UNIT 2

COMBINATIONAL LOGIC

2.1 INTRODUCTION

A combinational circuit consists of logic gates whose outputs at any time are determined directly from the present combination of input without regard to previous inputs. A combination circuit consists of input variables, logic gates and output variables. The logic gates accept signals from the inputs and generate signals to the outputs. This process transforms binary information from the given input data to the required output data. A block diagram of a combinational circuit is shown in **Figure 2.1**. It accepts '*n*' binary input variables and generate 'm' output variables depending on the logical combination of gates.

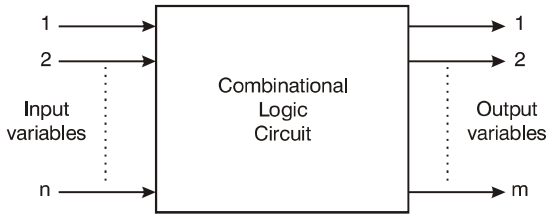


Fig. 2.1 : Block diagram of a combinational circuit

The possible representations for a combinational logic function:

- A truth table
- An algebraic sum of minterms, the canonical sum
- A minterm list using the $\sum$ notation
- An algebraic product of maxterms, the canonical product
- A maxterm list using the π notation.

2.2 DESIGN PROCEDURE

Any combinational circuit can be designed by following the design procedure given below:

- From the given word description of the problem, identify the number of input variables and required output.
- The variables input and output variables are assigned letter symbols.
- Draw a truth table such that it completely describes the operation of the circuit for different combinations of inputs.
- Obtain the Boolean expression for each output using either algebraic or K-map method.
- Obtain the logic diagram.

In practical design method, some constraints are considered:

- Minimum number of gates.

- ❖ Minimum number of inputs to a gate.
- ❖ Minimum propagation time of the signal through the circuit.
- ❖ Minimum number of interconnections.
- ❖ Limitations of the driving capabilities of each gate.

Example 2.1: Design a combination logic circuit with three input variables that will produce a logic 1 output when more than one input variables are logic 1.

Solution: Number of Input variables = 3

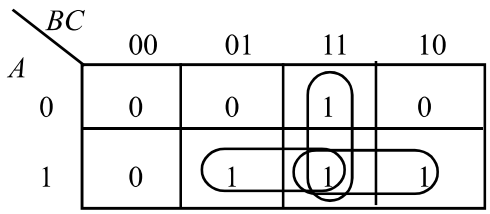
Number of Output variables = 1

Let assign the letter symbols *A, B* and *C* to three input variables and assign '*Y*' to one output variable. The relationship between input variables and output variable is tabulated in truth table as given in **Table 2.1**.

TABLE 2.1: Truth Table

A	*B*	*C*	*Y*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Using *K* map method, find the Boolean expression for *Y*.



$$Y = AB + AC + BC$$

Draw the logic diagram for $Y = AB + AC + BC$.

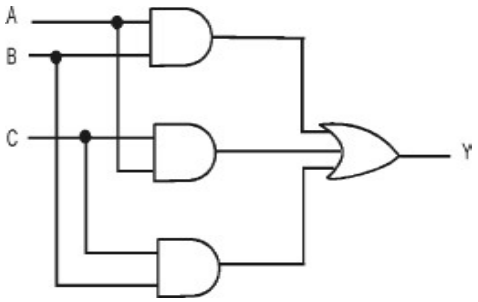


Fig. 2.2: Logic diagram

Example 2.2: Design a combinational logic circuit that has four inputs and one output. The output is high if both inputs A and *B* are high or both inputs *C* and *D* are high.

Solution: The relationship between input variables (*A, B, C, D*) and output variable (*Y*) is tabulated as shown in **Table 2.2.**

TABLE 2.2 : Truth Table

A	*B*	*C*	*D*	*Y*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Using *K* map obtain the Boolean expression for output variable *Y*

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

$Y = AB + CD$

Draw the logic diagram for $Y = AB + CD$.

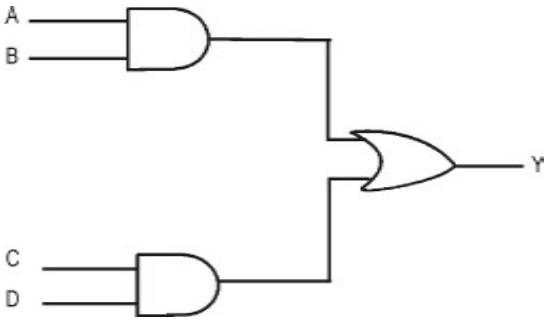
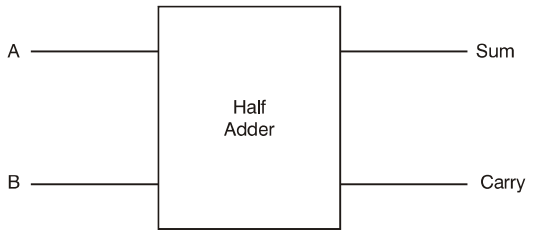


Fig. 2.3 : Logic diagram

2.3 HALF ADDER

"The half adder accepts two binary digits on its inputs and produces two binary digits on its outputs, a sum and a carry bit."

Figure 2.4(a) shows the logic symbol of a half-adder and **Figure 2.4(b)** shows the truth table for the half-adder.



(a) Logic symbol

Inputs		Outputs	
A	***B***	***S***	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	0	0	1

(b) Truth Table

Fig. 2.4: Half adder

The half adder follows the basic rules for addition:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

By using K-map, determine the expression for output variables (sum and carry out)

A \ B	0	1
0	0	(1)
1	(1)	0

Sum = $A\overline{B} + \overline{A}B$

$= A \oplus B$

A \ B	0	1
0	0	0
1	0	(1)

Carry = $C_{out} = AB$

The output carry is produced with an AND gate with A and B on the inputs and the output sum is produced with an EX-OR gate as shown in **Figure 2.5.**

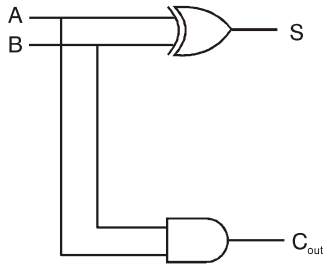


Fig. 2.5: Half adder logic diagram

2.4 FULL ADDER

"The full adder accepts three inputs-two input bits and input carry and generates sum output and output carry."

A half adder has only two inputs and there is no provision to add a carry coming from the lower order bits when multi addition is performed. For this purpose a full adder is used. A full adder has 3 inputs – A, B, C_{in} and two outputs – Sum (s) and carry out (C_{out}). **Figure 2.6(a)** shows the logic symbol and **Figure 2.6(b)** shows the truth table for full adder.

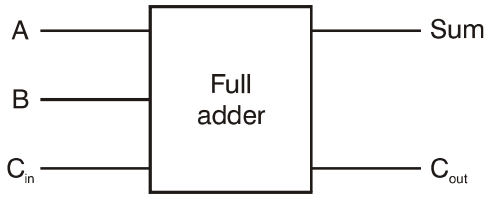


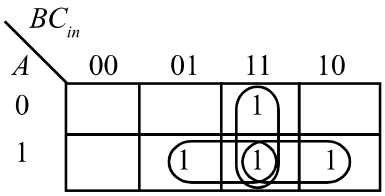
Fig. 2.6(a): Logic symbol

INPUTS			OUTPUTS	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig. 2.6(b): Truth Table

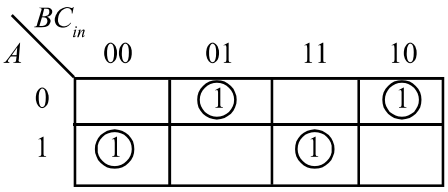
By using K-map simplification, determine the expression for output variables S and C_{out}.

(i) Expression for Carry



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

(ii) Expression for Sum



$$S = \overline{A}\,\overline{B}\,C_{in} + \overline{A}\,B\,\overline{C}_{in} + A\,\overline{B}\,\overline{C}_{in} + \overline{A}\,\overline{B}\,\overline{C}_{in} + ABC_{in}$$

$$= \overline{A}\left(\overline{B}\,C_{in} + B\overline{C}_{in}\right) + A\left(\overline{B}\,\overline{C}_{in} + BC_{in}\right)$$

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

Let $X = B \oplus C_{in}$, then $S = \overline{A}X + A\overline{X}$

$$= A \oplus X$$

Replacing X with $B \oplus C_{in}$, then

$$\boxed{\begin{aligned} S &= A \oplus B \oplus C_{in} \\ C_{out} &= AC_{in} + AB + BC_{in} \end{aligned}}$$

The logic diagram is constructed by logic gates as shown in **Figure 2.7**.

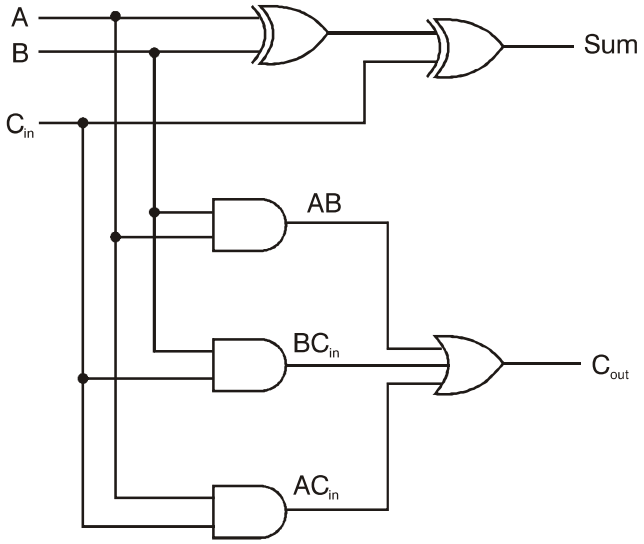


Fig. 2.7 : Full adder logic diagram

2.4.1 Implementation of full adder using Half adders

The full adder can also be implemented with two half-adders and one OR gate as shown in **Figure 2.8.**

For a half adder, Sum = $A \oplus B$

$$C_{out} = AB$$

For a full adder, Sum = $(A \oplus B) \oplus C_{in}$

$$\begin{aligned}
C_{out} &= AB + AC_{in} + BC_{in} \\
&= AB + AC_{in} + BC_{in}(A + \overline{A}) \\
&= AB + AC_{in} + ABC_{in} + \overline{A}BC_{in} \\
&= AB(1 + C_{in}) + AC_{in} + \overline{A}BC_{in} \\
&= AB + AC_{in} + \overline{A}BC_{in} \\
&= AB + AC_{in}(B + \overline{B}) + \overline{A}BC_{in} \\
&= AB + ABC_{in} + A\overline{B}C_{in} + \overline{A}BC_{in} \\
&= AB(1 + C_{in}) + A\overline{B}C_{in} + \overline{A}BC_{in} \\
&= AB + A\overline{B}C_{in} + \overline{A}BC_{in} \\
&= AB + C_{in}(A\overline{B} + \overline{A}B) \\
&= AB + C_{in}(A \oplus B)
\end{aligned}$$

Using these expressions draw the logic diagram for full adder.

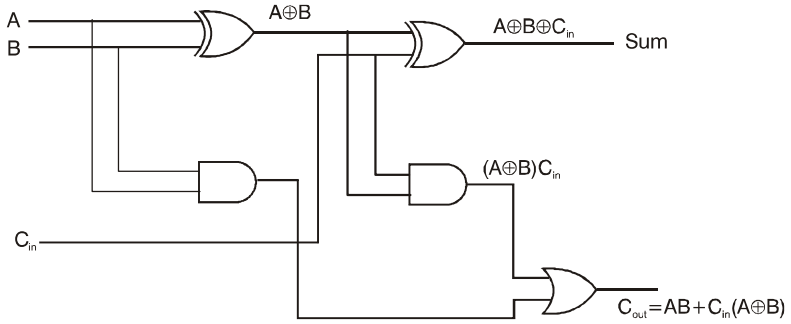


Fig. 2.8: Implementation of full adder

Notice in **Figure 2.8,** there are two half adders, connected as shown in the block diagram **Figure 2.9**, with their output carries ORed.

