7.2 DESIGN PROCEDURE

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

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

7.3 ADDERS

Digital computers perform a variety of information processing tasks. Among the basic tasks encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of 4 possible operations, namely,

$$0 + 0 = 0, \quad 0 + 1 = 1, \quad 1 + 0 = 1, \quad \text{and} \quad 1 + 1 = 10.$$

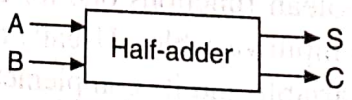
The first three operations produce a sum whose length is one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a carry. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half-adder*. One that performs the addition of three bits (two significant bits and previous carry) is called a *full-adder*. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder.

7.3.1 The Half-Adder

A half-adder is a combinational circuit with two binary inputs (augend and addend bits) and two binary outputs (sum and carry bits). It adds the two inputs (A and B) and produces the sum (S) and the carry (C) bits. It is an arithmetic circuit used to perform the arithmetic operation of addition of two single bit words. The truth table and block diagram of a half-adder are shown in Figure 7.2.

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) Truth table



(b) Block diagram

Figure 7.2 Half-adder.

The sum (S) bit and the carry (C) bit, according to the rules of binary addition, are given by: The sum (S) is the X-OR of A and B (It represents the LSB of the sum). Therefore,

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

The carry (C) is the AND of A and B (It is 0 unless both the inputs are 1). Therefore,

$$C = AB$$

A half-adder can, therefore, be realized by using one X-OR gate and one AND gate as shown in Figure 7.3a. Realization using AOI logic is shown in Figure 7.3b.

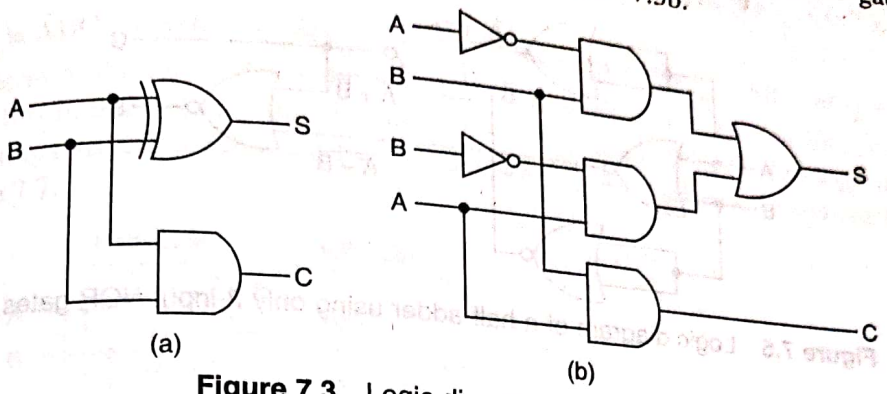


Figure 7.3 Logic diagrams of half-adder.

A half-adder can also be realized in universal logic by using either only NAND gates or only NOR gates as shown in Figures 7.4 and 7.5 respectively.

NAND logic

$$S = A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B}$$
$$= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B})$$
$$= A \cdot \overline{AB} + B \cdot \overline{AB}$$
$$= \overline{A \cdot \overline{AB} \cdot B \cdot \overline{AB}}$$
$$C = AB = \overline{\overline{AB}}$$

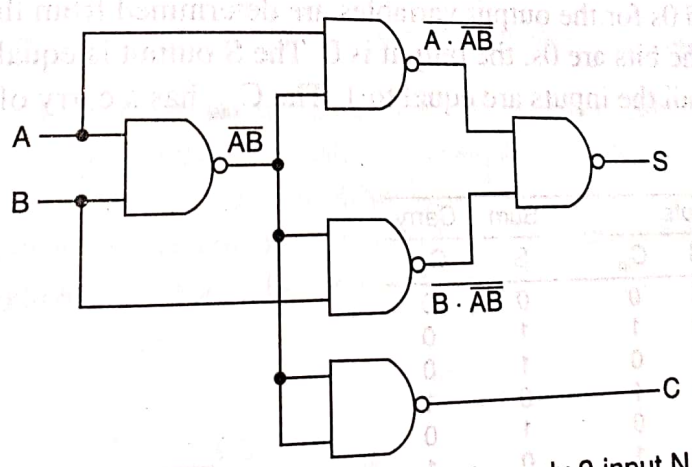


Figure 7.4 Logic diagram of a half-adder using only 2-input NAND gates.

NOR logic

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

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

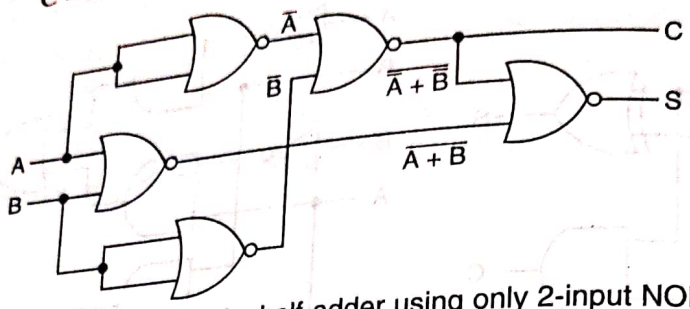


Figure 7.5 Logic diagram of a half-adder using only 2-input NOR gates.

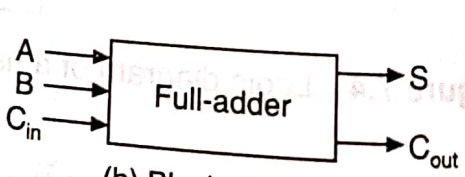
7.3.2 The Full-Adder

A full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. When we want to add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column. So, in the second and higher columns, the two data bits of that column and the carry bit generated from the addition in the previous column need to be added.

The full-adder adds the bits A and B and the carry from the previous column called the carry-in C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out}. The variable S gives the value of the least significant bit of the sum. The variable C_{out} gives the output carry. The block diagram and the truth table of a full-adder are shown in Figure 7.6. The eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have. The 1s and 0s for the output variables are determined from the arithmetic sum of the input bits. When all the bits are 0s, the output is 0. The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. The C_{out} has a carry of 1 if two or three inputs are equal to 1.

Inputs			Sum	Carry
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

(a) Truth table



(b) Block diagram

Figure 7.6 Full-adder.

From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of A, B, and C_{in} is described by

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

and

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

The sum term of the full-adder is the X-OR of A, B, and C_{in}, i.e. the sum bit is the modulo sum of the data bits in that column and the carry from the previous column. The logic diagram of the full-adder using two X-OR gates and two AND gates (i.e. two half-adders) and one OR gate is shown in Figure 7.7.

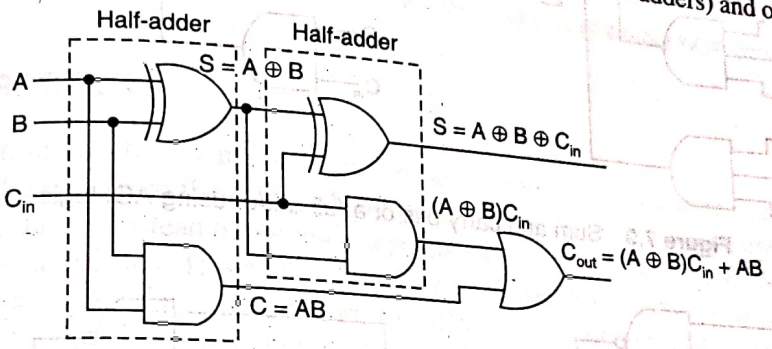


Figure 7.7 Logic diagram of a full-adder using two half-adders.

The block diagram of a full-adder using two half-adders is shown in Figure 7.8.

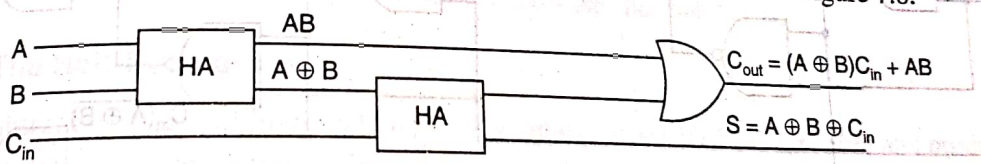


Figure 7.8 Block diagram of a full-adder using two half-adders.

Even though a full-adder can be constructed using two half-adders as shown in Figure 7.7, the disadvantage is that the bits must propagate through several gates in succession, which makes the total propagation delay greater than that of the full-adder circuit using AOI logic shown in Figure 7.9.

The full-adder can also be realized using universal logic, i.e. either only NAND gates or only NOR gates as shown in Figures 7.10 and 7.11 respectively.

NAND logic

We know that

$$A \oplus B = \overline{\overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}}$$

Then

$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) \cdot \overline{(A \oplus B)C_{in}}} \cdot \overline{C_{in} \cdot \overline{(A \oplus B)C_{in}}}}$$

$$C_{out} = C_{in}(A \oplus B) + AB = \overline{\overline{C_{in}(A \oplus B)} \cdot \overline{AB}}$$

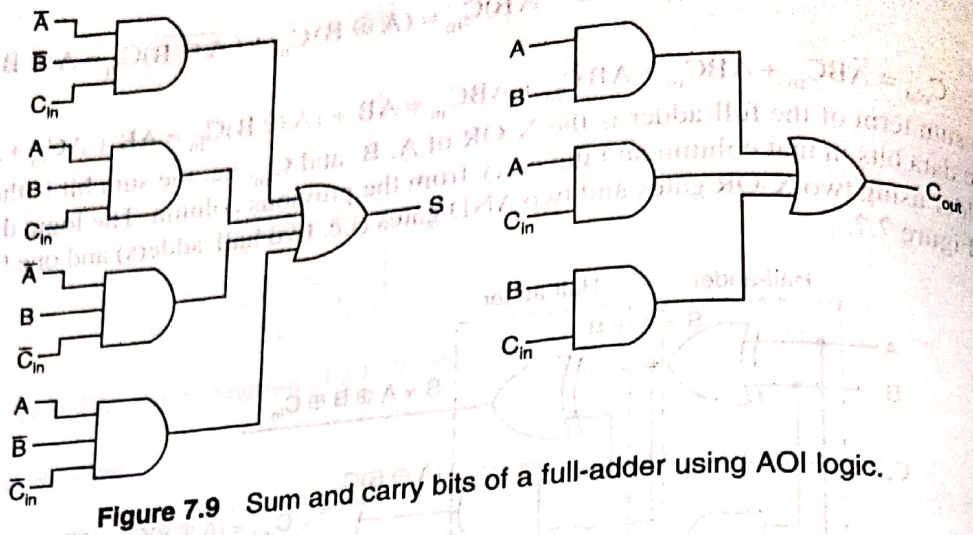


Figure 7.9 Sum and carry bits of a full-adder using AOI logic.

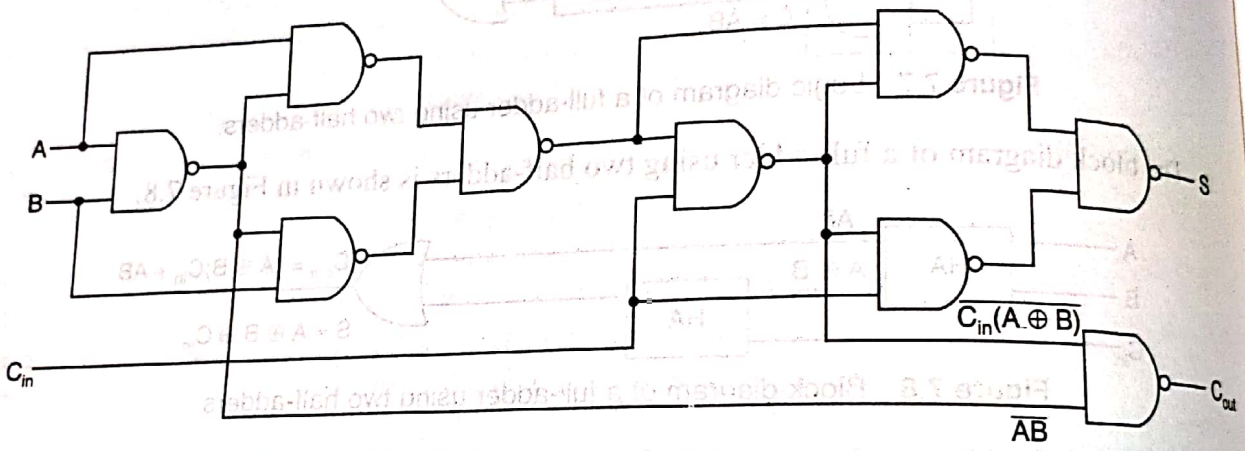


Figure 7.10 Logic diagram of a full-adder using only 2-input NAND gates.

NOR logic

We know that

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

Then

$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) + C_{in}} + \overline{(A \oplus B) + \overline{C}_{in}}}$$

$$C_{out} = AB + C_{in}(A \oplus B) = \overline{\overline{A} + \overline{B} + \overline{C}_{in} + \overline{A \oplus B}}$$

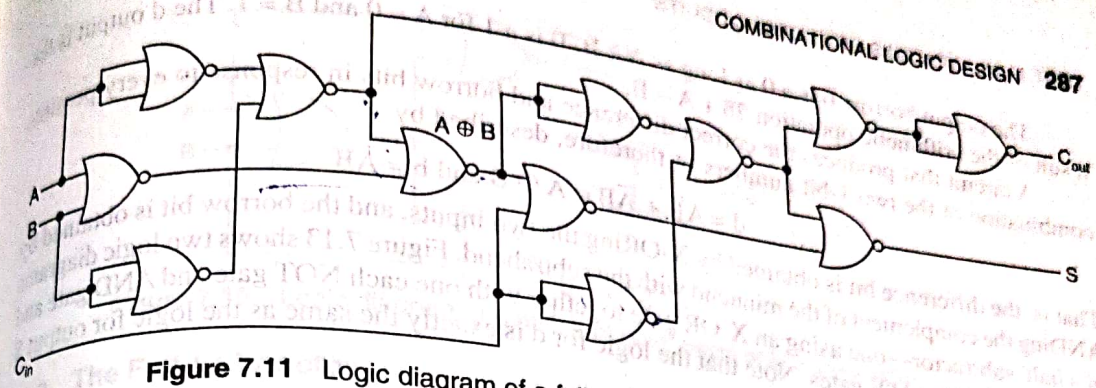


Figure 7.11 Logic diagram of a full-adder using only 2-input NOR gates.

7.4 SUBTRACTORS

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this method, the subtraction operation becomes an addition operation and instead of having a separate circuit for subtraction, the adder itself can be used to perform subtraction. This results in reduction of hardware. In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. The fact that a 1 has been borrowed must be conveyed to the next higher pair of bits by means of a signal coming out (output) of a given stage and going into (input) the next higher stage. Just as there are half- and full-adders, there are half- and full-subtractors.

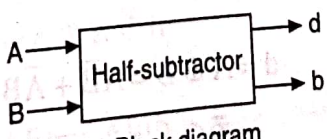
7.4.1 The Half-Subtractor

A half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference. It also has an output to specify if a 1 has been borrowed. It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.

A half-subtractor shown in Figure 7.12 is a combinational circuit with two inputs A and B and two outputs d and b. d indicates the difference and b is the output signal generated that informs the next stage that a 1 has been borrowed. We know that, when a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as follows.

Inputs		Outputs	
A	B	d	b
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

(a) Truth table



(b) Block diagram

Figure 7.12 Half-subtractor.

The output borrow b is a 0 as long as $A \geq B$. It is a 1 for $A = 0$ and $B = 1$. The d output is the result of the arithmetic operation $2b + A - B$.

A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is, therefore, described by

$$d = A\bar{B} + \bar{A}B = A \oplus B \text{ and } b = \bar{A}B$$

That is, the difference bit is obtained by X-ORing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend. Figure 7.13 shows two logic diagrams of a half-subtractor—one using an X-OR gate together with one each NOT gate and AND gate and the other using the AOI gates. Note that the logic for d is exactly the same as the logic for output S in the half-adder.

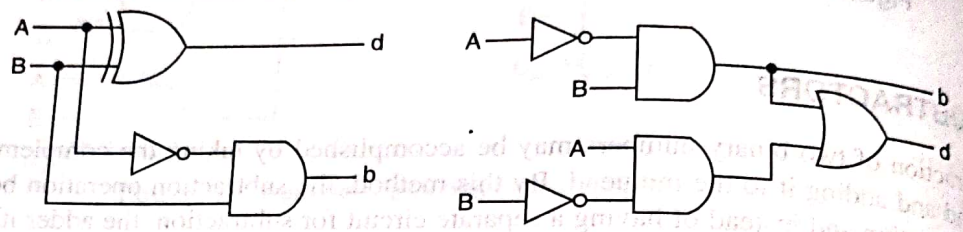


Figure 7.13 Logic diagrams of a half-subtractor.

A half-subtractor can also be realized using universal logic—either using only NAND gates or using only NOR gates—as shown in Figures 7.14 and 7.15 respectively.

NAND logic

$$d = A \oplus B = \overline{\overline{A \cdot \overline{AB}} \cdot \overline{B \cdot \overline{AB}}}$$

$$b = \bar{A}B = B(\bar{A} + \bar{B}) = B(\overline{AB}) = \overline{B \cdot \overline{AB}}$$

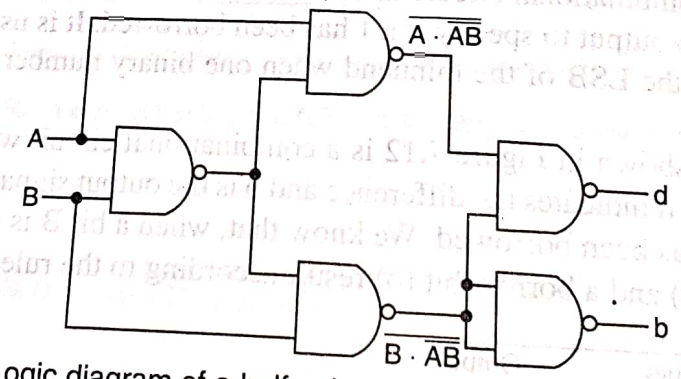


Figure 7.14 Logic diagram of a half-subtractor using only 2-input NAND gates.

NOR logic

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

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

$$d = \bar{A}B = \bar{A}(A + B) = \overline{\overline{\bar{A}(A + B)}} = \overline{A + (\overline{A + B})}$$

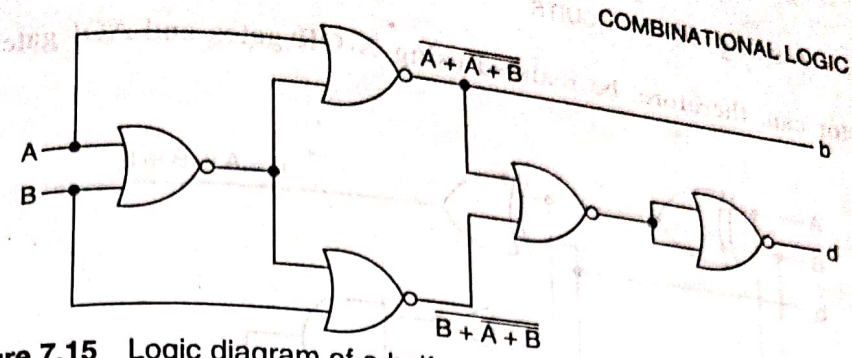


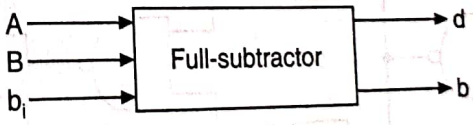
Figure 7.15 Logic diagram of a half-subtractor using only 2-input NOR gates.

7.4.2 The Full-Subtractor

The half-subtractor can be used only for LSB subtraction. If there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column. Such a subtraction is performed by a full-subtractor. It subtracts one bit (B) from another bit (A), when already there is a borrow b_i from this column for the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit (b) required from the next column. So a full-subtractor is a combinational circuit with three inputs (A, B, b_i) and two outputs d and b. The two outputs present the difference and output borrow. The 1s and 0s for the output variables are determined from the subtraction of $A - B - b_i$. The truth table and the block diagram of a full-subtractor are shown in Figure 7.16.

Inputs			Difference	Borrow
A	B	b_i	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

(a) Truth table



(b) Block diagram

Figure 7.16 Full-subtractor.

From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possible combination of A, B, and b_i is described by

$$\begin{aligned} d &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + A\bar{B}\bar{b}_i + ABb_i \\ &= b_i(AB + \bar{A}\bar{B}) + \bar{b}_i(A\bar{B} + \bar{A}B) \\ &= b_i(\overline{A \oplus B}) + \bar{b}_i(A \oplus B) = A \oplus B \oplus b_i \end{aligned}$$

and

$$b = \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + \bar{A}Bb_i + ABb_i = \bar{A}B(b_i + \bar{b}_i) + (AB + \bar{A}\bar{B})b_i$$

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

A full-subtractor can, therefore, be realized using X-OR gates and AOI gates as shown in Figure 7.17.

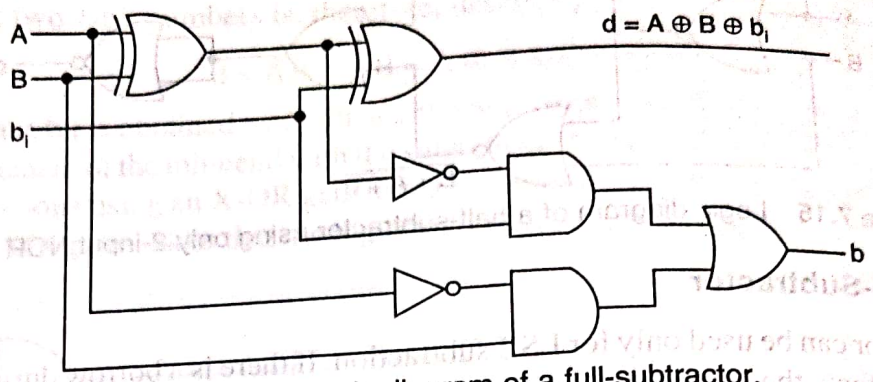


Figure 7.17 Logic diagram of a full-subtractor.

The full subtractor can also be realized in universal logic using either only NAND gates or only NOR gates as shown in Figures 7.18 and 7.19 respectively.

