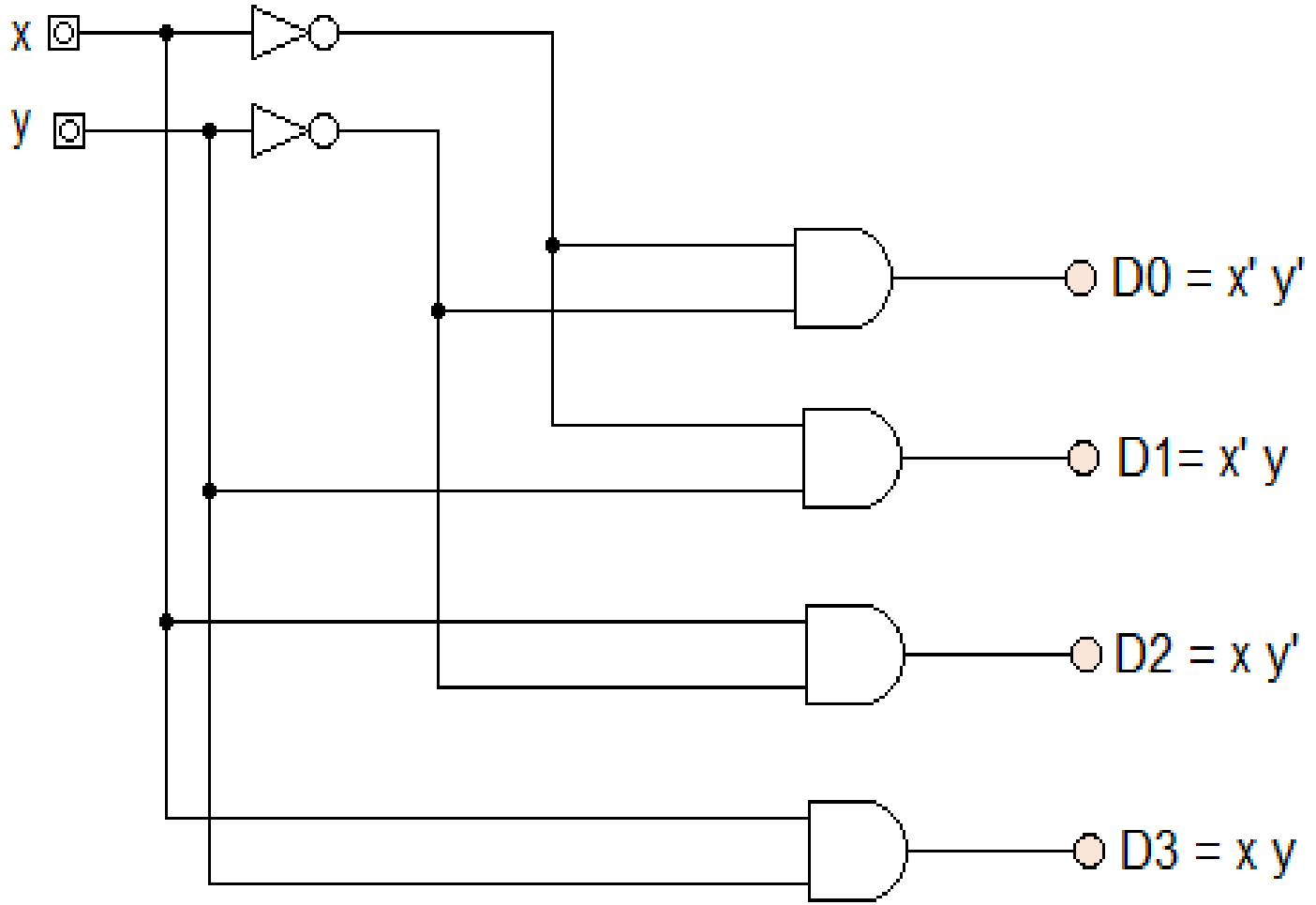
Decoder: Decoder is a combinational circuit that converts binary information from n input lines to maximum of 2^n unique output lines. If the n-bit decoded information has unused or don't care combinations, the decoder output will have less than 2^n outputs. There should be only one output line of decoder high at a time.

2 – to – 4 line decoder: It consists of two inputs and four output lines.

Truth Table:

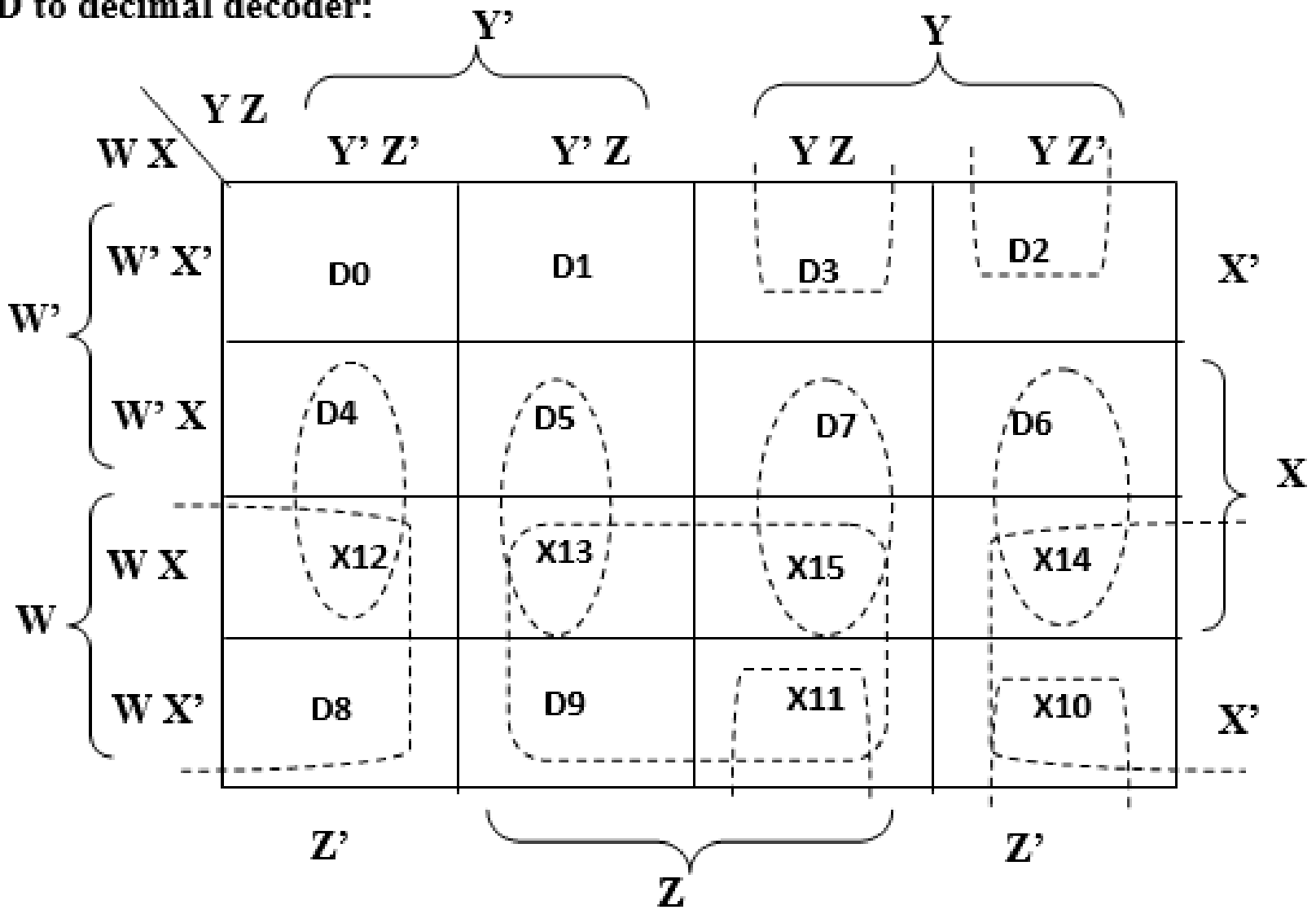
Inputs	Outputs
X Y	D_0 D_1 D_2 D_3
0 0	1 0 0 0
0 1	0 1 0 0
1 0	0 0 1 0
1 1	0 0 0 1

Logic diagram of 2 – to – 4 line decoder:



BCD – to Decimal Decoder: It consists of 4 inputs and 10 output lines: The decoder which converts binary coded decimal code (BCD) into decimal values is called BCD to decimal decoder. The BCD code uses four bits and therefore there should be 2^4 input combinations. This produces 16 different output signals, but the decimal digits are from 0 to 9 (equal to 10 different combinations). Hence other 6 input combinations are not used in decimal system. Therefore we can use don't cares to represent 10 to 15 and use K-Map to simplify the circuit of BCD to decimal decoder.

K-Map for BCD to decimal decoder:



Simplified expressions for different outputs:

D0 = w' x' y' z'

D1 = w' x' y' z

D2 = x' y z'

D3 = x' y z

D4 = x y' z'

D5 = x y' z

D6 = x y z'

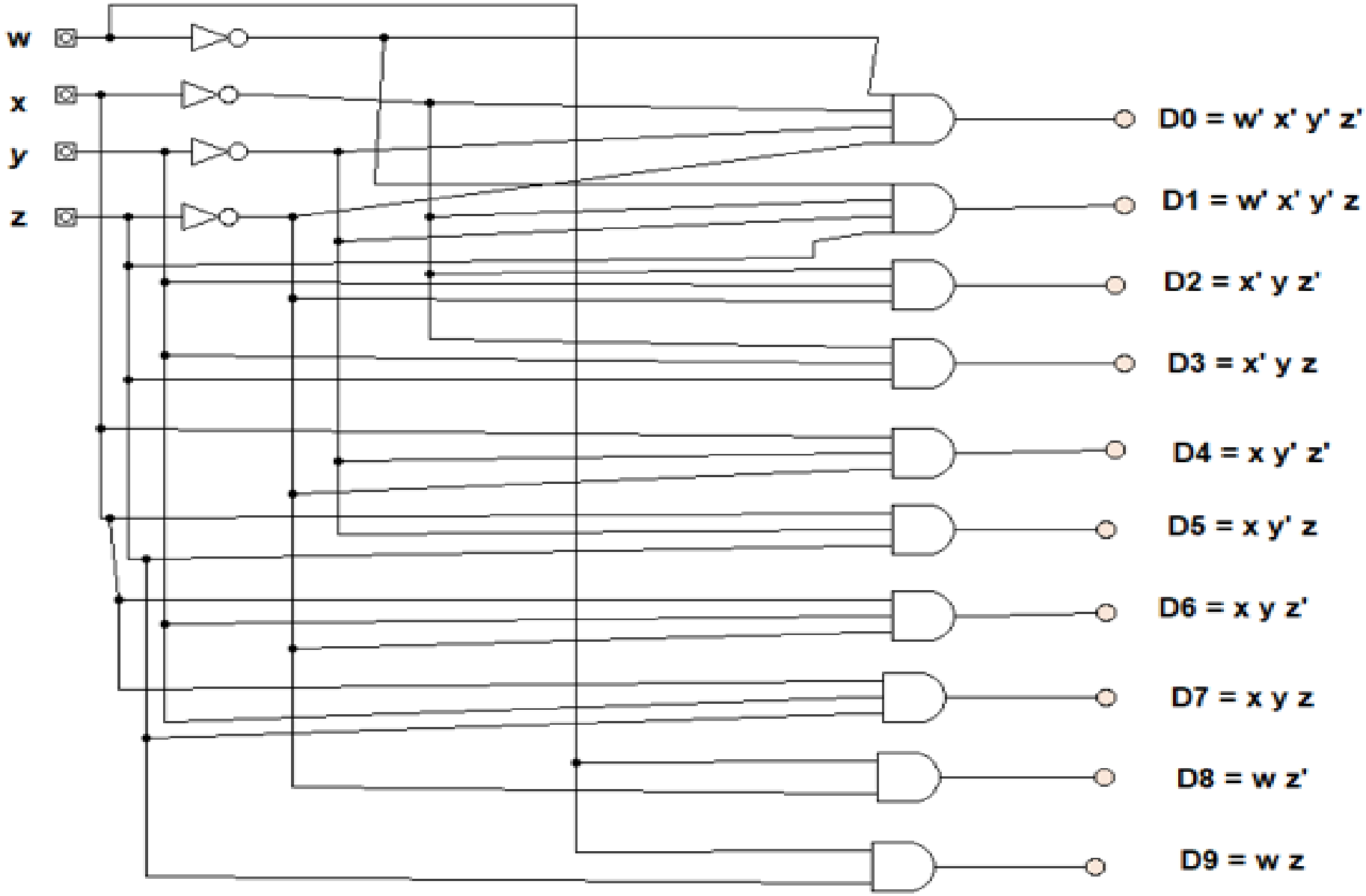
D7 = x y z

D8 = w z'

D9 = w z

- In the above K-Map D0 and D1 cannot be combined with any don't cares.
- D2 can be combined with don't care X10
- D3 with don't care X11
- D4 with don't care X12
- D5 with don't care X13
- D6 with don't care X14
- D7 with don't care X15
- D8 with don't cares X10, X12 and X14
- D9 with don't care X11, X13 and 15

Logic diagram of BCD to Decimal Decoder:



Truth Table of BCD to decimal decoder:

W	X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1

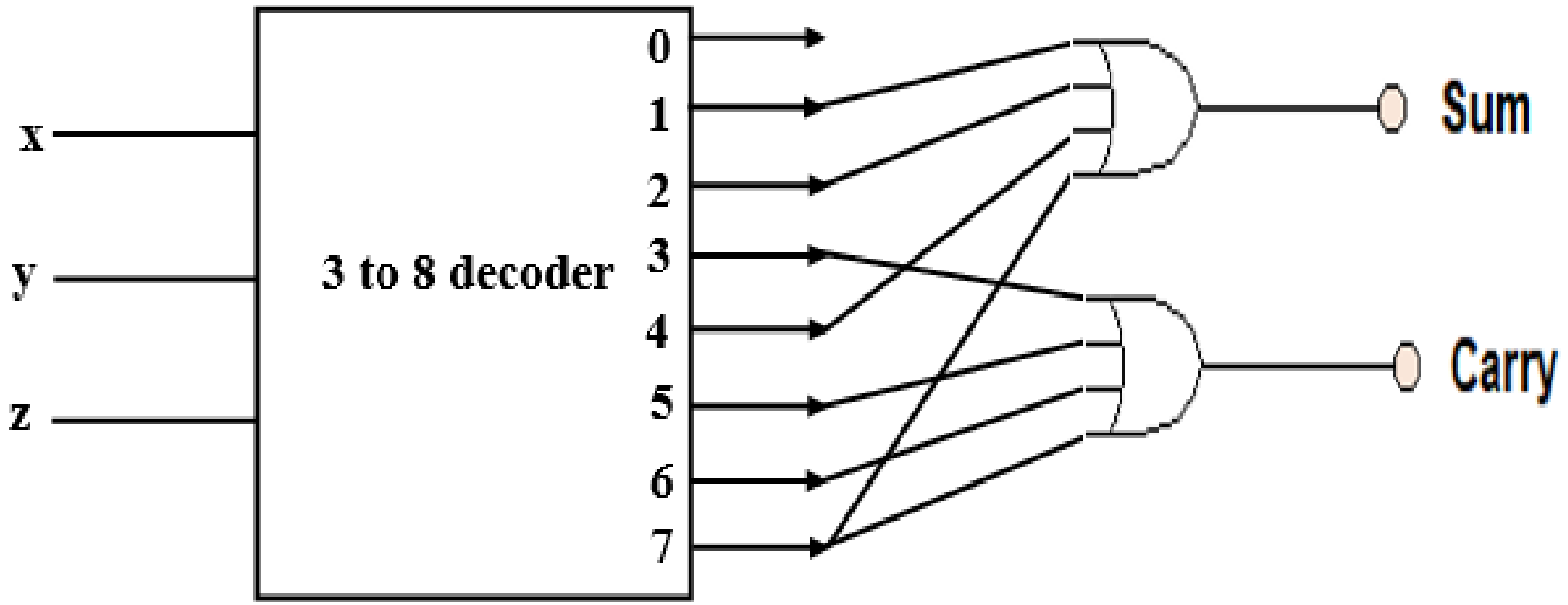
Implementation of Full-Adder circuit using 3-to-8 line decoder and two OR gates:

Truth Table of Full-Adder:

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

Implementation of Full-Adder circuit using 3-to-8 line decoder and two OR gates:

Logic Circuit:



Encoder: Encoder is a combinational logic circuit that produces a reverse operation from that of a decoder. It has 2^n input lines and n output lines. The output lines generate the binary code for the 2^n input variables.

Octal to binary encoder: It consists of eight inputs one for each of the eight digits and three outputs that generate the corresponding binary number. It is constructed with OR gates whose inputs can be determined from the truth table.