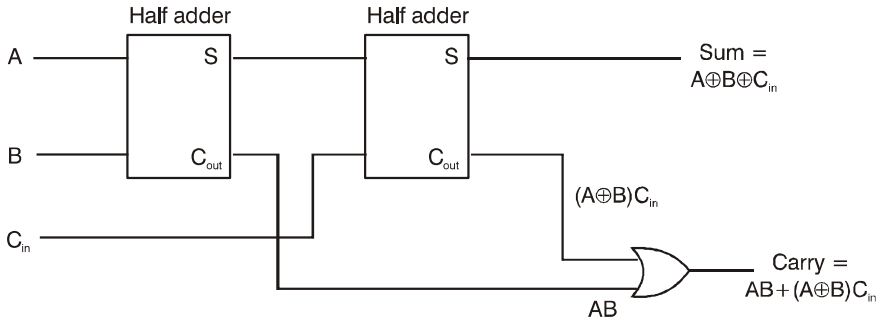
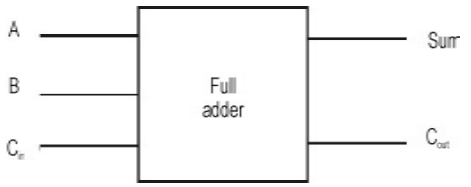


Fig. 2.9 : Implementation of full adder with half adders

2.5 HALF SUBTRACTOR

A half subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. Let us designate minuend bit as A and the subtrahend bit as B. The result of operation $A - B = D$ and the borrow is B_{out}. **Figure 2.10(a)** shows the logic symbol of a half subtractor and **Figure 2.10(b)** shows the truth table for the half-subtractor.



(a) Logic symbol

Inputs		Outputs	
A	***B***	***D***	B_{out}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

(b) Truth table

Fig. 2.10 : Half subtractor

The half subtractor follows the basic rules for subtraction:

$0 - 0 = 0$

$0 - 1 = 1$ with 1 borrow

$1 - 0 = 1$

$1 - 1 = 0$

By using K-map, determine the expression for output variables, Difference (D) and Borrow out (B_{out}).

A \ B	0	1
0	0	(1)
1	(1)	0

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

$= A \oplus B$

A \ B	0	1
0	0	(1)
1	0	0

$B_{out} = \overline{A}B$

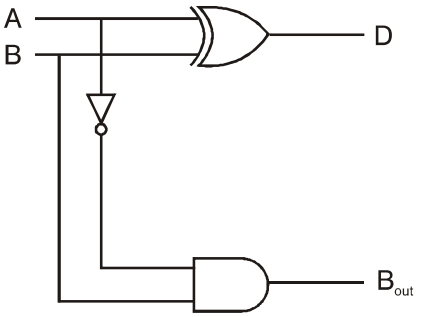


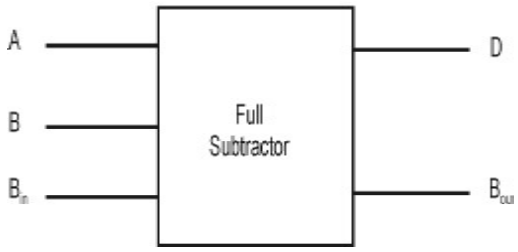
Fig. 2.11: Half subtractor

The difference output (D) is produced with an EX-OR gate and the borrow output (B_{out}) is produced with an AND gate with $\overline{A}$ and B as shown in **Figure 2.11**.

2.6 FULL SUBTRACTOR

"The full subtractor accepts three inputs – minuend, subtrahend and borrow from the previous stage and generates difference output and borrow output."

The full subtractor has 3 inputs-Minuend (A), Subtrahend (B) and borrow from the previous stage (B_{in}) and two outputs-difference (D) and borrow out (B_{out}). **Figure 2.12(a)** shows the logic symbol and **Figure 2.12(b)** shows the truth table for full subtractor.



(a) Logic symbol

Inputs			Outputs	
A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(b) Truth Table

Fig. 2.12 : Full Subtractor

K-map Simplification

(i) Expression for 'D':

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

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

(ii) Expression for Borrow (B_{out})

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

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

The Boolean function for D is simplified as,

$$\begin{aligned} D &= \overline{A}\,\overline{B}\,B_{in} + \overline{A}\,B\,\overline{B}_{in} + ABB_{in} + A\overline{B}\overline{B}_{in} \\ &= B_{in}\left(\overline{A}\,\overline{B} + AB\right) + \overline{B}_{in}\left(\overline{A}B + A\overline{B}\right) \\ &= B_{in}\left(\overline{A \oplus B}\right) + B_{in}\left(A \oplus B\right) \\ &= \left(B_{in} \cdot \overline{B}_{in}\right) + \left(A \oplus B \cdot \overline{A \oplus B}\right) \\ &= B_{in} \oplus \left(A \oplus B\right) \end{aligned}$$

The logic diagram of full subtractor is constructed by logic gates is shown in **Figure 2.13**.

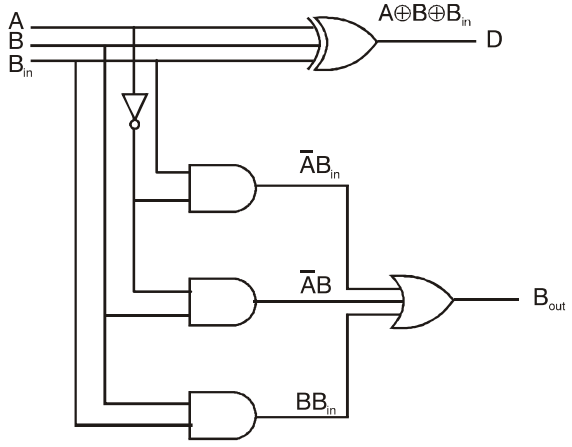


Fig. 2.13 : Full subtractor-logic diagram

2.6.1 Implementation of Full Subtractor using Half Subtractor

A full subtractor can also be implemented with two half subtractors and one OR gate as shown in **Figure 2.14**.

For a half subtractor, $D = A \oplus B$

$$B_{out} = \overline{A}\,B$$

For a full subtractor, $D = A \oplus B \oplus B_{in}$

$$\begin{aligned}
B_{out} &= \overline{A}B + \overline{A}B_{in} + BB_{in} \\
&= \overline{A}B + \overline{A}B_{in}\left(B + \overline{B}\right) + BB_{in} \\
&= \overline{A}B + \overline{A}BB_{in} + \overline{A}\,\overline{B}\,B_{in} + BB_{in} \\
&= \overline{A}B\left(1 + B_{in}\right) + \overline{A}\,\overline{B}B_{in} + BB_{in} \\
&= \overline{A}B + BB_{in} + \overline{A}\,\overline{B}B_{in} \\
&= \overline{A}B + BB_{in}\left(A + \overline{A}\right) + \overline{A}\,\overline{B}B_{in} \\
&= \overline{A}B + ABB_{in} + \overline{A}BB_{in} + \overline{A}\,\overline{B}\,B_{in} \\
&= \overline{A}B\left(1 + B_{in}\right) + ABB_{in} + \overline{A}\,\overline{B}\,B_{in} \\
&= \overline{A}B + ABB_{in} + \overline{A}\,\overline{B}\,B_{in} \\
&= \overline{A}B + B_{in}\left(AB + \overline{A}\,\overline{B}\right)
\end{aligned}$$

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

Using these expressions, draw the logic diagram for full subtractor as given below:

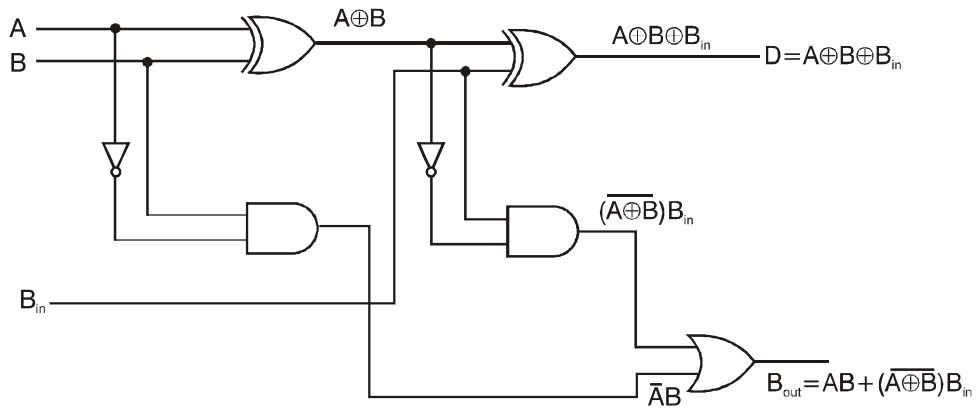


Fig. 2.14 : Implementation of full subtractor

Notice in **Figure 2.14**, there are two half subtractors connected as shown in the block diagram **Figure 2.15**, with their output borrows ORed.

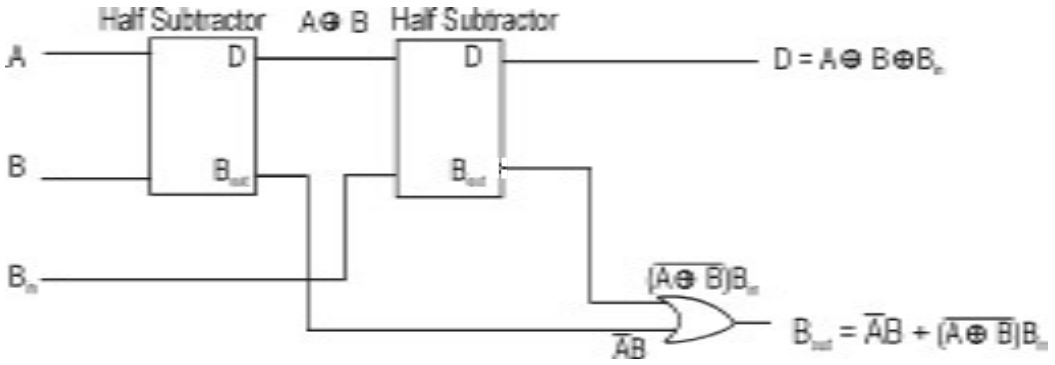


Fig. 2.15 : Implementation of full subtractor using half subtractors

2.7 PARALLEL BINARY ADDERS

A single full adder is capable of adding two 1 bit numbers and an input carry. To add binary numbers with more than one bit, additional full-adders are required. In order to add two binary numbers, a full adder is required for each bit in the number. Thus for two-bit numbers, we need two full-adders. Similarly for the addition of four-bit numbers, we need four full-adders and so on.

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. The least significant bits (LSB) of the two binary numbers being added go into the right most full adder whereas the higher order bits are applied as shown to the successive higher-order adders. The most significant bits (MSB) of the two binary numbers are applied to the left-most full-adder. The block diagram of 4 bit parallel adder is shown in **Figure 2.16**.

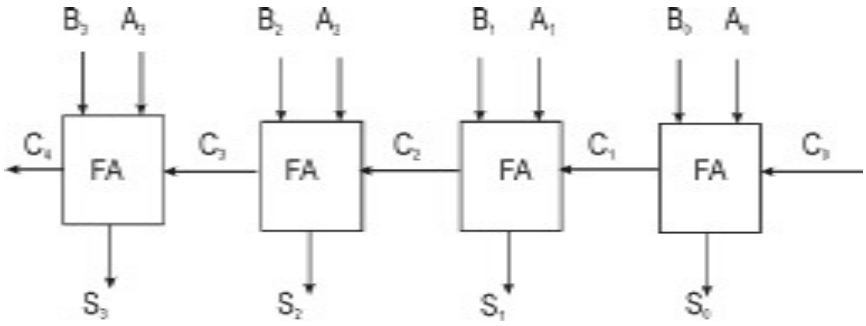


Fig. 2.16 : 4 bit Parallel Adder

The input carry is labelled as C_0 and the output carry is labelled as C_4.

There are several parallel adders that are available as ICs. **Table 2.3** shows the most commonly used 4 bit parallel adders. In the IC package, a 4 bit parallel adder consisting 4 terminals for the augend bits, 4 terminals for addend bits, 4 terminals for the sum bits and 2 terminals for the input and output carries. The logic symbol of a 4 bit parallel adder is shown in **Figure 2.17**.

TABLE 2.3: IC 4 Bit Parallel Adder

Family	*IC*
TTL	7483A
LSTTL	4LS83A
CMOS	74HC283

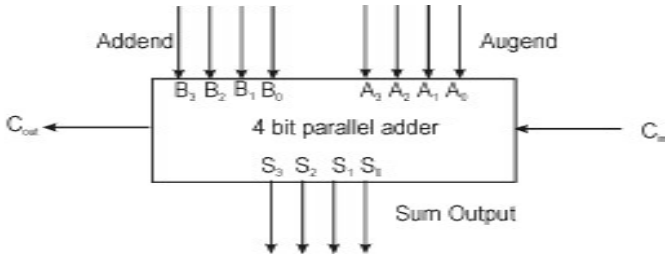


Fig. 2.17 : Logic symbol of a 4-bit parallel adder

8 bit Parallel adder: In order to accomplish addition of large binary numbers, two or more IC adders can be connected together, i.e., cascaded. **Figure 2.18**. Shows two 74LS83A adders connected to add two 8 bit numbers, $A_7\,A_6\,A_5\,A_4\,A_3\,A_2\,A_1\,A_0$ and $B_7\,B_6\,B_5\,B_4\,B_3\,B_2\,B_1\,B_0$.