NAND logic

$$d = A \oplus B \oplus b_i = \overline{\overline{(A \oplus B) \oplus b_i}} = \overline{\overline{(A \oplus B)(\overline{A \oplus B})b_i} \cdot \overline{b_i(\overline{A \oplus B})b_i}}$$

$$b = \bar{A}B + b_i(\overline{A \oplus B}) = \overline{\overline{\bar{A}B} + \overline{b_i(\overline{A \oplus B})}}$$

$$= \overline{\overline{\bar{A}B} \cdot \overline{b_i(\overline{A \oplus B})}} = \overline{\overline{B(\bar{A} + \bar{B})} \cdot \overline{b_i(\bar{b}_i + (\overline{A \oplus B}))}}$$

$$= \overline{\overline{B \cdot \overline{AB}} \cdot \overline{b_i[b_i \cdot (\overline{A \oplus B})]}}$$

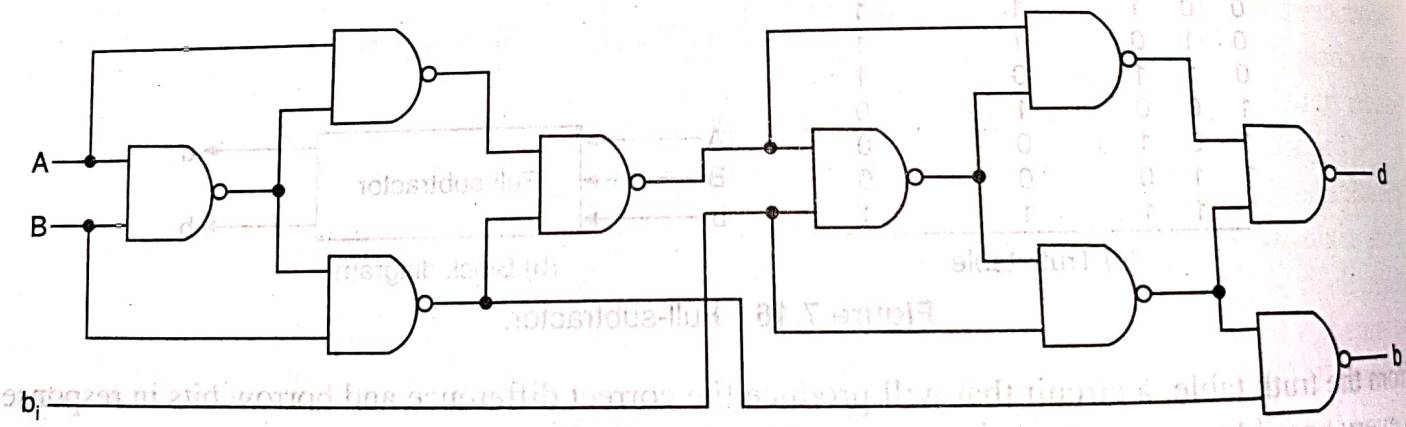


Figure 7.18 Logic diagram of a full-subtractor using only 2-input NAND gates.

NOR logic

$$d = A \oplus B \oplus b_i = \overline{\overline{(A \oplus B) \oplus b_i}}$$

$$= \overline{(A \oplus B)b_i + (\overline{A \oplus B})\bar{b}_i}$$

$$= \overline{[(A \oplus B) + (\overline{A \oplus B})\bar{b}_i][b_i + (\overline{A \oplus B})\bar{b}_i]}$$

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

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

$$b = \bar{A}B + b_i(\overline{A \oplus B})$$

$$= \bar{A}(A + B) + (\overline{A \oplus B})[(A \oplus B) + b_i]$$

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

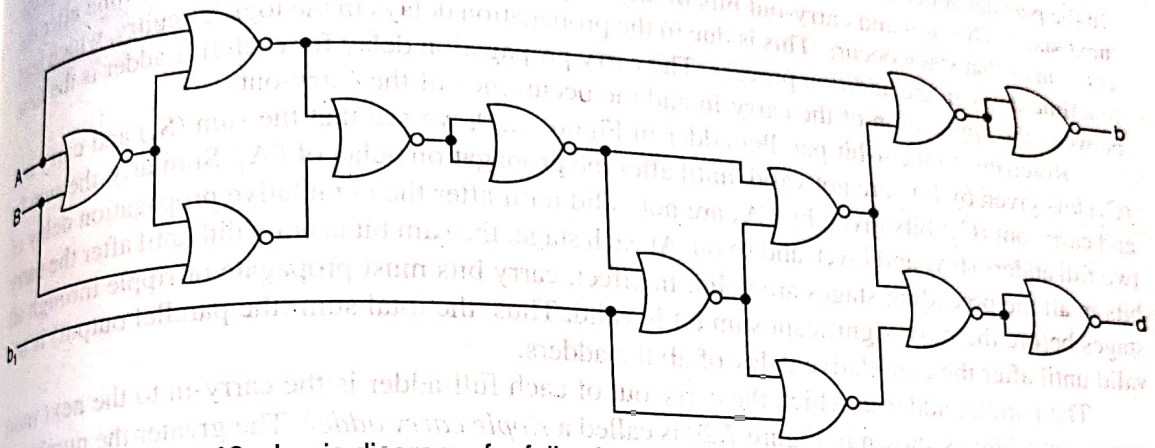


Figure 7.19 Logic diagram of a full subtractor using only 2-input NOR gates.

7.5 BINARY PARALLEL ADDER

A binary parallel adder is a digital circuit that adds two binary numbers in parallel form and produces the arithmetic sum of those numbers in parallel form. 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.

Figure 7.20 shows the interconnection of four full-adder (FA) circuits to provide a 4-bit parallel adder. The augend bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the lower-order bit. The carries are connected in a chain through the full-adders. The input carry to the adder is C_{in} and the output carry is C_4. The S outputs generate the required sum bits. 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

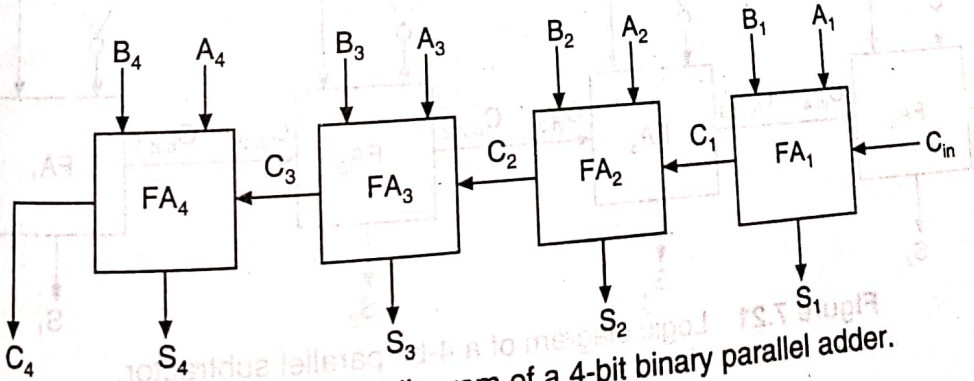


Figure 7.20 Logic diagram of a 4-bit binary parallel adder.

for the sum bits, and two terminals for the input and output carries. An n-bit parallel adder requires n-full adders. It can be constructed from 4-bit, 2-bit, and 1-bit full adder ICs by cascading several packages. The output carry from one package must be connected to the input carry of the one with the next higher-order bits. The 4-bit full adder is a typical example of an MSI function.

7.5.1 The Ripple Carry Adder

In the parallel adder discussed above, the carry-out of each stage is connected to the carry-in of the next stage. The sum and carry-out bits of any stage cannot be produced, until some time after the carry-in of that stage occurs. This is due to the propagation delays in the logic circuitry, which lead to a time delay in the addition process. The carry propagation delay for each full-adder is the time between the application of the carry-in and the occurrence of the carry-out.

Referring to the 4-bit parallel adder in Figure 7.20, we see that the sum (S_1) and carry-out (C_1) bits given by FA_1 are not valid, until after the propagation delay of FA_1. Similarly, the sum S_2 and carry-out (C_2) bits given by FA_2 are not valid until after the cumulative propagation delay of two full adders (FA_1 and FA_2), and so on. At each stage, the sum bit is not valid until after the carry bits in all the preceding stages are valid. In effect, carry bits must propagate or ripple through all stages before the most significant sum bit is valid. Thus, the total sum (the parallel output) is not valid until after the cumulative delay of all the adders.

The parallel adder in which the carry-out of each full-adder is the carry-in to the next most significant adder as shown in Figure 7.20 is called a *ripple carry adder*. The greater the number of bits that a ripple carry adder must add, the greater the time required for it to perform a valid addition. If two numbers are added such that no carries occur between stages, then the add time is simply the propagation time through a single full-adder.

7.6 4-BIT PARALLEL SUBTRACTOR

The subtraction of binary numbers can be carried out most conveniently by means of complements as discussed in Chapter 1. Remember that the subtraction A – B can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters as shown in Figure 7.21.

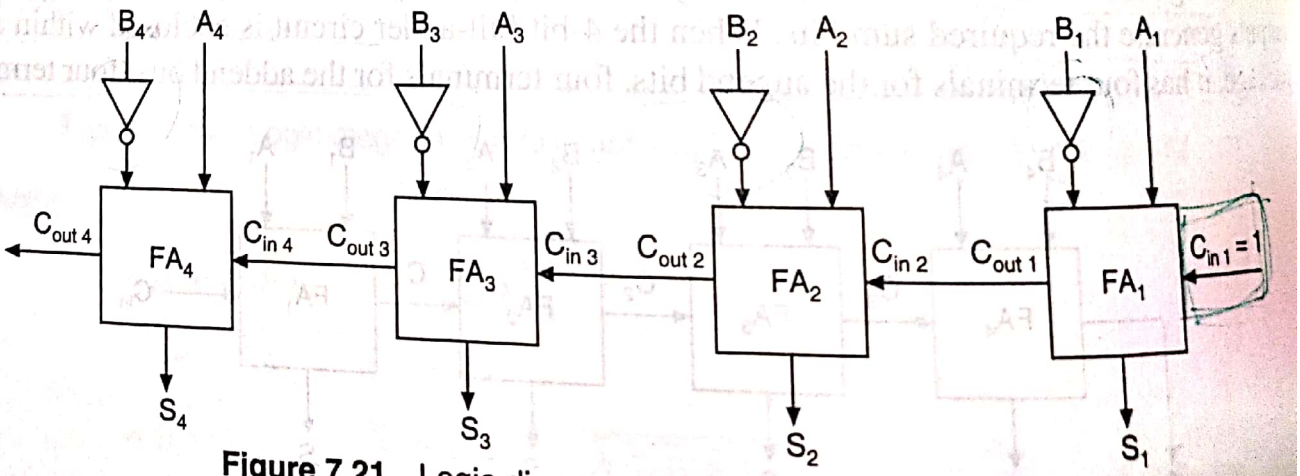


Figure 7.21 Logic diagram of a 4-bit parallel subtractor.

7.7 BINARY ADDER-SUBTRACTOR

Figure 7.22 shows a 4-bit adder-subtractor circuit. Here the addition and subtraction operations are combined into one circuit with one common binary adder. This is done by including an X-OR gate with each full-adder. The mode input M controls the operation. When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor. Each X-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full-adder receives the value of B, the input carry is 0 and the circuit performs $A + B$. When $M = 1$, we have $B \oplus 1 = \overline{B}$ and $C_1 = 1$. The B inputs are complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B.

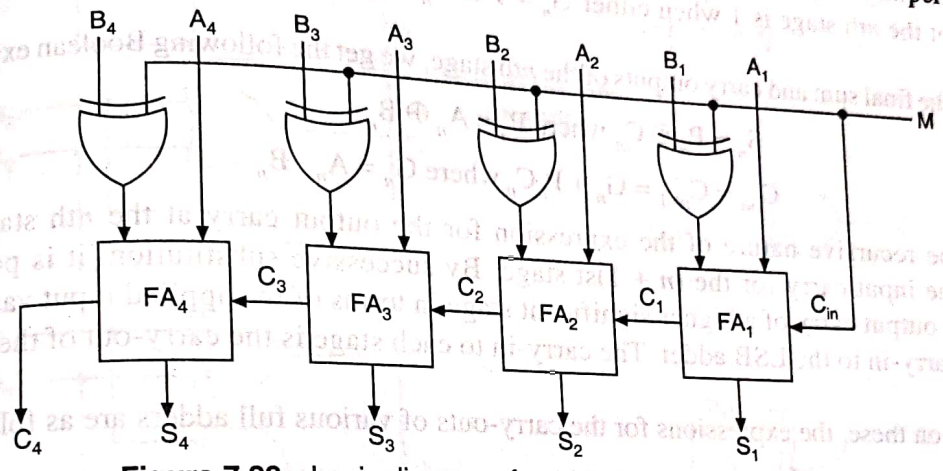


Figure 7.22 Logic diagram of a 4-bit binary adder-subtractor.

7.8 THE LOOK-AHEAD-CARRY ADDER

In the case of the parallel adder, the speed with which an addition can be performed is governed by the time required for the carries to propagate or ripple through all of the stages of the adder. The look-ahead-carry adder speeds up the process by eliminating this ripple carry delay. It examines all the input bits simultaneously and also generates the carry-in bits for all the stages simultaneously.

The method of speeding up the addition process is based on the two additional functions of the full-adder, called the *carry generate* and *carry propagate functions*.

Consider one full adder stage; say the *n*th stage of a parallel adder shown in Figure 7.23. We know that it is made of two half-adders and that the half-adder contains an X-OR gate to produce the sum and an AND gate to produce the carry. If both the bits A_n and B_n are 1s, a carry has to be generated in this stage regardless of whether the input carry C_{in} is a 0 or a 1. This is called generated carry, expressed as $G_n = A_n \cdot B_n$ which has to appear at the output through the OR gate as shown in Figure 7.23.

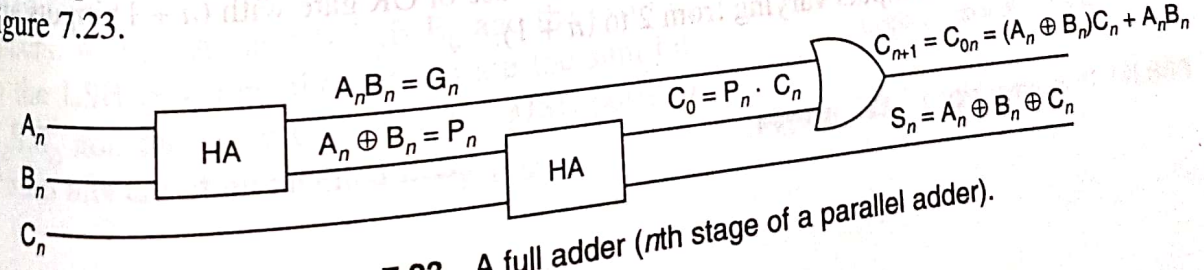


Figure 7.23 A full adder (*n*th stage of a parallel adder).

There is another possibility of producing a carry out. X-OR gate inside the half-adder at the input produces an intermediary sum bit—call it P_n—which is expressed as $P_n = A_n \oplus B_n$. Next P_n and C_n are added using the X-OR gate inside the second half adder to produce the final sum bit $S_n = P_n \oplus C_n = A_n \oplus B_n \oplus C_n$ and output carry $C_0 = P_n \cdot C_n = (A_n \oplus B_n)C_n$ which becomes input carry for the $(n + 1)$th stage.

Consider the case of both P_n and C_n being 1. The input carry C_n has to be propagated to the output only if P_n is 1. If P_n is 0, even if C_n is 1, the AND gate in the second half-adder will inhibit C_n. We may thus call P_n as the propagated carry as this is associated with enabling propagation of C_n to the carry output of the nth stage which is denoted as C_{n+1} or C_{0n}. So, we can say that the carryout of the nth stage is 1 when either $G_n = 1$ or $P_n \cdot C_n = 1$ or both G_n and $P_n \cdot C_n$ are equal to 1.

For the final sum and carry outputs of the nth stage, we get the following Boolean expressions.

$$S_n = P_n \oplus C_n \text{ where } P_n = A_n \oplus B_n$$

$$C_{on} = C_{n+1} = G_n + P_n C_n \text{ where } G_n = A_n \cdot B_n$$

Observe the recursive nature of the expression for the output carry at the nth stage which becomes the input carry for the $(n + 1)$st stage. By successive substitution, it is possible to express the output carry of a higher significant stage in terms of the applied input variables A, B and the carry-in to the LSB adder. The carry-in to each stage is the carry-out of the previous stage.

Based on these, the expressions for the carry-outs of various full adders are as follows:

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

The general expression for n stages designated as 0 through $(n - 1)$ would be

$$C_n = G_{n-1} + P_{n-1} \cdot C_{n-1} = G_{n-1} + P_{n-1} \cdot G_{n-2} + P_{n-1} \cdot P_{n-2} \cdot G_{n-3} + \ldots + P_{n-1} \cdot \ldots P_0 \cdot C_0$$

Observe that the final output carry is expressed as a function of the input variables in SOP form, which is a two-level AND-OR or equivalent NAND-NAND form. To produce the output carry for any particular stage, it is clear that it requires only that much time required for the signals to pass through two levels only. Hence the circuit for look-ahead-carry introduces a delay corresponding to two gate levels. The block diagram of a 4 stage look-ahead-carry parallel adder is shown in Figure 7.24.

Observe that the full look-ahead-scheme requires the use of OR gate with $(n + 1)$ inputs and AND gates with number of inputs varying from 2 to $(n + 1)$.

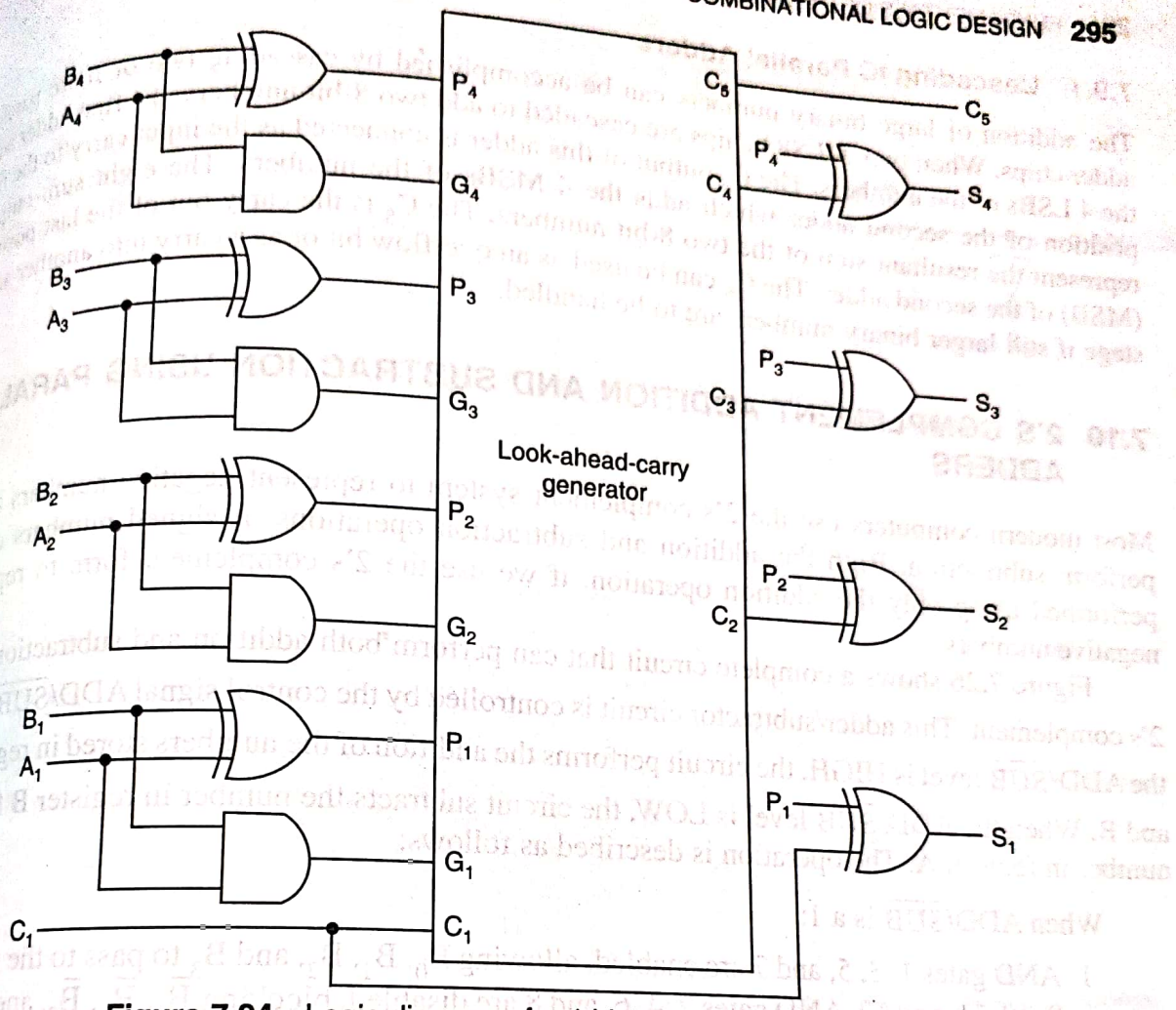


Figure 7.24 Logic diagram of a 4-bit look-ahead-carry adder.

7.9 IC PARALLEL ADDERS

Several parallel adders are available as ICs. The most common one is a 4-bit parallel adder IC that contains four interconnected full-adders and a look-ahead-carry circuit needed for high speed operation. The 7483A, the 74LS83A, the 74283, and the 74LS283 are all TTL 4-bit parallel adder chips. Figure 7.25 shows the functional symbol for the 74LS83 4-bit parallel adder (and its equivalents). The inputs to this IC are two 4-bit numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ and the carry C_0 into the LSB position; the outputs are the sum bits $S_3S_2S_1S_0$ and the carry C_4 out of the MSB position. The sum bits are often labelled $\Sigma_3\Sigma_2\Sigma_1\Sigma_0$.

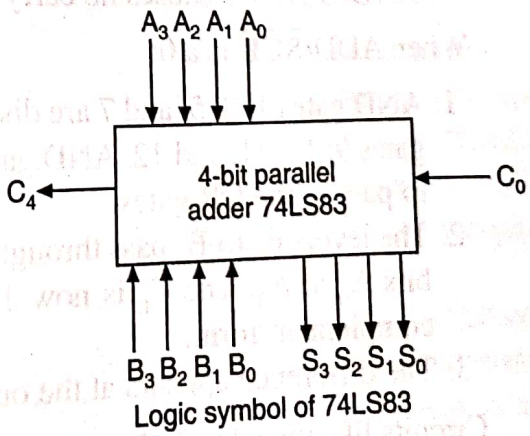


Figure 7.25 Logic symbol of 74LS83.

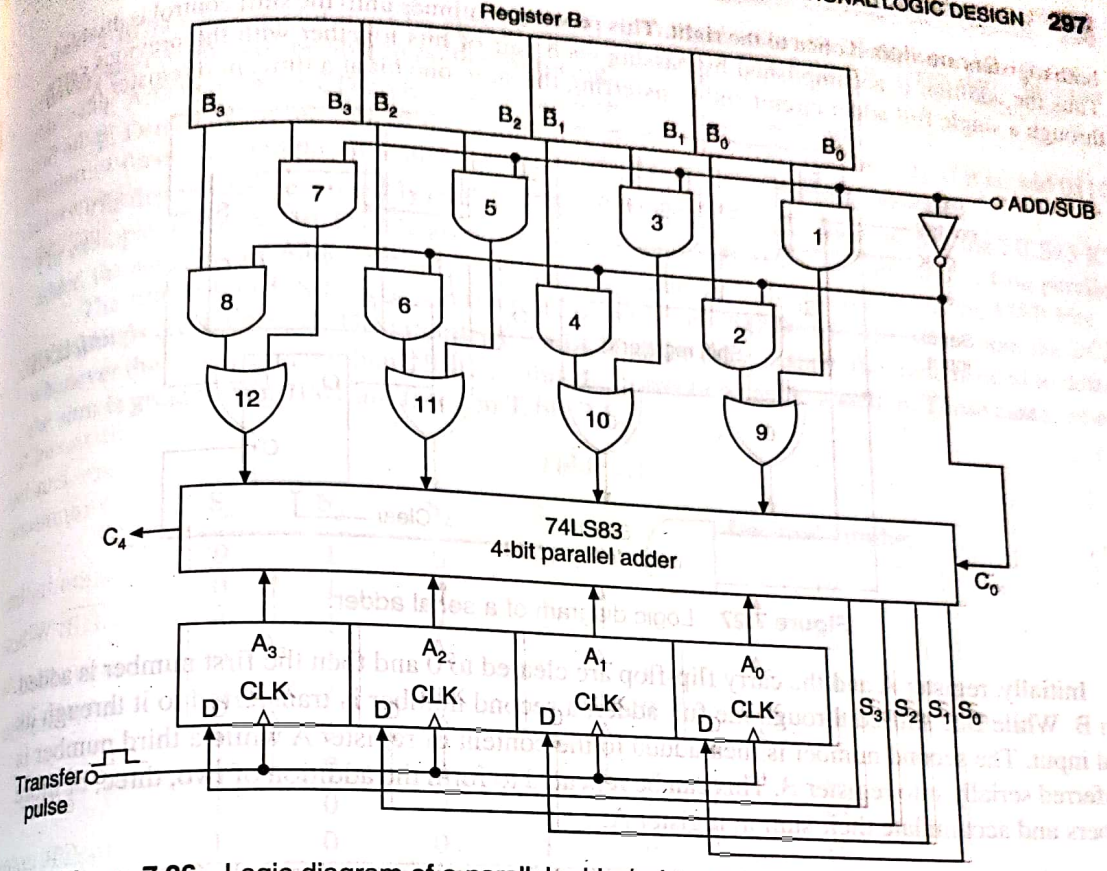


Figure 7.26 Logic diagram of a parallel adder/subtractor using 2's complement system.