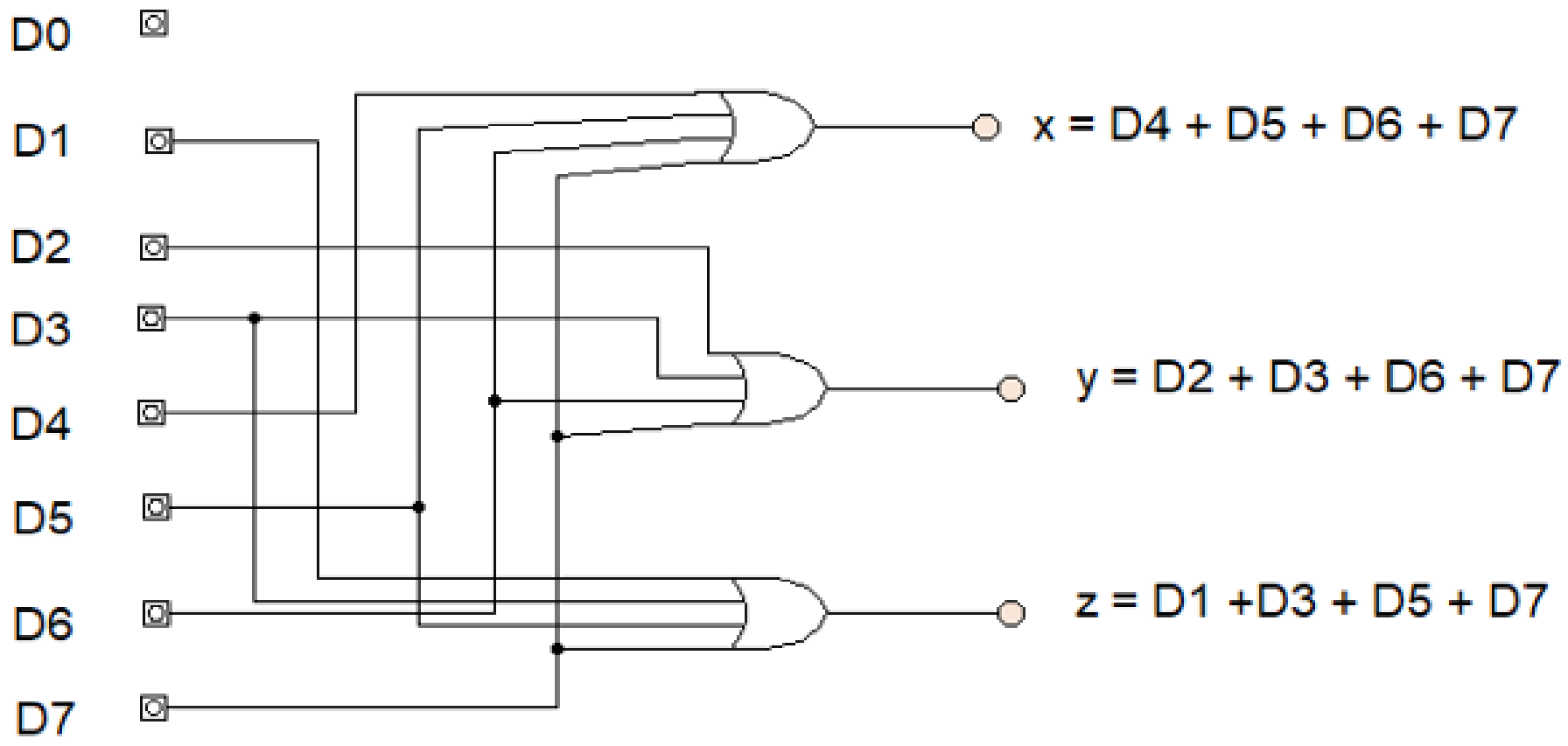
Truth Table for Octal to Binary Encoder:

Inputs								Outputs		
D0	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**x**	**y**	**z**
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

From the truth table we can find that the output x is high for the octal inputs 4, 5, 6, 7 and we can connect D4, D5, D6 and D7 to the first OR gate x. similarly, we can follow the same process for y and z.

$$x = D4 + D5 + D6 + D7$$
$$y = D2 + D3 + D6 + D7$$
$$z = D1 + D3 + D5 + D7$$

Logic diagram of Octal to Binary Encoder



Code Conversion: There are varieties of binary codes and different digital systems use different codes for the same discrete elements. Circuits are sometimes designed in such a way the output of one system becomes the input of another system. A code conversion circuit may be required in between two systems if each uses different codes for the same information. Therefore a combinational circuit that makes two different systems with different binary codes compatible with each other is called **code converter**.

To convert any binary code A into code B, the input lines must have bit combination of elements as available in A and produce corresponding bit combination of B. The code converter uses logic gates to make the conversion.

BCD to Excess-3 Code converter:

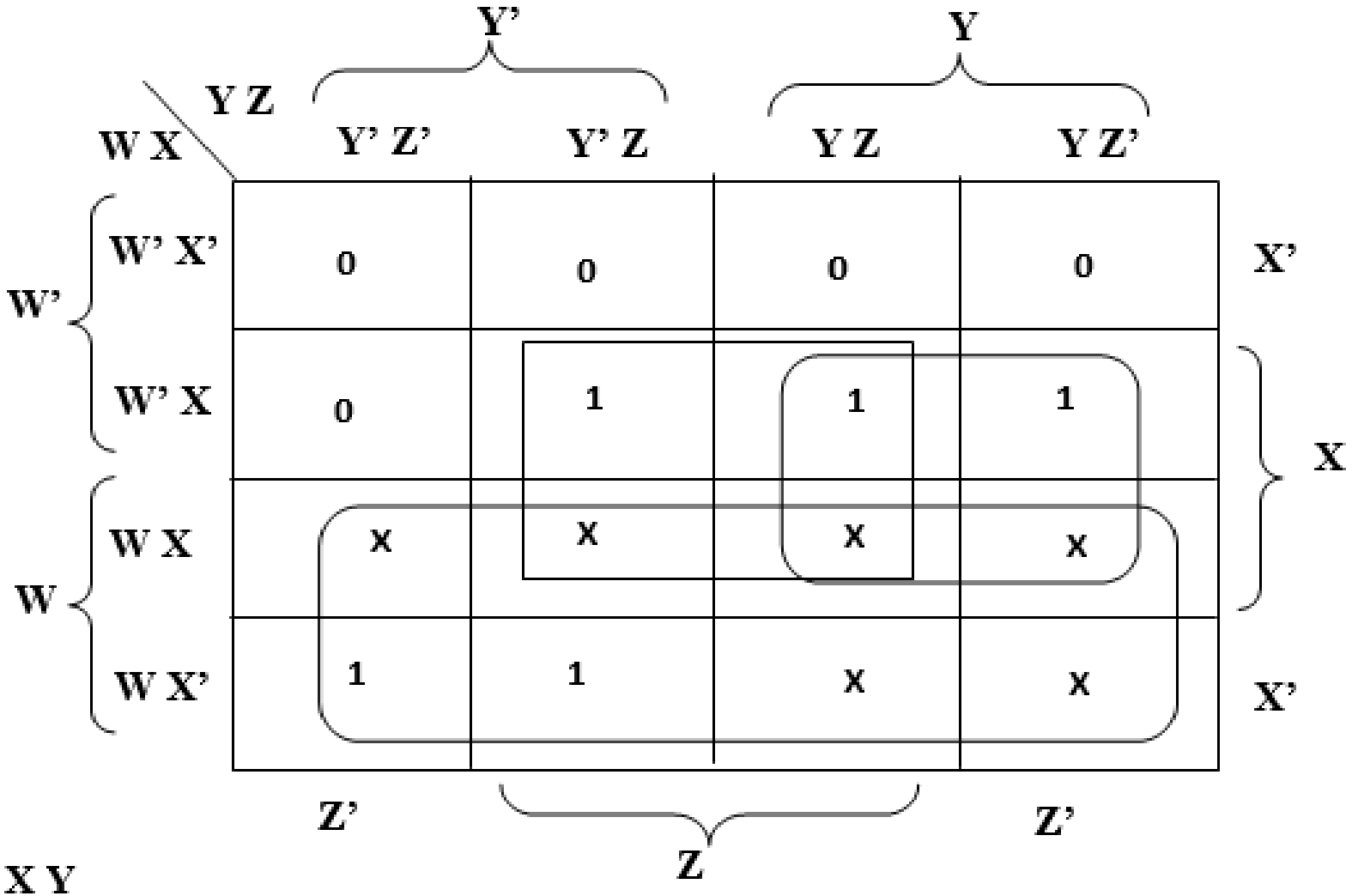
Truth table of BCD to Excess-3 Converter:

Inputs BCD				Outputs Excess-3 Code			
W	**X**	**Y**	**Z**	**A**	**B**	**C**	**D**
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

We can use four variable K-map to minimize and design the BCD to Excess-3 Code Converter Circuit.

K-Map for A:

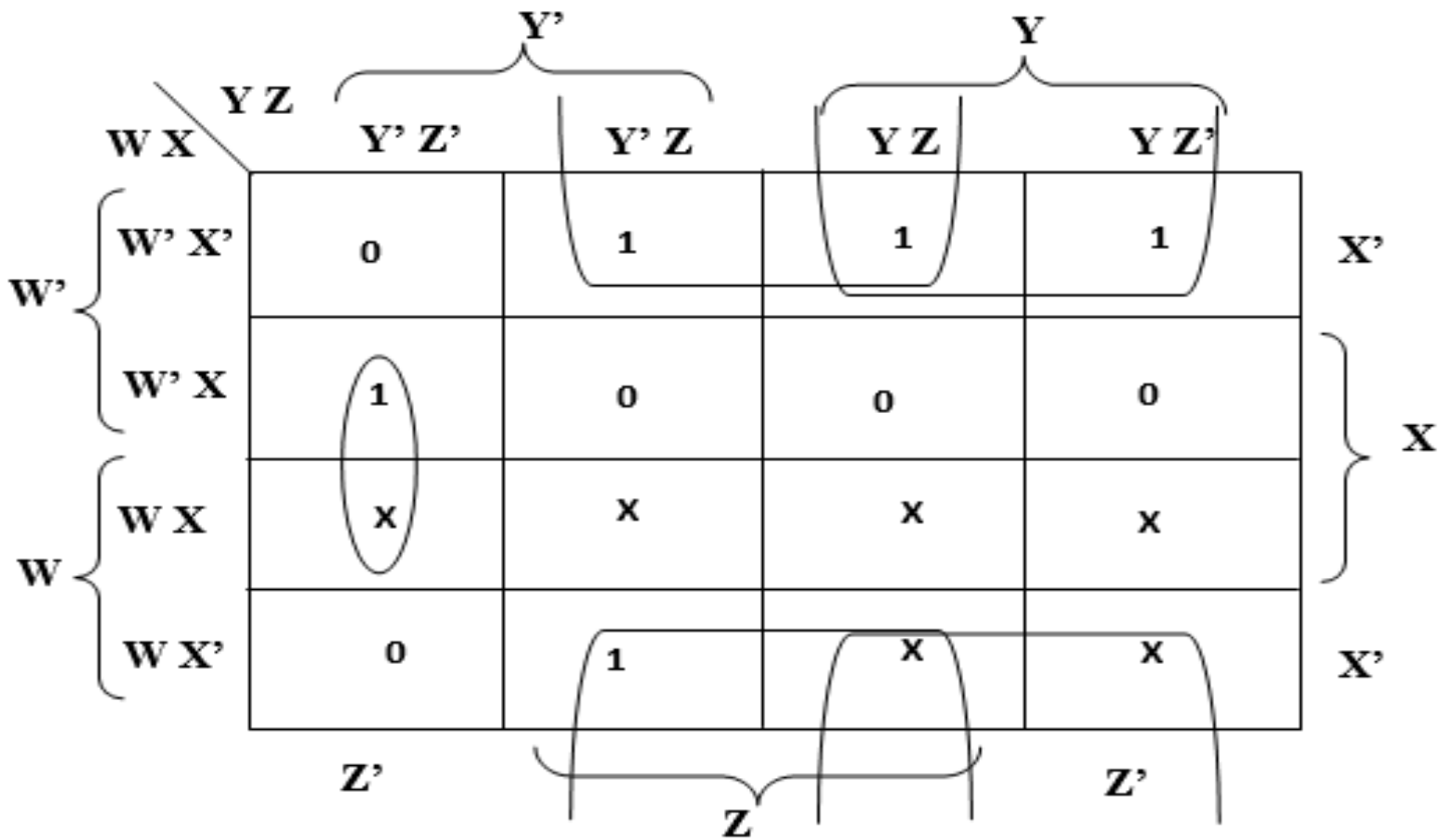
A = W' X Y' Z + W' X Y Z' + W' X Y Z + W X' Y' Z' + W X' Y' Z



A = W + X Z + X Y

K-Map for B:

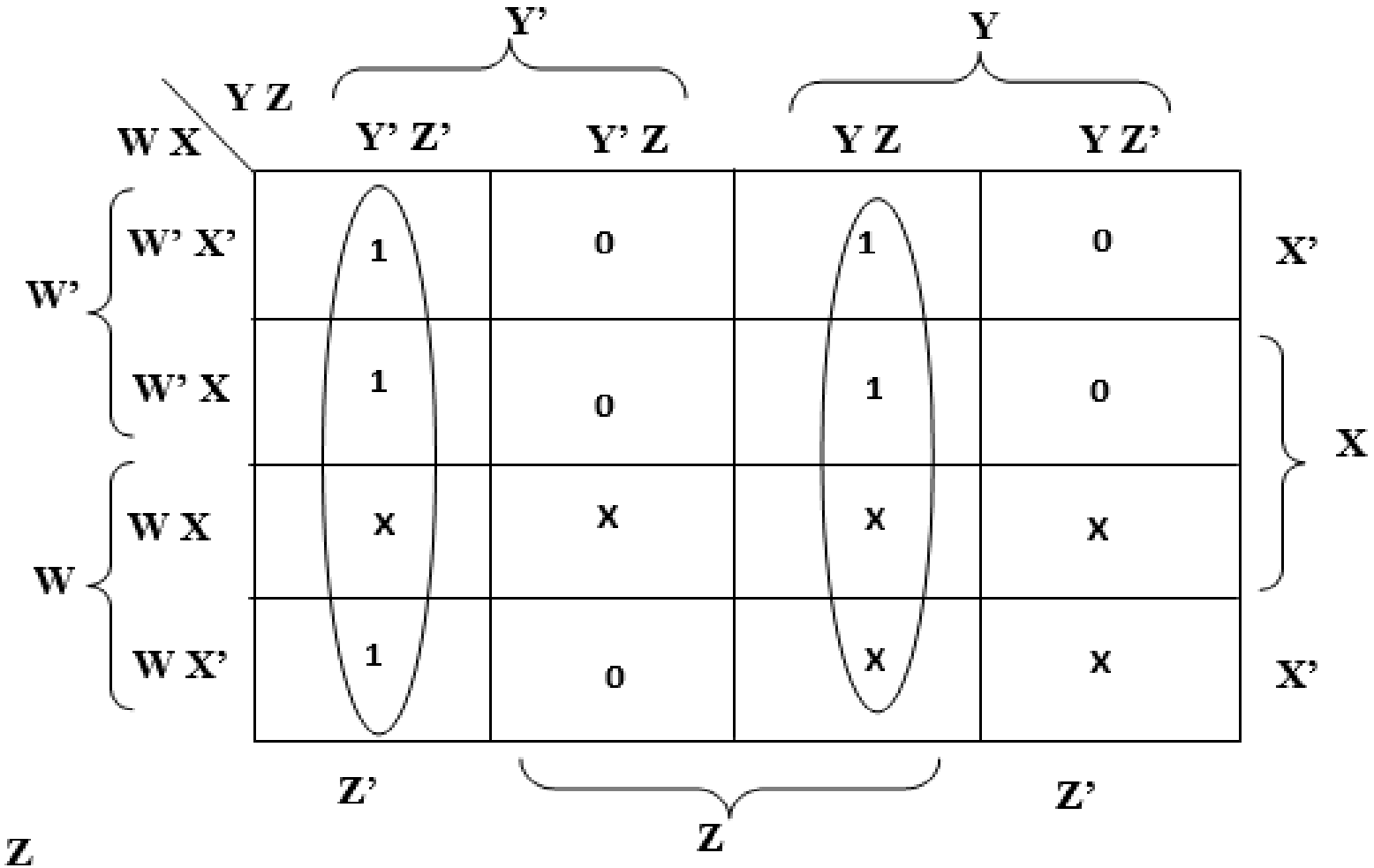
B= W' X' Y' Z + W' X' Y Z' + W' X' Y Z + W' X Y' Z' + W X' Y' Z



B = X Y' Z' + X' Z + X' Y

K-Map for C:

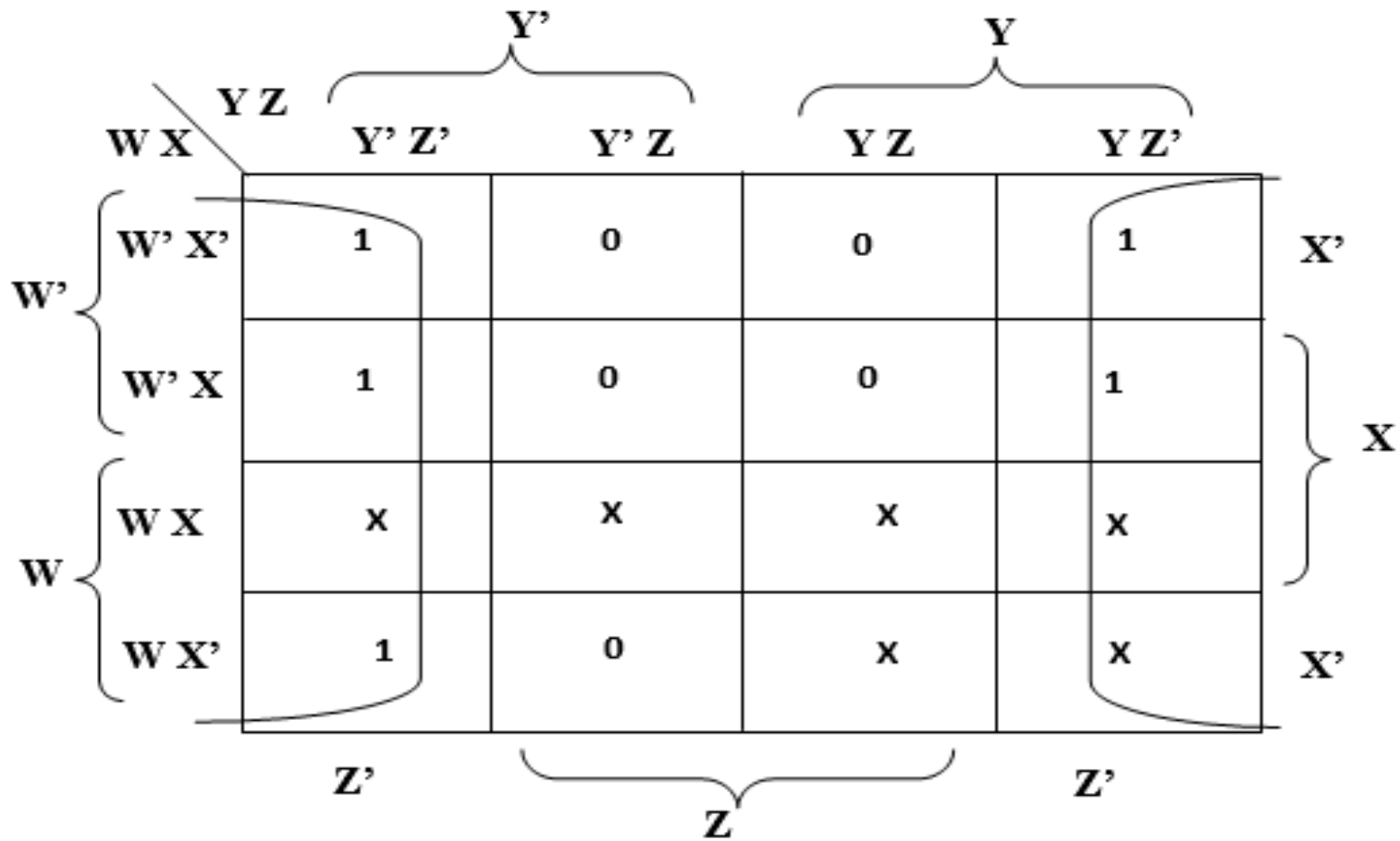
C = W' X' Y' Z' + W' X' Y Z + W' X Y' Z' + W' X Y Z + W X' Y' Z



C = Y' Z' + Y Z

K-Map for D:

D = W' X' Y' Z' + W' X' Y Z' + W' X Y' Z' + W' X Y Z' + W X' Y' Z'



D = Z'

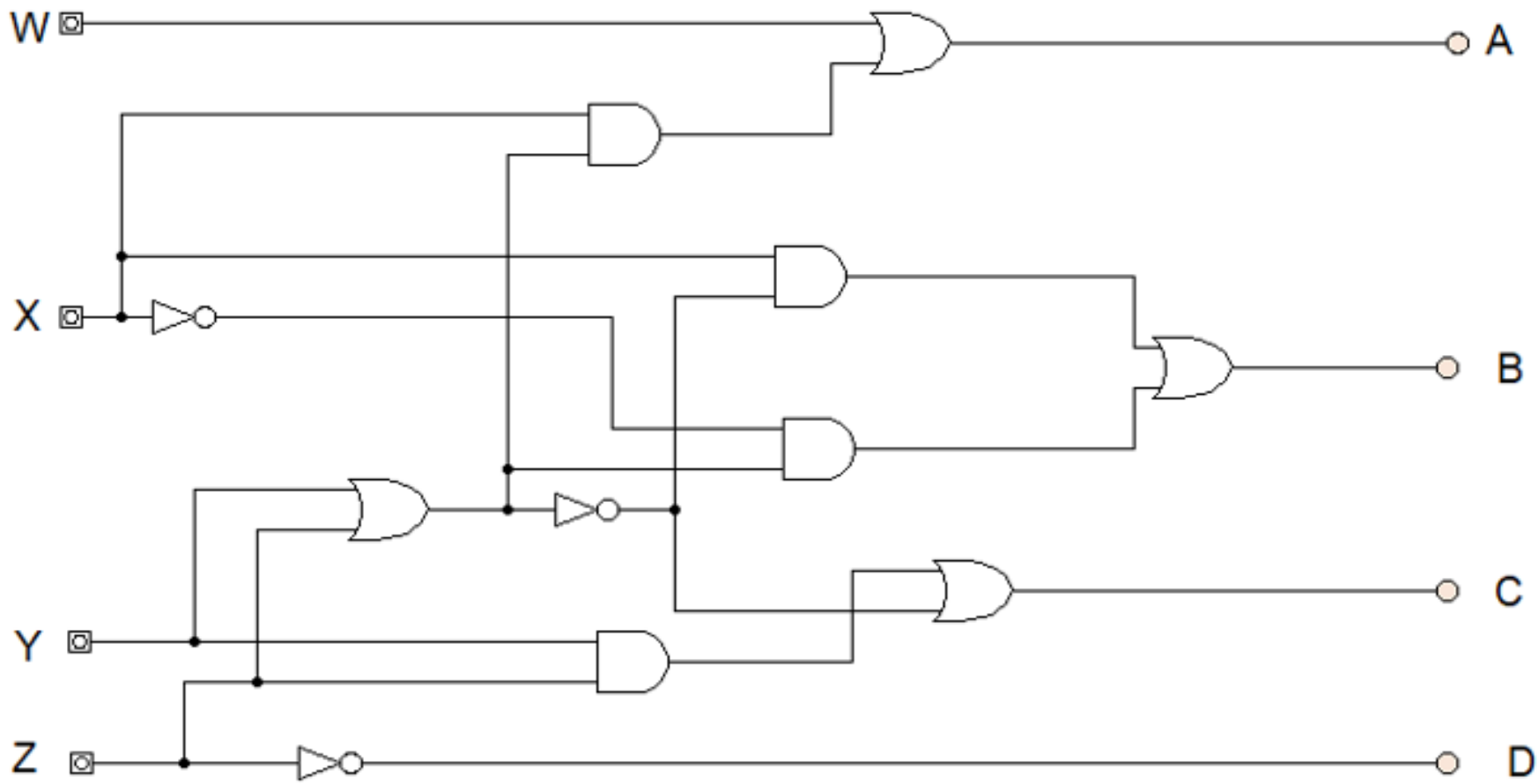
$A = W + X\,Z + X\,Y = W + X(Y + Z)$

$B = X\,Y'\,Z' + X'\,Z + X'\,Y = X\,Y'Z' + X'(Y + Z) = X(Y + Z)' + X'(Y + Z)$

$C = Y'\,Z' + Y\,Z = YZ + (Y + Z)'$

$D = Z'$

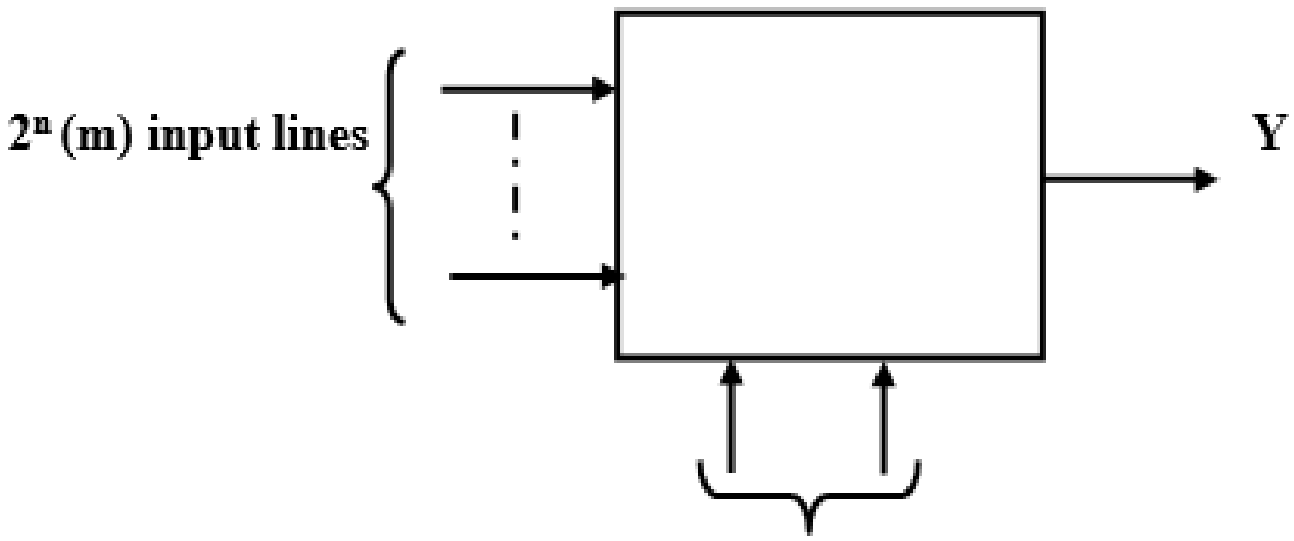
Logic Diagram of BCD to Excess-3 code converter



Multiplexer (MUX): Multiplexing means transmitting a large number of information units over a smaller number of channels or lines.

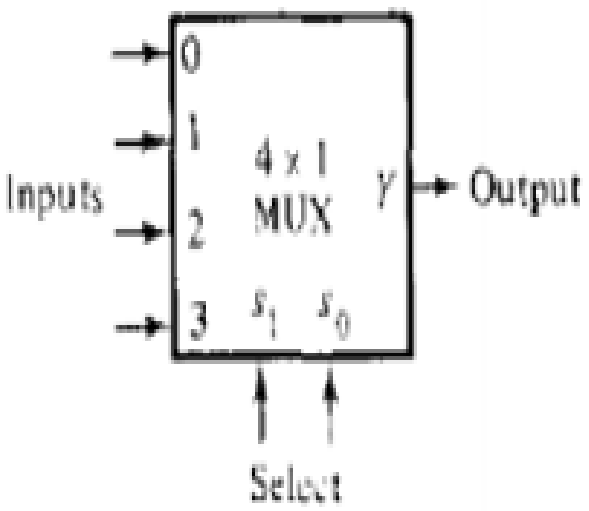
Multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. It is also called data selector or input selector, because it selects only one input at a time and directs it to the output. The selection of particular input depends on the bit combination of selection input. For n selection line input line combination is 2^n.

Block diagram of MUX:



4 x 1 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

$$Y=S1' S0' I0 + S1' S0 I1 + S1 S0' I2 + S1 S0 I3$$

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

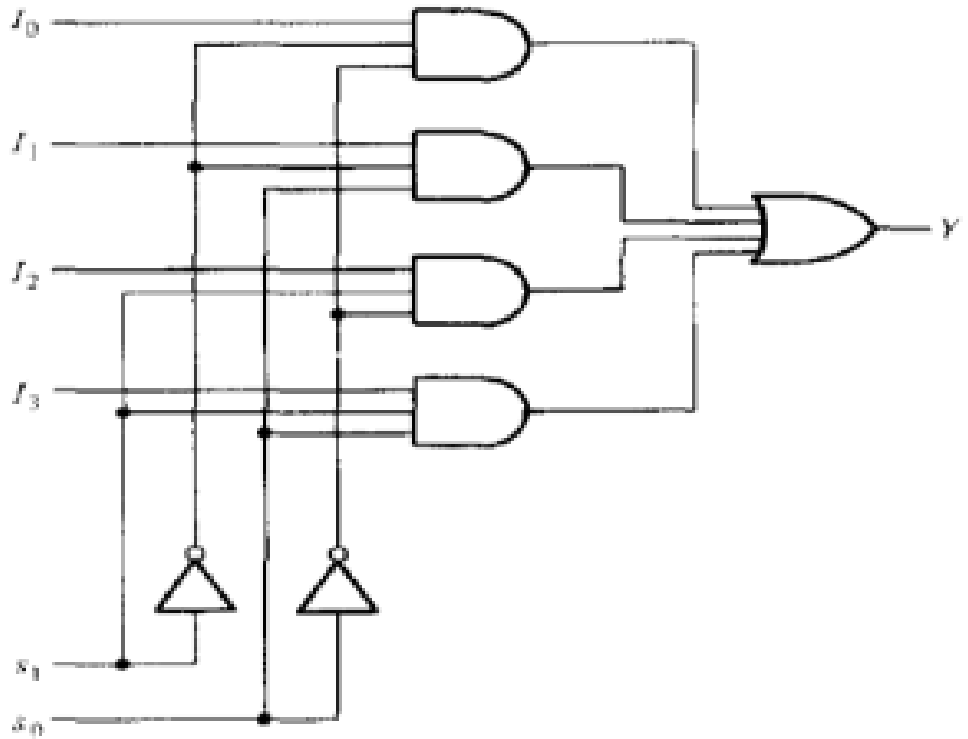
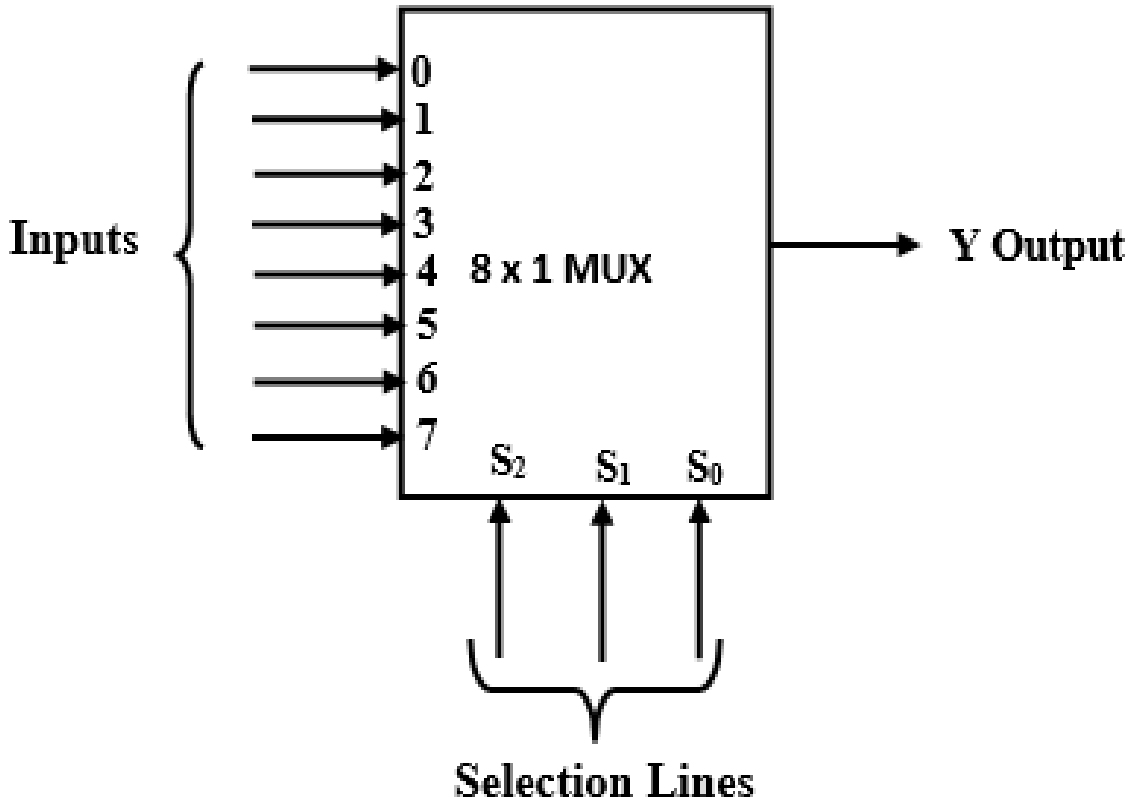


Fig: Logic Diagram: 4-to-1 line Multiplexer

8 x 1 Multiplexer:

8 x 1 Multiplexer has EIGHT data inputs I_7, I_6, I_5, I_4, I_3, I_2, I_1 & I_0, three selection lines s_2, s_1 & s_0 and one output Y. The **block diagram** of 8 x 1 Multiplexer is shown in the following figure.