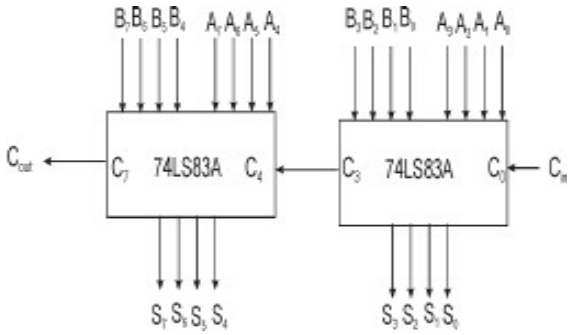


Fig. 2.18 : Cascading of two 74L383A

2.8 PARALLEL SUBTRACTOR

The subtraction of binary numbers can be done here by means of 2's complement method. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. In the parallel subtractor, the 1's complement can be implemented with inverters added with each data input B and a one can be added to the sum through the input carry as shown in **Figure 2.19.**

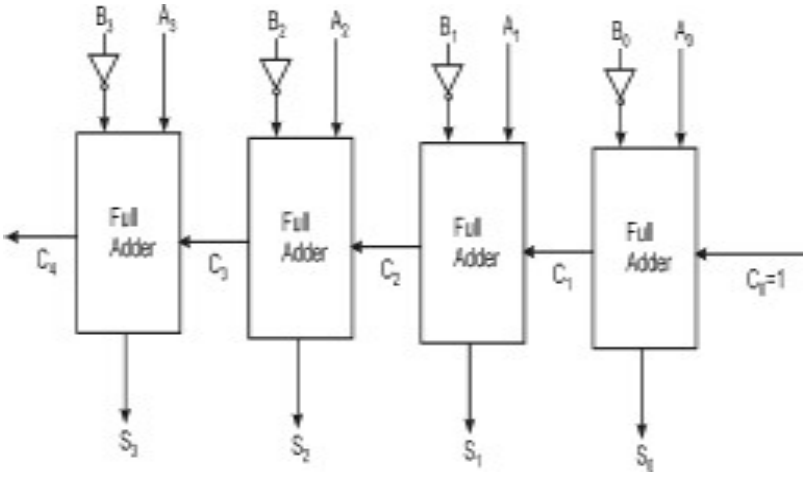


Fig. 2.19 : 4 Bit Parallel Subtractor

2.9 BINARY ADDER/SUBTRACTOR

The 4 bit binary adder/subtractor circuit is shown in **Figure 2.20**. It performs the operations of both addition and subtraction. It has two 4 bit input $A_0\,A_1\,A_2\,A_3$ and $B_0\,B_1\,B_2\,B_3$. The $\overline{\text{ADD}}$ /SUB line is the control line, connected with input carry C_0 of the least significant bit of the full adder, is used to perform the operations of addition and subtraction.

The addition and subtraction operations can be combined into one circuit with one common binary adder. This is done by including an EX-OR gate with each full- adder. When $\overline{\text{ADD}}$ /SUB = 0, the circuit is an adder and when $\overline{\text{ADD}}$ /SUB = 1, the circuit is a subtractor. Each EX-OR gate receives input $\overline{\text{ADD}}$ /SUB and one of the inputs of B.

ADDER

When $\overline{\text{ADD}}$ /SUB = 0, the operation is $B \oplus 0 = 0$. The full-adder receives the value of B and the input carry $C_0 = 0$. Thus the circuit performs the addition operation, $A + B$.

SUBTRACTOR

When $\overline{\text{ADD}}$ /SUB = 1, the operation is $B \oplus 1 = \overline{B}$. The full adder receives the value of $\overline{B}$ and the input carry $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry (C_0). Thus the circuit performs the subtraction operation, i.e.,

$$A + (2\text{'s complement of } B) = A - B.$$

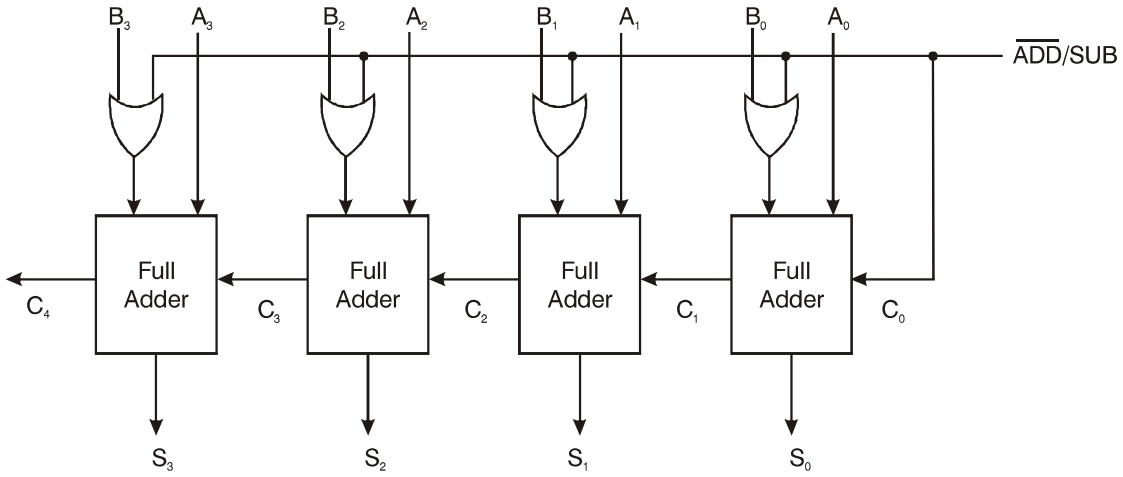


Fig. 2.20 : 4 Bit Adder/Subtractor

2.10 SERIAL ADDER

In the serial adder, the addition operation is done by bit-by-bit. The serial adder requires simpler circuit than a parallel adder. The speed of operation of serial adder is lower than the parallel adder. The operation of serial adder as follows:

Two shift registers A and B are used to store the numbers to be added serially. A single full adder is used to add one pair of bits at a time along with the carry. The D-flipflop is used to store the carry output of the full adder, so that it can be added to the next significant position of the numbers in the registers. The contents of the shift registers shift from left to right and their outputs starting from A_0 and B_0 are fed into a single full adder along with the output of the D-flipflop upon application of each clock pulse. The sum output of the full adder is fed to MSB bit (S_3) of the sum register. For each succeeding clock pulse, the contents of the both shift registers are shifted once to the right and new carry bit are transferred to sum register and D- flipflop respectively. This process continues untill all the pairs of bits are added. The diagram of a serial adder is shown in **Figure 2.21** and a four bit serial adder is shown in **Figure 2.22.**

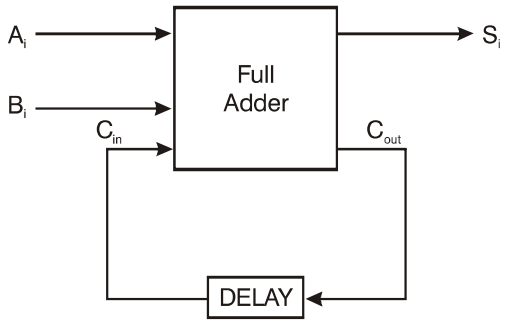


Fig. 2.21 : Serial Adder

2.13 BINARY MULTIPLIER

Multiplication operation can be carried out by (i) Multipliers using partial product addition and shifting and (ii) Parallel multipliers.

Multiplier using Shift Method

To understand the multiplication process using shift method, consider the multiplication of two 4-bit binary numbers 1010 and 1011, as an example.

	1 0 1 0	→	Multiplicand
x	1 0 1 1	→	Multiplier
	1 0 1 0	→	Partial product 1
	1 0 1 0	→	Partial product 2
	0 0 0 0	→	Partial product 3
	1 0 1 0	→	Partial product 4
	1 1 0 1 1 1 0		

From the above multiplication process, one can easily understand that if the multiplier bit is 1, then the multiplicand is simply copied as a partial product; if the multiplier bit is 0, then the partial product is 0. Whenever a partial product is obtained, it is shifted one bit to the left of the previous partial product. This process is continued until all the multiplier bits are checked, and then the partial products are added to this multiplication process, i.e. multiplication by partial product addition and shifting, can be implemented using the block diagram shown in figure.

In the shown figure, the 4-bit multiplier is stored in register Y ($Y_3Y_2Y_1Y_0$); the 4-bit multiplicand is stored in register M ($M_3M_2M_1M_0$), and the X register ($X_4X_3X_2X_1X_0$) is initially cleared to 00000. Here, to perform multiplication, the least significant bit of the multiplier bit (Y_0) is checked whether it is 0 or 1. If $Y_0 = 1$, the number in the multiplicand register (M) is added with the least significant 4-bits of X register ($X_3X_2X_1X_0$;X_4 is to store carry in addition process) and the combined X and Y register is shifted to the right by 1 bit. If $Y_0 = 0$, the combined X and Y register is shifted to the right by 1 bit without performing any addition. This process has to be repeated four times to perform 4-bit multiplication. Now, the multiplication result ($R_7R_6R_5R_4R_3R_2R_1R_0$) will be available in X and Y registers ($X_3X_2X_1X_0Y_3Y_2Y_1Y_0$).

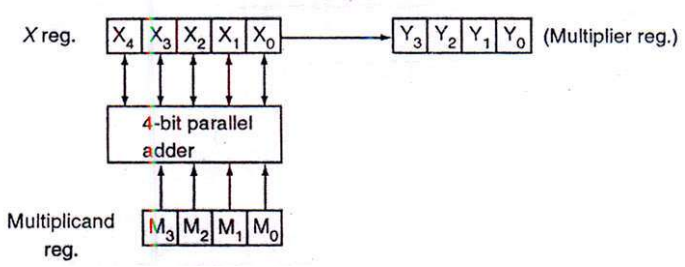


Fig.2.24: 4-bit binary multiplier using shift method

Parallel Multiplier

The 4-bit multiplier using shift method requires 4 cycles of addition and shifting operations, but it requires only a single 4-bit parallel adder. The speed of multiplication process can be increased considerably in parallel multiplier at the extra cost of increased hardware. The circuit diagram for a 4-bit parallel mutliplier is shown in figure.

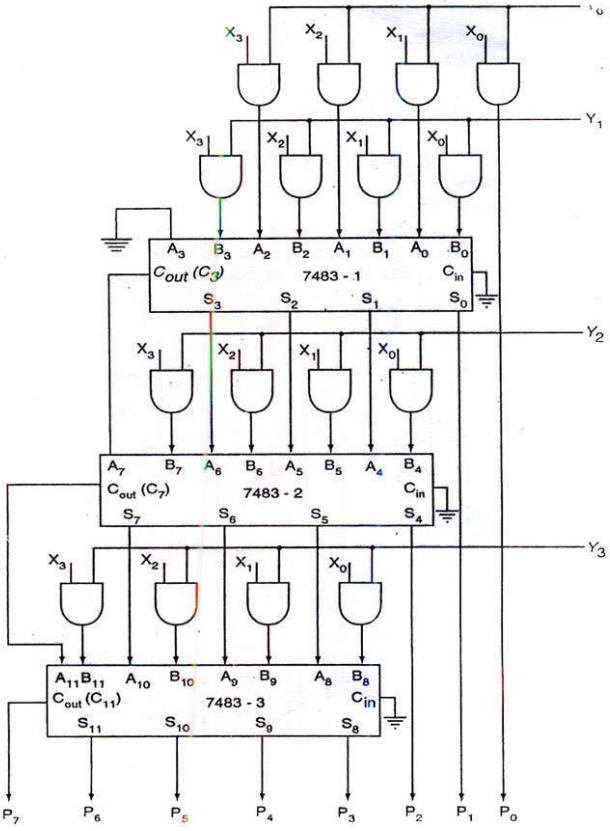


Fig.2.25:4-bit parallel multiplier

It requires three 4-bit parallel binary adders and 16 numbers of 2-input AND gates. Here, each group of 4 AND gates is used to obtain partial products while 4-bit parallel adders are used to add the partial products. Since the generation of partial products and their additions are performed in parallel in the group of AND gates and 4-bit adders respectively, the multiplication result ($P_7P_6P_5P_4P_3P_2P_1P_0$) will be available at the output immediately after the propagation delay in the multiplier circuit.