7.11 SERIAL ADDER

A serial adder is used to add binary numbers in serial form. The two binary numbers to be added serially are stored in two shift registers A and B. Bits are added one pair at a time through a single full adder (FA) circuit as shown in Figure 7.27. The carry out of the full-adder is transferred to a D flip-flop. The output of this flip-flop is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full adder could be transferred to a third shift register. By shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both augend and the sum bits. The serial input register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is as follows. Initially register A holds the augend, register B holds the addend and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full-adder at x and y. The shift control enables both registers and carry flip-flop; so, at the clock pulse both registers are shifted once to the right, the sum bit from S enters the left most flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits of the registers. For each succeeding clock pulse a new sum bit is transferred to A, a new carry is transferred to Q, and

both registers are shifted once to the right. This process continues until the shift control is disabled. Thus the addition is accomplished by passing each pair of bits together with the previous carry through a single full adder circuit and transferring the sum, one bit at a time, into register A.

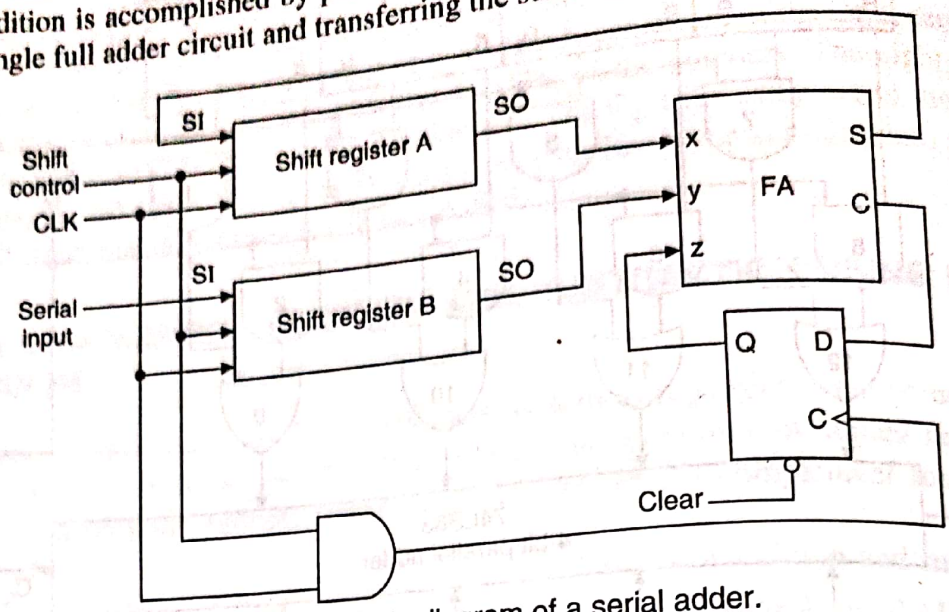


Figure 7.27 Logic diagram of a serial adder.

Initially, register A and the carry flip-flop are cleared to 0 and then the first number is added from B. While B is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the content of register A while a third number is transferred serially into register B. This can be repeated to form the addition of two, three, or more numbers and accumulate their sum in register A.

7.11.1 Difference between Serial and Parallel Adders

The parallel adder uses registers with parallel load, whereas the serial adder uses shift registers. The number of full adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit. The sequential circuit in the serial adder consists of a full-adder and a flip-flop that stores the output carry.

7.12 BCD ADDER

The BCD addition process has been discussed in Chapter 1. It is briefly reviewed here.

1. Add the 4-bit BCD code groups for each decimal digit position using ordinary binary addition.
2. For those positions where the sum is 9 or less, the sum is in proper BCD form and no correction is needed.
3. When the sum of two digits is greater than 9, a correction of 0110 should be added to the sum, to produce the proper BCD result. This will produce a carry to be added to the next decimal position.

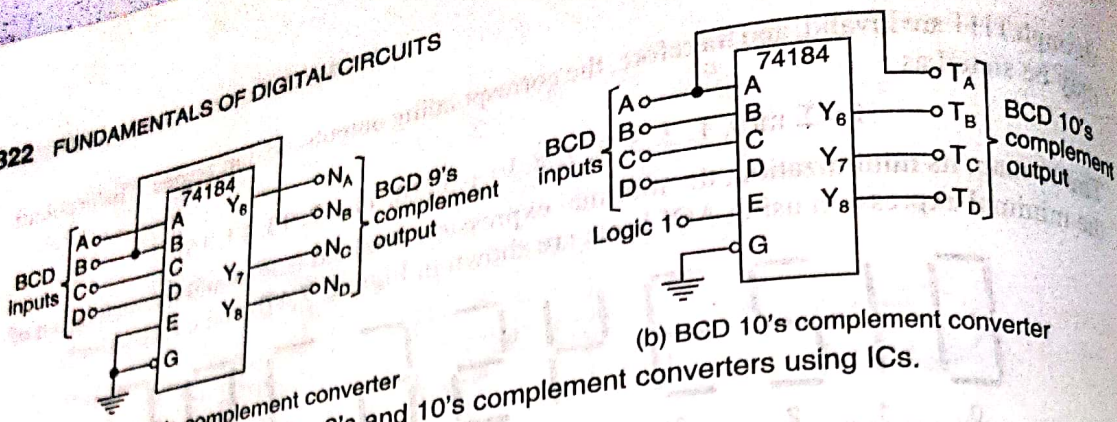


Figure 7.49 9's and 10's complement converters using ICs.

7.17 PARITY BIT GENERATORS/CHECKERS

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. Binary data, when transmitted and processed, is susceptible to noise that can alter its 1s to 0s and 0s to 1s. To detect such errors, an additional bit called the *parity bit* is added to the data bits and the word containing the data bits and the parity bit is transmitted. At the receiving end the number of 1s in the word received are counted and the error, if any, is detected. This parity check, however, detects only single bit errors.

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

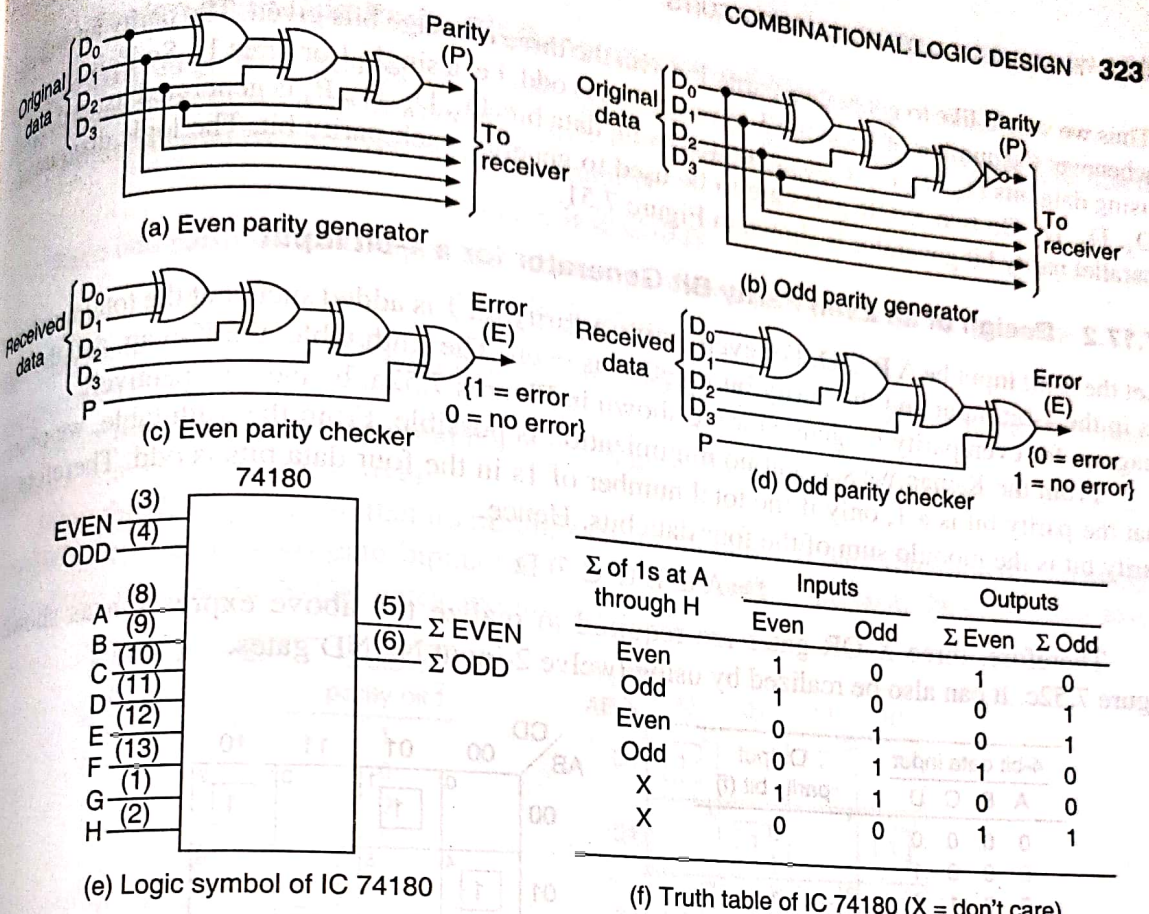
A parity bit, a 0 or a 1 is attached to the data bits such that the total number of 1s in the word is even for even parity and odd for odd parity. The parity bit can be attached to the code group either at the beginning or at the end depending on system design. A given system operates with either even or odd parity but not both. So, a word always contains either an even or an odd number of 1s.

At the receiving end, if the word received has an even number of 1s in the odd parity system or an odd number of 1s in the even parity system, it implies that an error has occurred.

In order to check or generate the proper parity bit in a given code word, the basic principle used is, "the modulo sum of an even number of 1s is always a 0 and the modulo sum of an odd number of 1s is always a 1". Therefore, in order to check for an error, all the bits in the received word are added. If the modulo sum is a 0 for an odd parity system or a 1 for an even parity system, an error is detected.

To generate an even parity bit, the four data bits are added using three X-OR gates. The sum bit will be the parity bit. Figure 7.50a shows the logic diagram of an even parity generator.

To generate an odd parity bit, the four data bits are added using three X-OR gates and the sum bit is inverted. Figure 7.50b shows the logic diagram of an odd parity generator. Figures 7.50c and d show an even bit parity checker and an odd parity checker, respectively. Also, Figure 7.50e shows the logic symbol of IC 74180, a 9-bit parity generator/checker. Figure 7.50f gives the truth table operation of this IC. This device can be used to check for odd or even parity in a 9-bit code (8 data bits and one parity bit), or it can be used to generate a 9-bit odd or even parity code.



Σ of 1s at A through H	Inputs		Outputs	
	Even	Odd	Σ Even	Σ Odd
Even	1	0	1	0
Odd	1	0	0	1
Even	0	1	0	1
Odd	0	1	1	0
X	1	1	0	0
X	0	0	1	1

(f) Truth table of IC 74180 (X = don't care)

Figure 7.50 Parity bit generator/checker.

7.17.1 Parallel Parity Bit Generator for Hamming Code

Consider the 7-bit Hamming code ($P_1P_2D_3P_4D_5D_6D_7$) used to correct single bit errors. Each word of the code contained 7 bits numbered 1–7. The bits 1, 2, and 4 are labelled as parity bits and called P_1, P_2, and P_4. The message bits in the word are labelled D_3, D_5, D_6 and D_7. Each parity bit is chosen in such a way that together with three other message bits it forms a 4-bit word with even parity, i.e. P_1 is chosen as a 0 or 1 so that $P_1D_3D_5D_7$ must have even parity. P_2 is chosen as a 0 or 1 so that $P_2D_3D_6D_7$ must have even parity. P_4 is chosen as a 0 or 1 so that $P_4D_5D_6D_7$ must have even parity.

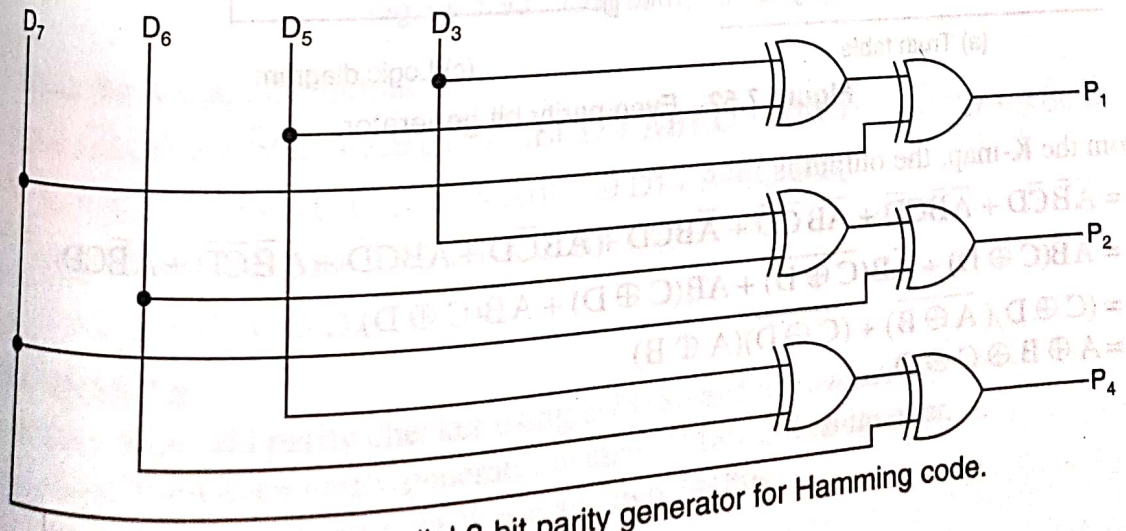


Figure 7.51 Parallel 3-bit parity generator for Hamming code.

Thus we would like to generate a parity bit with the three message bits given. The parity bit will be 1 whenever the number of 1's in the 3-message bits is odd, i.e. a single 1 or three 1s. So P_1 is generated using data bits D_3, D_5, D_7. P_2 is generated using data bits D_3, D_6, D_7. P_4 is generated using data bits D_5, D_6, D_7. So two X-OR gates are to be used to generate each parity bit. The logic diagram of a parallel parity bit generator is shown in Figure 7.51.

7.17.2 Design of an Even Parity Bit Generator for a 4-bit Input

Let the 4-bit input be A B C D. For even parity, a parity bit 1 is added such that the total number of 1s in the 4-bit input and the parity bit together is even. The truth table, the K-map, and the logic diagram for even parity bit generator are shown in Figures 7.52a, b, and c respectively.

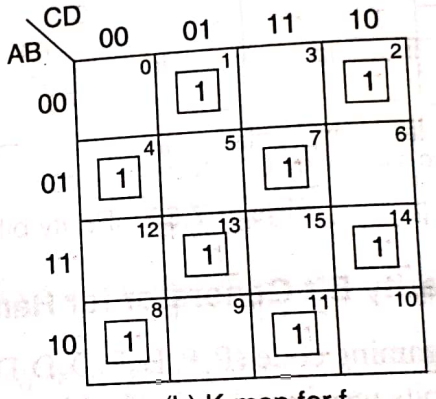
From the K-map we see that no minimization is possible. From the truth table, we observe that the parity bit is a 1, only if the total number of 1s in the four data bits is odd. Therefore, the parity bit is the modulo sum of the four data bits. Hence,

$$f = A \oplus B \oplus C \oplus D$$

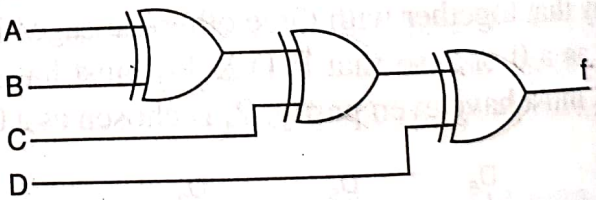
Therefore, three X-OR gates are required to realize the above expression as shown in Figure 7.52c. It can also be realized by using twelve 2-input NAND gates.

4-bit data input A	B	C	D	Output parity bit (f)
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(a) Truth table



(b) K-map for f



(c) Logic diagram

Figure 7.52 Even parity bit generator.

From the K-map, the output is

$$f = \bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D} + \bar{A}B\bar{C}\bar{D} + \bar{A}BCD + AB\bar{C}D + ABC\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}CD$$

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

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

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

7.17.3 Design of an Odd Parity Bit Generator for a 4-bit Input

An odd parity bit generator outputs a 1, when the number of 1s in the data bits is even, so that the total number of 1s in the data bits and the parity bit together is odd.

The expression for the even parity generator is

$$f = A \oplus B \oplus C \oplus D$$

Since odd parity is the complement of even parity, we get for odd parity generator

$$\bar{f} = \overline{A \oplus B \oplus C \oplus D}$$

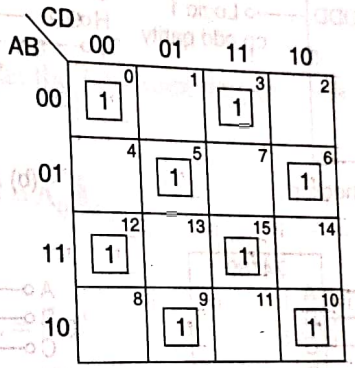
So, we put an inverter at the output of an even parity bit generator. The same result is directly obtained as follows.

The truth table, the K-map, and the logic diagram for the odd parity generator are shown in Figures 7.53a, b, and c, respectively.

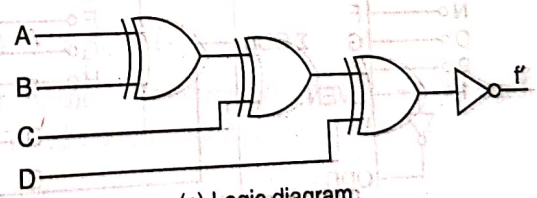
From the K-map, we see that no minimization is possible. If the expression for the parity bit is implemented as it is, 40 gate inputs are required. To reduce the cost, the expression may be manipulated in terms of X-OR gates and implemented.

4-bit data input				Output
A	B	C	D	parity bit $\bar{f}$
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(a) Truth table



(b) K-map for $\bar{f}$



(c) Logic diagram

Figure 7.53 Odd parity bit generator.

From the K-map, the output is

$$\begin{aligned} f &= \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + AB\bar{C}\bar{D} + ABCD + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + A\bar{B}\bar{C}D + A\bar{B}C\bar{D} \\ &= \bar{A}\bar{B}\overline{(C \oplus D)} + AB\overline{(C \oplus D)} + \bar{A}B(C \oplus D) + A\bar{B}(C \oplus D) \\ &= \overline{(C \oplus D)}\,\overline{(A \oplus B)} + (C \oplus D)(A \oplus B) \\ &= \overline{(A \oplus B) \oplus (C \oplus D)} \end{aligned}$$

EXAMPLE 7.8

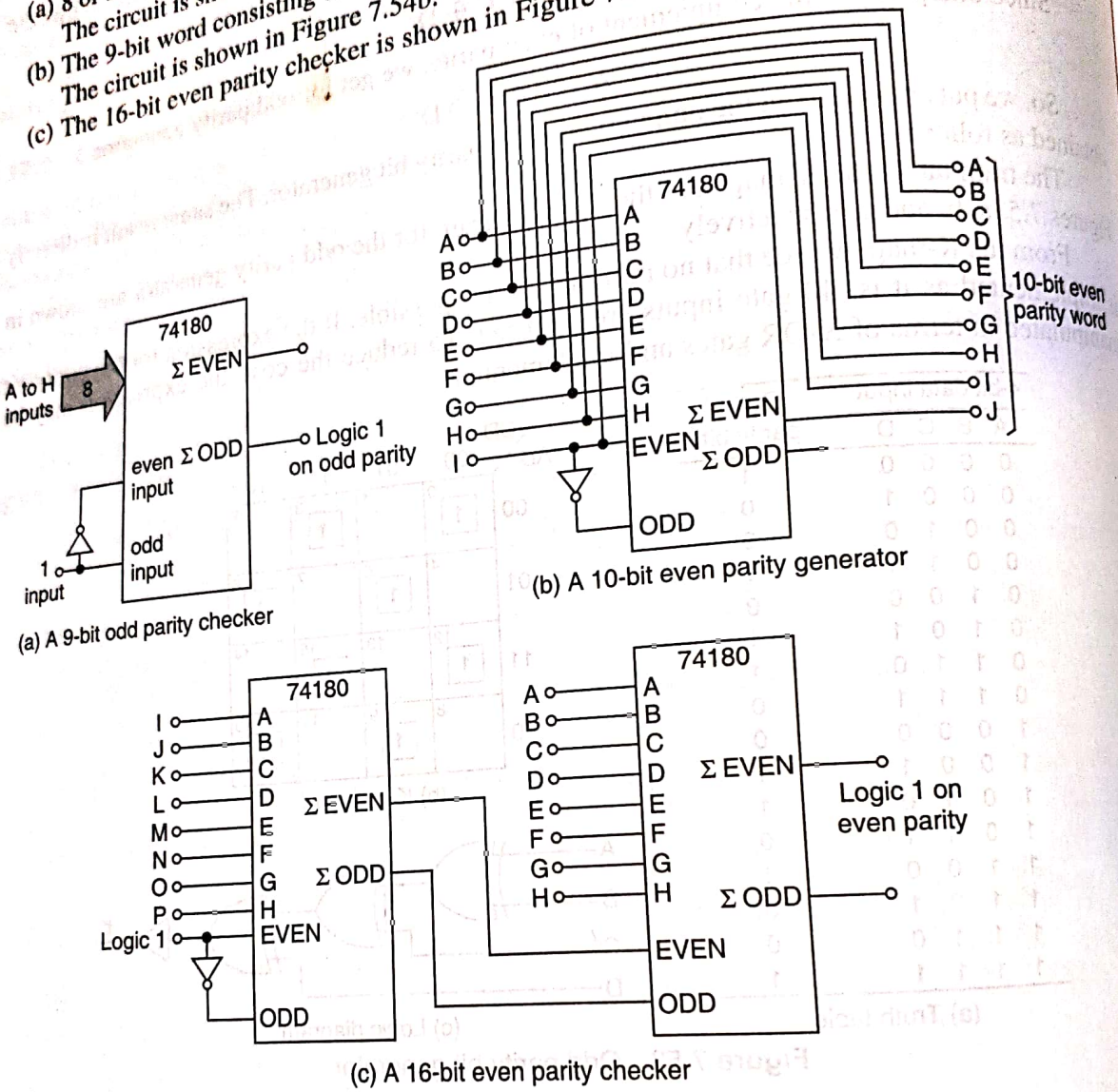
(a) Make a 9-bit odd parity checker using a 74180 and an inverter.
(b) Make a 10-bit even parity generator using a 74180 and an inverter.
(c) Make a 16-bit even parity checker using two 74180s.