One of these 8 inputs will be connected to the output based on the combination of inputs present at these three selection lines.

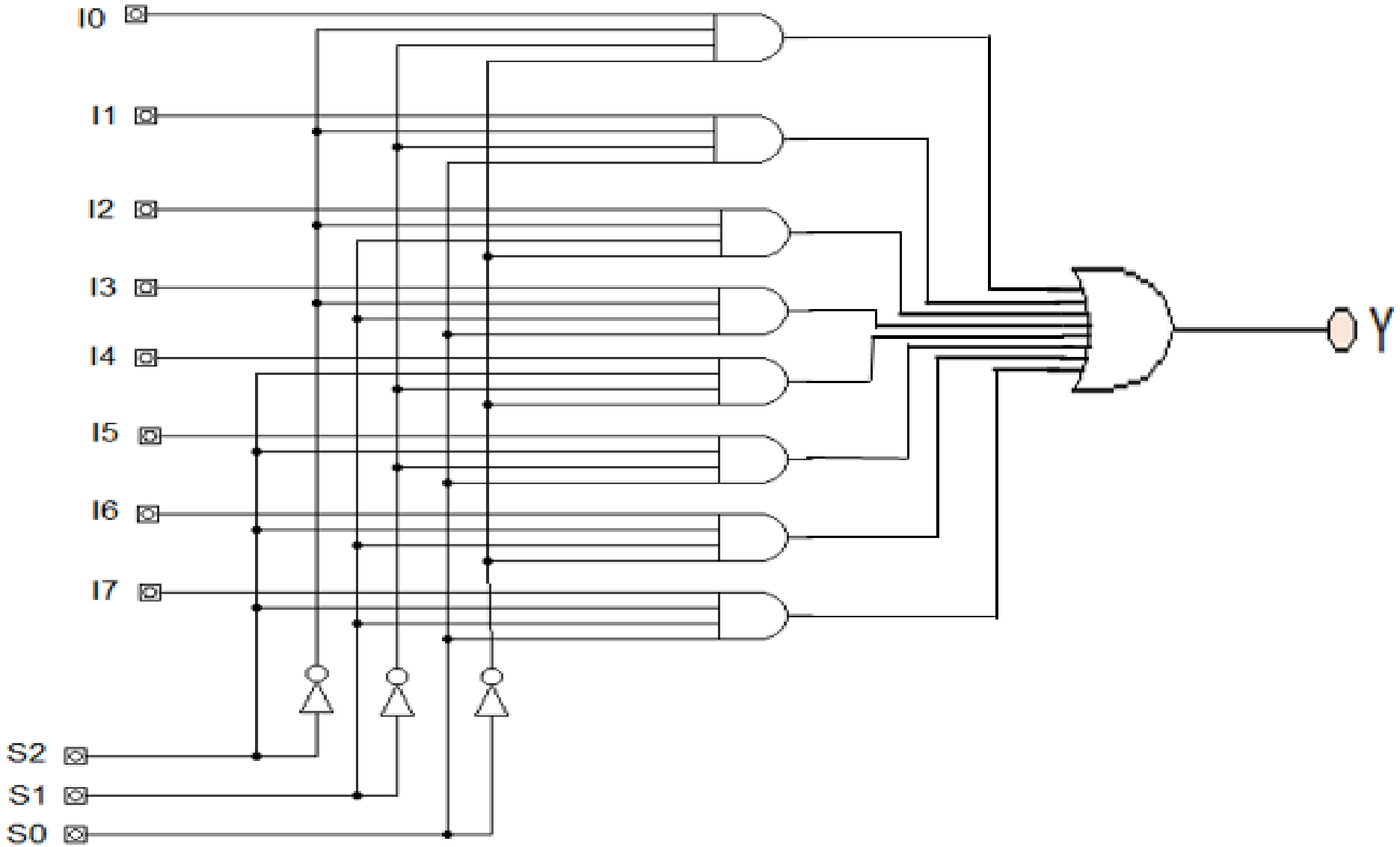
Truth table of 8x1 Multiplexer:

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

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

$$Y=S2' S1' S0' I0 + S2' S1' S0 I1 + S2' S1 S0' I2 + S2' S1 S0 I3 + S2 S1' S0' I4 + S2 S1' S0 I5 + S2 S1 S0' + S2 S1 S0 I7$$

Logic diagram of 8 x 1 Multiplexer:



Boolean Function implementation: As decoder can be used to implement a Boolean function by employing an external OR gate, we can implement any Boolean function (in SOP) with multiplexer since multiplexer is essentially a decoder with the OR gate already available.

- If we have a Boolean function of n + 1 variables, we take n of these variables and connect them to the selection lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the selection lines, one can implement any Boolean function with a multiplexer.
- So, it is possible to generate any function of n + 1 variables with a 2n -to-1 multiplexer.

Example: *Implement Boolean function* $F(A, B, C) = \Sigma(1, 3, 5, 6)$ *with multiplexer.*

Solution: The function can be implemented with a 4-to-1 multiplexer, as shown in Fig. below. Two of the variables, B and C, are applied to the selection lines in that order, i.e., B is connected to s1 and C to s0. The inputs of the multiplexer are 0, 1, A and A'.

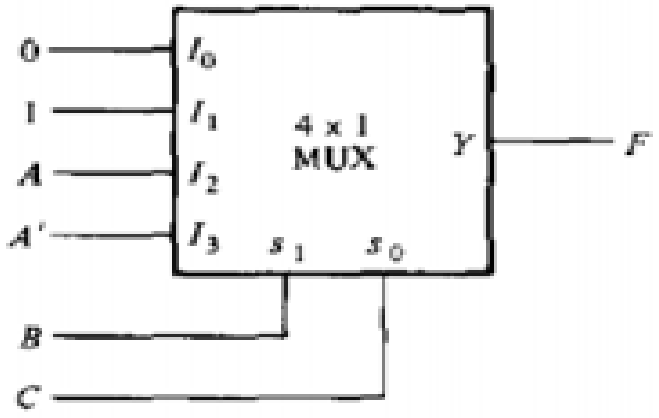
Truth Table:

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

Implementation Table:

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

Multiplexer Implementation:



Most important thing during this implementation is the **implementation table** which is derived from following rules:

List the inputs of the multiplexer and under them list all the minterms in two rows. The first row lists all those minterms where *A* is complemented, and the second row all the minterms with *A* uncomplemented, as shown in above example. Circle all the minterms of the function and inspect each column separately.

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

Question: Implement the following function with a multiplexer:

$$F(A, B, C, D) = \Sigma(0, 1, 3, 4, 8, 9, 15)$$

Solution: The function can be implemented with a 8-to-1 multiplexer, as shown in Fig. below. Three of the variables, B, C and D are applied to the selection lines in that order, i.e., B is connected to s2, C to s1 and D to S0. The first half of the midterms are associated with A' and the second half with A.

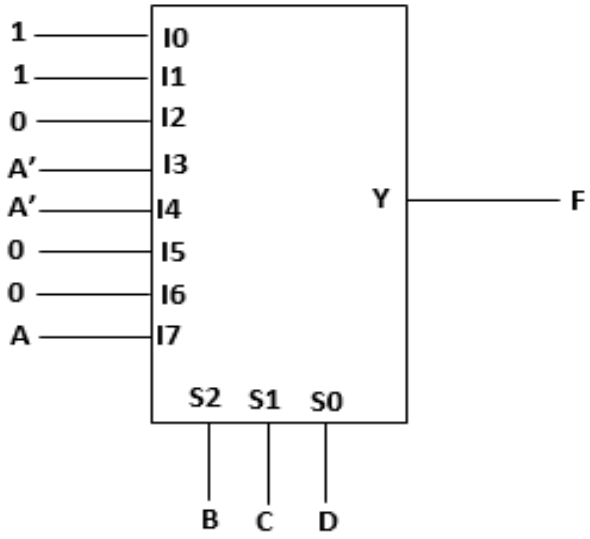
Truth Table:

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

Implementation Table:

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	(0)	(1)	2	(3)	(4)	5	6	7
A	(8)	(9)	10	11	12	13	14	(15)
	1	1	0	A'	A'	0	0	A

Multiplexer Implementation:



Q. Implement 4 x 1 MUX using 2 x 1 MUX.

Solution;

Here,

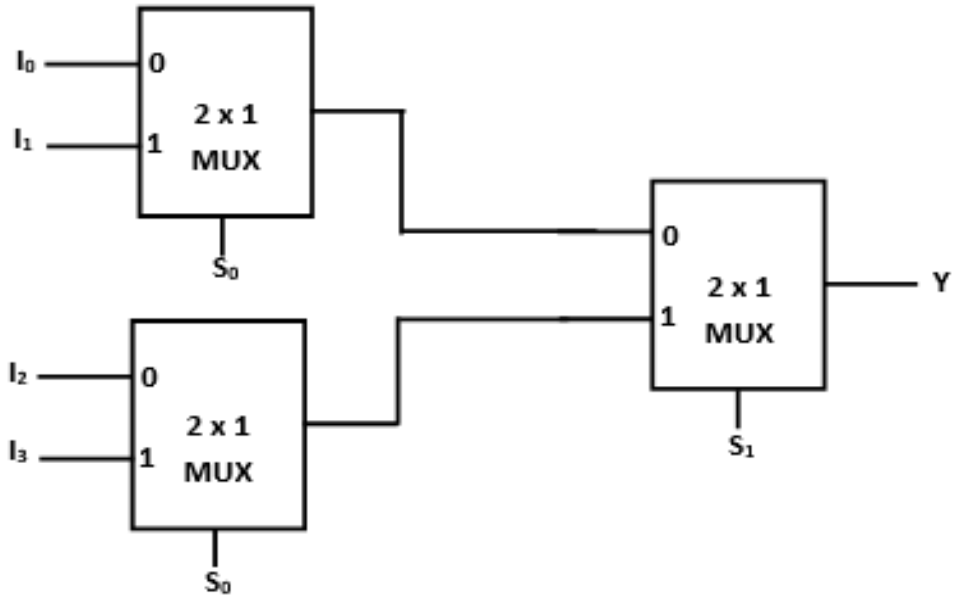
Required inputs $n_1 = 4$

Available inputs $n_2 = 2$

$n_1 / n_2 = 4/2 = 2$ 1st order

$= 2 / 2 = 1$ 2nd order

Total 2 x 1 MUX required= 2 + 1 = 3



Truth Table

Selection Lines		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Suppose, $S_0 = S_1 = 0$, upper MUX selects I_0 and lower MUX selects I_2 and 3rd MUX selects I_0, hence I_0 output will be selected.

Q. Implement 8 x 1 MUX using 4 x 1 MUX.

Solution;

Here,

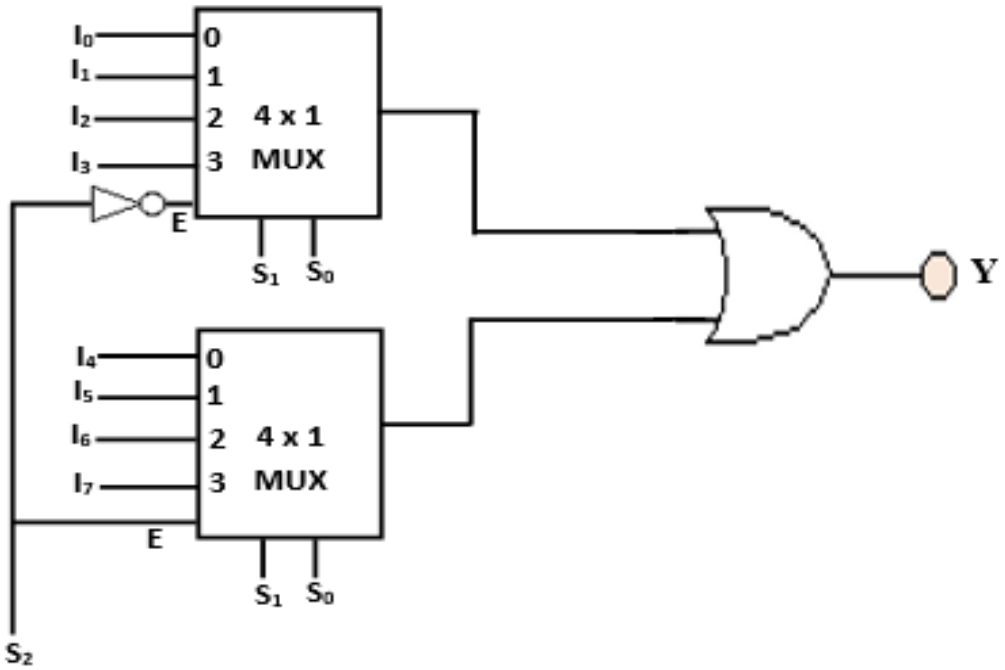
Required inputs $n_1 = 8$

Available inputs $n_2 = 4$

$n_1/n_2 = 8/4 = 2$

$= 2 / 4 = 0.5$

We need two 4 x 1 MUX and one OR gate



S2	S1	S0	Y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

When $S_2 = 0$, then upper MUX works.

When $S_2 = 1$, then lower MUX works.

Suppose,

Case: $S_2 = 0$, $S_1 = 1$, $S_3 = 1$, from upper MUX, I_3 is selected and from Lower MUX, $I_7 = 0$ is selected.

Therefore, $I_3 + 0 = I_3$ is final output.

Case: $S_2 = S_1 = S_0 = 1$, from upper MUX $I_3 = 0$ is selected and from lower MUX I_7 is selected.

Therefore, $0 + I_7 = I_7$ is final output.

Q. Implement 8 x 1 MUX using 2 x 1 MUX.

Solution:

n1= 8

n2 = 2

n1/ n2 = 8/2 = 4

4/2 = 2

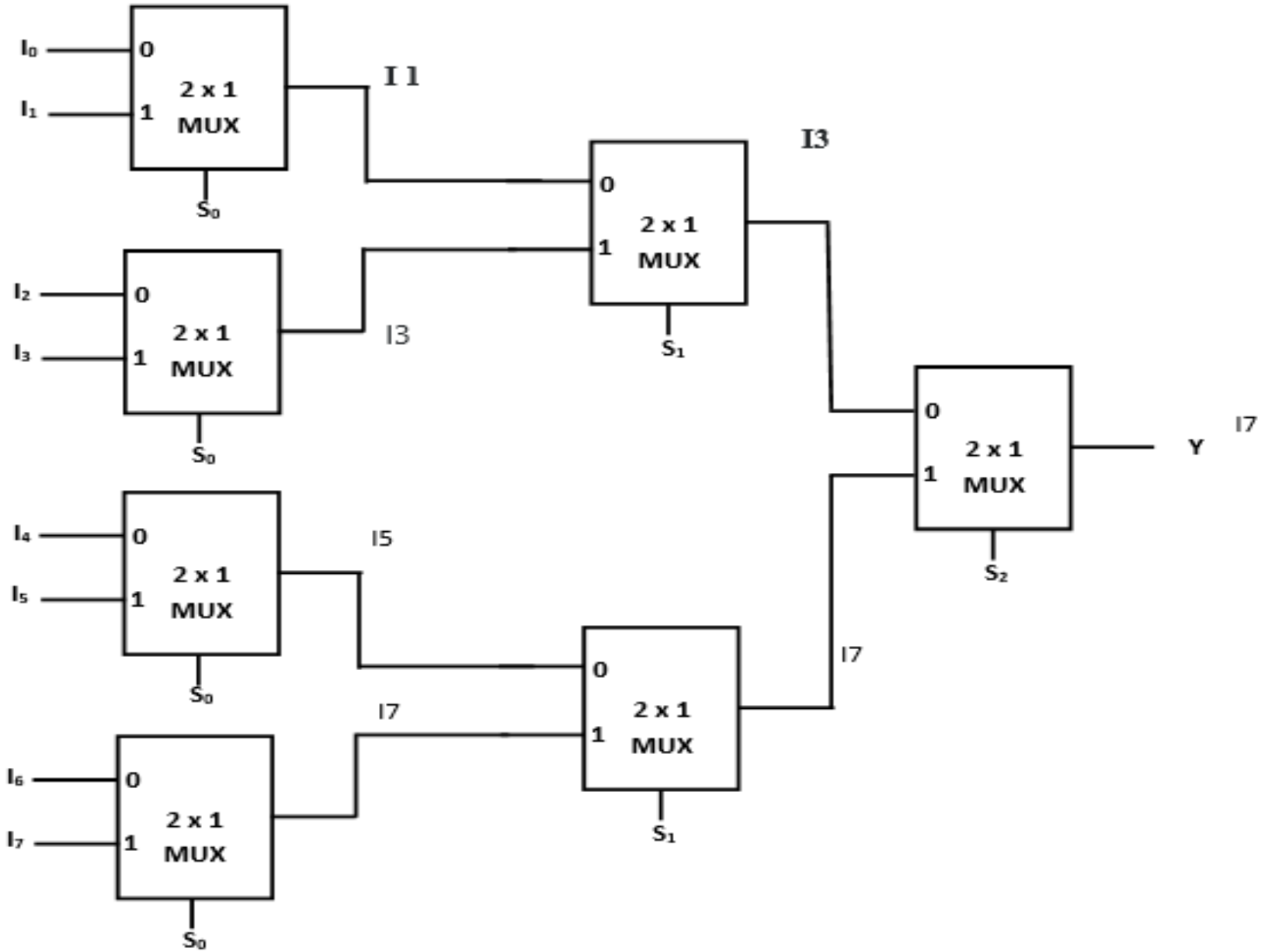
2/2 =1

4 +2 +1 =7

Truth Table of 8 x 1 MUX:

S2	S1	S0	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

Implementation of 8 x 1 MUX using 2 x 1 MUX.