The operation of the parallel multiplier can be understood in a better manner from the symbolic form of binary multiplication process shown in figure.

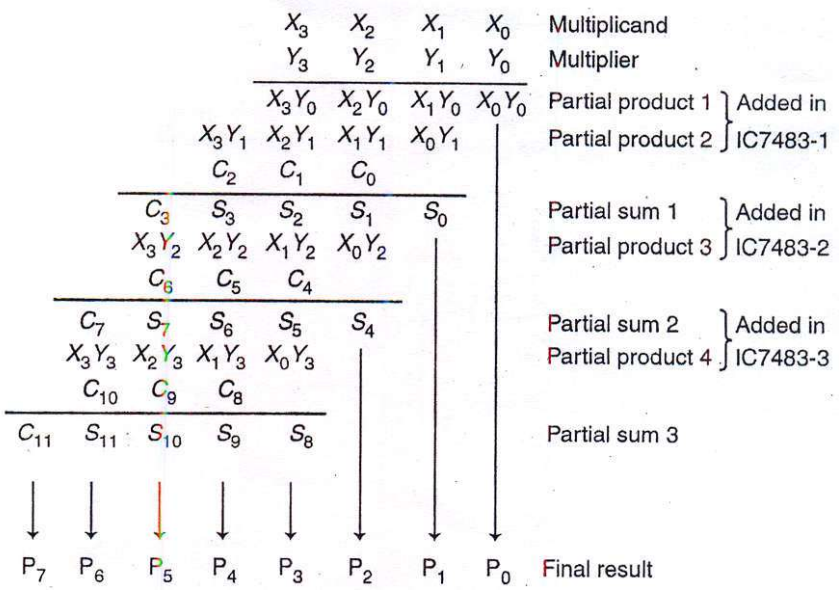


Fig.2.26: Symbolic form of binary multiplication process

2.14 CARRY LOOK AHEAD ADDER

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

For example, addition of two numbers (0011 + 0101) gives the result as 1000. Addition of the LSB position produces a carry into the second position. This carry when added to the bits of the second position, produces a carry into the third position. This carry when added to the bits of the third position, produces a carry into the last position. The sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous positions. i.e., the adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example if each full adder is considered to have a propagation delay of 30 ns, then S_3 will not react its correct value until 90 ns after LSB carry is generated. Therefore total time required to perform addition is 90 + 30 = 120 ns.

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

Carry generate, $G_i = A_i B_i$

Carry propagate, $P_i = A_i \oplus B_i$

Sum, $S_i = P_i \oplus C_i$

$= A_i \oplus B_i \oplus C_i$

Carry, $C_{i+1} = G_i + P_i C_i$

The structure of one stage of a carry look ahead adder is shown in **Figure 2.27.**

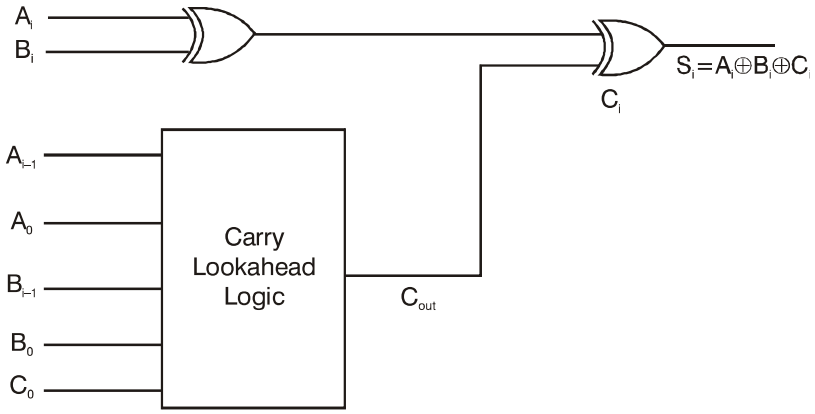


Fig. 2.27 : Structure of one stage of a carry look ahead adder

G_i (carry generate) generates carry if both A_i and B_i are 1 regardless of the input carry.

P_i (carry propagate) propagates carries if atleast one of its addend bits is 1. The carry output of a stage can now be written in terms of the generate and propagate signals:

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

To eliminate carry ripple, expand the C_i term for each stage and multiply out to obtain a 2-level AND-OR expression. Using this technique, we can obtain the following carry equations for the first four adder stages are obtained.

C_1

$C_2 = G_1 + P_1 C_1$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 C_1)$

$= G_2 + P_2 G_1 + P_2 P_1 C_1$

$C_4 = G_3 + P_3 C_3 = G_3 + P_3 (G_2 + P_2 G_1 + P_2 P_1 C_1)$

$= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_1$

From these equations, it can be seen that C_4 does not have to wait for C_3 and C_2 to propagate. Infact, C_4 is propagated at the same time as C_2 and C_3. **Figure 2.28**shows the implementation of carry equations for C_2, C_3 and C_4 using AND-OR logic.

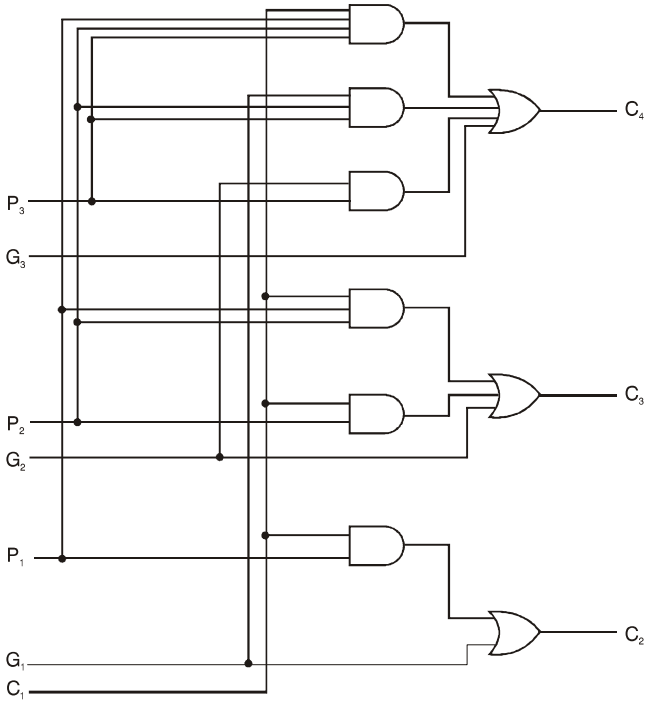


Fig. 2.28 : Logic diagram of a look ahead carry generator

2.15 BCD ADDER

The BCD adder is used to add two BCD digits and produces a sum in BCD digit. We know that, BCD number means 0 to 9 (10 digits) and are represented in the binary form 0000 to 1001.

BCD numbers cannot be greater than 9 and 10 is represented in BCD as 0001 0000. In BCD addition, the sum is greater than 9(1001), we obtain a non-valid BCD representation. The addition of binary 6(0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

Let us consider BCD addition of 5 and 10,

$5 \Rightarrow 0101$

$10 \Rightarrow 1010$ (+)

$15 \Rightarrow 1111$

1111 is an invalid BCD number. It can be corrected by the addition of 6(0110) to the invalid BCD number.

$15 \Rightarrow \quad 1111$

$6 \Rightarrow \quad 0110$ (+)

$0001\ 0101 \Rightarrow 15$ (BCD)

Thus we can summarize the BCD addition procedure as follows:

1. Add two BCD numbers using ordinary binary addition.
2. If the result is equal to or less than 9, no correction is needed. The sum is in correct BCD form.
3. If the result is greater than 9 or if a carry is generated from the result, the result is invalid and the correction is needed.
4. To correct the invalid sum add 6(0110) to the result. If a carry results from this addition, add it to the next higher order BCD digit.

   Implementation of BCD adder using logic circuit as follows:

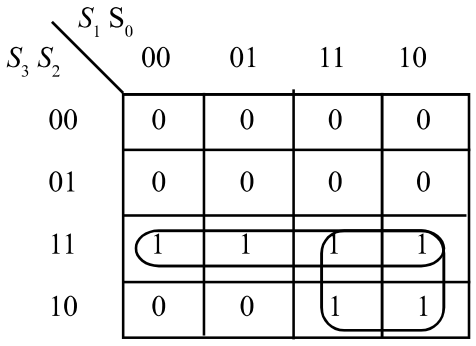
   - 4-bit binary adder for initial addition.
   - Logic circuit to detect sum greater than 9.
   - Second 4-bit binary adder to add 6(0110) to the result if result is greater than 9 or carry is 1.

The logic circuit to detect result greater than 9 can be determined by simplifying the boolean expression of given **Table 2.4.**

TABLE 2.4: Truth Table

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

Draw the K-map for the above truth table and find out the expression



$$Y = S_3 S_2 + S_3 S_1$$

This Binary to BCD correction expression is shown in **Figure 2.29.**

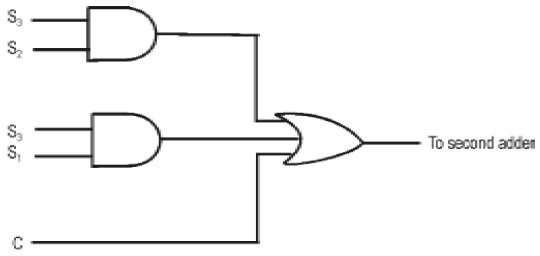


Fig. 2.29 : Binary to BCD Correction

In first binary adder, two BCD numbers together with input carry are added. When the result is equal to zero (i.e., result $\leq$ 9 or $C_{out} = 0$), nothing is added to the result. When the result is one (i.e., result $\geq$ 9 or $C_{out} = 1$) binary 0110 is added to the result through second binary adder. The C_{out} generated by second adder can be ignored, since it supplies information already available at the output carry terminal. The single digit BCD adder is shown in **Figure 2.30.** Multiple digit BCD adders can be constructed by cascading as many single digit adders as needed. The BCD carry-out from each stage would be connected to the carry-in of the next higher order stage.

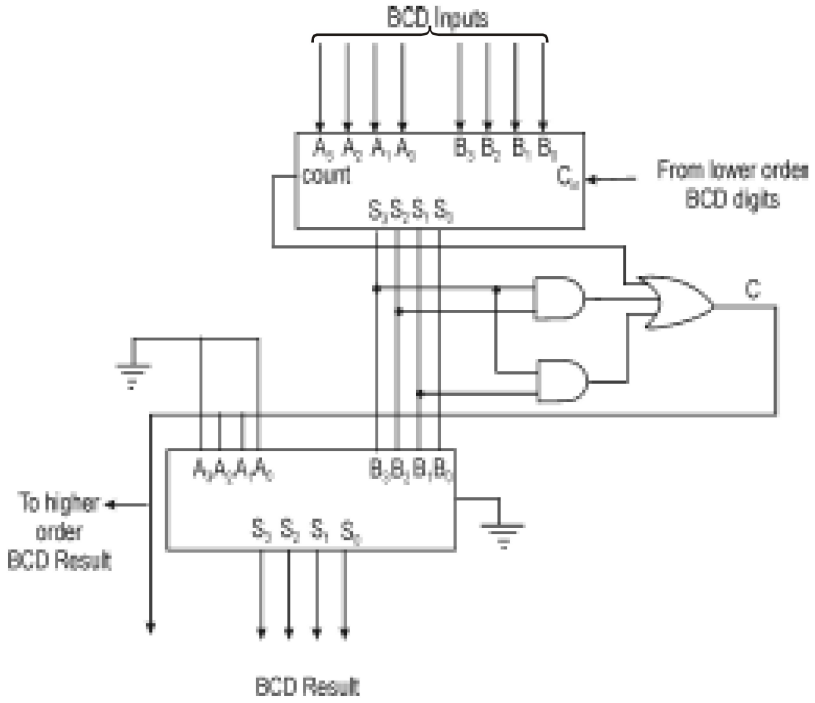


Fig. 2.30 : Single Digit BCD Adder

2.16 MAGNITUDE COMPARATOR

A magnitude comparator is a combinational circuit that compares the magnitude of two numbers *A* and *B* and generates one of the following outputs:

$$A = B$$

$$A < B$$

$$A > B .$$

The block diagram of n-bit magnitude comparator is shown in **Figure 2.31**.