Solution

(a) 8 of the 9-bits are applied at A-H inputs and the ninth bit, I, is applied to the ODD input. The circuit is shown in Figure 7.54a.

(b) The 9-bit word consisting of A through I is converted to a 10-bit word with even parity. The circuit is shown in Figure 7.54b.

(c) The 16-bit even parity checker is shown in Figure 7.54c.



(a) A 9-bit odd parity checker

(b) A 10-bit even parity generator

(c) A 16-bit even parity checker

Figure 7.54 Example 7.8: Logic circuits.

7.18 COMPARATORS

A comparator is a logic circuit used to compare the magnitudes of two binary numbers. Depending on the design, it may either simply provide an output that is active (goes HIGH for example) when the two numbers are equal, or additionally provide outputs that signify which of the numbers is greater when equality does not hold.

The X-NOR gate (coincidence gate) is a basic comparator, because its output is a 1 only if its two input bits are equal, i.e. the output is a 1 if and only if the input bits coincide.

Two binary numbers are equal, if and only if all their corresponding bits coincide. For example, two 4-bit binary numbers, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are equal, if and only if, $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$. Thus, equality holds when A_3 coincides with B_3, A_2 coincides with B_2, A_1 coincides with B_1, and A_0 coincides with B_0. The implementation of this logic,

$$\text{EQUALITY} = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

is straightforward. It is obvious that this circuit can be expanded or compressed to accommodate binary numbers with any other number of bits.

The block diagram of a 1-bit comparator which can be used as a module for comparison of larger numbers is shown in Figure 7.55.

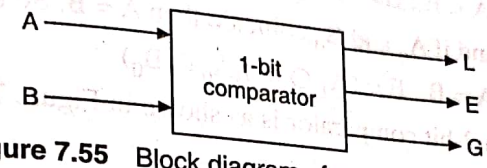


Figure 7.55 Block diagram of a 1-bit comparator.

7.18.1 1-bit Magnitude Comparator

The logic for a 1-bit magnitude comparator: Let the 1-bit numbers be $A = A_0$ and $B = B_0$.
If $A_0 = 1$ and $B_0 = 0$, then $A > B$.
Therefore,

$$A > B: G = A_0\bar{B}_0$$

If $A_0 = 0$ and $B_0 = 1$, then $A < B$.
Therefore,

$$A < B: L = \bar{A}_0B_0$$

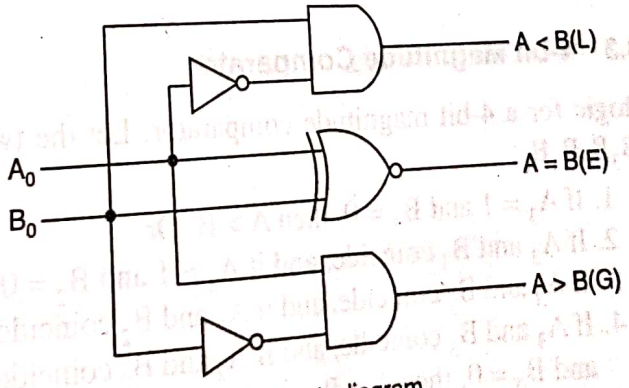
If A_0 and B_0 coincide, i.e. $A_0 = B_0 = 0$ or if $A_0 = B_0 = 1$, then $A = B$.
Therefore,

$$A = B : E = A_0 \odot B_0$$

The truth table and the logic diagram for the 1-bit comparator are shown in Figure 7.56. The logic expressions for G, L, and E can also be obtained from the truth table.

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

(a) Truth table



(b) Logic diagram

Figure 7.56 1-bit comparator.

7.18.2 2-bit Magnitude Comparator

The logic for a 2-bit magnitude comparator: Let the two 2-bit numbers be $A = A_1 A_0$ and $B = B_1 B_0$

1. If $A_1 = 1$ and $B_1 = 0$, then $A > B$ or
2. If A_1 and B_1 coincide and $A_0 = 1$ and $B_0 = 0$, then $A > B$. So the logic expression for $A > B$ is

$$A > B : G = A_1\bar{B}_1 + (A_1 \odot B_1)A_0\bar{B}_0$$

1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or
2. If A_1 and B_1 coincide and $A_0 = 0$ and $B_0 = 1$, then $A < B$. So the expression for $A < B$ is

$$A < B : L = \bar{A}_1 B_1 + (A_1 \odot B_1)\bar{A}_0 B_0$$

If A_1 and B_1 coincide and if A_0 and B_0 coincide then $A = B$. So the expression for $A = B$ is

$$A = B : E = (A_1 \odot B_1)(A_0 \odot B_0)$$

The logic diagram for a 2-bit comparator is as shown in Figure 7.57.

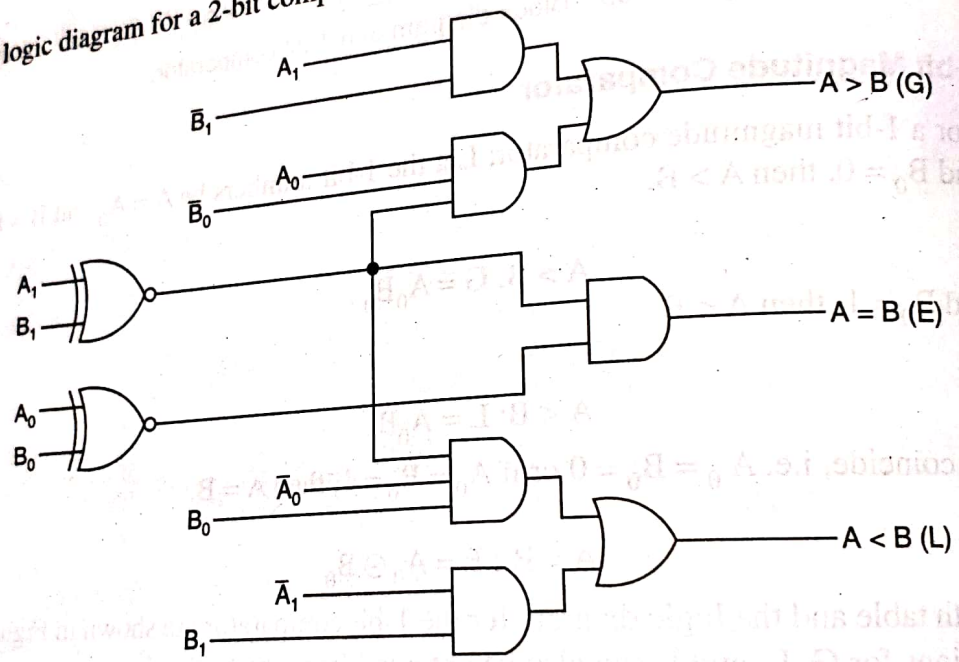


Figure 7.57 Logic diagram of a 2-bit magnitude comparator.

7.18.3 4-bit Magnitude Comparator

The logic for a 4-bit magnitude comparator: Let the two 4-bit numbers be $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.

1. If $A_3 = 1$ and $B_3 = 0$, then $A > B$. Or
2. If A_3 and B_3 coincide, and if $A_2 = 1$ and $B_2 = 0$, then $A > B$. Or
3. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if $A_1 = 1$ and $B_1 = 0$, then $A > B$. Or
4. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if A_1 and B_1 coincide, and if $A_0 = 1$ and $B_0 = 0$, then $A > B$.

From these statements, we see that the logic expression for $A > B$ can be written as

$$(A > B) : G = A_3\bar{B}_3 + (A_3 \odot B_3)A_2\bar{B}_2 + (A_3 \odot B_3)(A_2 \odot B_2)A_1\bar{B}_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)A_0\bar{B}_0$$

Similarly, the logic expression for A < B can be written as

$$(A < B): L = \bar{A}_3B_3 + (A_3 \odot B_3)\bar{A}_2B_2 + (A_3 \odot B_3)(A_2 \odot B_2)\bar{A}_1B_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)\bar{A}_0B_0$$

If A_3 and B_3 coincide and if A_2 and B_2 coincide and if A_1 and B_1 coincide and if A_0 and B_0 coincide, then A = B.

So the expression for A = B can be written as

$$(A = B): E = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$

Figure 7.58 shows the logic diagram of a comparator that implements the logic we have described. Note that, it provides three active-HIGH outputs: A > B, A < B, and A = B.

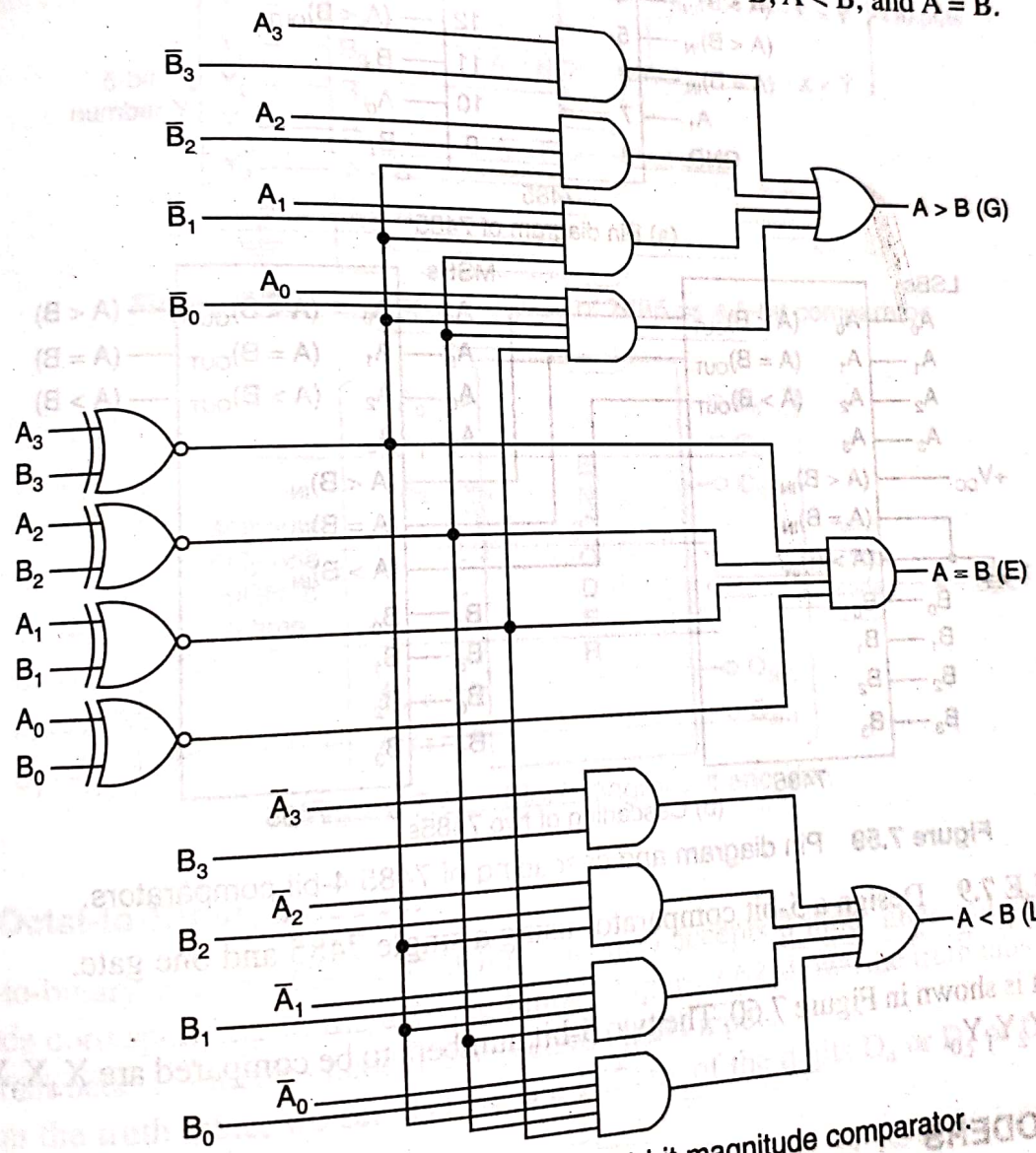
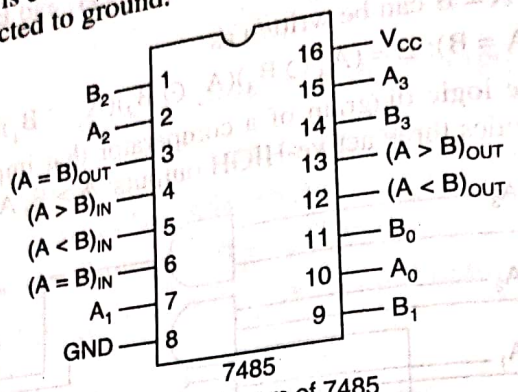


Figure 7.58 Logic diagram of a 4-bit magnitude comparator.

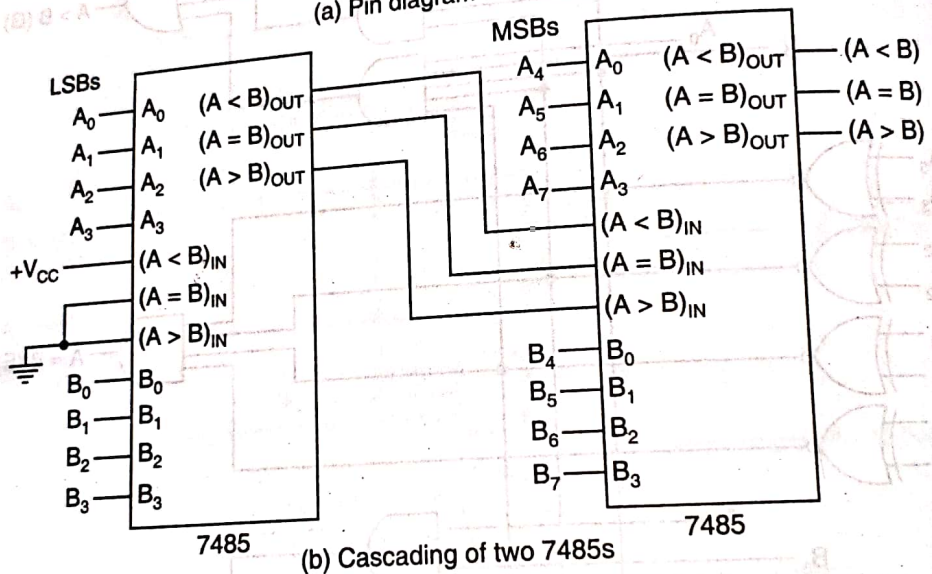
7.19 IC COMPARATOR

Figure 7.59a shows the pin diagram of IC 7485, a 4-bit comparator. Pins labelled $(A < B)_{IN}$, $(A = B)_{IN}$ and $(A > B)_{IN}$ are used for cascading. Figure 7.59b shows how two 4-bit comparators

are cascaded to perform 8-bit comparisons. The $(A < B)_{OUT}$, $(A = B)_{OUT}$ and $(A > B)_{OUT}$ outputs from the lower order comparator used for the least significant 4 bits, are connected to the $(A < B)_{IN}$, $(A = B)_{IN}$, and $(A > B)_{IN}$ inputs of the higher-order comparator. Note that, $(A < B)_{IN}$ input of the lower order comparator is connected to V_{CC}, and $(A = B)_{IN}$ and $(A > B)_{IN}$ inputs of the lower order comparator are connected to ground.



(a) Pin diagram of 7485



(b) Cascading of two 7485s

Figure 7.59 Pin diagram and cascading of 7485 4-bit comparators.

EXAMPLE 7.9 Design a 5-bit comparator using a single 7485 and one gate.

Solution

The circuit is shown in Figure 7.60. The two 5-bit numbers to be compared are $X_4X_3X_2X_1X_0$ and $Y_4Y_3Y_2Y_1Y_0$.

7.20 ENCODERS

An encoder is a device whose inputs are decimal digits and/or alphabetic characters and whose outputs are the coded representation of those inputs, i.e. an encoder is a device which converts familiar numbers or symbols into coded format. In other words, an encoder may be said to be a combinational logic circuit that performs the 'reverse' operation of the decoder. The opposite of the decoding process is called encoding, i.e. encoding is a process of converting familiar numbers

or symbols into a coded format. An encoder has a number of input lines, only one of which is activated at a given time, and produces an N-bit output code depending on which input is activated. Figure 7.61 shows the block diagram of an encoder with M inputs and N outputs. Here the inputs are active HIGH, which means they are normally LOW.

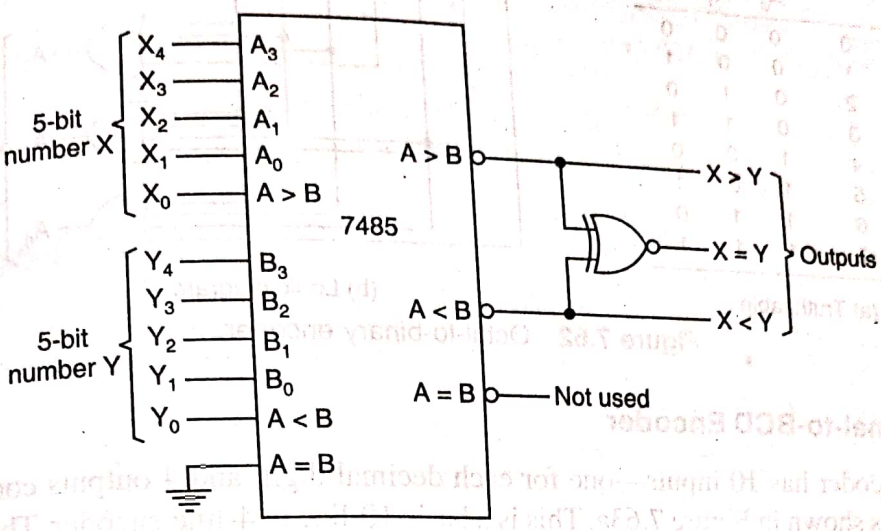


Figure 7.60 Example 7.9: Use of 7485 as a 5-bit comparator.

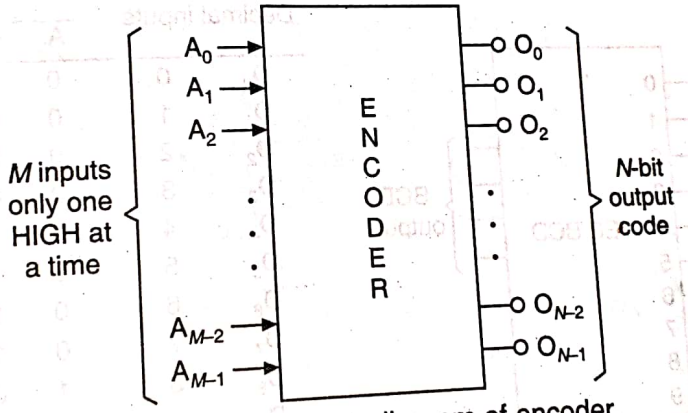


Figure 7.61 Block diagram of encoder.

7.20.1 Octal-to-Binary Encoder

An octal-to-binary encoder (8-line to 3-line encoder) accepts 8 input lines and produces a 3-bit output code corresponding to the activated input. Figure 7.62 shows the truth table and the logic circuit for an octal-to-binary encoder with active HIGH inputs.

From the truth table, we see that A_2 is a 1 if any of the digits D_4 or D_5 or D_6 or D_7 is a 1. Therefore,

$$A_2 = D_4 + D_5 + D_6 + D_7$$

Similarly,

$$A_1 = D_2 + D_3 + D_6 + D_7$$

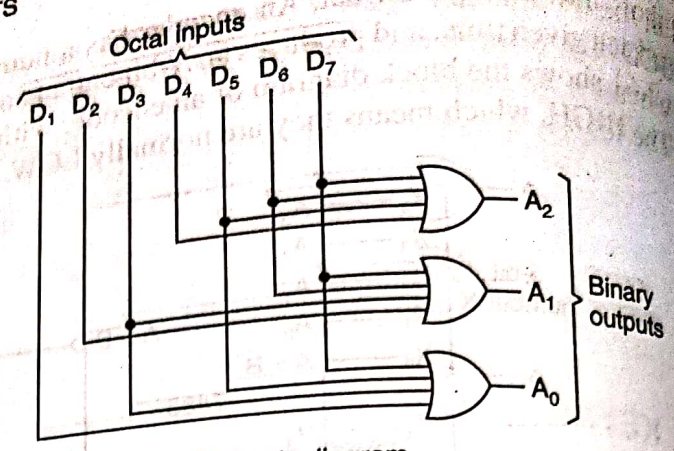
and

$$A_0 = D_1 + D_3 + D_5 + D_7$$

We see that D_0 is not present in any of the expressions. So, D_0 is a don't care.

Octal digits		Binary A_2	A_1	A_0
D_0	0	0	0	0
D_1	1	0	0	1
D_2	2	0	1	0
D_3	3	0	1	1
D_4	4	1	0	0
D_5	5	1	0	1
D_6	6	1	1	0
D_7	7	1	1	1

(a) Truth table

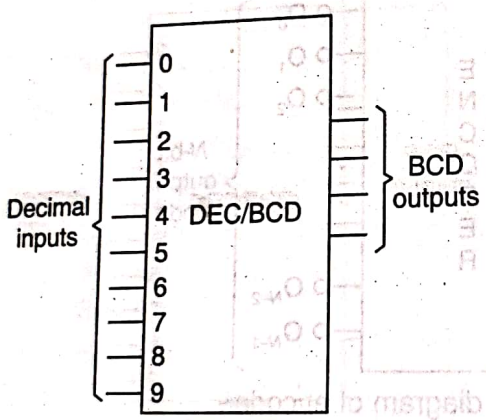


(b) Logic diagram

Figure 7.62 Octal-to-binary encoder.