Applications of Multiplexer

Mux is implemented in various domains where there is a necessity of transmitting a large amount of data with the use of single line.

1. Communication System

A Mux is implemented in this system to increase efficiency. Using a single transmission line, various types of data (audio, video, etc.) are transmitted at the same instant.

2. Computer Memory

In a computer, the huge quantity of memory is implemented by means of the Mux. It also has an advantage of a reduction in the number of copper lines which are used for the connection of memory to other parts of the computer.

Applications of Multiplexer

3. Computer System of a Satellite Transmission

Mux is used for the data signals to be transmitted from spacecraft or computer system of a satellite to the earth by means of GPS.

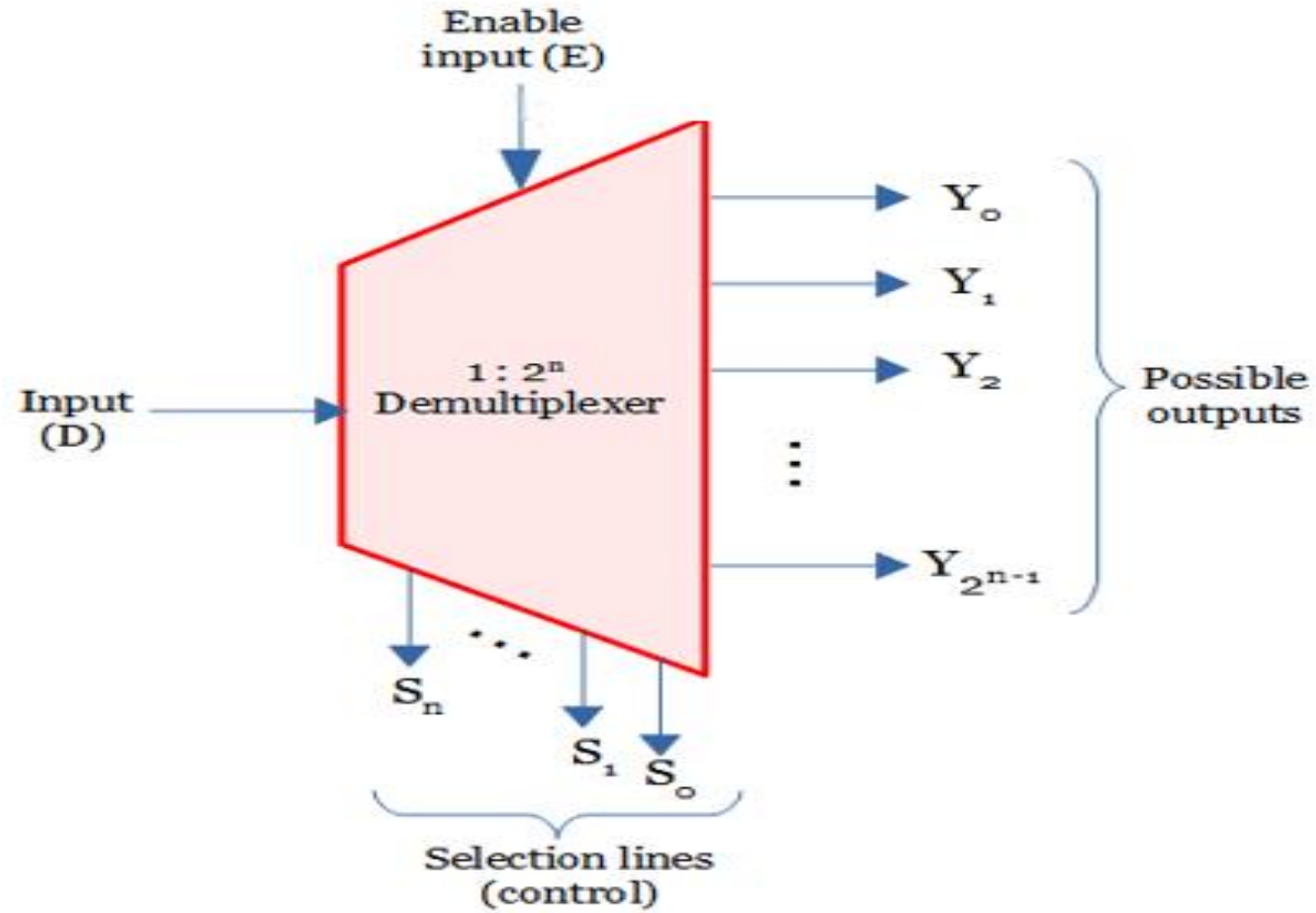
4. Telephone Network

In a telephone network, the multiple audio signals are brought into a single line and transmitted with the implementation of a Mux. By this way, the numerous audio signals are made isolated and ultimately the recipient will receive the required audio signals.

De-Multiplexer (De-MUX)

De-multiplexer or De-mux is a combinational circuit that distributes the single input data to a specific output line. The control inputs or selection lines are used to select a specific output line from the possible output lines. De-multiplexer works opposite to that of the multiplexer. De-mux has one input, 2^n possible outputs and n control or selection lines. It is also called a data distributor.

Block Diagram of De-Multiplexer:

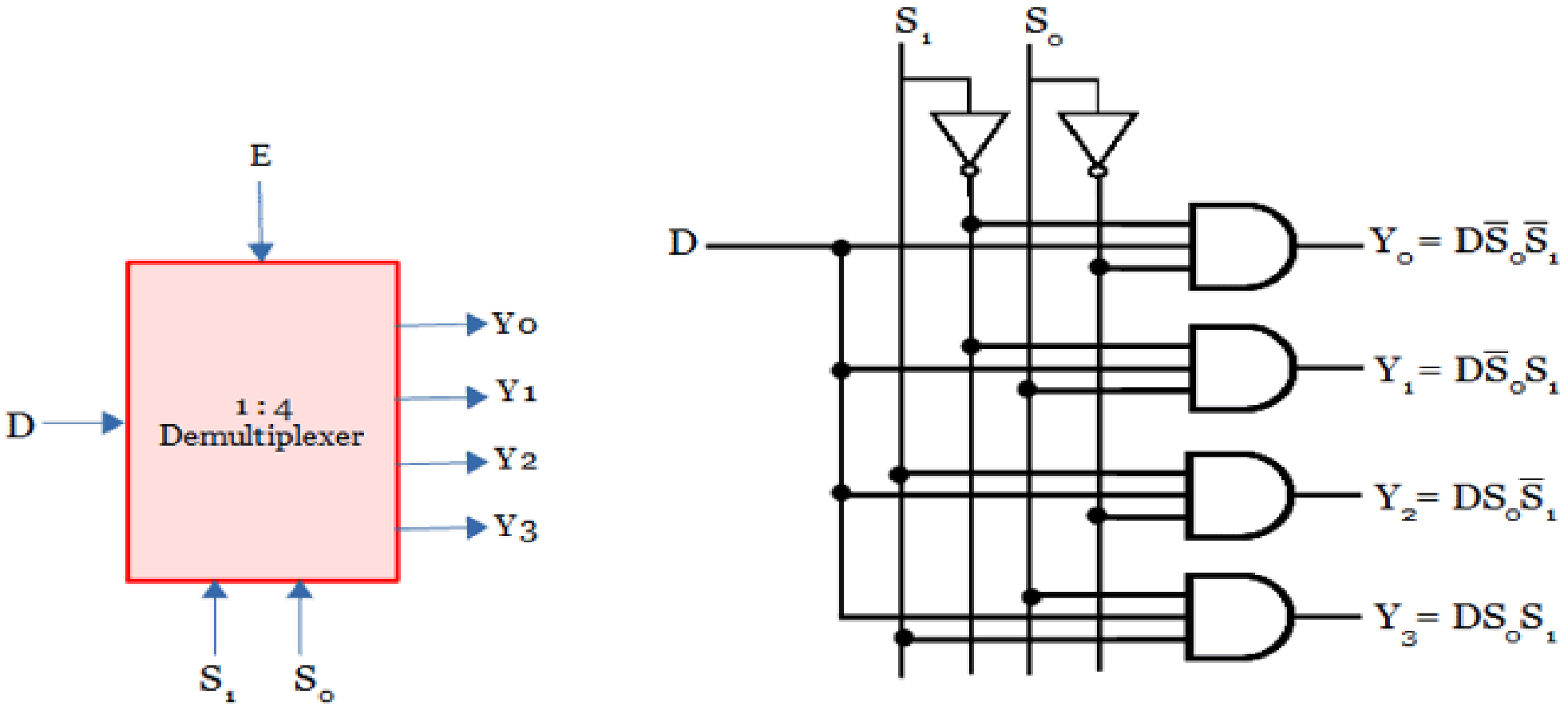


The de-multiplexer circuit is shown in the above diagram. It has one data input (D), 2^n possible outputs (Y_0, Y_1, Y_2,...Y_{2^n-1}), n selection lines (S_0, S_1,...S_n). It also has an enable input. The de-mux will work only when the enable is set to logic 1.

As like multiplexer, the de-mux also has several types based on the number of possible outputs. It includes 1-to-4 de-mux, 1-to-8 de-mux, etc.

1 : 4 De-multiplexer:

The block diagram and circuit of 1-to-4 de-multiplexer are shown below. There are four possible outputs Y_0, Y_1, Y_2, Y_3 and a single input D. The single data input is sent to one of the four outputs as per the selection line input.



Block diagram and circuit of 1 : 4 de-mux

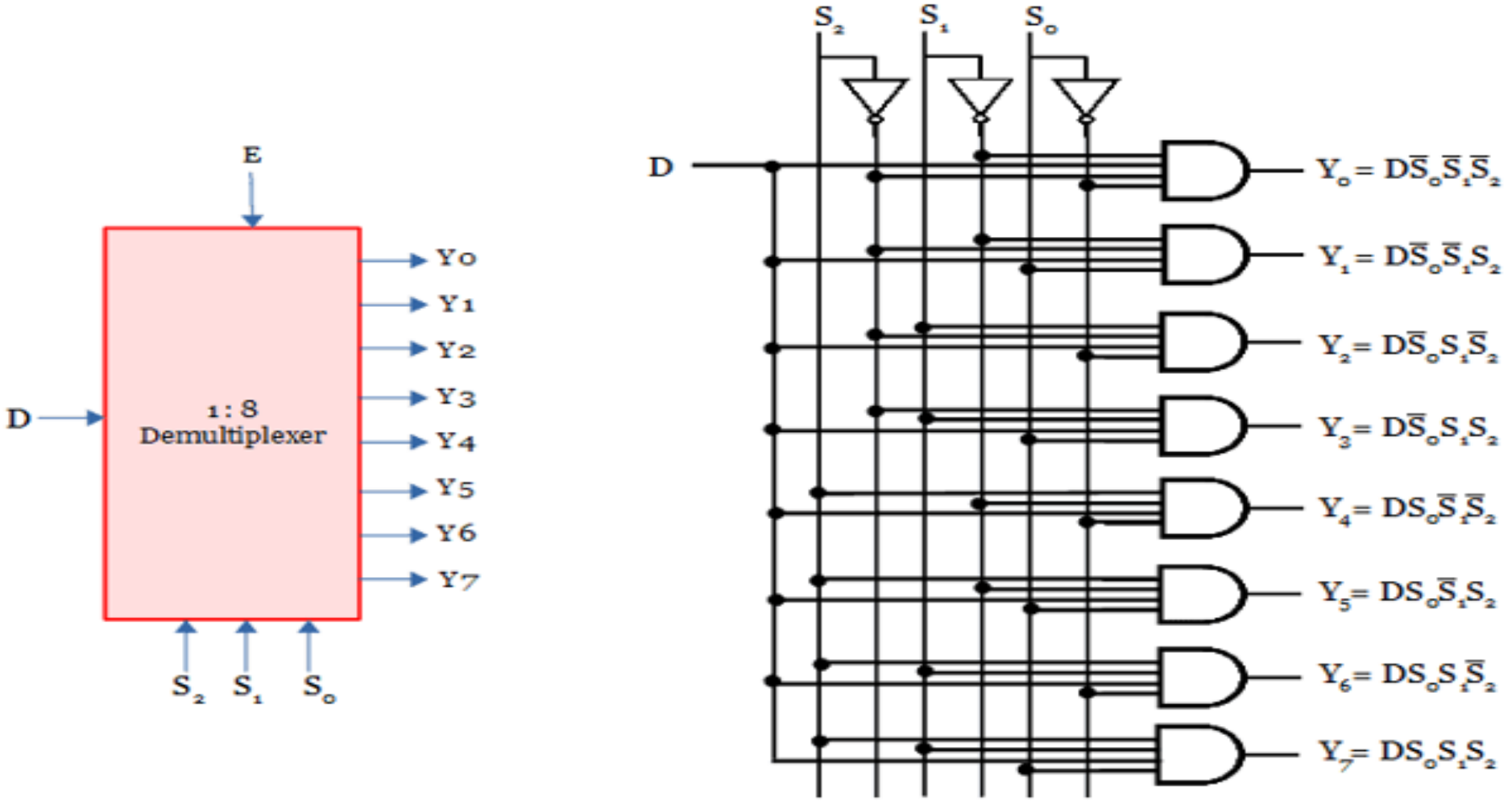
If the selection line input $S_1S_0 = 00$, the first AND gate in the above circuit diagram gets enabled. It is because both the AND gate receives the inverted input. Since all the remaining AND gates get 0 from the S_1S_0 at any one of the inputs, they get disabled for this input. Thus the data input is routed to the output Y_0.
When the selection line input, $S_1S_0 = 01$, the second AND gate is enabled and so the data input is directed to the output Y_1.
When $S_1S_0 = 10$, the third AND gate gets enabled, which will drive the data input D to the output terminal Y_2.
Similarly, for $S_1S_0 = 11$, the AND gate at the bottom will be enabled and so the data input D will be at the output Y_3.
The truth table shown below explains the operation of 1 : 4 de-multiplexer.

Enable (E)	Selection lines		Data input (D)	Outputs			
	S_1	S_0		Y_0	Y_1	Y_2	Y_3
0	x	x	x	x	x	x	x
1	0	0	1	1	0	0	0
1	0	1	1	0	1	0	0
1	1	0	1	0	0	1	0
1	1	1	1	0	0	0	1

Function table of 1 : 4 De-mux

1 : 8 De-multiplexer:

Similar to the 1 to 4 de-mux, 1-to-8 de-multiplexer performs the transfer of single data to any one of the 8 possible outputs. It has 3 selection lines to distribute the data to the output.



Block diagram and circuit of 1 : 8 De-mux

The operation is similar to a 1-to-4 de-mux. The following truth table or function table shows the operation of the 1-to-8 de-multiplexer.

Enable (E)1	Selection lines			Data input (D)	Outputs							
	S_2	S_1	S_0		Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	x	x	x	x	x	x	x	x	x	x	x	x
1	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	1	1	0	1	0	0	0	0	0	0
1	0	1	0	1	0	0	1	0	0	0	0	0
1	0	1	1	1	0	0	0	1	0	0	0	0
1	1	0	0	1	0	0	0	0	1	0	0	0
1	1	0	1	1	0	0	0	0	0	1	0	0
1	1	1	0	1	0	0	0	0	0	0	1	0
1	1	1	1	1	0	0	0	0	0	0	0	1

Function table of 1 : 8 De-mux

Applications of De-multiplexer:

1. The de-multiplexers are used along with multiplexers. So, in the communication system, the multiplexer is used for transmitting the information, whereas de-mux is used to retrieve the original message at the receiving end.
2. It is used in applications where serial to parallel conversion of binary data is needed.
3. In time-division multiplexing, used to route the single input to multiple output lines at the receiving end.

Binary Adder: This circuit sums up two binary numbers A and B of n-bits using full-adders to add each bit-pair & carry from previous bit position. The sum of A and B can be generated in two ways: either in a serial fashion or in parallel.