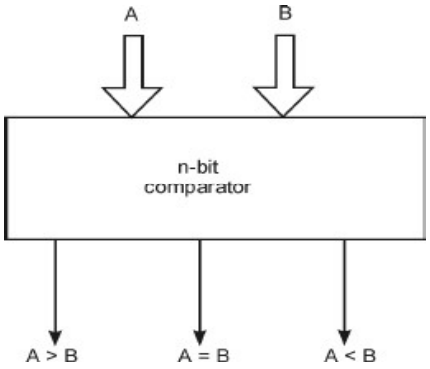
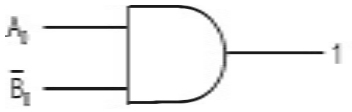


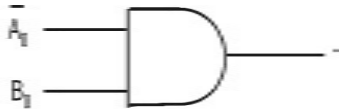
Fig. 2.31 : Block diagram of n-bit magnitude comparator

To implement the magnitude comparator, the EX-NOR gates and AND gates are used. The EX-NOR gate is used to find whether the two binary digits are equal or not, and the AND gates are used to find whether a binary digit is less than or greater than another binary digit as shown in **Figure 2.32**. **Figure 2.33**shows a single bit magnitude comparator.

$\mathbf{A_0 = B_0}$



$\mathbf{A_0 > B_0}$
$\mathbf{(A_0 = 1, B_0 = 0)}$



$\mathbf{A_0 < B_0}$
$\mathbf{(A_0 = 0, B_0 = 1)}$

Fig. 2.32 : Comparator Operation

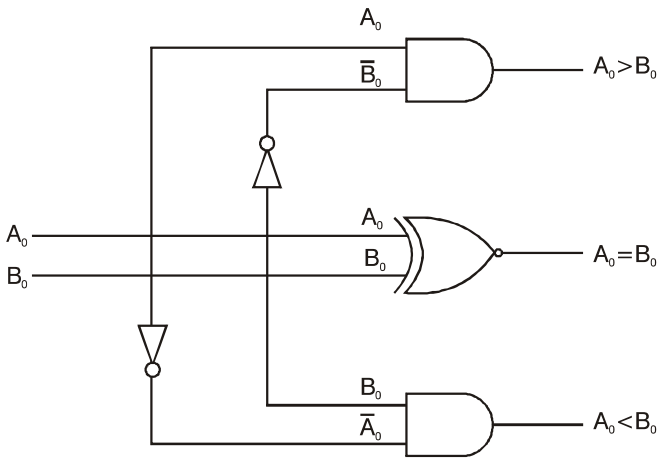


Fig. 2.33: Single bit magnitude comparator

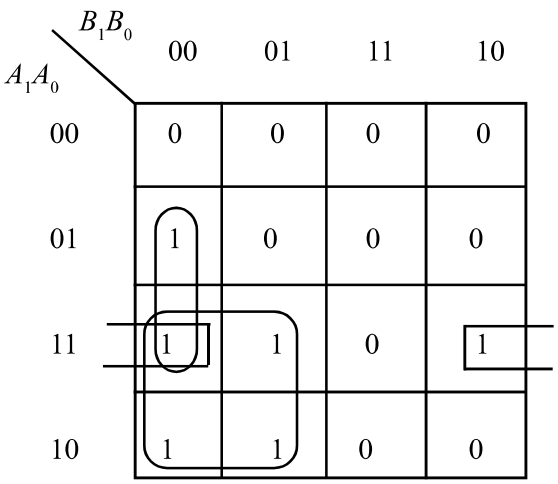
The same principle can be extended to an n-bit magnitude comparator. The design of a 2 bit magnitude comparator is as follows:

The truth table for 2 bit comparator is given in **Table 2.5**.

TABLE 2.5 : Comparator Truth Table

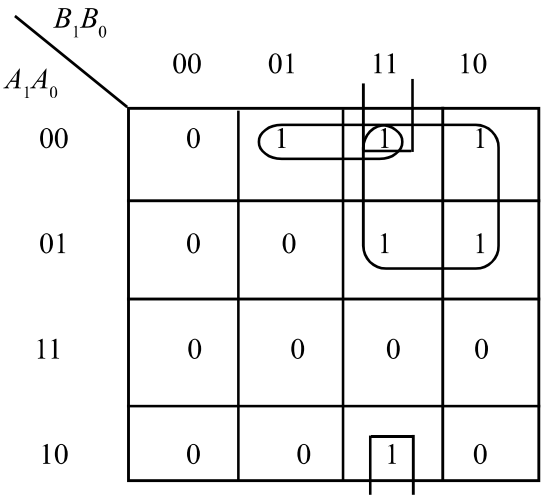
Inputs				*Outputs*		
A_1	A_0	B_1	B_0	A > B	A = B	A < B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

For $A > B$



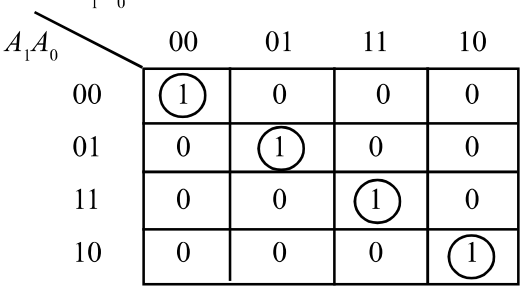
$A_0\overline{B}_1\overline{B}_0 + A_1\overline{B}_1 + A_1A_0\overline{B}_0$

For $A < B$



$\overline{A}_0B_1B_0 + \overline{A}_1B_1 + \overline{A}_1\overline{A}_0B_0$

For $A = B$



$$\overline{A}_1\,\overline{A}_0\,\overline{B}_1\,\overline{B}_0 + \overline{A}_1\,A_0\,\overline{B}_1\,B_0 + A_1A_0B_1B_0 + A_1\,\overline{A}_0\,B_1\,\overline{B}_0$$

$$= \overline{A}_1\,\overline{B}_1\left(\overline{A}_0\,\overline{B}_0 + A_0\,B_0\right)\overline{A}_1\,B_1\left(A_0B_0 + \overline{A}_0\,\overline{B}_0\right)$$

$$= \left(\overline{A}_1\,\overline{B}_1 + A_1\,B_1\right)\left(\overline{A}_0\overline{B}_0 + A_0\,B_0\right) = \left(\overline{A_1 \oplus B_1}\right)\left(\overline{A_0 \oplus B_0}\right)$$

The expressions for determining whether,

$A > B$ is $A_0\overline{B}_1\overline{B}_0 + A_1\overline{B}_1 + A_1A_0\overline{B}_0$

$A < B$ is $\overline{A}_0B_1B_0 + \overline{A}_1B_1 + \overline{A}_1\overline{A}_0B_0$

$A = B$ is, $\left(\overline{A_1 \oplus B_1}\right)\left(\overline{A_0 \oplus B_0}\right)$

The implementation of 2 bit magnitude comparator using EX-NOR and AND gates using the above expressions is shown in **Figure 2.34**.

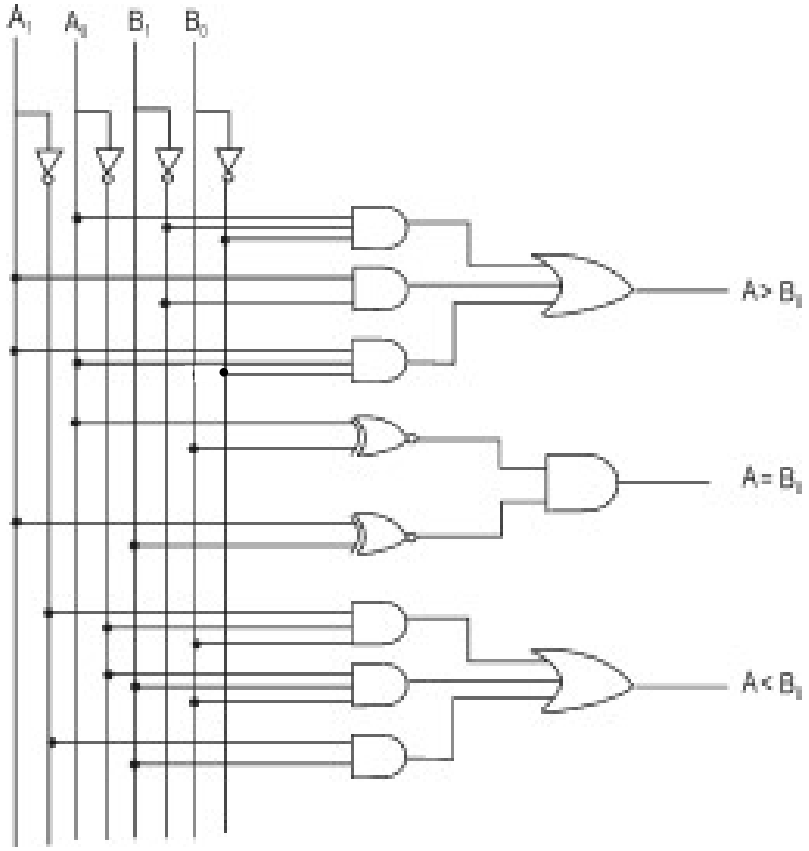


Fig. 2.34 : 2-bit Magnitude comparator

2.17 PARITY GENERATOR AND CHECKER

A parity bit is used for the purpose of detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1s either odd or even. The message including the parity bit is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a parity generator and the circuit that checks the parity bit in the transmitter is called a parity generator and the circuit that checks the parity in the receiver is called a parity checker.

Even parity means an 'n' bit input has an even number of 1s. For example, 110101 has even parity because it contains four 1s.

Odd parity means an 'n' bit input has an odd number of 1s. For example, 110100 has odd parity because it contains three 1s.

2.19 DECODERS

Decoder is a digital device that converts coded information into another code or non-coded form. It is a multi-input multi-output logic circuit. The number of outputs is greater than the number of inputs ($n : 2^n$). The encoded information is presented as 'n' inputs producing 2^n possible outputs as shown in **Figure 2.42**.

If the number of inputs and outputs are equal in a digital system, then it can be called as code converters. (BCD to XS-3 code, Binary to BCD code, Gray to Binary Code converters, etc.)

Fig. 2.42: Decoder

2.19.1 Binary Decoder

A binary decoder has 'n' bit binary input and a one activated output out of 2^n outputs. A binary decoder is used when it is necessary to activate exactly one of 2^n outputs based on an n-bit input value. **Figure 2.43**shows 2 to 4 line decoder. 2 inputs are decoded into 4 outputs, each output representing one of the minterms of the 2 input variables. The **Table 2.10** shows the truth table for 2 to 4 line decoder.

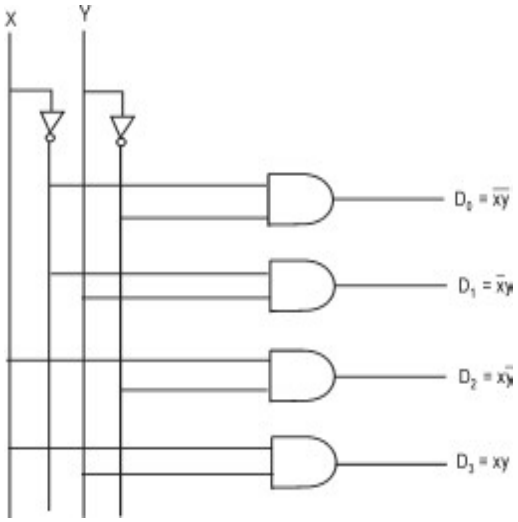


Fig. 2.43: 2 to 4 line decoder

TABLE 2.10: Truth Table

Inputs		*Outputs*			
X	*Y*	D_0D_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

2.19.2 3-to-8 line Decoder

A 3-to-8 line decoder has three inputs (*x, y, z*) and eight outputs (D_0–D_7). Based on the 3 inputs one of the 8 outputs is selected.