7.20.2 Decimal-to-BCD Encoder

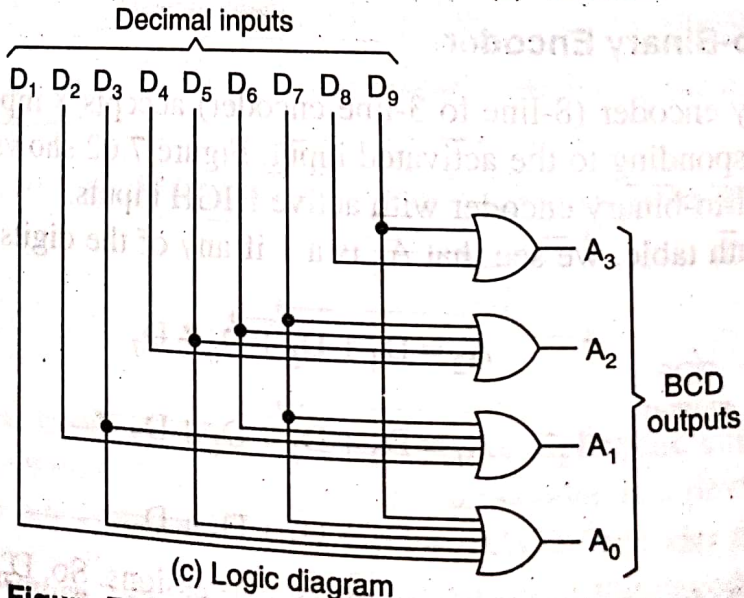
This type of encoder has 10 inputs—one for each decimal digit, and 4 outputs corresponding to the BCD code as shown in Figure 7.63a. This is a basic 10-line to 4-line encoder. The BCD code is



(a) Logic symbol

Decimal inputs		Binary A_3	A_2	A_1	A_0
D_0	0	0	0	0	0
D_1	1	0	0	0	1
D_2	2	0	0	1	0
D_3	3	0	0	1	1
D_4	4	0	1	0	0
D_5	5	0	1	0	1
D_6	6	0	1	1	0
D_7	7	0	1	1	1
D_8	8	1	0	0	0
D_9	9	1	0	0	1

(b) Truth table



(c) Logic diagram

Figure 7.63 Decimal-to-BCD encoder.

listed in the truth table (Figure 7.63b) and from this we can determine the relationships between each BCD bit and the decimal digits. There is no explicit input for a decimal 0. The BCD output is 0000 when the decimal inputs 1– 9 are all 0.

The logic circuit of the encoder is shown in Figure 7.63c. From the table, we get

$$A_3 = D_8 + D_9$$
$$A_2 = D_4 + D_5 + D_6 + D_7$$
$$A_1 = D_2 + D_3 + D_6 + D_7$$
$$A_0 = D_1 + D_3 + D_5 + D_7 + D_9$$

7.21 KEYBOARD ENCODERS

Figure 7.64 shows a typical keyboard encoder consisting of a diode matrix, used to encode the 10 decimal digits in 8-4-2-1 BCD.

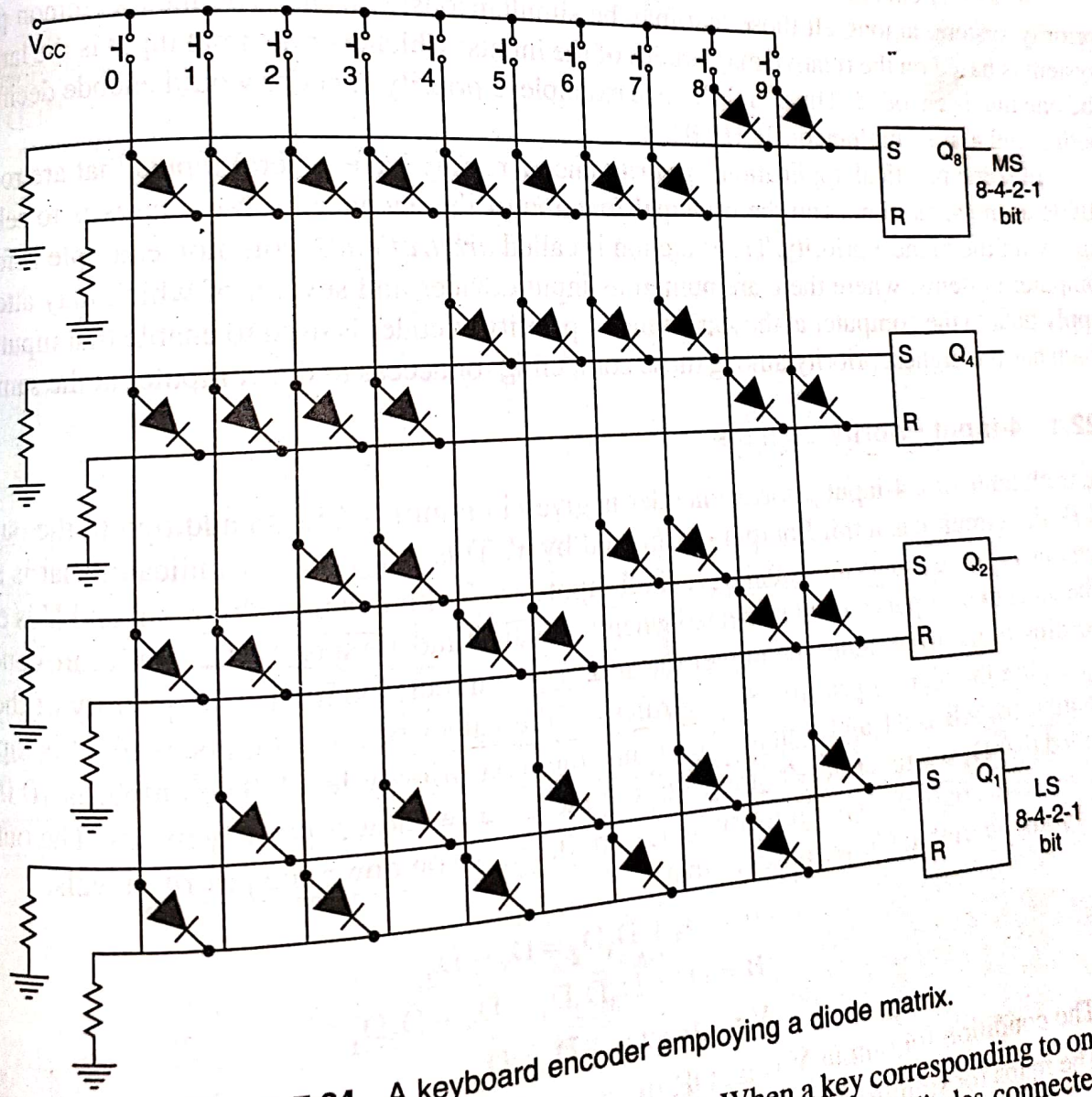


Figure 7.64 A keyboard encoder employing a diode matrix.

The S-R flip-flops are used to store the BCD output. When a key corresponding to one of the decimal [illegible] positive voltage forward biases the selected diodes, connected to the [illegible]ops. The diodes are so arranged that each flip-flop sets

is gi… input and the … inputs. A hexadecimal-to-binary encoder which is also a very useful circuit because of the widespread use of the hexadecimal code in computers, microprocessors, etc. can be designed using this facility.

… are also active LOW … inputs and active LOW outputs. The enable … this operation. Its pin diagram … ering of long

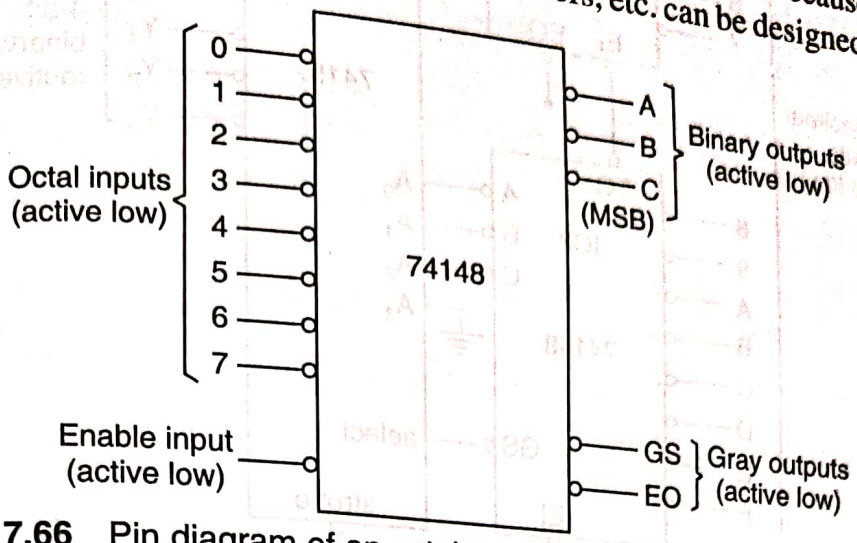


Figure 7.66 Pin diagram of an octal-to-binary priority encoder (IC 74148).

EXAMPLE 7.10 Design a hexadecimal-to-binary encoder using 74148 encoders and 74157 multiplexer.

Solution

Since there are 16 symbols (0–F) in the hexadecimal number system, two 74148 encoders are required. Hexadecimal inputs 0–7 are applied to IC1 input lines and hexadecimal inputs 8–F to IC2 input lines. Whenever one of the inputs of IC2 is active (LOW), IC1 must be disabled. On the other hand, if all the inputs of IC2 are HIGH, then IC1 must be enabled. This is achieved by connecting the EO line of IC2 to the EI line of IC1. A quad 2:1 multiplexer is required to get the proper 4-bit binary outputs. The complete circuit is shown in Figure 7.67. The GS output of 74148 goes LOW whenever one of its inputs is active. Therefore, the GS of IC2 is connected to the SELECT input of 74157. The 74157 selects its A inputs if the SELECT input is LOW, otherwise B inputs are selected. The outputs of the multiplexer are the required binary outputs and are active LOW. This circuit is also a priority encoder.

7.23 DECODERS

A decoder is a logic circuit that converts an N-bit binary input code into M output lines such that only one output line is activated for each one of the possible combinations of inputs. In other words, we can say that a decoder identifies or recognizes or detects a particular code. Figure 7.68 shows the general decoder diagram with N inputs and M outputs. Since each of the N inputs can be a 0 or a 1, there are 2^N possible input combinations or codes. For each of these input combinations,

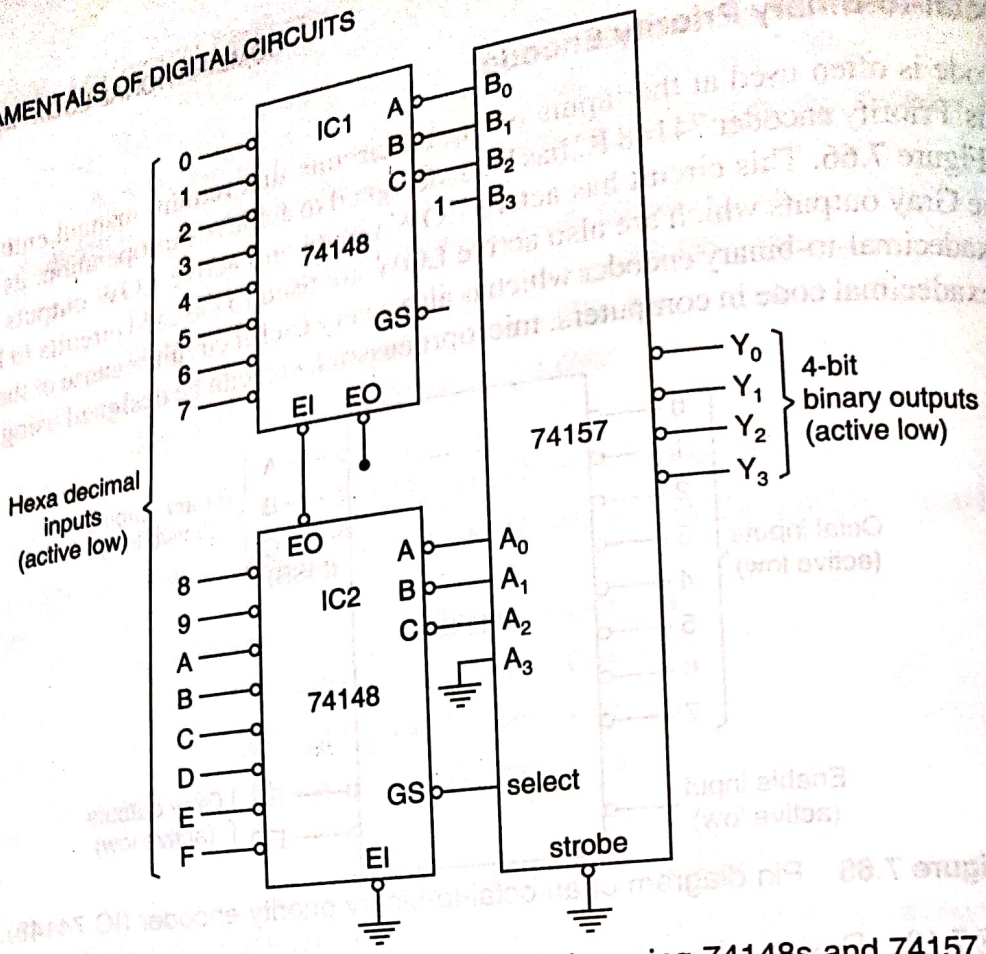


Figure 7.67 Hexadecimal-to-binary encoder using 74148s and 74157.

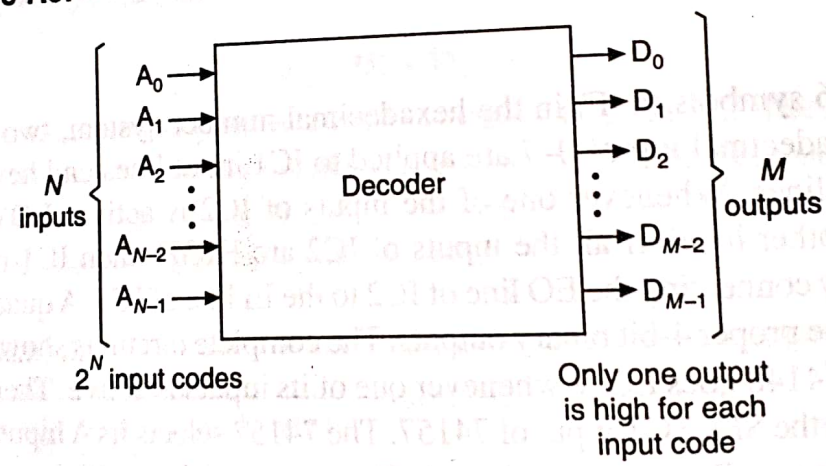


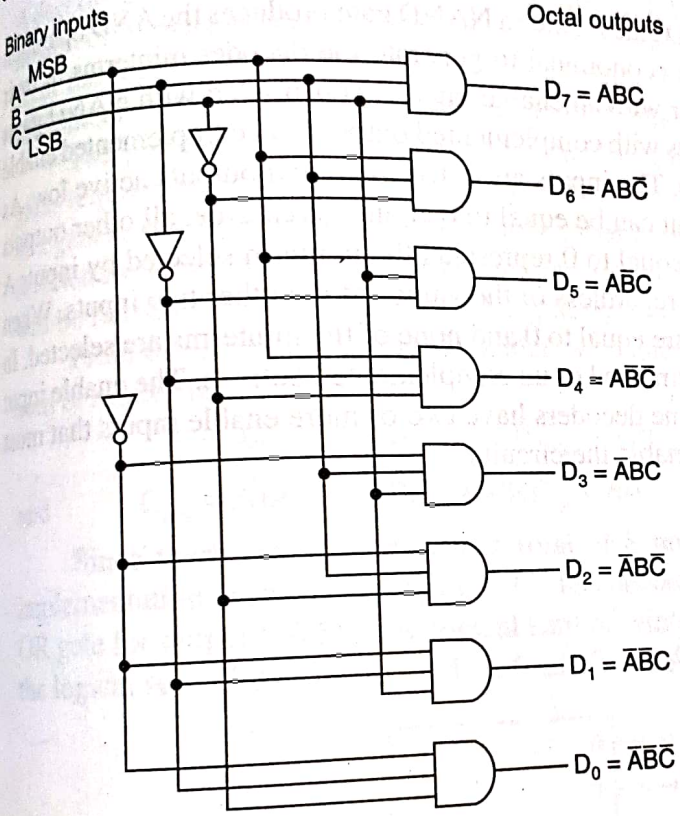
Figure 7.68 General block diagram of a decoder.

only one of the M outputs will be active (HIGH), all the other outputs will remain inactive (LOW). Some decoders are designed to produce active LOW output, while all the other outputs remain HIGH.

Some decoders do not utilize all of the 2^N possible input codes. For example, a BCD to decimal decoder has a 4-bit input code and 10 output lines that correspond to the 10 BCD code groups 0000 through 1001. Decoders of this type are often designed so that if any of the unused codes are applied to the input, none of the outputs will be activated.

7.23.1 3-Line-to-8-Line Decoder

Figure 7.69a shows the circuitry for a decoder with three inputs and eight outputs. It uses all AND gates, and therefore, the outputs are active-HIGH. For active-LOW outputs, NAND gates are used. The truth table of the decoder is shown in Figure 7.69b. This decoder can be referred to in several ways. It can be called a 3-line to 8-line decoder because it has three input lines and eight output lines. It is also called a binary-to-octal decoder because it takes a 3-bit binary input code and activates one of the eight (octal) outputs corresponding to that code. It is also referred to as a *1-of-8 decoder* because only one of the eight outputs is activated at one time.



(a) Logic diagram

Inputs			Outputs							
A	B	C	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

(b) Truth table

Figure 7.69 3-line to 8-line decoder.

7.23.2 Enable Inputs

Some decoders have one or more ENABLE inputs that are used to control the operation of the decoder. For example, in the 3-line to 8-line decoder, if a common ENABLE line is connected to the fourth input of each gate, a particular output as determined by the A, B, C input code will go HIGH only when the ENABLE line is held HIGH. When the ENABLE is held LOW, however, all the outputs will be forced to the LOW state regardless of the levels at the A, B, and C inputs. Thus, the decoder is enabled only when the ENABLE is HIGH. IC74 LS138 is a 3-line to 8-line decoder.

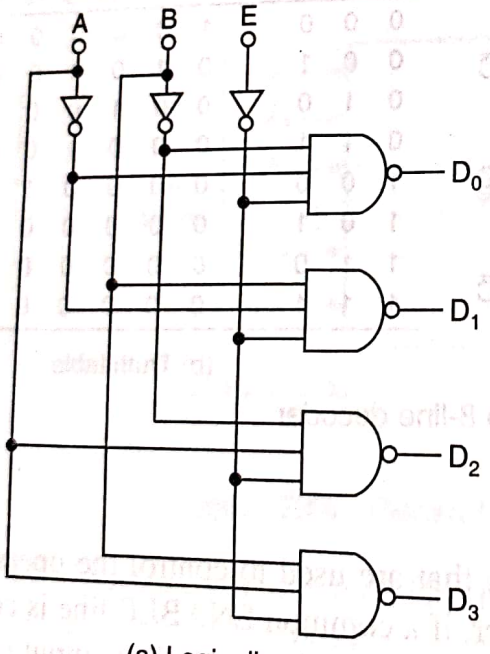
7.23.3 BCD-to-Decimal Decoder (7442)

The BCD to-decimal Decoder is also called a 4-line to 10-line or 4-to-10 decoder or 1-of-10 decoder. It has 4-input lines (for A_3, A_2, A_1, A_0) and 10 output lines (for D_0, D_1, D_2, D_3, D_4, D_5,

D_6, D_7, D_8, D_9). Only one output line is active at time. 6 of the 10 input ... are invalid and for input combinations that are invalid for BCD none of the outputs will be activated. The inputs and outputs can be active high or active low. IC 7442 is a BCD to decimal decoder with active low inputs and outputs. The TTL 7445 IC is a BCD-to-decimal decoder/driver. The term driver is added to its description because this IC has open collector outputs that can operate at higher current and voltage limits than a normal TTL output. It makes 7445 suitable for directly driving loads such as indicator LEDs or lamps, relays or DC motors.

7.23.4 2-Line-to-4-Line Decoder with NAND Gates

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. A 2-to-4 line decoder with an enable input is constructed with NAND gates as shown in Figure 7.70.The circuit operates with complemented outputs and complemented enable input. The decoder is enabled when E = 0. The inputs are active high and outputs active low. As indicated by the truth table, only one output can be equal to 0 at any given time, all other outputs are equal to 1. The output whose value is equal to 0 represents the minterm selected by inputs A and B. The circuit is disabled when E = 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected. In general a decoder may operate with complemented or un complemented outputs. The enable input may be activated with a 0 or a 1 signal. Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuits.



(a) Logic diagram

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

(b) Truth table

Figure 7.70 2 line-to-4 line decoder with NAND gates.

A decoder with enable input can function as a demultiplexer. A demultiplexer is a circuit that receives information from a single line and directs it to one of the 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines.

The decoder of Figure 7.70 can function as a 1-to-4 line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path

to all 4 outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of two selection lines A and B. This can be verified from the truth table of the circuit. For example, if the selection lines AB = 10, the output D_2 will be same as the input value, while all other outputs are maintained at a 1. Because decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a decoder/demultiplexer.

7.23.5 Combinational Logic Implementation

A decoder provides 2^n minterms of n input variables. Since any Boolean function can be expressed in sum of minterms, one can use a decoder to generate the minterms and an external OR gate to form the logic sum. In this way any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean function for the circuit is expressed in sum of minterms. A decoder that generates all the minterms of the input variables is then chosen. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an example that implements a full adder.

From the truth table of the full adder, we obtain the functions for the combinational circuit in sum of minterms.

$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in} = A \oplus B \oplus C_{in} = \Sigma\, m(1, 2, 4, 7)$$

and

$$C_{out} = \bar{A}BC_{in} + A\bar{B}C_{in} + AB\bar{C}_{in} + ABC_{in} = AB + (A \oplus B)C_{in} = \Sigma\, m(3, 5, 6, 7)$$

Since there are 3 inputs and a total of 8 minterms, we need a 3-to-8 line decoder. The implementation is shown in Figure 7.71. The decoder generates the 8 minterms for A, B, C_{in}. The OR gate for output S forms the logical sum of minterms 1, 2, 4 and 7. The OR gate for C_{out} forms the logical sum of the minterms 4, 5, 6 and 7.

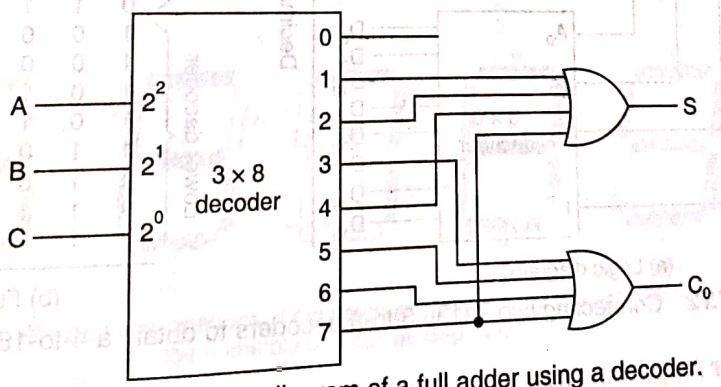


Figure 7.71 Logic diagram of a full adder using a decoder.

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of K minterms can be expressed in its complemented form $\bar{F}$ with 2^n–K terms. If the number of minterms in a function is greater than $2^n/2$, then $\bar{F}$ can be expressed with fewer minterms. In such a case, it is advantageous to use a NOR gate to sum the minterms of $\bar{F}$.

7.24 MULTIPLEXERS (DATA SELECTORS)

Multiplexing means sharing. There are two types of multiplexing—time multiplexing and frequency multiplexing. A common example of multiplexing or sharing occurs when several peripheral devices share a single transmission line or bus to communicate with a computer. To accomplish this sharing, each device in succession is allocated a brief time to send or receive data. At any given time, one and only one device is using the line. This is an example of time multiplexing, since each device is given specific time intervals to use the line. In frequency multiplexing, several devices share a common line by transmitting at different frequencies. In a large mainframe computer, numerous users are time-multiplexed to the computer in such a rapid succession that all appear to be using the computer simultaneously.