- The serial addition method uses only one full-adder circuit and a storage device to hold the generated output carry. The pair of bits in A and B are transferred serially, one at a time, through the single full-adder to produce a string of output bits for the sum. The stored output carry from one pair of bits is used as an input carry for the next pair of bits.

- The parallel method uses n full-adder circuits, and all bits of A and B are applied simultaneously. The outputs carry from one full-adder is connected to the input carry of the full-adder one position to its left. As soon as the carries are generated, the correct sum bits emerge from the sum outputs of all full-adders.

Binary Parallel adder: A binary parallel adder is a digital circuit that produces the arithmetic sum of two binary numbers in parallel. It consists of full-adders connected in a chain, with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

Diagram below shows the interconnection of four full-adder (FA) circuits to provide a 4-bit binary parallel adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left. The carries are connected in a chain through the full-adders. The S outputs generate the required sum bits. The input carry to the adder is C1 and the output carry is C5. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augend bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries.

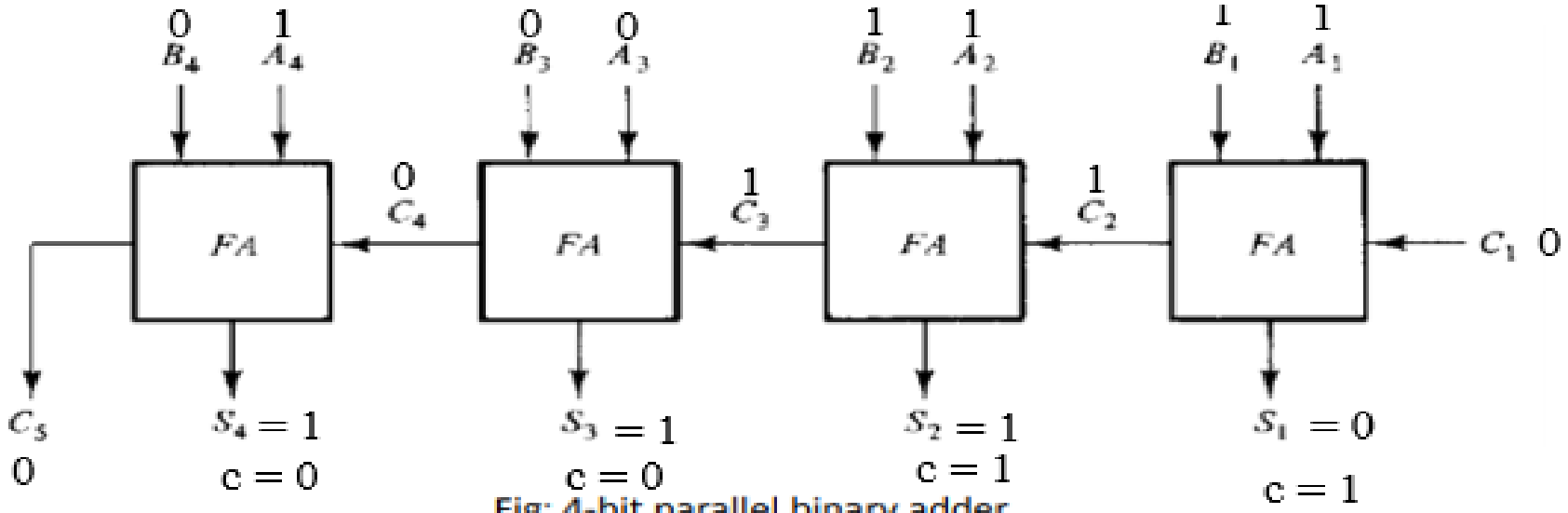


Fig: 4-bit parallel binary adder

Sum = A ⊕ B ⊕C

Carry = AB + BC +AC

E.g. A = 1 0 1 1

B = 0 0 1 1

x	**Input Carry**	**0 1 1 0**	**C_i**
y	**Augend**	**1 0 1 1**	**A_i**
z	**Addend**	**0 0 1 1**	**B_i**
s	**Sum**	**1 1 1 0**	**S_i**
c	**Output Carry**	**0 0 1 1**	**C_{i+1}**

BCD Adder:

This combinational circuit adds up two decimal numbers when they are encoded with binary-coded decimal (BCD) form.

- Adding two decimal digits in BCD, together with a possible carry, the output sum cannot be greater than $9 + 9 + 1 = 19$.
- Applying two BCD digits to a 4-bit binary adder, the adder will form the sum in binary ranging from 0 to 19. These binary numbers are listed in Table below and are labeled by symbols K, Z8, Z4, Z2, and Z1. K is the carry, and the subscripts under the letter z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code.

Truth table for BCD Adder:

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

BCD Adder:

- The first column in the table lists the binary sums as they appear in the outputs of a 4- bit binary adder.
- The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the second column.
- The problem is to find a simple rule by which the binary number, in the first column can be converted to the correct BCD-digit representation of the number in the second column.

Looking at the table, we see that:

When (binary sum) <= 1001

Corresponding BCD number is identical, and therefore no conversion is needed.

When (binary sum) > 1001

Non-valid BCD representation is obtained. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

BCD Adder:

The logic circuit that detects the necessary correction can be derived from the table entries.

- Correction is needed when
- The binary sum has an output carry K = 1.
- The other six combinations from 1010 to 1111 that have Z8=1. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z8, we specify further that either Z4 or Z2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

- When C = 1, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

BCD Adder:

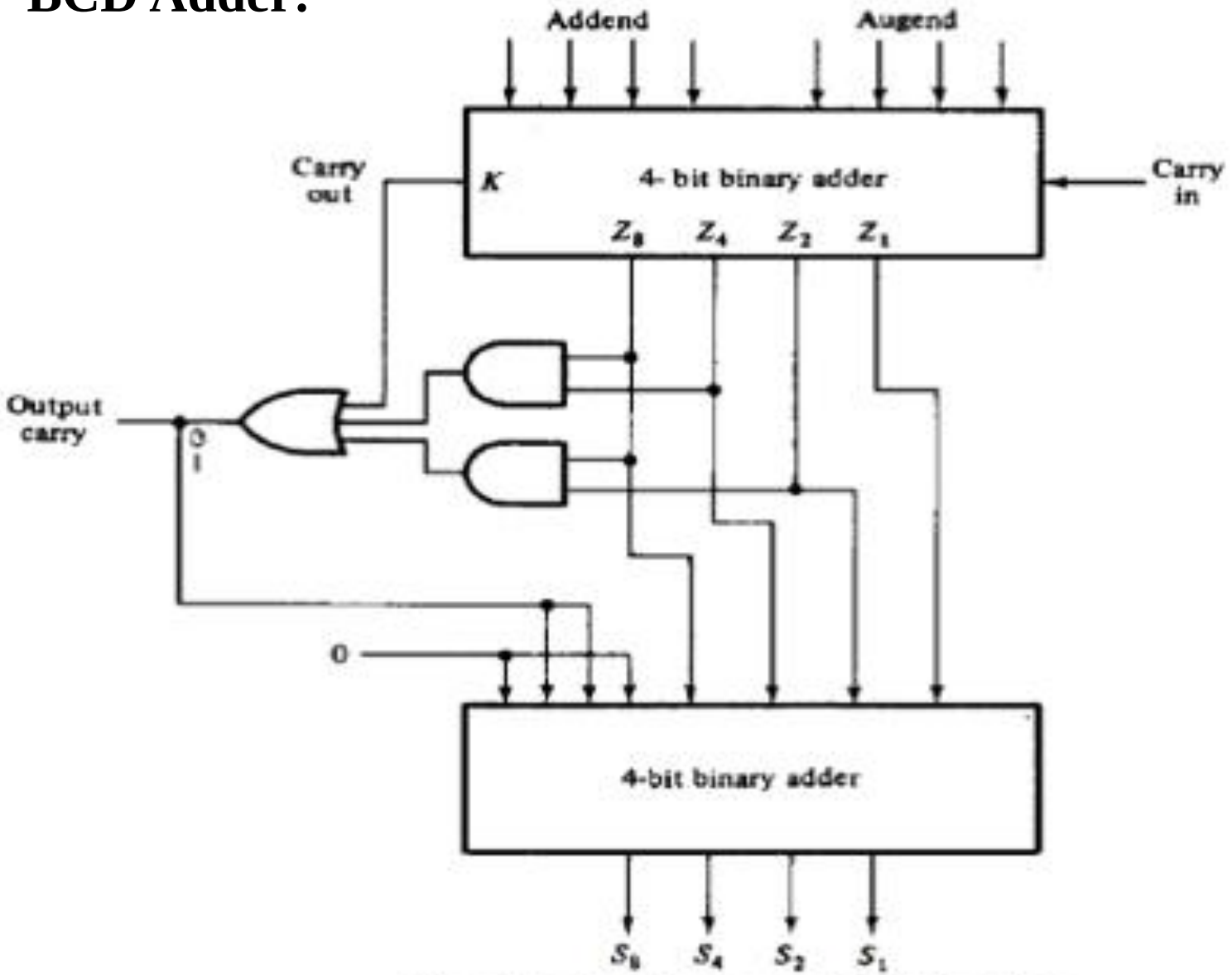


Fig: Block diagram of BCD adder

BCD Adder:

- A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD.
- BCD adder must include the correction logic in its internal construction.
- To add 0110 to the binary sum, we use a second 4-bit binary adder, as shown in diagram.
- The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum.
- When the output carry = 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder.
- The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

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

4-bitMagnitude comparator: It is a combinational circuit that compares two binary numbers, each of 4-bits.

Let the two binary numbers be A and B. Where, A = A_3 A_2 A_1 A_0 and B = B_3 B_2 B_1 $B_{0.}$ Since there are 8-bits in the inputs, it is very difficult to design the logic circuit using truth table (because 2^8 = 256 combinations).

Therefore, it is necessary to design an algorithm to build a logic circuit for 4-bit magnitude comparator.

Truth table for A>B, A = B and A<B

A B	A = B	A < B	A > B
0 0	1	0	0
0 1	0	1	0
1 0	0	0	1
1 1	1	0	0

Now, for $(A = B) = \overline{A}\ \overline{B} + A\ B$

$(A < B) = \overline{A}\ B$

$(A > B) = A\ \overline{B}$

Design of Equal (EQ): (EQ, A = B)

	A_3	A_2	A_1	A_0
E.g. A =	1	1	1	1
B =	1	1	1	1
	B_3	B_2	B_1	B_0

For A = B, All the bits in A and B should be equal. That is $A_3 = B_3$ AND $A_2 = B_2$ AND $A_1 = B_1$ AND $A_0 = B_0$

All the conditions should be true i.e. 1.

Let, $x_i = A_i B_i + \overline{A_i}\,\overline{B_i}$, Where i = 0, 1, 2, 3

Therefore, $x_i = 1$, iff $A_i = B_i$ and $x_i = 0$, iff $A_i \neq B_i$

$EQ = A_i \odot B_i$, EX-NOR gate gives output 1, if all the inputs are same.

Condition for A = B

EQ = 1 (i.e. A = B), iff

$A_3 = B_3 \longrightarrow (x_3 = 1)$ &

$A_2 = B_2 \longrightarrow (x_2 = 1)$ &

$A_1 = B_1 \longrightarrow (x_1 = 1)$ &

$A_0 = B_0 \longrightarrow (x_0 = 1)$

Therefore,

$EQ = 1$, iff $x_3\ x_2\ x_1\ x_0 = 1$

$\mathbf{EQ = x_3\ x_2\ x_1\ x_0}$

Design of GT output (A > B)

Design of GT output (A > B)

- If $A_3 > B_3$, then A > B (GT =1), irrespective of relativize values of other bits of A and B.

  This can be stated as GT = 1,

  If $A_3 \overline{B_3} = 1$, if $A_3 = B_3$ ($x_3 = 1$)

  Compare next significant bits i.e. A_2 and B_2

- If $A2 > B_2$ then A > B irrespective of other values of other bits in A and B.

  This can be stated as GT = 1

  If $x_3 A_2 \overline{B_2} = 1$

  If $A_3 = B_3$ ($x_3 = 1$) and $A_2 = B_2$ ($x_2 = 1$)

Design of GT output (A > B)

Compare next significant pairs of bits i.e. A_1 and B_1

- If $A_1 > B_1$, then A > B (GT = 1) irrespective of the relative values of remaining bits A_0 and B_0

This can be stated as, $x_3\, x_2\, A_1\, \overline{B_1} = 1$

If $A_3 = B_3$ ($x_3 = 1$) and $A_2 = B_2$ ($x_2 = 1$) and $A_1 = B_1$ ($x_1 = 1$)

Compare next pair of bits A_0 and B_0

- If $A_0 > B_0$ and A > B(GT = 1), then A > B can be stated as GT = 1,

If $x_3\, x_2\, x_1\, A_0 \overline{B_0} = 1$

Therefore,

$A_3 \overline{B_3} = 1$

$x_3\, A_2 \overline{B_2} = 1$

$x_3\, x_2\, A_1 \overline{B_1} = 1$

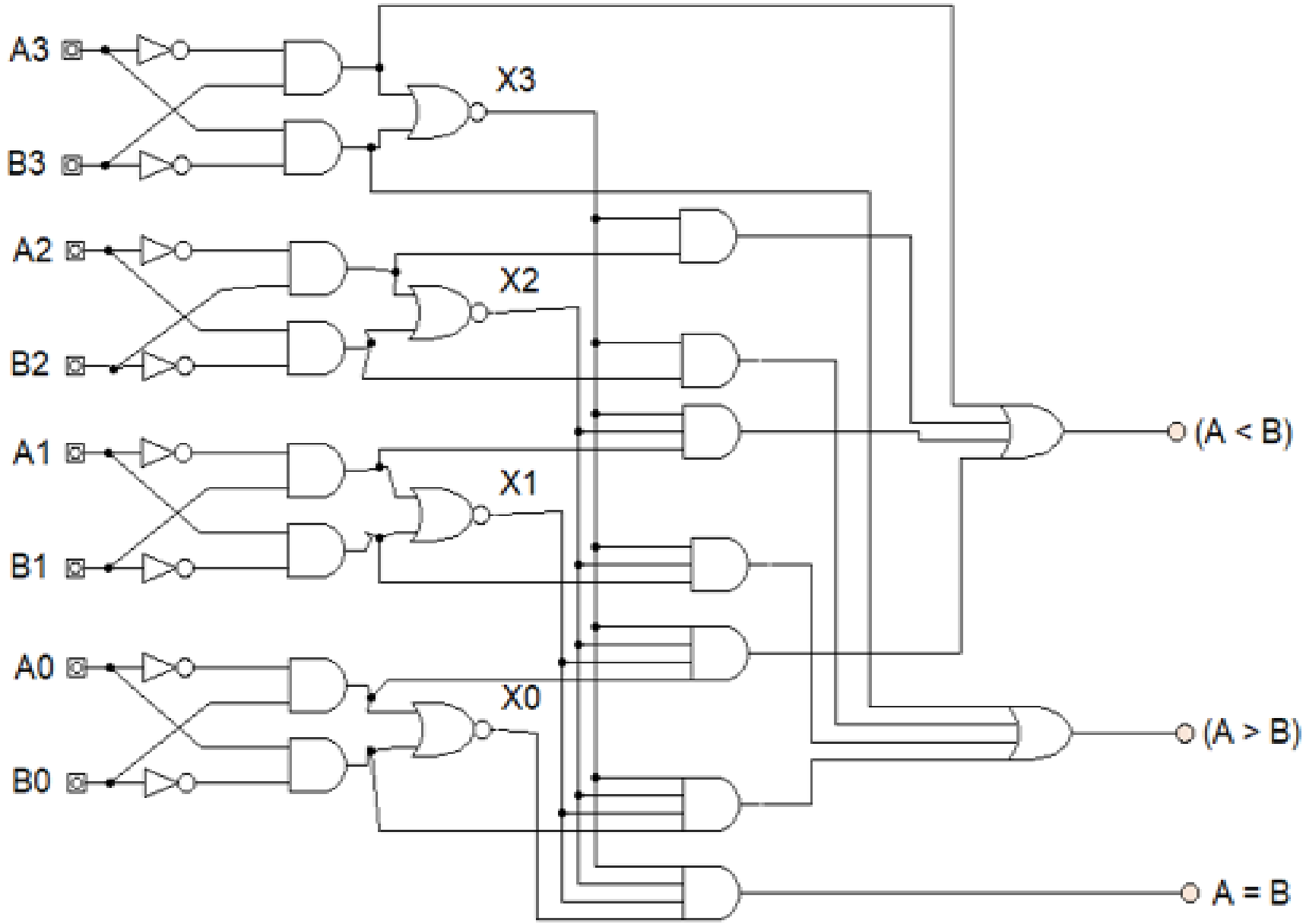
$x_3\, x_2\, x_1\, A_0 \overline{B_0} = 1$

$$\mathbf{GT = A_3 \overline{B_3} + x_3\, A_2 \overline{B_2} + x_3\, x_2\, A_1 \overline{B_1} + x_3\, x_2\, x_1\, A_0 \overline{B_0}}$$

Similarly, for LT output (A < B) follow the same procedure as GT, which gives

$$\mathbf{LT = \overline{A_3}\, B_3 + x_3\, \overline{A_2}\, B_2 + x_3\, x_2\, \overline{A_1}\, B_1 + x_3\, x_2\, x_1\, \overline{A_0}\, B_0}$$

Logic diagram of 4-bit Magnitude comparator:



Programmable Logic Devices (PLDs): Programmable Logic Devices are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

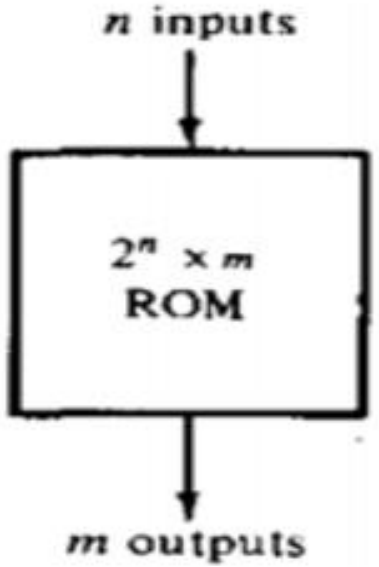
- Programmable Read Only Memory
- Programmable Array Logic
- Programmable Logic Array

The process of entering the information into these devices is known as programming. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Read Only Memory (ROM): ROM is a storage device in which a fixed set of binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established for a ROM, it remains fixed even when power is turned off and turned on.