The logic diagram of 3-to-8 line decoder is shown in **Figure 2.44** and the truth table of this decoder is given in **Table 2.11**. The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. This decoder is used for binary-to-octal conversion. The input variables may represent a binary number and the outputs will represent the eight digits in the octal number system. The output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

TABLE 2.11: Truth Table

Inputs			*Outputs*							
x	*y*	*z*	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

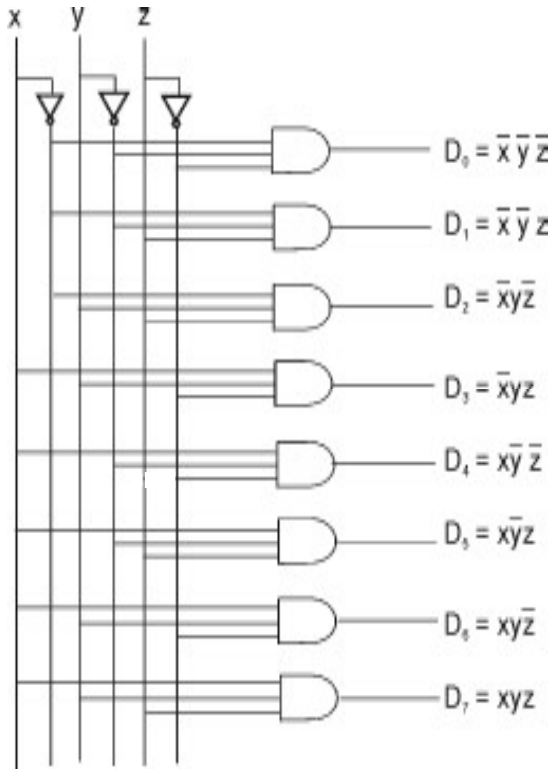


Fig. 2.44: 3-to-8 line decoder

2.19.3 1-of-16 Decoder

The 1-of-16 decoder is shown in **Figure 2.45**. The 4 inputs ABCD are the control bits. It has 16 output lines and only 1 of the 16 output lines is high. For instance, when $ABCD = 0001$, only the Y_1 AND gate has all inputs high, therefore only the Y_1 output is high. If $ABCD = 0100$, only the Y_4 AND gate has all input high, therefore only the Y_4 output goes high. This circuit is also known as 4-to-16 line decoder.

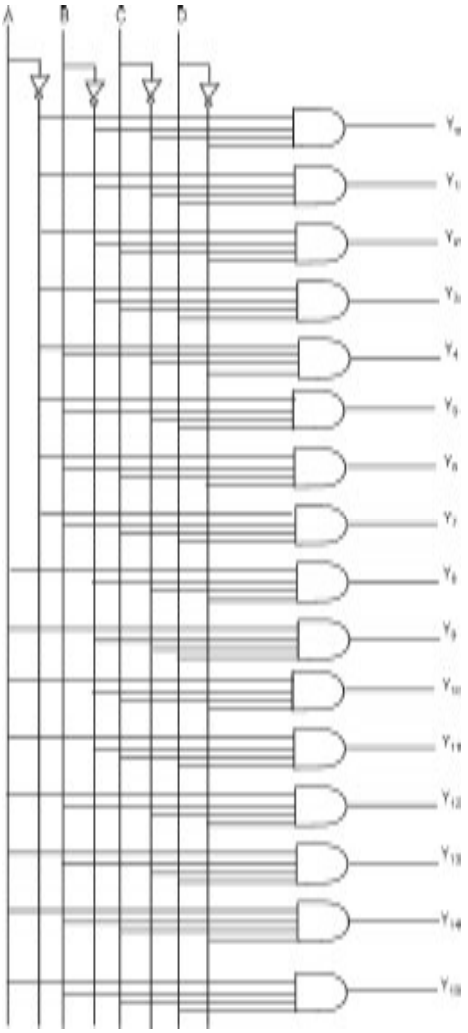


Fig. 2.45: 1-of-16 [4-to-16 line] Decoder

2.19.4 BCD-to-Decimal Decoder

The **Figure 2.46**shows a 1-of-10-decoder because only one of the 10 output lines is high. For example, when $ABCD = 0011$, only the Y_3 AND gate has all high inputs, therefore only the Y_3 output is high. If $ABCD$ changes to 1000, only the Y_8 AND gate has all high inputs, therefore only the Y_8 output goes high. The $ABCD$ possibilities are from 0000 to 1001 (9). The high output always equals the decimal equivalent of the input BCD digit. For this reason, this circuit is also called a BCD-to-Decimal converter.

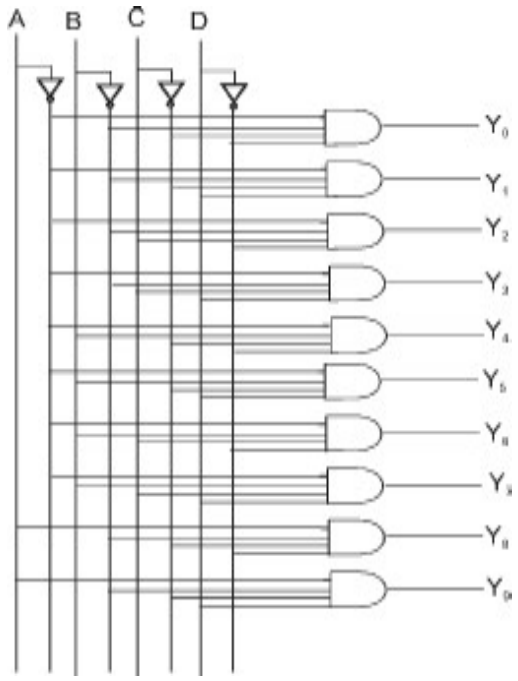


Fig. 2.46: BCD-to-Decimal Decoder

2.20 ENCODERS

An encoder is a digital circuit that performs the inverse operation of a decoder. It is a combinational logic circuit, that output logic circuit, that output lines generate the binary code corresponding to the input value.

It has 2^n input lines and 'n' output lines. The octal-to-binary encoder has $8(2^3)$ inputs, one for each of the octal digits and three ($n = 3$) outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time; otherwise the circuit has no meaning.

2.20.1 Octal-to-Binary encoder

The octal-to-binary encoder truth table is given in **Table 2.12**. The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6 or 7 and output x is 1 for digits 4, 5, 6 or 7. These conditions can be expressed by the following output Boolean functions:

$$z = D_1 + D_3 + D_5 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$x = D_4 + D_5 + D_6 + D_7$$

The octal-to-binary encoder is implemented for these Boolean functions using OR gates. The octal-to-binary encoder is shown in **Figure 2.47**.

TABLE 2.12 : Truth Table

Inputs								*Outputs*		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	*x*	*y*	*z*
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

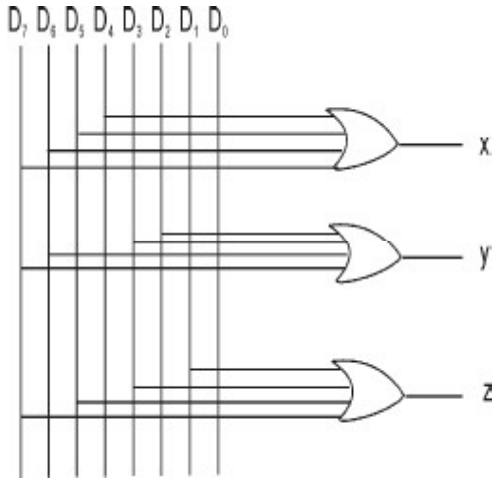


Fig. 2.47: Octal-to-Binary Encoder

2.20.2 Priority Encoder

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a 4 input priority encoder is given in **Table 2.13**.

Input D_3 has the highest priority. When $D_3 = 1$, the output XY = 11. D_2 has the next priority level. If $D_2 = 1$, $D_3 = 0$, the output $xy = 10$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher-priority inputs are 0 and so on down the priority level.

The output V (Valid-output indicator) = 1 only when one or more inputs are equal to 1. $V = 0$, if all inputs are 0 and the other two outputs (X and Y) of the circuit are not used.

TABLE 2.13 : Truth Table of a Priority Encodes

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

Although the **Table 2.13** has only five rows, when each don't care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example the third row in the table with X100 represents minterms 0100 and 1100. The don't care condition is replaced by 0 and 1 as shown in **Table 2.14.**

TABLE 2.14: Modified Truth Table

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

For *x*

D_2D_1 \ D_3D_2	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

$$x = D_2 + D_3$$

For *y*

D_0D_1 \ D_3D_2	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

$$y = D_3 + D_1\,\overline{D}_2$$

For *V*

$D_0\,D_1$ \ $D_2\,D_3$	00	0 1	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$V = D_0 + D_1 + D_2 + D_3$$

The simplified Boolean expressions for the priority encoder are obtained from the K-maps as follows:

$$x = D_2 + D_3$$

$$y = D_3 + D_1\,\overline{D}_2$$

$$v = D_0 + D_1 + D_2 + D_3$$

The priority encoder is implemented in **Figure 2.48** according to the above Boolean functions.

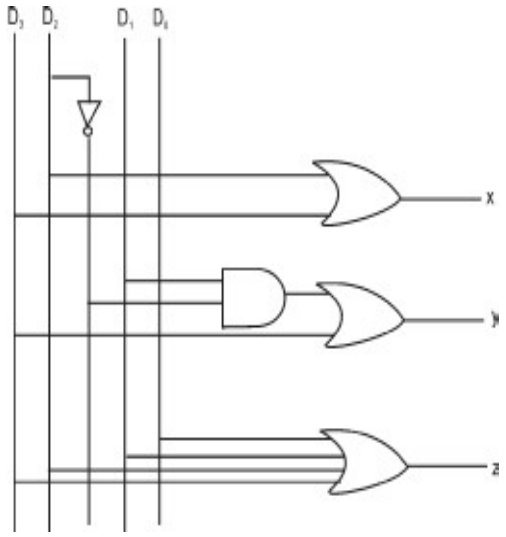


Fig. 2.48: 4 input priority encoder

2.21 MULTIPLEXERS

Multiplex means "many into one". A multiplexer is a combinational circuit with many inputs but only one output. By applying control signals, we can steer any input to the output. It has '*n*' input signals, '*m*' control signals and 1 output signal. Multiplexer is called as **data selector** or because the output bit depends on the input data bit that is selected. The block diagram of multiplexer is shown in **Figure 2.49**.

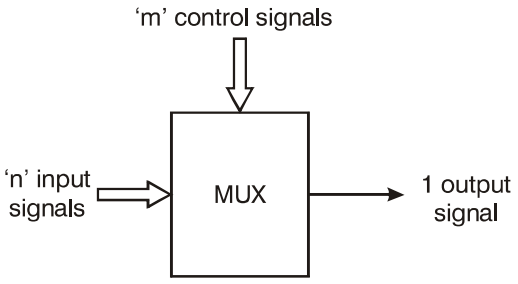


Fig. 2.49

2.21.1 4-to-1 Line Multiplexer

A 4-to-1 line multiplexer has four (n) input lines, two (m) select lines and one output line. The selection (control) lines decide the number of input lines. If the number of 'n' input lines is equal to 2^m, then 'm' select lines are required to select one of the 'n' input lines.

The logic symbol of a 4-to-1 multiplexer is shown in **Figure 2.50**. If has 4 input lines (I_0 to I_3), two select lines (S_0, S_1) and a single output line. The function table of 4-to-1 multiplexer is shown in **Table 2.15**.

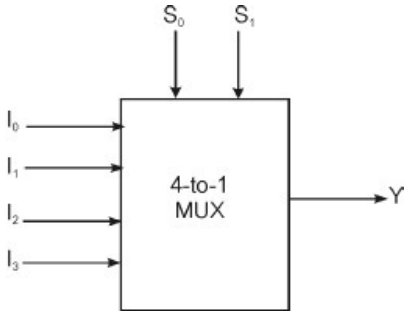


Fig. 2.50: Logic Symbol

TABLE 2.15 : Function Table

Select Lines		**Output**
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

1. $Y = I_0$ **iff** $S_1 = 0$, $S_0 = 0$

   $\therefore\ Y = I_0\overline{S}_1\overline{S}_0 = I_0 \cdot 1 \cdot 1 = I_0$

2. $Y = I_1$ **iff** $S_1 = 0, S_0 = 1$

   $\therefore Y = I_1\overline{S}_1 S_0 = I_1$, when $S_1S_0 = 01$

3. $Y = I_2$ **iff** $S_1 = 1$, $S_0 = 0$

   $\therefore Y = I_2 S_1\overline{S}_0 = I_2$ when $S_1S_0 = 10$

4. $Y = I_3$ **iff** $S_1 = S_0 = 1$

   $\therefore\ Y = I_3S_1S_0 = I_3$ when $S_1S_0 = 11$

The final expression for the data output,

$$Y = I_0\overline{S}_1\overline{S}_0 + I_1\overline{S}_1 S_0 + I_2 S_1\overline{S}_0 + I_3 S_1 S_0$$

Using this expression, the 4-to-1 multiplexer can be implemented using gates as shown in **Figure 2.51**.