A multiplexer (MUX) or data selector is a logic circuit that accepts several data inputs and allows only one of them at a time to get through to the output. The routing of the desired data input to the output is controlled by SELECT inputs (sometimes referred to as ADDRESS inputs). Figure 7.75 shows the functional diagram of a general multiplexer. In this diagram, the inputs and outputs are drawn as large arrows to indicate that they may constitute one or more signal lines. Normally there are 2^n input lines and n select lines whose bit combinations determine which input is selected.

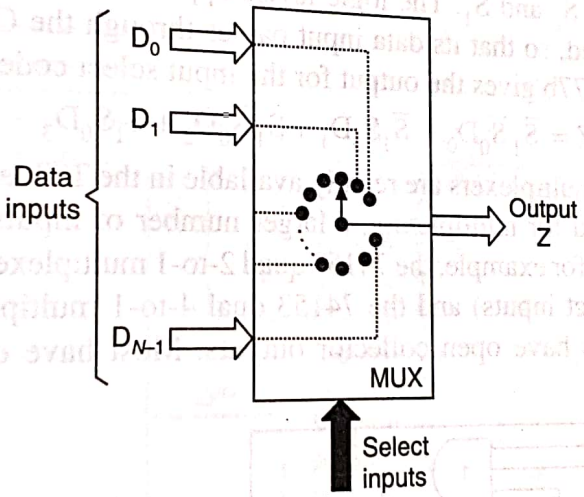


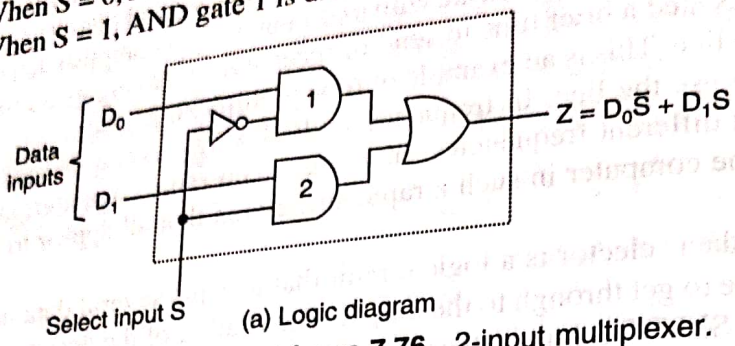
Figure 7.75 Functional diagram of a digital multiplexer.

The multiplexer acts like a digitally controlled multi-position switch. The digital code applied to the SELECT inputs determines which data inputs will be switched to the output. For example, the output Z will equal the data input D_0 for some particular input code; Z will equal D_1 for another particular code, and so on. In other words, we can say that a multiplexer selects 1-out-of-N input data sources and transmits the selected data to a single output channel. This is called *multiplexing*.

7.24.1 Basic 2-Input Multiplexer

Figure 7.76 shows the logic circuitry and function table for a 2-input multiplexer with data inputs D_0 and D_1, and data select input S. It connects two 1-bit sources to a common destination. It has two input lines, one data select line and one output line. The logic level applied to the S input

determines which AND gate is enabled, so that its data input passes through the OR gate to the output. The output, $Z = D_0\overline{S} + D_1S$.
When $S = 0$, AND gate 1 is enabled and AND gate 2 is disabled. So, $Z = D_0$.
When $S = 1$, AND gate 1 is disabled and AND gate 2 is enabled. So, $Z = D_1$.



(a) Logic diagram

S	Output (Z)
0	D_0
1	D_1

(b) Function table

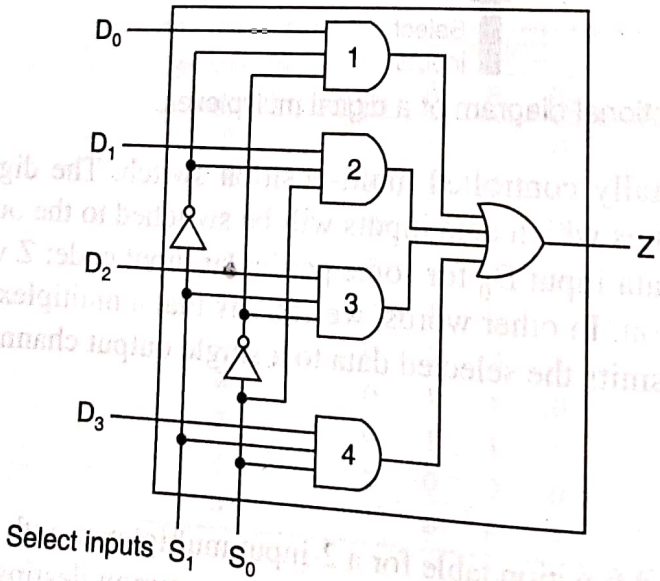
Figure 7.76 2-input multiplexer.

7.24.2 The 4-Input Multiplexer

Figure 7.77a shows the logic circuitry for a 4-input multiplexer with data inputs D_0, D_1, D_2, and D_3, and data select inputs S_0 and S_1. The logic levels applied to the S_0 and S_1 inputs determine which AND gate is enabled, so that its data input passes through the OR gate to the output. The function table in Figure 7.77b gives the output for the input select codes as

$$Z = \overline{S}_1\overline{S}_0D_0 + \overline{S}_1S_0D_1 + S_1\overline{S}_0D_2 + S_1S_0D_3$$

The 2-4-8-16-input multiplexers are readily available in the TTL and CMOS families. These basic ICs can be combined for multiplexing a larger number of inputs. Some packages contain more than one multiplexer, for example, the 74157 quad 2-to-1 multiplexer (four 2-to-1 multiplexers having the same data select inputs) and the 74153 dual 4-to-1 multiplexer. Some designs have 3-state outputs and others have open collector outputs. Most have enable inputs to facilitate cascading.



(a) Logic diagram

Select inputs		Output
S_1	S_0	Z
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

(b) Function table

Figure 7.77 4-input multiplexer.

The AND gates and inverters in the multiplexer resemble a decoder circuit and, indeed, they decode the selection input lines. In general, a 2^n- to-1 line multiplexer is constructed from an n-to-2^n decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate. The size of multiplexer is specified by the number 2^n of its data lines and the single output line. The n selection lines are implied from the 2^n data lines. As in decoders multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled and when it is in the active state, the circuit functions as a normal multiplexer.

7.24.3 The 16-Input Multiplexer from Two 8-Input Multiplexers

Figure 7.78 shows an arrangement to use two 8-input multiplexers (74151A) to get a 16-input multiplexer. One OR gate and one inverter are also required. The four select inputs S_3, S_2, S_1, and S_0 will select one of the 16 inputs to pass through to X. The S_3 input determines which multiplexer is enabled. When $S_3 = 0$, the left multiplexer is enabled and S_2, S_1, and S_0 inputs determine which of its data inputs will appear at its output and pass through the OR gate to X. When $S_3 = 1$, the right multiplexer is enabled and S_2, S_1, and S_0 inputs select one of its data inputs for passage to output X. This arrangement is also called multiplexer tree. Figure 7.94 shows an arrangement to obtain a 32 × 1 mux using two 16 × 1 muxes and one 2 × 1 mux.

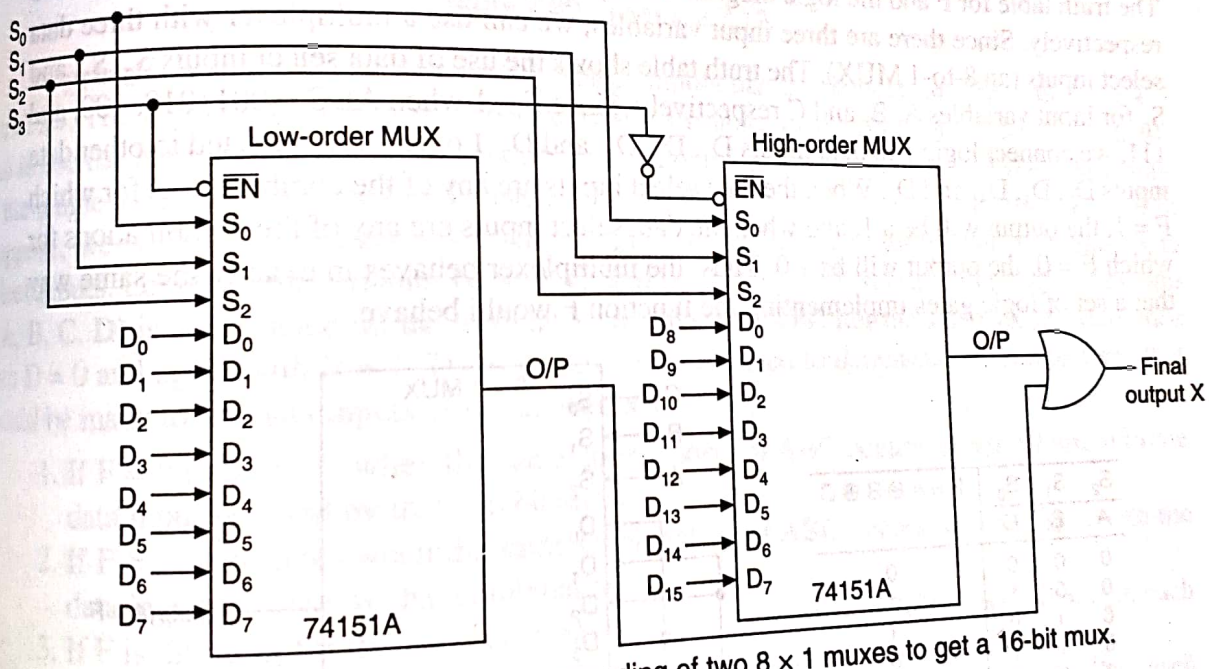


Figure 7.78 Logic diagram for cascading of two 8 × 1 muxes to get a 16-bit mux.

7.25 APPLICATIONS OF MULTIPLEXERS

Multiplexers find numerous and varied applications in digital systems of all types. These applications include data selection, data routing, operation sequencing, parallel-to-serial conversion, waveform generation, and logic function generation.

7.25.1 Logic Function Generator

A multiplexer can be used in place of logic gates to implement a logic expression. It can be so connected that it duplicates the logic of any truth table, i.e. it can generate any Boolean algebraic function of a set of input variables. In such applications, the multiplexer can be viewed as a function generator, because we can easily set or change the logic function it implements. One advantage of using a multiplexer in place of logic gates is that, a single integrated circuit can perform a function that might otherwise require numerous integrated circuits. Moreover, it is very easy to change the logic function implemented, if and when redesign of a system becomes necessary.

The first step in the design of a function generator using a multiplexer is to construct a truth table for the function to be implemented. Then, connect logic 1 to each data input of the multiplexer corresponding to each combination of the input variables, for which the truth table shows the function to be equal to 1. Logic 0 is connected to the remaining data inputs. The variables themselves are connected to the data select inputs of the multiplexer. For example, suppose the truth table specifies that the function F equals 1 for the input combination 110. So, when $S_2S_1S_0 = 110$, the data input 6, which is connected to logic 1, will be selected. This will route a 1 to the output of the multiplexer.

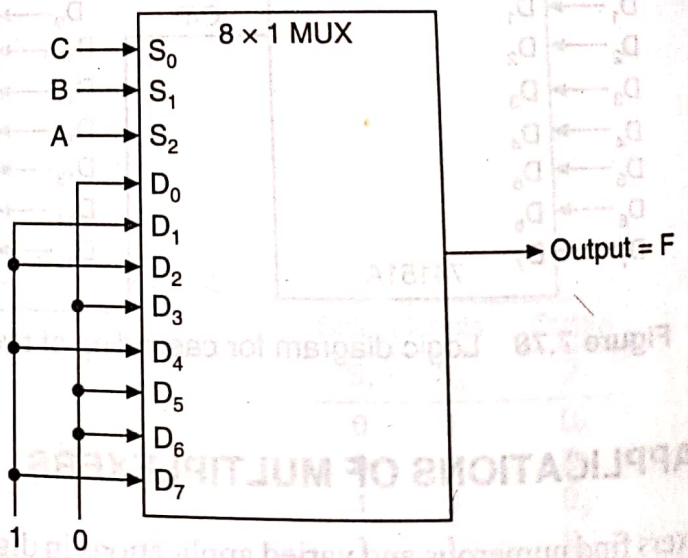
EXAMPLE 7.11 Use a multiplexer to implement the logic function $F = A \oplus B \oplus C$.

Solution

The truth table for F and the logic diagram to implement F are shown in Figures 7.79a and b respectively. Since there are three input variables, we can use a multiplexer with three data select inputs (an 8-to-1 MUX). The truth table shows the use of data select inputs S_2, S_1, and S_0 for input variables A, B, and C respectively. Since F = 1 when ABC = 001, 010, 100, and 111, we connect logic 1 to data inputs D_1, D_2, D_4, and D_7. Logic 0 is connected to other data inputs D_0, D_3, D_5, and D_6. When the data select inputs are any of the combinations for which F = 1, the output will be a 1, and when the data select inputs are any of the combinations for which F = 0, the output will be a 0. Thus, the multiplexer behaves in exactly the same way that a set of logic gates implementing the function F would behave.

S_2	S_1	S_0	F = A ⊕ B ⊕ C
A	B	C	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a) Truth table



(b) Logic diagram

Figure 7.79 Example 7.11: Use of 74151A to implement the logic function $F = A \oplus B \oplus C$.

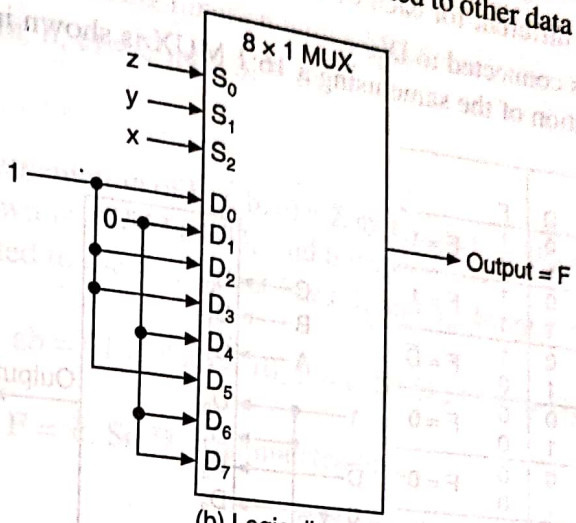
EXAMPLE 7.12 Implement the following function using 8-to-1 MUX

$$F(x, y, z) = \Sigma\, m(0, 2, 3, 5)$$

Solution
The truth table for F and the logic diagram to implement the function F using an 8 : 1 MUX are shown in Figures 7.80a and b respectively. The inputs x, y and z are applied to the data select inputs S_2, S_1 and S_0 respectively. Since F = 1 when xyz = 000, 010, 011, and 101, logic 1 is connected to data inputs D_0, D_2, D_3 and D_5. Logic 0 is connected to other data inputs D_1, D_4, D_6 and D_7.

S_2	S_1	S_0	
x	y	z	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

(a) Truth table



(b) Logic diagram

Figure 7.80 Example 7.12.

In general, a multiplexer with *n*-data select inputs can implement any function of $n + 1$ variables. The key to this design is to use the first *n* variables of the function as the select inputs and to use the least significant input variable and its complement to drive some of the data inputs. If the single variable is denoted by D, each data output of the multiplexer will be D, $\bar{D}$, 1, or 0. Suppose, we wish to implement a 4-variable logic function using a multiplexer with three data select inputs. Let the input variables be A, B, C, and D; D is the LSB. A truth table for the function F(A, B, C, D) is constructed. In the truth table, we note that ABC has the same value twice once with D = 0 and again with D = 1. The following rules are used to determine the connections that should be made to the data inputs of the multiplexer.

1. If F = 0 both times when the same combination of ABC occurs, connect logic 0 to the data input selected by that combination.
2. If F = 1 both times when the same combination of ABC occurs, connect logic 1 to the data input selected by that combination.
3. If F is different for the two occurrences of a combination of ABC, and if F = D in each case, connect D to the data input selected by that combination.
4. If F is different for the two occurrences of a combination of ABC, and if F = $\bar{D}$ in each case, connect $\bar{D}$ to the data input selected by that combination.

EXAMPLE 7.13 Use a multiplexer having three data select inputs to implement the logic for the function given below. Also realize the same using a 16:1 MUX.

$$F = \Sigma\, m(0, 1, 2, 3, 4, 10, 11, 14, 15).$$

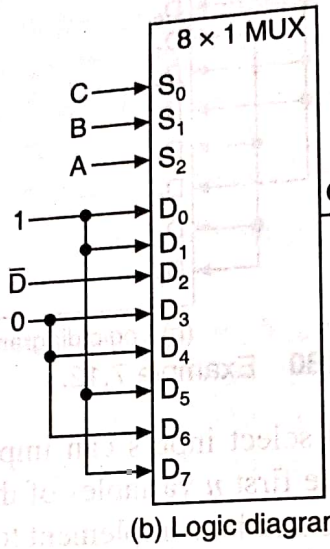
Solution
The truth table for the given function is shown in Figure 7.81a. Since the given function is of four variables, we can use a multiplexer with three data select inputs (i.e. 8:1 mux) as shown

in Figure 7.81b. As seen from the truth table, since F is same for each of the two occurrences of ABC = 000, ABC = 001, ABC = 101, and ABC = 111 and since F = 1 in both cases, 1 is connected to D_0, D_1, D_5, D_7. Since F is the same for each of two occurrences of ABC = 011, ABC = 100 and ABC = 110 and since F = 0 in both cases, 0 is connected to D_3, D_4 and D_6. Since F is different for each of the two occurrences of ABC = 010 and since F = $\overline{D}$ in both cases, $\overline{D}$ is connected to D_2.

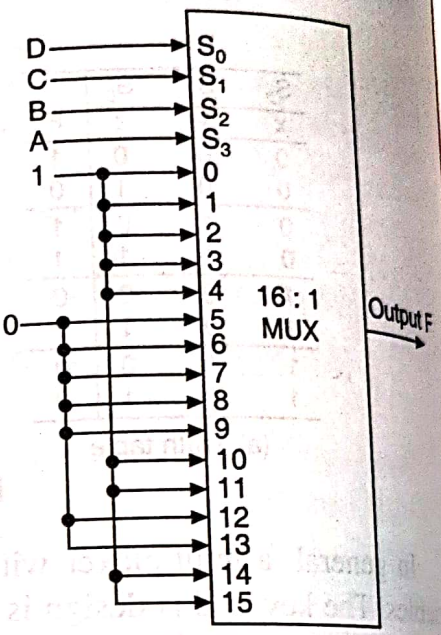
Realization of the same using a 16:1 MUX is shown in Figure 7.81c.

S_2 A	S_1 B	S_0 C	D	F	
0	0	0	0	1	F = 1
0	0	0	1	1	
0	0	1	0	1	F = 1
0	0	1	1	1	
0	1	0	0	1	F = $\overline{D}$
0	1	0	1	0	
0	1	1	0	0	F = 0
0	1	1	1	0	
1	0	0	0	0	F = 0
1	0	0	1	0	
1	0	1	0	1	F = 1
1	0	1	1	1	
1	1	0	0	0	F = 0
1	1	0	1	0	
1	1	1	0	1	F = 1
1	1	1	1	1	

(a) Truth table



(b) Logic diagram



(c) Logic diagram

Figure 7.81 Example 7.13.

EXAMPLE 7.14 Use a 4 × 1 MUX to implement the logic function

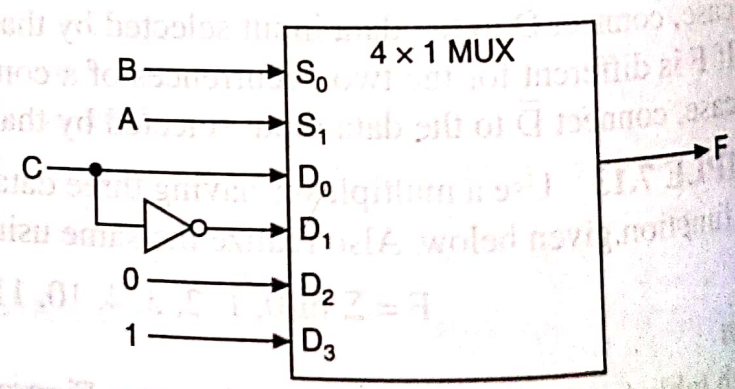
$$F(A, B, C) = \Sigma\, m(1, 2, 4, 7).$$

Solution

The logic function $F(A, B, C) = \Sigma\, m(1, 2, 4, 7)$ can be implemented with a 4-to-1 multiplexer as shown in Figure 7.82. The given function is a 3 variable function. The two variables A and B are applied to the selection lines in that order. A is connected to S_1 input and B is connected to S_0 input. The values for the data input lines are determined from the truth table of the function. When

S_1 A	S_0 B	C	F	
0	0	0	0	F = C
0	0	1	1	
0	1	0	1	F = $\overline{C}$
0	1	1	0	
1	0	0	0	F = 0
1	0	1	0	
1	1	0	1	F = 1
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Figure 7.82 Example 7.14.

The inputs A, B, and C are connected to the data select lines S_2, S_1, and S_0. From the truth table we observe that

1. For both values of ABC = 000, F = 0. So D_0 is connected to 0.
2. For both values of ABC = 100, and ABC = 101, F = 1. So D_4 and D_5 are connected to 1.
3. For both values of ABC = 010, ABC = 011, ABC = 110, and ABC = 111, F = D. So D_2, D_3, D_6 and D_7 are connected to D.
4. For both values of ABC = 001, $F = \overline{D}$. So D_1 is connected to $\overline{D}$.

7.26 DEMULTIPLEXERS (DATA DISTRIBUTORS)

A multiplexer takes several inputs and transmits one of them to the output. A demultiplexer performs the reverse operation; it takes a single input and distributes it over several outputs. So a demultiplexer can be thought of as a 'distributor', since it transmits the same data to different destinations. Thus, whereas a multiplexer is an *N*-to-1 device, a demultiplexer is a 1-to-*N* (or 2^n) device. Figure 7.88 shows the functional diagram for a demultiplexer (DEMUX). The large arrows for inputs and outputs can represent one or more lines. The 'select' input code determines the output line to which the input data will be transmitted. In other words, the demultiplexer takes one input data source and selectively distributes it to 1-of-*N* output channels just like a multi-position switch.

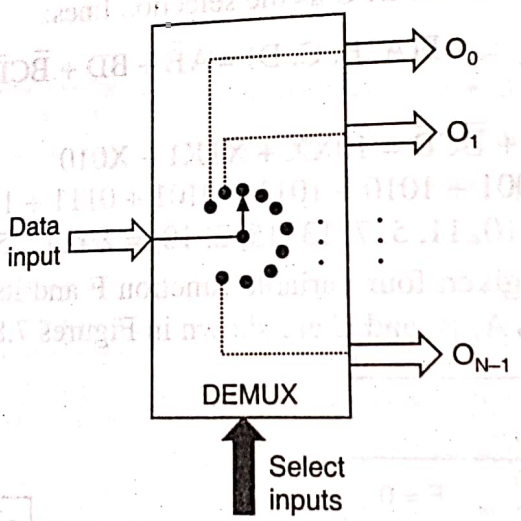


Figure 7.88 Functional diagram of a general demultiplexer.