A read-only memory (ROM) is a device that includes both the decoder and the OR gates within a single IC package. The connections between the outputs of the decoder and the inputs of the OR gates can be specified for each particular configuration. The ROM is used to implement complex combinational circuits within one IC package or as permanent storage for binary information.

Symbolically ROM is indicated as:



It consists of n input lines and m output lines. Each bit combination of the input variables is called an address. Each bit combination that comes out of the output lines is called a word. The number of bits per word is equal to the number of output lines, m. An address is essentially a binary number that denotes one of the minterms of n variables. The number of distinct addresses possible with n input variables is 2n

Combinational Logic Implementation: Each output of ROM provides the sum of all the minterms of the n input variables. Remember that any Boolean function can be expressed in sum of minterms form. By breaking the links of those minterms not included in the function, each ROM output can be made to represent the Boolean function.

- For an n-input, m-output combinational circuit, we need a 2n x m ROM.
- The blowing of the fuses is referred to as programming the ROM.
- The designer need only specify a ROM program table that gives the information for the required paths in the ROM. The actual programming is a hardware procedure that follows the specifications listed in the program table.

Example: Consider a following truth table:

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

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

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

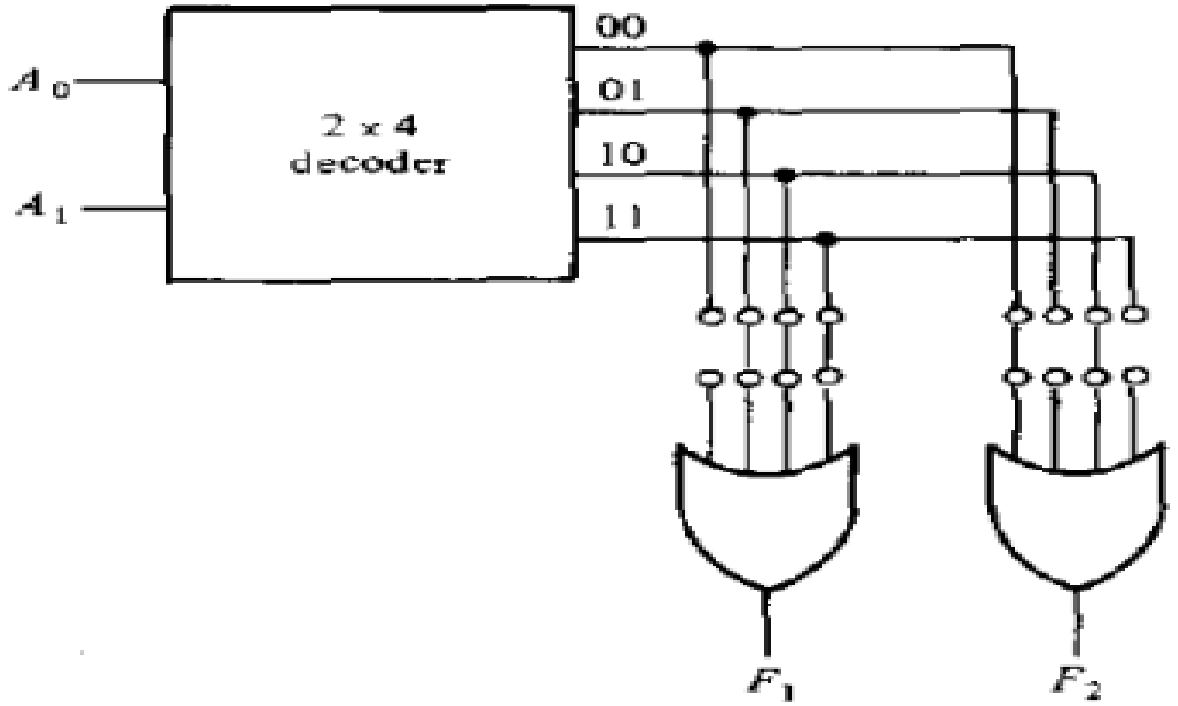


Fig: Combinational-circuit implementation with a 4 x 2 ROM

Diagram shows the internal construction of a 4X2 ROM. It is now necessary to determine which of the eight available fuses must be blown and which should be left intact. This can be easily done from the output functions listed in the truth table. Those minterms that specify an output of 0 should not have a path to the output through the OR gate. Thus, for this particular case, ti1e truth table shows three 0's, and their corresponding fuses to the OR gates must be blown.

Types of ROM:

Programmable Read Only Memory PROM

Read Only Memory ROM is a memory device, which stores the binary information permanently. That means, we can’t change that stored information by any means later. If the ROM has programmable feature, then it is called as **Programmable ROM**. The user has the flexibility to program the binary information electrically once by using PROM programmer.

Erasable Programmable Read Only Memory (EPROM):

The hardware procedure for programming ROMs or PROMs is irreversible and, once programmed, the fixed pattern is permanent and cannot be altered. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed. A third type of unit available is called erasable PROM, or EPROM. EPROMs can be restructured to the initial value (all 0's or all 1's) even though they have been changed previously. When an EPROM is placed under a special ultraviolet light for a given period of time, the shortwave radiation discharges the internal gates that serve as contacts. After erasure, the ROM returns to its initial state and can be reprogrammed.

Electrically Erasable Programmable Read Only Memory (EEPROM):

Certain ROMs can be erased with electrical signals instead of ultraviolet light, and these are called electrically erasable PROMs, or EEPROMs.

Programmable Logic Array (PLA): A combinational circuit may occasionally have don't-care conditions. When implemented with a ROM, a don't care condition becomes an address input that will never occur. The words at the don't-care addresses need not be programmed and may be left in their original state (all 0's or all 1's). The result is that not all the bit patterns available in the ROM are used, which may be considered a waste of available equipment. PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD.

Block Diagram of PLA:

A block diagram of the PLA is shown in Fig. below. It consists of n inputs, m outputs, k product terms, and m sum terms. The product terms constitute a group of k AND gates and the sum terms constitute a group of m OR gates. Fuses are inserted between all n inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gates and the inputs of the OR gates.

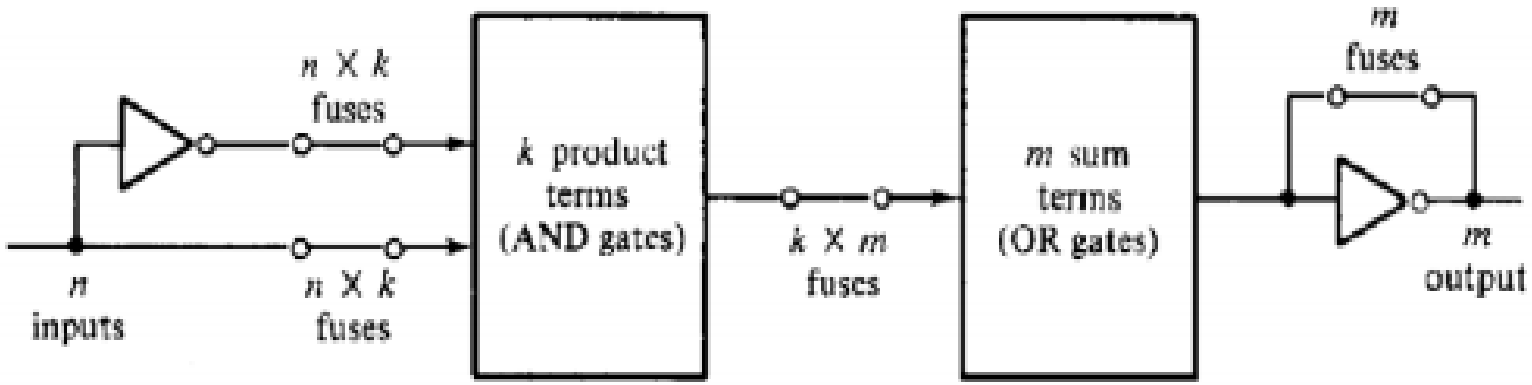


Fig: PLA block diagram

Total Program link=2n x k + k x m + m

PLA program table and Boolean function Implementation:

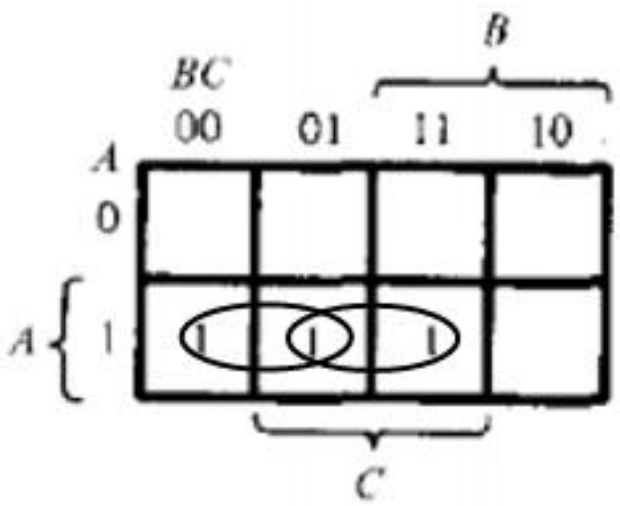
The use of a PLA must be considered for combinational circuits that have a large number of inputs and outputs. It is superior to a ROM for circuits that have a large number of don't-care conditions. Let me explain the example to demonstrate how PLA is programmed.

Consider a truth table of the combinational circuit:

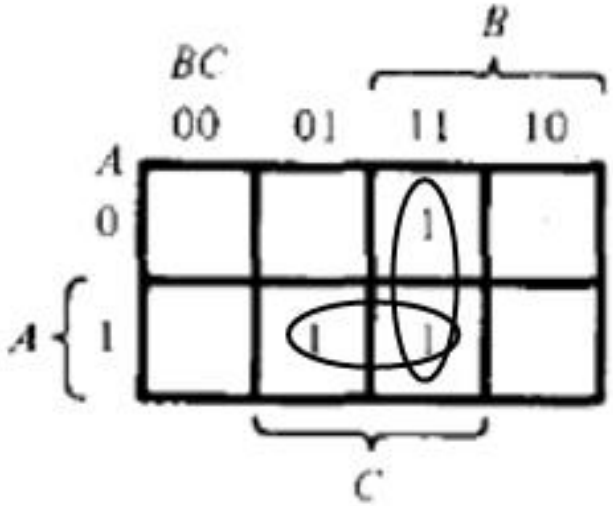
A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

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

The simplified functions in sum of products are obtained from the following K-maps:



$F_1 = AB' + AC$



$F_2 = AC + BC$

There are three distinct product terms in this combinational circuit: AB', AC and BC. The circuit has three inputs and two outputs; so the PLA can be drawn to implement this combinational circuit.

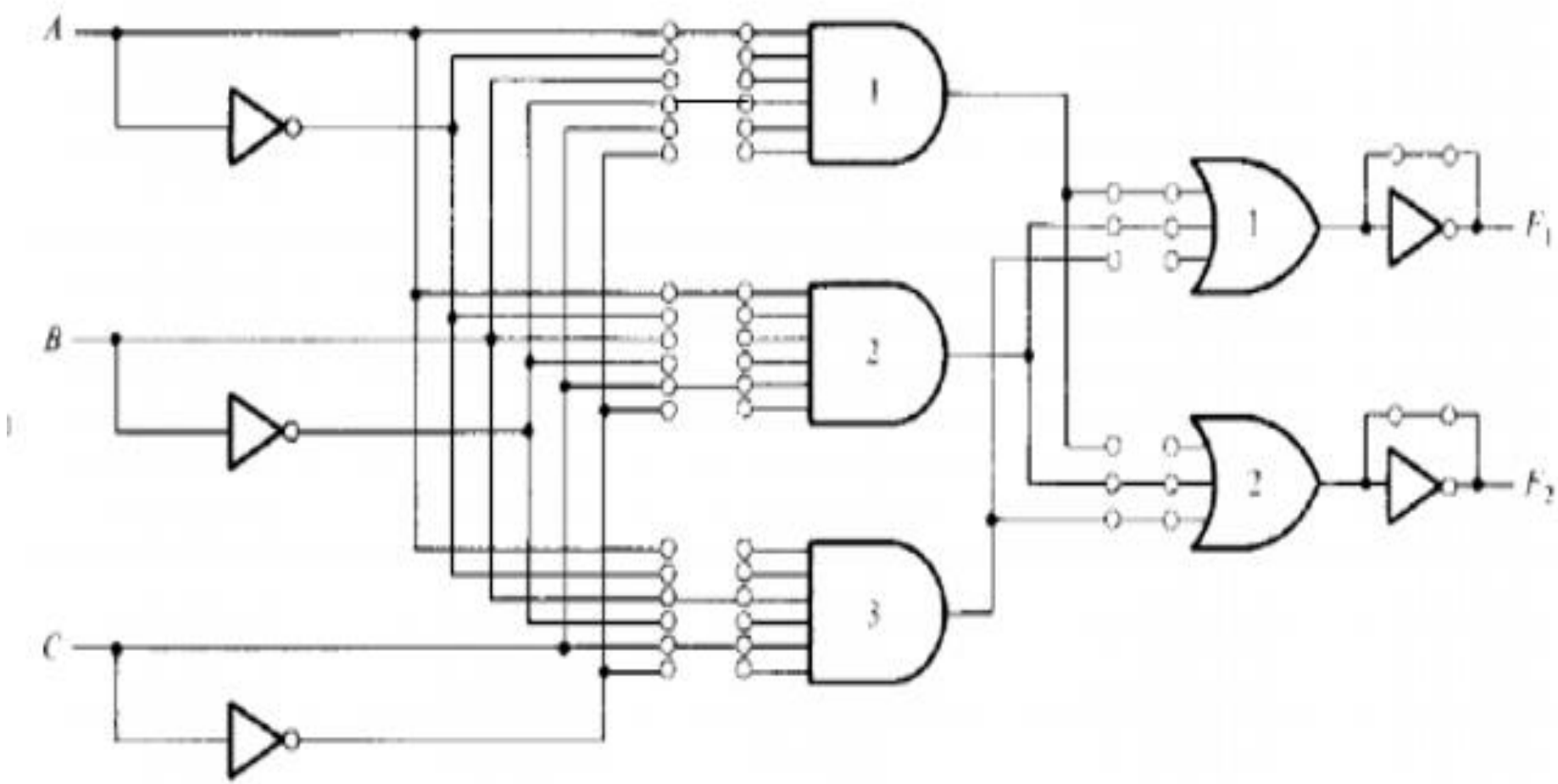


Fig: PLA with 3 inputs, 3 product terms, and 2 outputs

Programming the PLA means, we specify the paths in its AND-OR-NOT pattern. A typical PLA program table consists of three columns.

First column: lists the product terms numerically.

Second column: specifies the required paths between inputs and AND gates.

Third column: specifies the paths between the AND gates and the OR gates.

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

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

Table: PLA program table

For each product term, the inputs are marked with 1, 0 or - (dash).

- If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1.
- If it appears complemented (primed), the corresponding input variable is marked with a 0.
- If the variable is absent in the product term, it is marked with a dash.

Each product term is associated with an AND gate. The paths between the inputs and the AND gates are specified under the column heading inputs. A 1 in the input column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.

The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input. The paths between the AND and OR gates are specified under the column heading outputs. The output variables are marked with 1's for all those product terms that formulate the function. We have F1 = AB' + AC So F1 is marked with 1's for product terms 1 and 2 and with a dash for product term 3. Each product term that has a 1 in the output column requires a path from the corresponding AND gate to the output OR gate.