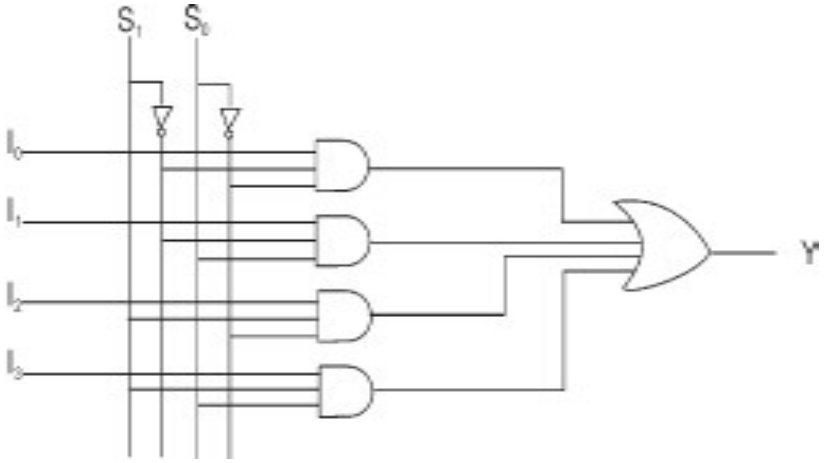


Fig. 2.51: Logic Diagram of 4-to-1 Multiplexer

2.21.2 16-to-1 Multiplexer

The 16-to-1 multiplexer has 16 data input lines (D_0–D_{15}), a single output line (Y) and 4 select lines (A, B, C, D) to select one of the 16 input lines. The truth table for a 16-to-1 multiplexer is shown in **Table 2.16**.

For example, when $ABCD = 0000$, the upper AND gate is enabled while all other AND gates are disabled. Therefore data bit D_0 is transmitted to the output.

$Y = D_0$

If $D_0 = 0$, $Y = 0$

$D_0 = 1$, $Y = 1$ i.e., Y depends only on the value of D_0

When $ABCD = 1111$, $Y = D_{15}$.

Thus the control bits (A, B, C, D) determines which of the input data bits is transmitted to the output.

Figure 2.52 shows the logic diagram of 16-to-1 multiplexer.

TABLE 2.16 : Truth Table of 16-to-1 MUX

Enable	*Select Inputs*				*Output*
$\overline{E}$	S_3	S_2	S_1	S_0	Y
0	0	0	0	0	$\overline{D}_0$
0	0	0	0	1	$\overline{D}_1$
0	0	0	1	0	$\overline{D}_2$
0	0	0	1	1	$\overline{D}_3$
0	0	1	0	0	$\overline{D}_4$
0	0	1	0	1	$\overline{D}_5$

0	0	1	1	0	$\overline{D}_6$
0	0	1	1	1	$\overline{D}_7$
0	1	0	0	0	$\overline{D}_8$
0	1	0	0	1	$\overline{D}_9$
0	1	0	1	0	$\overline{D}_{10}$
0	1	0	1	1	$\overline{D}_{11}$
0	1	1	0	0	$\overline{D}_{12}$
0	1	1	0	1	$\overline{D}_{13}$
0	1	1	1	0	$\overline{D}_{14}$
0	1	1	1	1	$\overline{D}_{15}$
1	X	X	X	X	1

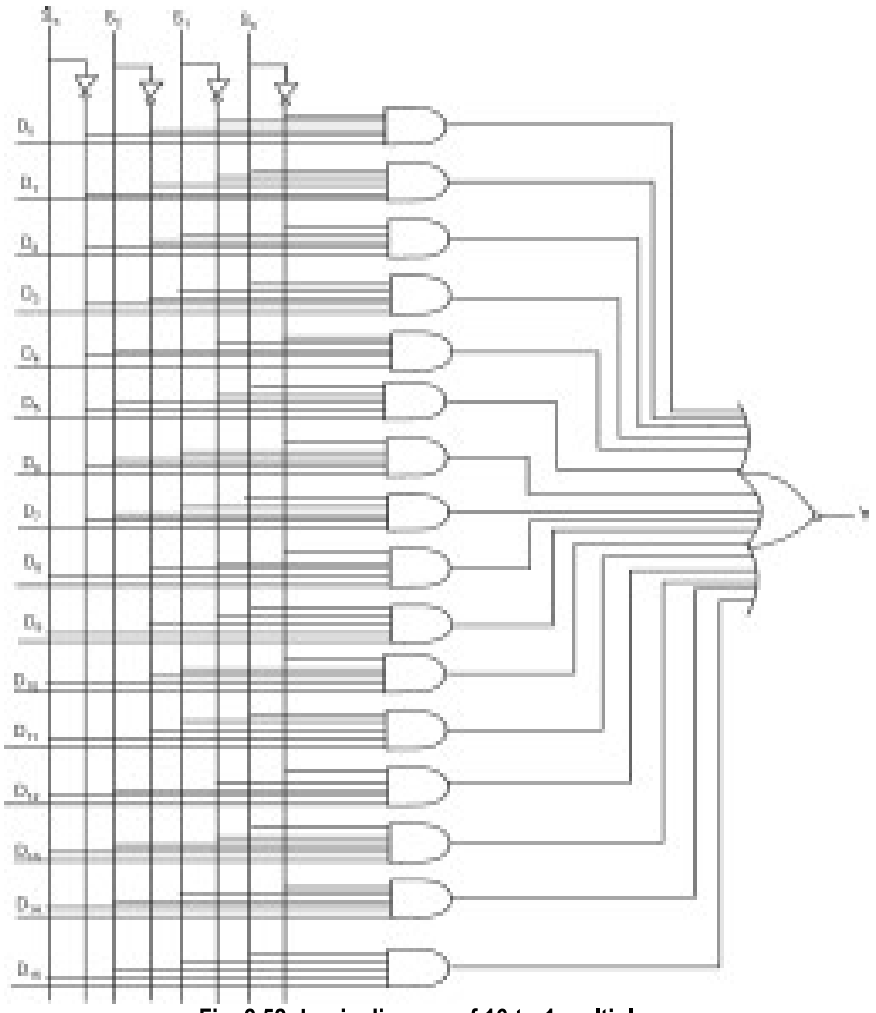


Fig. 2.52: Logic diagram of 16-to-1 multiplexer

We can implement a 16-to-1 multiplexer using two 8-to-1 multiplexer as shown in **Figure 2.53**.

To select one of the 16 inputs, 4 select lines ($S_3S_2S_1S_0$) are required. Among the 4 select lines ($S2\ S_1S_0$) are connected with 3 select inputs of both 4 to 1 multiplexers. S_3 is connected directly to $\overline{E}$ (Enable) input of MUX1 and it is connected through an NOT gate to $\overline{E}$ input of MUX2. Therefore, when $S_3 = 0$, MUX1 is selected and the inputs (D_0 to D_7) are multiplexed to the output and MUX2 is disabled. When $S_3 = 1$, the MUX 1 is disabled while MUX 2 is enabled and the inputs (D_8 to D_{15}) are multiplexed to the output.

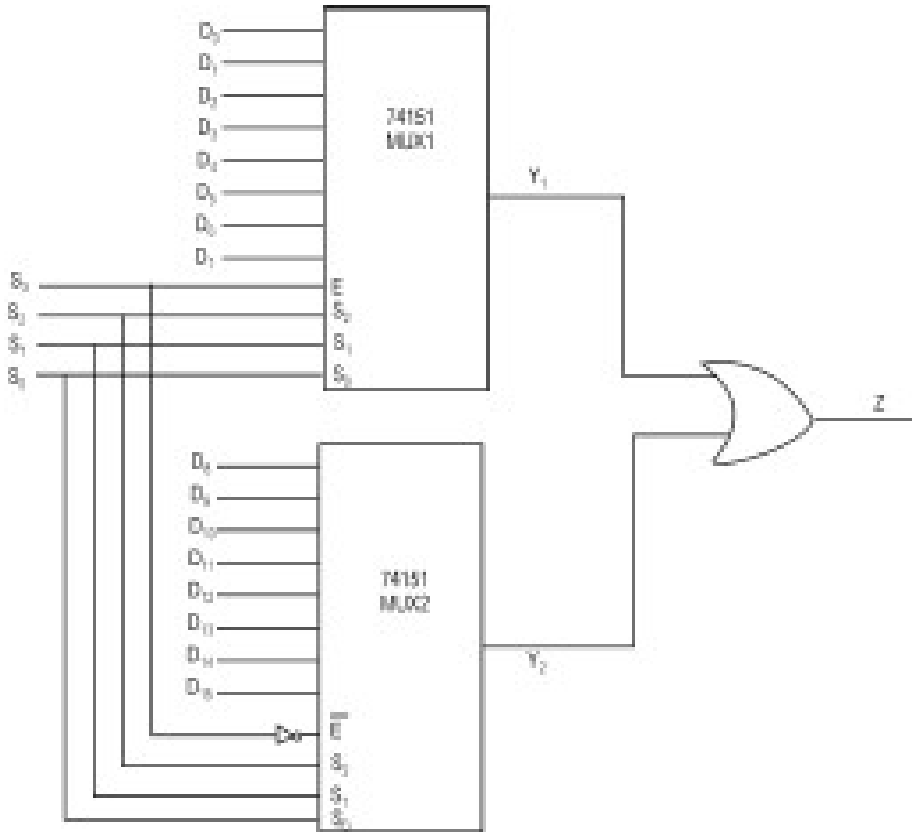


Fig. 2.53: 16-to-1 Multiplexer using IC 74151

2.22 DEMULTIPLEXERS

Demultiplex means "one into many". A demultiplexer is a combinational logic circuit with one input and many outputs. By applying control signal, we can steer the input signal to one of the output lines. **Figure 2.54** shows the block diagram of demultiplexer. It has 1 input signal, 'm' control signals and 'n' output signals.

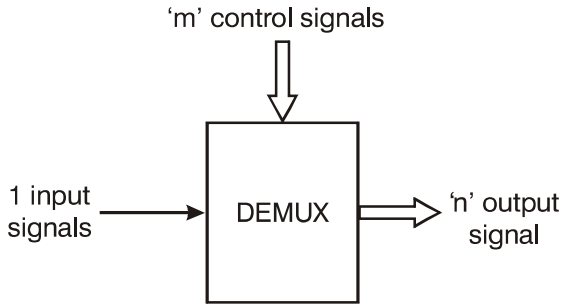


Fig. 2.54: Block diagram of demultiplexer

The select inputs (Control Signals) determine to which output the data input will be connected. As the serial data is changed to parallel data, i.e., the input caused to appear on one of the '*n*' output lines, the demultiplexer is called **data distributor** or **serial-to-parallel converter**.

2.22.1 1-to-4 Demultiplexer

A 1-to-4 demultiplexer has a single input (D), 4 outputs ($Y_0Y_1Y_2Y_3$) and two select lines (S_1S_0). The truth table of the 1 to 4 demultiplexer is shown in **Table 2.17**.

When $S_1S_0 = 00$, the data input is connected to output Y_0.

$$\therefore \qquad Y_0 = \overline{S}_1\overline{S}_0D = D$$

When $S_1S_0 = 01$, the data input is connected to output Y_1

$$\therefore \qquad Y_1 = \overline{S}_1S_0D = D$$

When $S_1S_0 = 10$, the data input is connected to output Y_2

$$Y_2 = S_1\overline{S}_0D = D$$

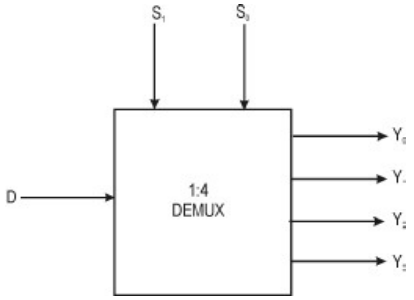
When $S_1S_0 = 11$, the data input is connected to output Y_3

$$Y_3 = S_1S_0D = D$$

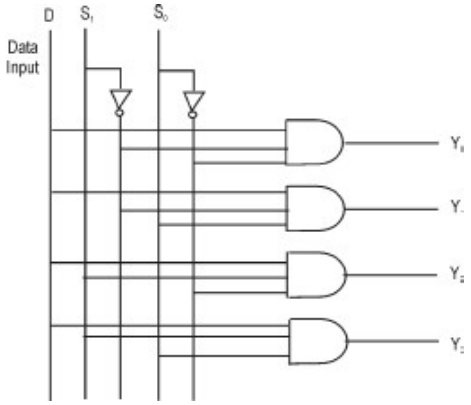
Using these expressions, a 1-to-4 demultiplexer is implemented using AND gates as shown in **Figure 2.55**.