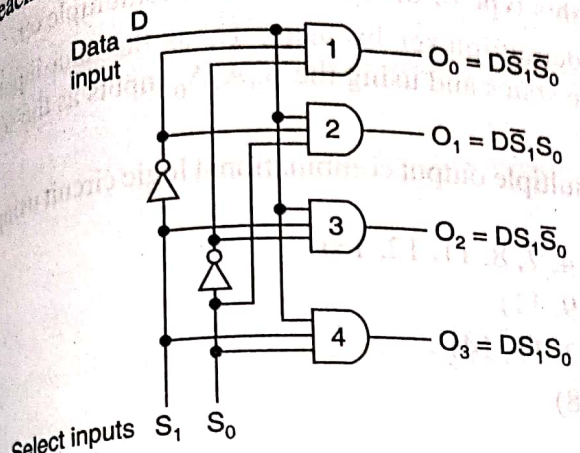
7.26.1 1-Line to 4-Line Demultiplexer

Figure 7.89 shows a 1-line to 4-line demultiplexer circuit. The input data line goes to all of the AND gates. The two select lines S_0 and S_1 enable only one gate at a time, and the data appearing on the input line will pass through the selected gate to the associated output line.

7.26.2 1-Line to 8-Line Demultiplexer

Figure 7.90a shows the logic diagram for a demultiplexer that distributes one input line to eight output lines. The single data input line D is connected to all eight AND gates, but only one of these gates will be enabled by the select input lines. For example, with $S_2\ S_1\ S_0 = 000$, only the AND gate

O_0 will be enabled, and the data input D will appear at output O_0. Other select codes cause input D to reach the other outputs. The truth table in Figure 7.90b summarizes the operation.

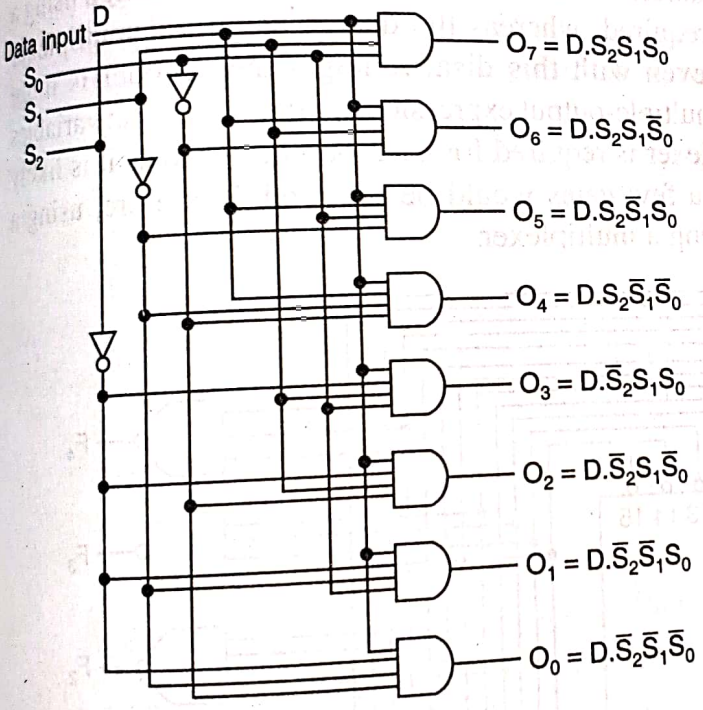


(a) Logic diagram

Select code		Outputs			
S_1	S_0	O_3	O_2	O_1	O_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

(b) Truth table

Figure 7.89 1-line to 4-line demultiplexer.



(a) Logic diagram

Select code			Outputs							
S_2	S_1	S_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	D
0	0	1	0	0	0	0	0	0	D	0
0	1	0	0	0	0	0	0	D	0	0
0	1	1	0	0	0	0	D	0	0	0
1	0	0	0	0	0	D	0	0	0	0
1	0	1	0	0	D	0	0	0	0	0
1	1	0	0	D	0	0	0	0	0	0
1	1	1	D	0	0	0	0	0	0	0

(b) Truth table

Figure 7.90 1-line to 8-line demultiplexer.

The demultiplexer circuit of Figure 7.90a is very similar to the 3-line to 8-line decoder circuit of Figure 7.69a, except that a fourth input D has been added to each gate. The inputs ABC of Figure 7.69b are here labelled S_2 S_1 S_0 and become the data select inputs.

In the 3-to-8 IC decoder, there are three input lines and eight output lines. The enable input $\bar{E}$ is used to enable or disable the decoding process. This 3-to-8 decoder can be used as a 1-to-8 demultiplexer as follows.

The enable input $\overline{E}$ is used as the data input D, and the binary code inputs are used as the select inputs. Depending on the select inputs, the data input will be routed to a particular output. For this reason, the IC manufacturers often call this type of device a decoder/demultiplexer.

The 74LS138 decoder can be used as a demultiplexer by using $\overline{E}_1$ as the data input D, holding the other two enable inputs in their active states and using the $A_2A_1A_0$ inputs as the select code.

EXAMPLE 7.20 Implement the following multiple output combinational logic circuit using a 4-line to 16-line decoder.

$$F_1 = \Sigma\, m(1, 2, 4, 7, 8, 11, 12, 13)$$
$$F_2 = \Sigma\, m(2, 3, 9, 11)$$
$$F_3 = \Sigma\, m(10, 12, 13, 14)$$
$$F_4 = \Sigma\, m(2, 4, 8)$$

Solution

The realization of the given multiple output logic circuit using a 4-line to 16-line decoder is shown in Figure 7.91. The decoder's outputs are active LOW; therefore, a NAND gate is required for every output of the combinational circuit. In combinational logic design using a multiplexer, additional gates are not required, whereas the design using a demultiplexer requires additional gates. However, even with this disadvantage, the decoder is more economical in cases where non-trivial, multiple-output expressions of the same input variables are required. In such cases, one multiplexer is required for each output, whereas it is likely that only one decoder supported with a few gates would be required. Therefore, using a decoder could have advantages over using a multiplexer.

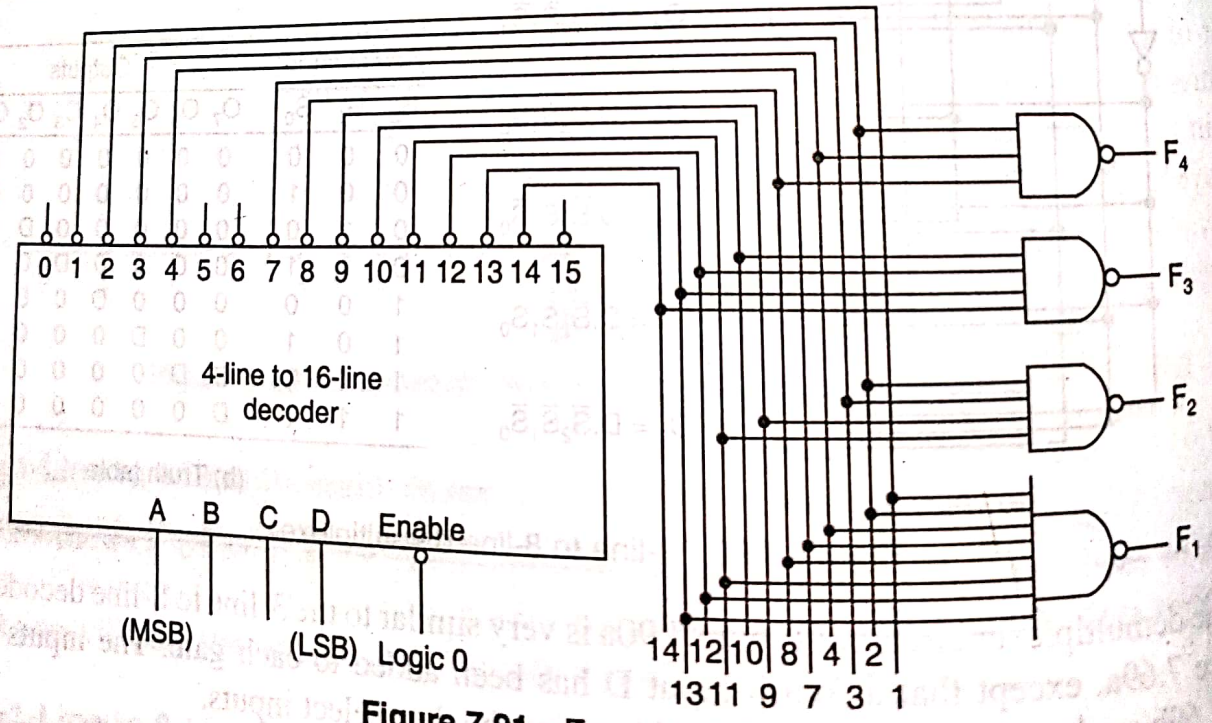


Figure 7.91 Example 7.20.

7.26.3 Demultiplexer Tree

Since 4-line to 16-line decoders are the largest available circuits in ICs, to meet the still larger input needs, there should be a provision for expansion. This is made possible by using enable input terminal. Figure 7.92 shows a 5-line

Once programmed, the data pattern can
referred to as ROM. ROMs are highly suited for very

Programmable read-only memory (PROM)

This type of memory comes from the manufacturer without any data stored in it, i.e. empty. The data pattern is programmed electrically by the user using a special circuit known as PROM *programmer*. It can be programmed only once during its life time. Once programmed, the data cannot be altered. This type of memory is known as PROM. These are highly suited for high volume usage due to their low cost of production.

Erasable programmable read-only memory (EPROM)

In this type of memory, data can be written any number of times, i.e. they are reprogrammable. Before it is reprogrammed, the contents already stored are erased by exposing the chip to ultraviolet radiation for about 30 minutes. This type of memory is referred to as EPROM. EPROMs are possible only in MOS technology. Programming is done using a PROM programmer.

Electrically erasable and programmable read-only memory (EEPROM or E^2PROM)

This is another type of reprogrammable memory in which erasing is done electrically rather than exposing the chip to the ultraviolet radiation. It is referred to as EEPROM or electrically alterable ROM (EAROM).

8.6 COMBINATIONAL PROGRAMMABLE LOGIC DEVICES

A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of products implementation. There are three major types of combinational PLDs and they differ in the placement of the programmable connection in the AND-OR array. The various PLDs used are PALs (programmable array logics), PLAs (programmable logic arrays) and PROMs (programmable read only memories).

Figure 8.7 shows the configuration of the 3 PLDs. The programmable read-only memory (PROM) has a fixed AND array constructed as a decoder and a programmable OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate. The programmable array logic (PAL) has programmable AND array and

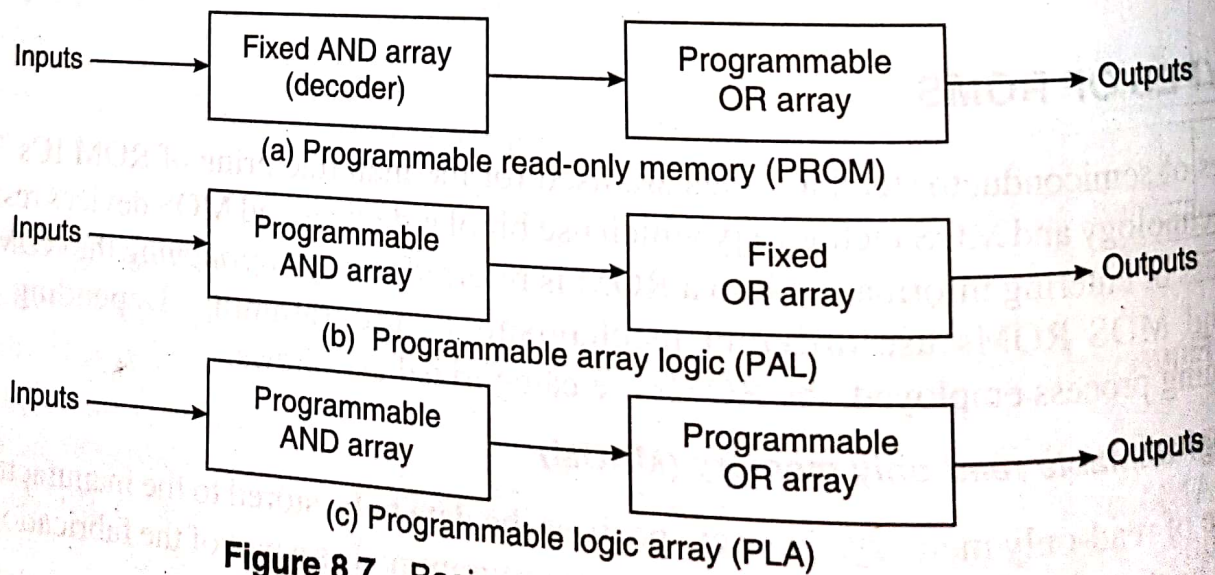


Figure 8.7 Basic configuration of three PLDs.

a fixed OR array. The most flexible PLD is the programmable logic array (PLA) where both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum of products implementation.

8.7 PROGRAMMABLE ARRAY LOGIC (PAL)

Programmable array logic (a registered trade mark of Monolithic Memories) is a particular family of programmable logic devices (PLDs) that is widely used and available from a number of manufacturers. The PAL circuits consist of a set of AND gates whose inputs can be programmed and whose outputs are connected to an OR gate, i.e. the inputs to the OR gate are hard-wired, i.e. PAL is a PLD with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program but is not as flexible as the PLA. Some manufacturers also allow output inversion to be programmed. Thus, like AND-OR and AND-OR-INVERT logic, they implement a sum of products logic function. Figure 8.8a shows a small example of the basic structure. The fuse symbols represent fusible links that can be burned open using equipment similar to a PROM programmer. Note that every input variable and its complement can be left either connected or disconnected from every AND gate. We then say that the AND gates are programmed. Figure 8.8b shows how the circuit is programmed to implement $F = \bar{A}BC + A\bar{B}C$. Note this important point. All input variables and their complements are left connected to the unused AND gate, whose output is, therefore, $A\bar{A}B\bar{B}C\bar{C} = 0$. The 0 has no affect on the output of the OR gate. On the other hand, if all inputs to the unused AND gate were burned open, the output of the AND gate would 'float' HIGH (logic 1), and the output of the OR gate in that case would remain permanently 1. The actual PAL circuits have several groups of AND gates, each group providing inputs to separate OR gates.

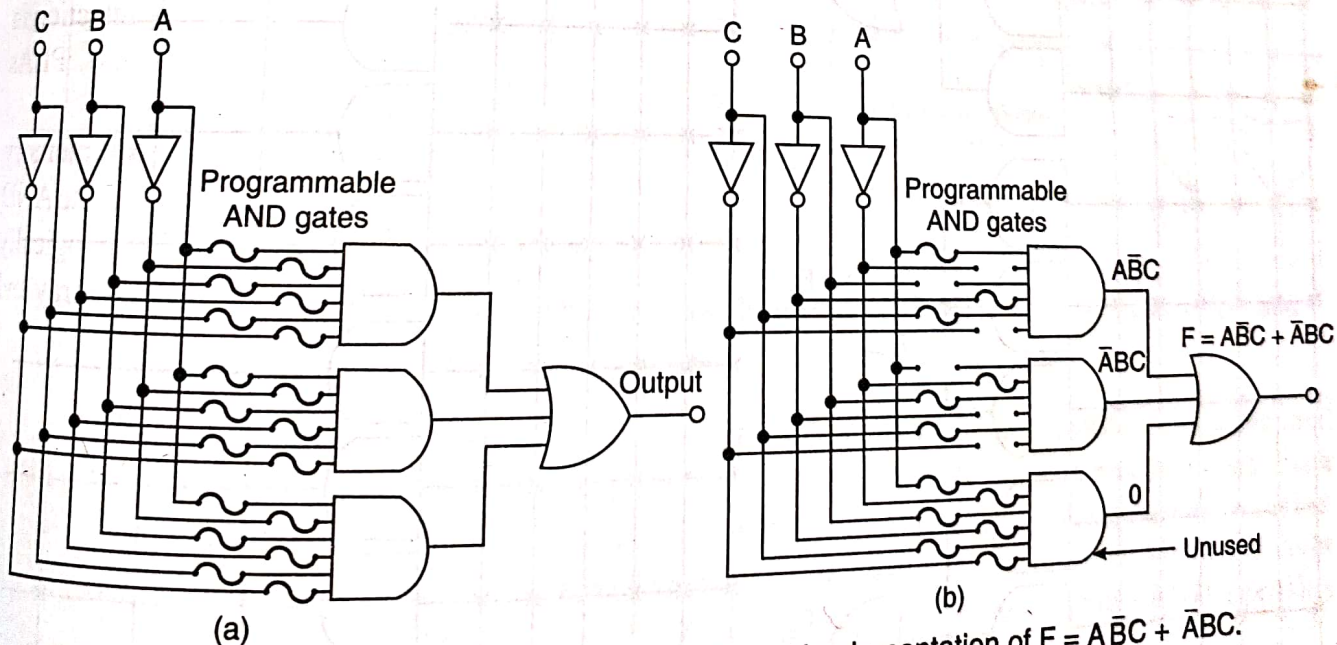


Figure 8.8 Basic structure of a PAL circuit, and implementation of $F = A\bar{B}C + \bar{A}BC$.

Figure 8.9a shows a conventional means for abbreviating PAL connection diagrams. Note that the AND gate is drawn with a single input line, whereas in reality, it has three inputs. The vertical lines denote the inputs and the horizontal lines feed the AND gates. An × sign denotes a connection through an intact fusible link and a dot sign represents a permanent connection. The

absence of any symbol represents an open or no connection by virtue of a burned-open link. In the example shown, input A is connected to the gate through a fusible link, input C is permanently connected, and input B is disconnected. Therefore, the output of the gate is AC.

Figure 8.9b shows an example of how the PAL structure is represented using the abbreviated connections. It is a 3-input 3-wide AND-OR structure. In this example, each function can have three minterms or product terms. Notice that there are nine AND gates, which implies only nine chosen products of not more than three variables ABC. Inputs to the OR gates at the outputs are fixed as shown by ×s marked on the vertical lines. The inputs to the AND gates are marked on the corresponding line by the ×s. Removing the × implies blowing off the corresponding fuse which in turn implies that the corresponding input variable is not applied to the particular AND gate. In this example, the circuit is unprogrammed because all the fusible links are intact. Note that, the 3-input OR gates in Figure 8.9c are also drawn with a single input line.

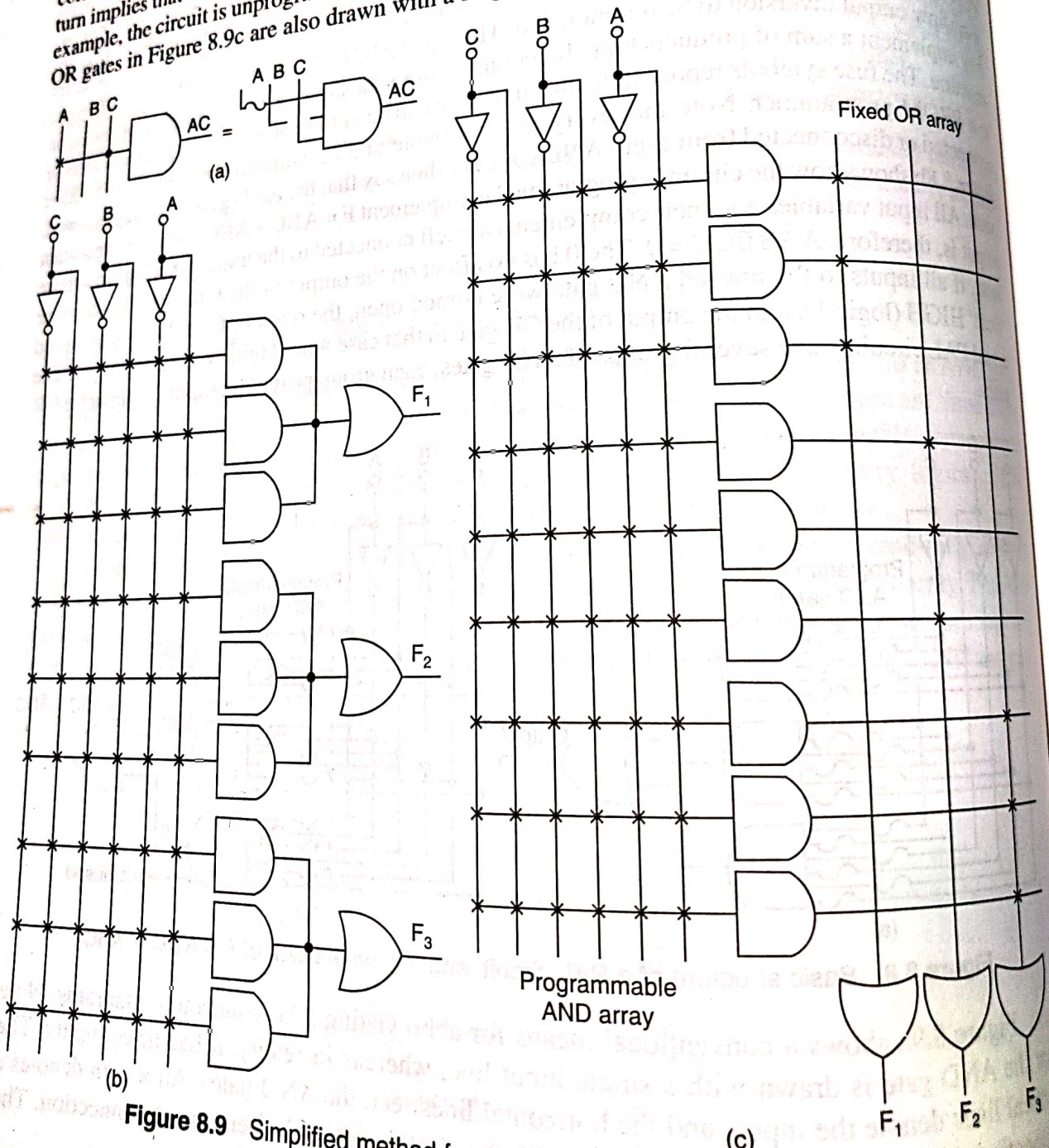


Figure 8.9 Simplified method for showing connections in PAL circuits.

EXAMPLE 8.4 Using the connection abbreviations, redraw the circuit in Figure 8.9c to show how it can be programmed to implement $F_1 = \bar{A}BC + A\bar{C} + A\bar{B}C$ and $F_2 = \bar{A}\bar{B}\bar{C} + BC$.

Solution

The redrawn circuit to implement the given functions is shown in Figure 8.10a. Note that one unused AND gate has all its links intact. All links intact can be represented by a × in the AND gate as shown in Figure 8.10b. Such a diagram is sometimes called the *fuse map*.