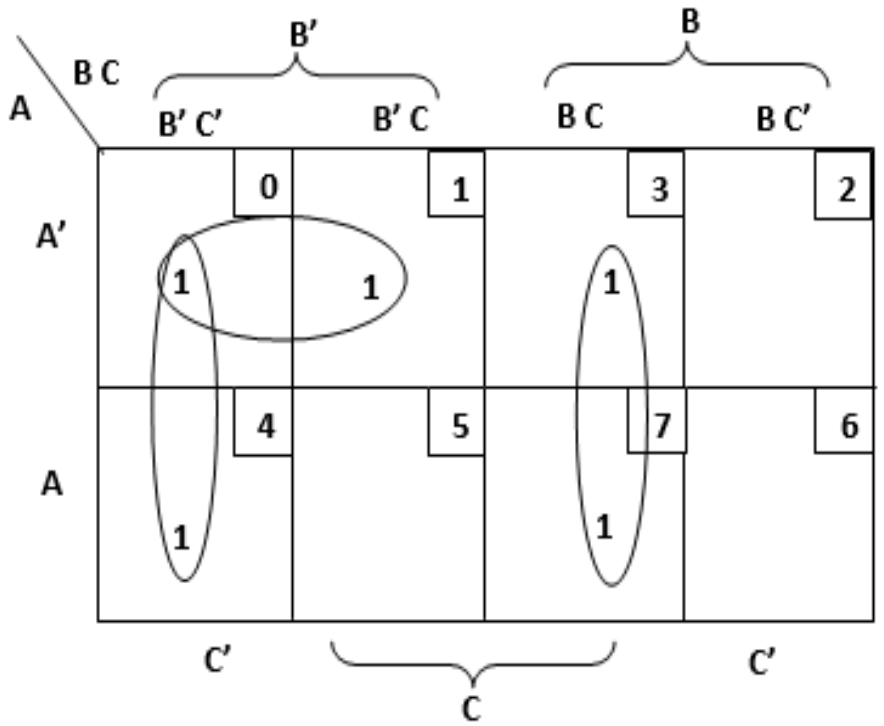
Example: Implement the given function with PLA:

$$F(A, B, C) = \sum(0, 1, 3, 4, 7)$$

Let us determine truth table from the given Boolean function:

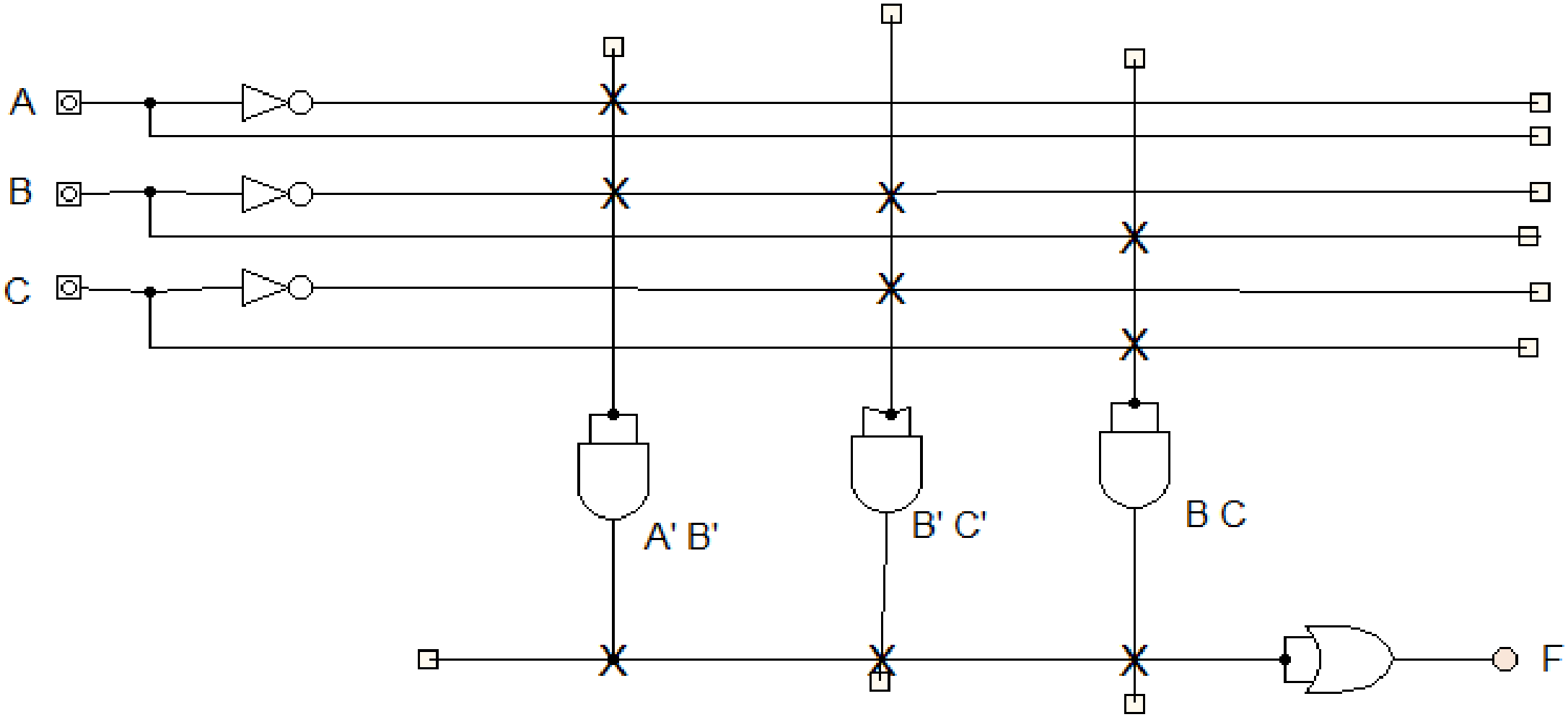
A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

K- Map for simplified expression:



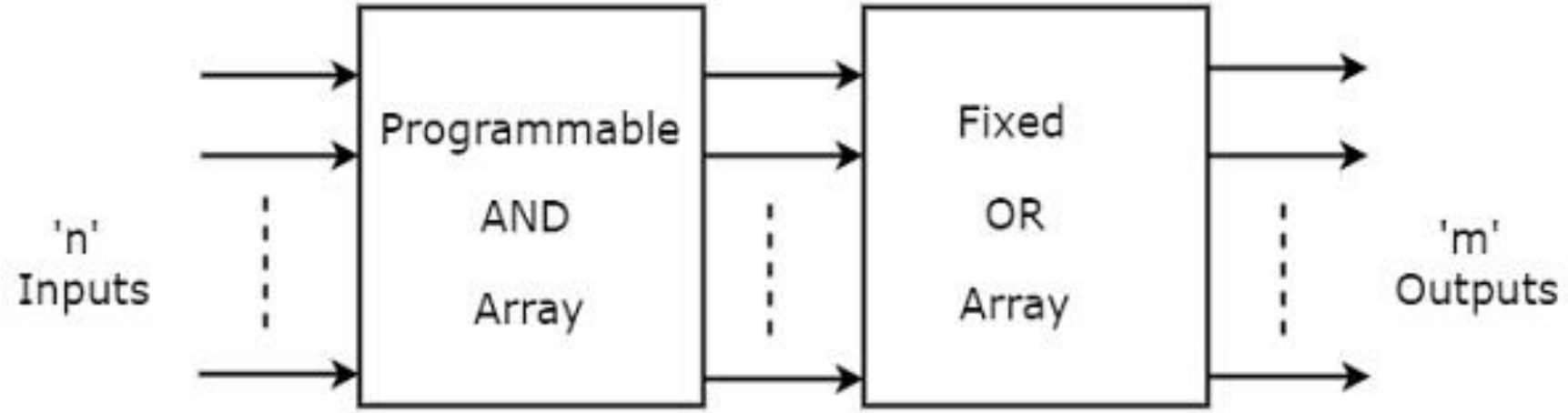
F = A' B' + B' C' + B C

Logic diagram implementation of F = A’ B’ + B’ C’ + B C with PLA



Programmable Array Logic PAL

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The **block diagram** of PAL is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Example

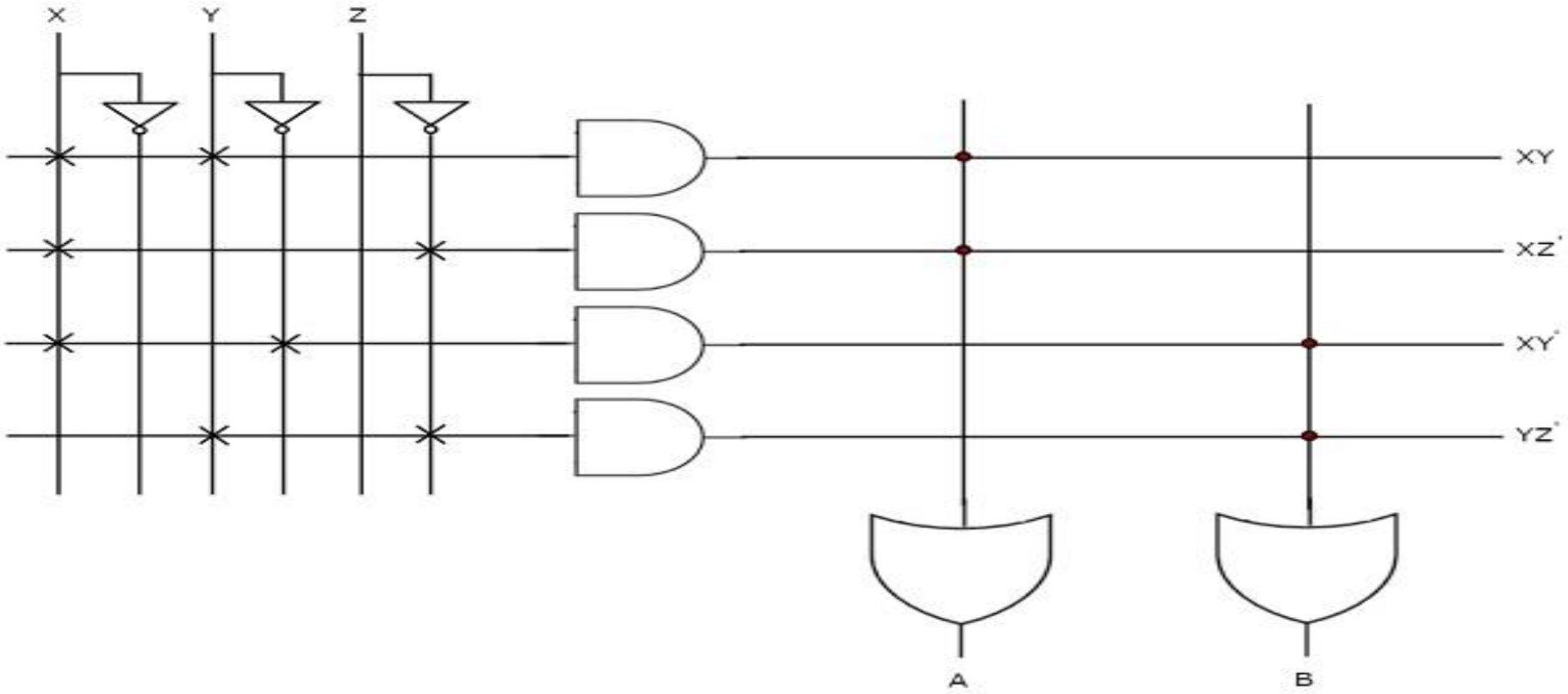
Let us implement the following **Boolean functions** using PAL.

$$A=XY+XZ'$$

$$B=XY'+YZ'$$

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions.

The corresponding **PAL** is shown in the following figure.



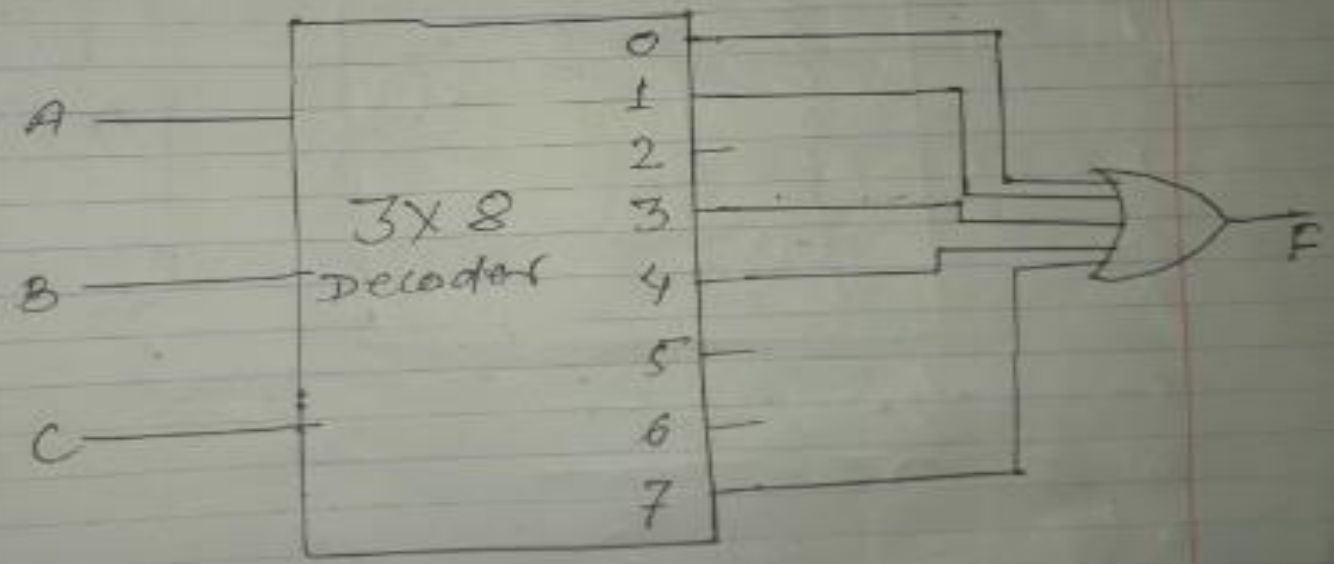
The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X′X′, Y, Y′Y′, Z & Z′Z′, are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate. The symbol ‘X’ is used for programmable connections.

Here, the inputs of OR gates are of fixed type. So, the necessary product terms are connected to inputs of each **OR gate**. So that the OR gates produce the respective Boolean functions. The symbol ‘.’ is used for fixed connections.

Q. Implement the following function
$F = \Sigma(0, 1, 3, 4, 7)$ using
i) Decoder
ii) Multiplexer
iii) PLA

Solution(i)

Since 3X8 decoder generates eight minterms for A, B and C. The OR gate for output sum forms the logical sum of minterms 0, 1, 3, 4, 7.



solution:1)

Implementing $F = \Sigma(0, 1, 3, 4, 7)$ using multiplexer.

Let $F(A, B, C) = \Sigma(0, 1, 3, 4, 7)$

Consider A as input, B and C as selection lines.

Truth Table

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

note:
8x1 Mux designed by 4x1 Mux

Implementation table:

	I_0	I_1	I_2	I_3
A'	(0)	(1)	2	(3)
A	(4)	5	6	(7)
	1	A'	0	1

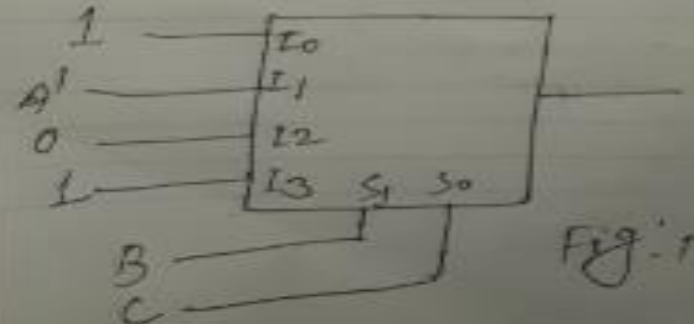


Fig: Multiplexer Implementation

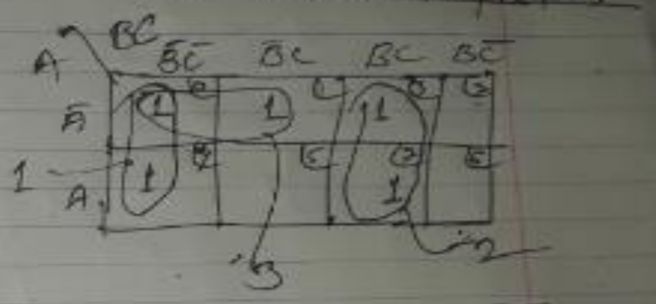
(ii) Implementation of given function

$F = \Sigma(0, 1, 3, 4, 7)$ using PLA.

Let us determine the truth table from given Boolean function.

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

K-Map simplification



$\therefore F = \bar{B}\bar{C} + BC + \bar{A}\bar{B}$

Logic diagram of PLA