TABLE 2.17 : Truth Table

Data Input	*Select Inputs*		*Outputs*			
D	S_1	S_0	Y_3	Y_2	Y_1	Y_0
D	0	0	0	0	0	D
D	0	1	0	0	D	0
D	1	0	0	D	0	0
D	1	1	D	0	0	0



(a) Logic Symbol



(b) Logic Diagram

Fig. 2.55: 1 to 4 multiplexer

2.23 IMPLEMENTATION OF COMBINATIONAL LOGIC USING MULTIPLEXER

Procedure

1. List the inputs of the multiplexer.
2. List under them all the given minterms in two rows. The first half of the minterms associated with $\overline{A}$ and the second half with the A.
3. The given function is implemented by circling the minterms of the function and applying the following rules to find the values for the inputs of the multiplexer:
   (i) If the two minterms in a column are not circled, apply 0 to the corresponding multiplexer input.
   (ii) If the two minterms in a column are circled, apply 1 to the corresponding multiplexer input.
   (iii) If the bottom minterm is circled and the top is not circled, apply A to the corresponding multiplexer input.
   (iv) If the top minterm is circled and the bottom is not circled, apply $\overline{A}$ to the corresponding multiplexer input.

4. Draw the multiplexer implementation diagram.

Example 2.4: Implement the following function using a multiplexer.

$$F(A, B, C) = \sum (1, 3, 5, 6)$$

Solution

Variables, $n = 3$ (A, B, C)

Select lines $= n - 1 = 2$ (S_1, S_0)

2^{n-1} to 1 MUX. i.e., 2^2 to 1 $\Rightarrow$ 4 to 1 MUX

Input lines $= 2^{n-1} = 2^2 = 4$ (I_0, I_1, I_2, I_3)

Implementation Table

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

$I_0 = 0,\ I_1 = 1, I_2 = \text{A}, I_3 = \overline{A}$

Multiplexer Implementation

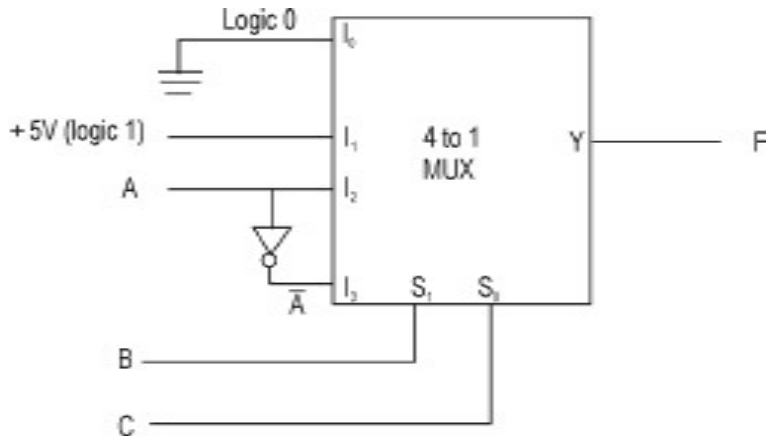


Fig. 2.56: 4 to 1 multiplexer

Example 2.5: Implement the following function with a multiplexer.

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

Solution: Variables, $n = 4$ (A, B, C, D)

Select lines, $n - 1 = 3$ (S_2, S_1, S_0)

Input lines, $2^{n-1} = 2^3 = 8$ (I_0–I_7)

2^{n-1} to 1 MUX = 2^3 to 1 $\Rightarrow$ 8 to 1 MUX

Use B, C and D variables as selection lines.

Implementation Table

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

Multiplexer Implementation

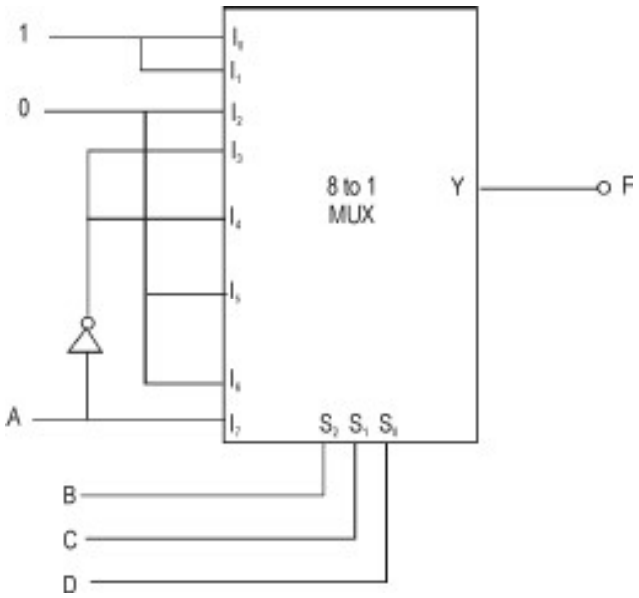


Fig. 2.57: 8 to 1 multiplexer

Example 2.6: Implement the following Boolean function using 8 : 1 MUX.

$$F(A,B,C,D) = \overline{A}B\overline{D} + ACD + \overline{B}CD + \overline{AC}D \qquad \textbf{(Dec. 2010)}$$

Solution:

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

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

$$= \overline{A}B\overline{D}C + \overline{A}B\overline{DC} + ACDB + ACD\overline{B} + \overline{B}CDA + \overline{B}CD\overline{A} + \overline{A}BD + \overline{AB}D + \overline{C}DA + \overline{C}D\overline{A}$$

$$= \overline{A}B\overline{D}C + \overline{A}B\overline{DC} + ABCD + A\overline{B}CD + A\overline{B}CD + \overline{AB}CD + \overline{A}BCD + \overline{A}B\overline{C}D$$

$$+\overline{AB}C\overline{D} + \overline{ABCD} + AB\overline{C}D + A\overline{BC}D + \overline{A}B\overline{C}D + \overline{ABC}D$$

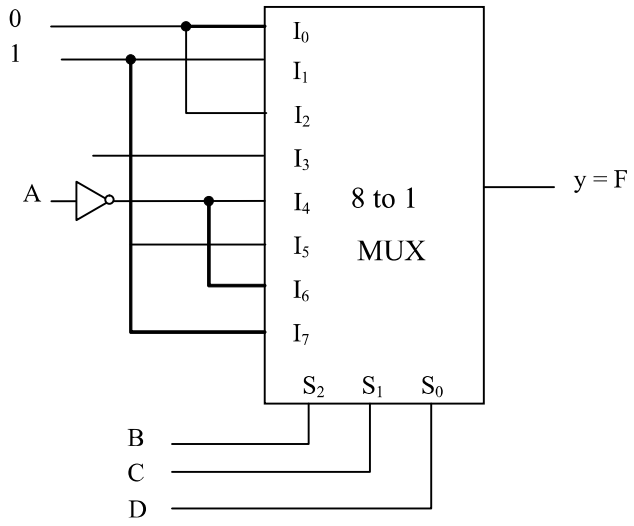
$$= \sum[6, 4, 15, 11, 11, 3, 7, 5, 3, 1, 13, 9, 5, 1]$$

$$= \sum[1, 3, 4, 5, 6, 7, 9, 11, 13, 15]$$

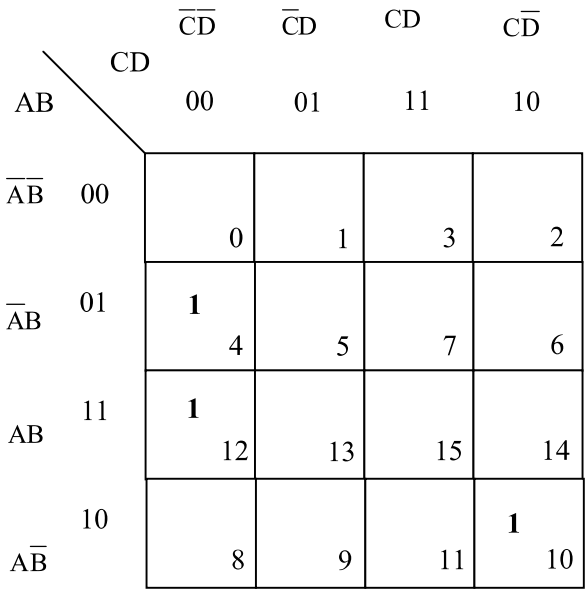
Implement table: [Use B, C, D as selection live]

	I_0	I_0	I_0	I_0	I_0	I_0	I_0	I_0
$\overline{A}$	0	(1)	2	(3)	(4)	5	6	7
A	8	(9)	10	(11)	12	(13)	14	(15)
	0	1	0	1	$\overline{A}$	1	$\overline{A}$	1

MUX Implementation



Example 2.7: Design a combinational logic using a suitable multiplexer to realize the following Boolean expression. $AD'B'C + BC'D'$. **(May 2011)**



F(A, B, C, D) = $\sum(4, 10, 12)$

Variables n = 4(A, B, C, D)

Select lines $\Rightarrow$ n - 1 = 4 - 1 = 3 (S_2, S_1, S_0)

Input lines, $2^{n-1} = 2^3 = 8$ (I_0 - I_7)

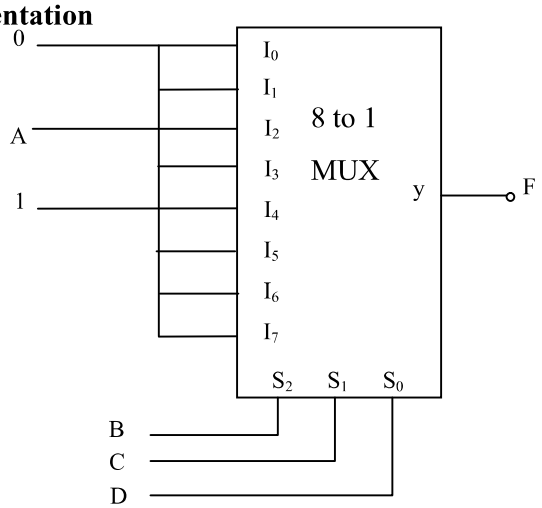
2n-1 to 1MUX = 2^3 to 1 = 8 to 1 MUX

Use B, C, D variables as selection lines.

Implenetation Table:

	I_0	I_0	I_0	I_0	I_0	I_0	I_0	I_0
$\overline{A}$	0	1	2	3	(4)	5	6	7
A	8	9	(10)	11	(12)	13	14	15
	0	0	A	0	1	0	0	0

Multiplexer Implementation



Example 2.8: Implement F(A, B, C, D) = ∑(1, 3, 4, 11, 12, 13, 14, 15) using 8 × 1 multiplexer.

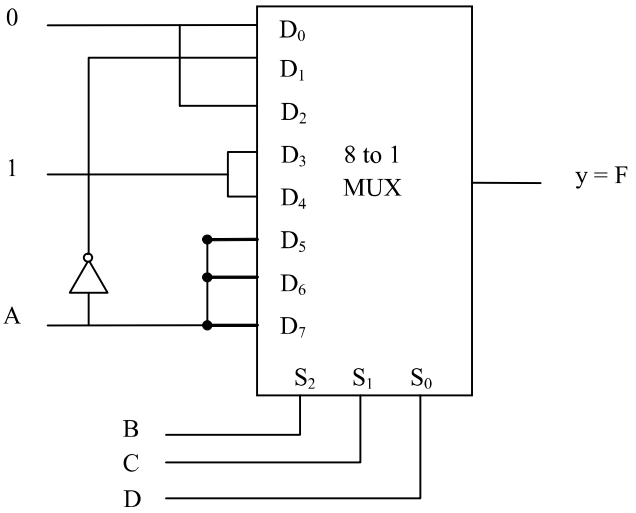
(Dec. 2012)

Solution

Implenetation Table:

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

Multiplexer Implementation



HARDWARE DESCRIPTION LANGUAGE

2.24 INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGE (HDL)

The complexity of digital designs has increased drastically with the advent of the smaller geometry semi-conductor process technology. This increase has made enormous demands on the industry, giving rise to Hardware Description Languages (HDL) which responded with the HDL-based design process, methodology and design tools. HDL not only manages the increased complexity, but it also allows a shorter design cycle.

HDL is a powerful language with numerous language constructs that are capable of describing very complex behaviour. The characteristics of an ideal HDL are:

- Supports multiple level of abstraction (gates to systems)
- Concise
- Describes functional blocks and their interconnections
- Wall suited for synthesis and verification

2.24.1 Types of HDLS

There are manyd ifferent systems for modeling and simulating hardware.

- Verilog
- VHDL
- Super log
- System C
- L-language and M-language (Mentor)
- Aida (IBM/HaL)
- and many others

2.24.2 Major HDLS

Two major HDLS are Verilog and VHDL. These both are programming languages. They are text-based, easier to create a design over schematic entry/capture. VHDL and Verilog both enjoy widespread use and share the logic synthesis market roughly 50/50.

Verilog has its synthetic roots in ‘C’ and is in some respects an easier language to learna nd use, which VHDL is more like ADA (U.S. Department of Decense–Sponsored Software programming language) and hase more features that support large project development.