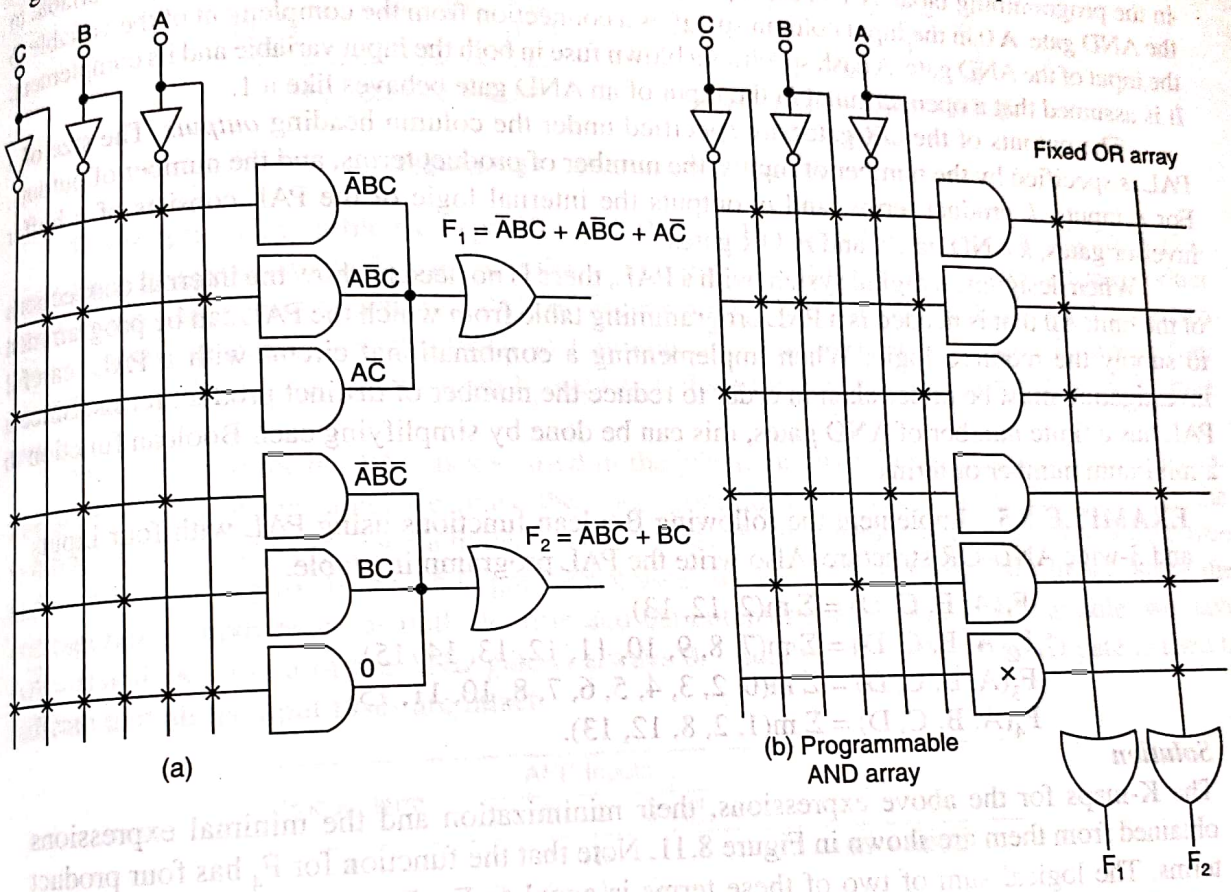


Figure 8.10 Example 8.4: PAL programmed to implement $F_1 = \bar{A}BC + A\bar{B}C + A\bar{C}$ and $F_2 = \bar{A}\,\bar{B}\,\bar{C} + BC$.

An example of an actual PAL IC is the PAL 18L8A from Texas Instruments. It is manufactured using low power Schottky technology and has ten logic inputs and eight output functions. Each output OR gate is hard-wired to seven AND gate outputs and therefore it can generate functions that include up to seven terms. An added feature of this particular PAL is that six of the eight outputs are fed back into AND array, where they can be connected as inputs to any AND gate. This makes the device very useful in generating all sorts of combinational logic.

8.7.1 PAL Programming Table

The fuse map of a PAL can be specified in a tabular form. The PAL programming table consists of three columns. The first column lists the product terms numerically. The second column specifies

the required paths between inputs and AND gates. The third column specifies the outputs of the OR gates. For each product term the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its true form, the corresponding input variable is marked with a 1. If it appears in complemented form, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked as a – (dash).

The paths between the inputs and the AND gates are specified under the column heading *inputs* in the programming table. A 1 in the input column specifies a connection from the input variable to the AND gate. A 0 in the input column specifies a connection from the complement of the variable to the input of the AND gate. A dash specifies a blown fuse in both the input variable and its complement. It is assumed that a open terminal in the input of an AND gate behaves like a 1.

The outputs of the OR gates are specified under the column heading *outputs*. The size of a PAL is specified by the number of inputs, the number of product terms, and the number of outputs. For n inputs, k product terms, and m outputs the internal logic of the PAL consists of n buffer inverter gates, k AND gates, and m OR gates.

When designing a digital system with a PAL, there is no need to show the internal connections of the unit. All that is needed is a PAL programming table from which the PAL can be programmed to supply the required logic. When implementing a combinational circuit with a PAL, careful investigation must be undertaken in order to reduce the number of distinct product terms. Since a PAL has a finite number of AND gates, this can be done by simplifying each Boolean function to a minimum number of terms.

EXAMPLE 8.5 Implement the following Boolean functions using PAL with four inputs and 3-wide AND-OR structure. Also write the PAL programming table.

$$F_1(A, B, C, D) = \Sigma\, m(2, 12, 13)$$
$$F_2(A, B, C, D) = \Sigma\, m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$
$$F_3(A, B, C, D) = \Sigma\, m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$
$$F_4(A, B, C, D) = \Sigma\, m(1, 2, 8, 12, 13).$$

Solution

The K-maps for the above expressions, their minimization and the minimal expressions obtained from them are shown in Figure 8.11. Note that the function for F_4 has four product terms. The logical sum of two of these terms is equal to F_1. By using F_1 it is possible to reduce the number of terms for F_4 from four to three. The implementation of the minimal logic expressions using PAL is shown in Figure 8.12b.

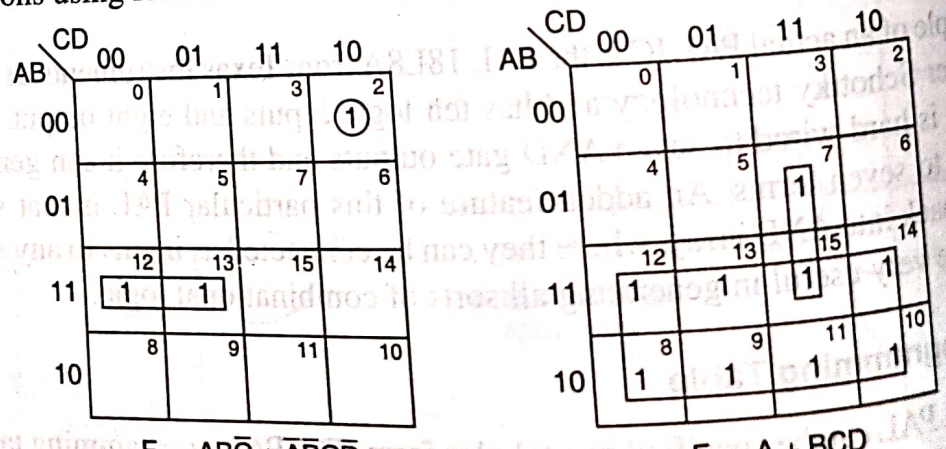


Figure 8.11 Example 8.5: K-maps *(Contd.)*

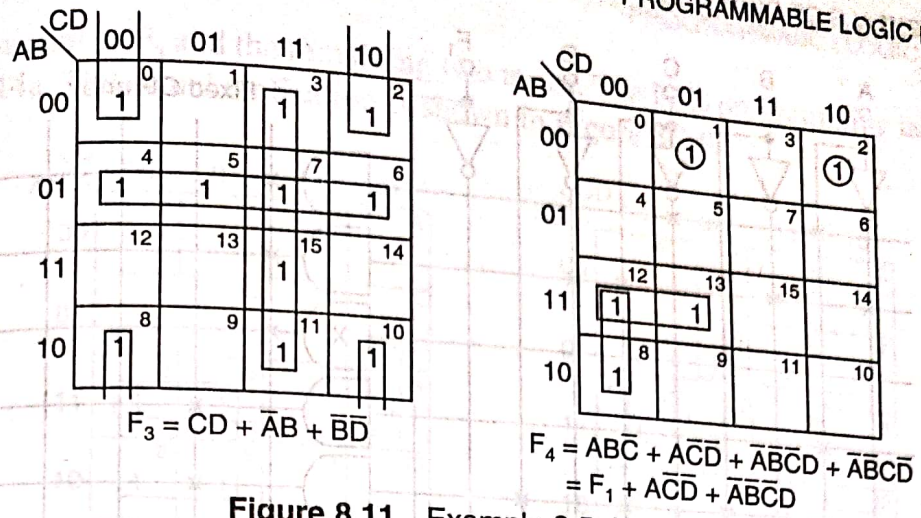


Figure 8.11 Example 8.5: K-maps.

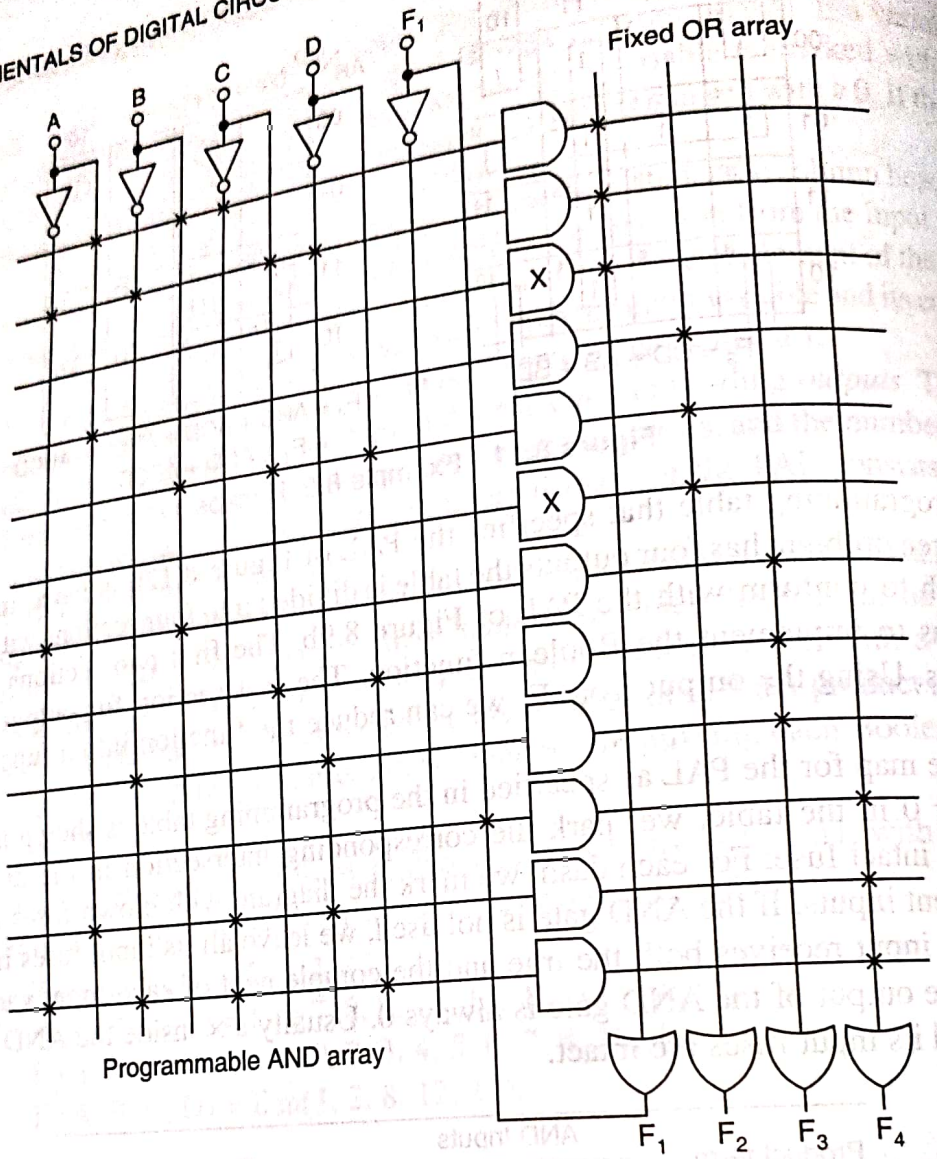
The programming table that specifies the PAL of Figure 8.12b is listed in Figure 8.12a. Since the given problem has four outputs the table is divided into four sections with three product terms in each to conform with the PAL of Figure 8.9b. The first two sections need only two product terms to implement the Boolean function. The last section for output F_4 needs four product terms. Using the output from F_1 we can reduce the function into a function with three terms.

The fuse map for the PAL as specified in the programming table is shown in Figure 8.12b. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash, we mark the diagram with blown fuses in both the true and complement inputs. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the true and the complement of each input variable, we have $A\bar{A} = 0$ and the output of the AND gate is always 0. Usually a × inside the AND gate is used to indicate that all its input fuses are intact.

Product term	AND Inputs A	B	C	D	F_1	Outputs
1	1	1	0	–	–	$F_1 = AB\bar{C} + \bar{A}B\bar{C}\bar{D}$
2	0	0	1	0	–	
3	–	–	–	–	–	
4	1	–	–	–	–	
5	–	1	1	1	–	$F_2 = A + BCD$
6	–	–	–	–	–	
7	0	1	–	–	–	$F_3 = \bar{A}B + CD + B\bar{D}$
8	–	–	1	1	–	
9	–	0	–	0	–	
10	–	–	–	–	1	$F_4 = F_1 + A\bar{C}\bar{D} + \bar{A}\bar{B}\bar{C}D$
11	1	–	0	0	–	
12	0	0	0	1	–	

(a) PAL programming table

Figure 8.12 Example 8.5 *(Contd.)*



(b) Realization of the example functions using PAL

Figure 8.12 Example 8.5.

EXAMPLE 8.6 Realize the following functions using a PAL with four inputs and 3-wide AND-OR structure. Also write the PAL programming table.

$F_1(A, B, C, D) = \Sigma\, m(6, 8, 9, 12\text{–}15)$ $\quad F_2(A, B, C, D) = \Sigma\, m(1, 4\text{–}7, 10\text{–}13)$

$F_3(A, B, C, D) = \Sigma\, m(4\text{–}7, 10\text{–}11)$ $\quad F_4(A, B, C, D) = \Sigma\, m(4\text{–}7, 9\text{–}15)$

Solution

The first step in the realization is to obtain the minimal sum of products form of all the given functions. The K-maps for the given functions, their minimization, and the minimal SOP expressions obtained from them are shown in Figure 8.13.

Each section in the PAL comprises three AND gates feeding the given OR gate. There are four such sections. Notice that F_2 has four product terms but the given PAL device has provision for three products only as inputs to OR gates. So some manipulation becomes necessary. Observe that out of four terms two of the terms of F_2 are equal to F_3 itself. So F_2 can be

written as the sum of 3 and the remaining two terms. The PAL programming table is shown in Figure 8.14a. The actual realization is shown in Figure 8.14b.

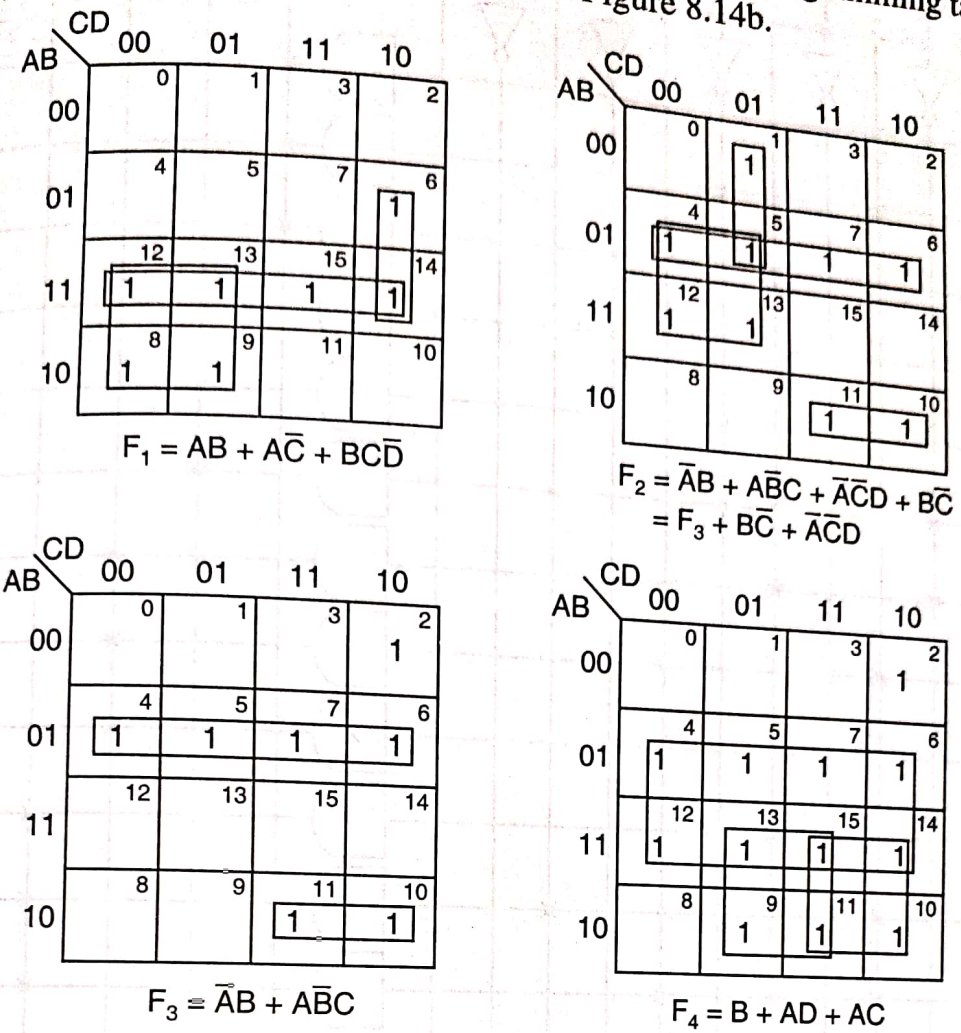
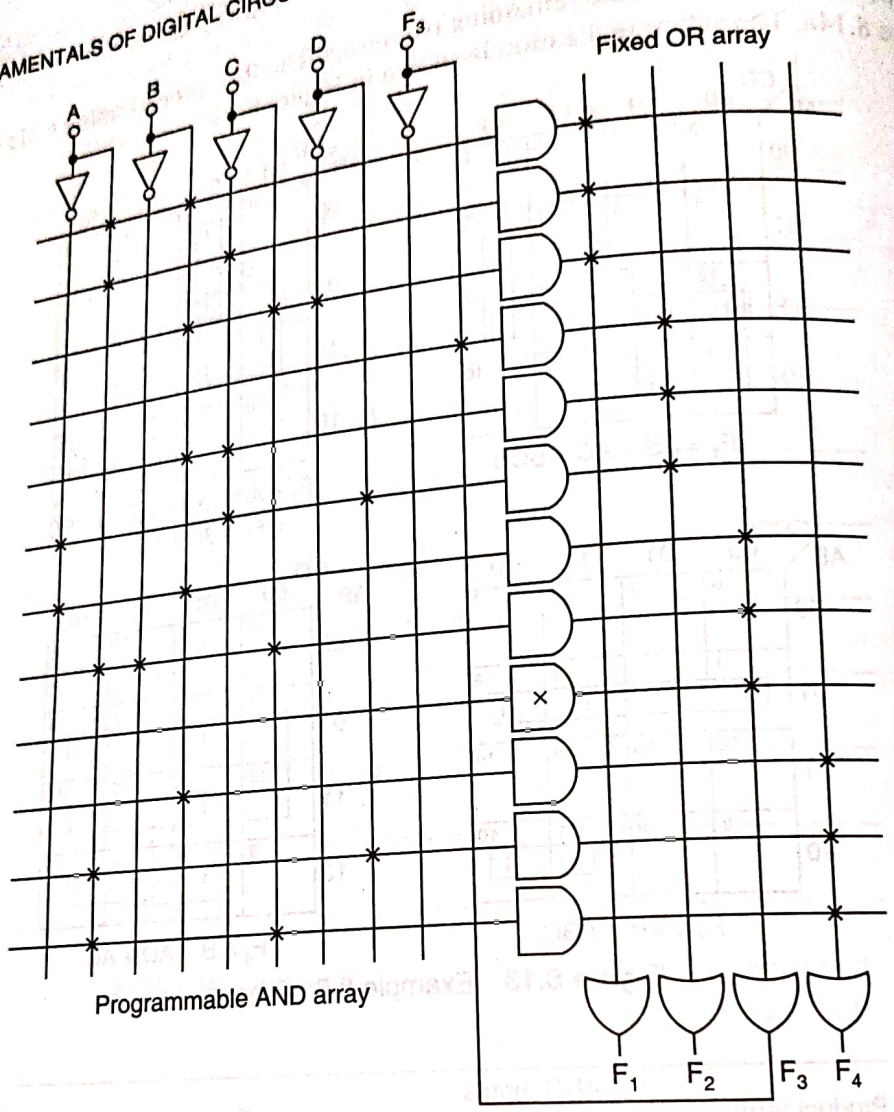


Figure 8.13 Example 8.6: K-maps.

Product term	AND Inputs A	B	C	D	F_3	Outputs
1	1	1	–	–	–	
2	1	–	0	–	–	$F_1 = AB + A\bar{C} + BC\bar{D}$
3	–	1	1	0	–	
4	–	–	–	–	1	$F_2 = \bar{A}B + A\bar{B}C + \bar{A}\bar{C}D + B\bar{C}$
5	0	–	0	1	–	$= F_3 + B\bar{C} + \bar{A}\bar{C}D$
6	–	1	0	–	–	
7	0	1	–	–	–	$F_3 = \bar{A}B + A\bar{B}C$
8	1	0	1	–	–	
9	–	–	–	–	–	
10	–	1	–	–	–	$F_4 = B + AD + AC$
11	1	–	–	1	–	
12	1	–	1	–	–	

(a) PAL programming table

Figure 8.14 Example 8.6 *(Contd.)*



(b) Realization of the example functions using PAL

Figure 8.14 Example 8.6.

8.8 PROGRAMMABLE LOGIC ARRAY (PLA)

The PLA represents another type of programmable logic but with a slightly different architecture. The PLA combines the characteristics of the PROM and the PAL by providing both a programmable OR array and a programmable AND array, i.e. in a PLA both AND gates and OR gates have fuses at the inputs. A third set of fuses in the output inverters allows the output function to be inverted if required. Usually X-OR gates are used for controlled inversion. This feature makes it the most versatile of the three PLDs. However, it has some disadvantages. Because it has two sets of fuses, it is more difficult to manufacture, program and test it than a PROM or a PAL. Figure 8.15 demonstrates the structure of a three-input, four-output PLA with every fusible link intact.

Like ROM, PLA can be mask programmable or field programmable. With a mask programmable PLA, the user must submit a PLA programming table to the manufacturer. This

table is used by the vender to produce a user made PLA that has the required internal paths between inputs and outputs. A second type of PLA available is called a field programmable logic array or FPLA. The FPLA can be programmed by the user by means of certain recommended procedures. FPLAs can be programmed with commercially available programmer units.

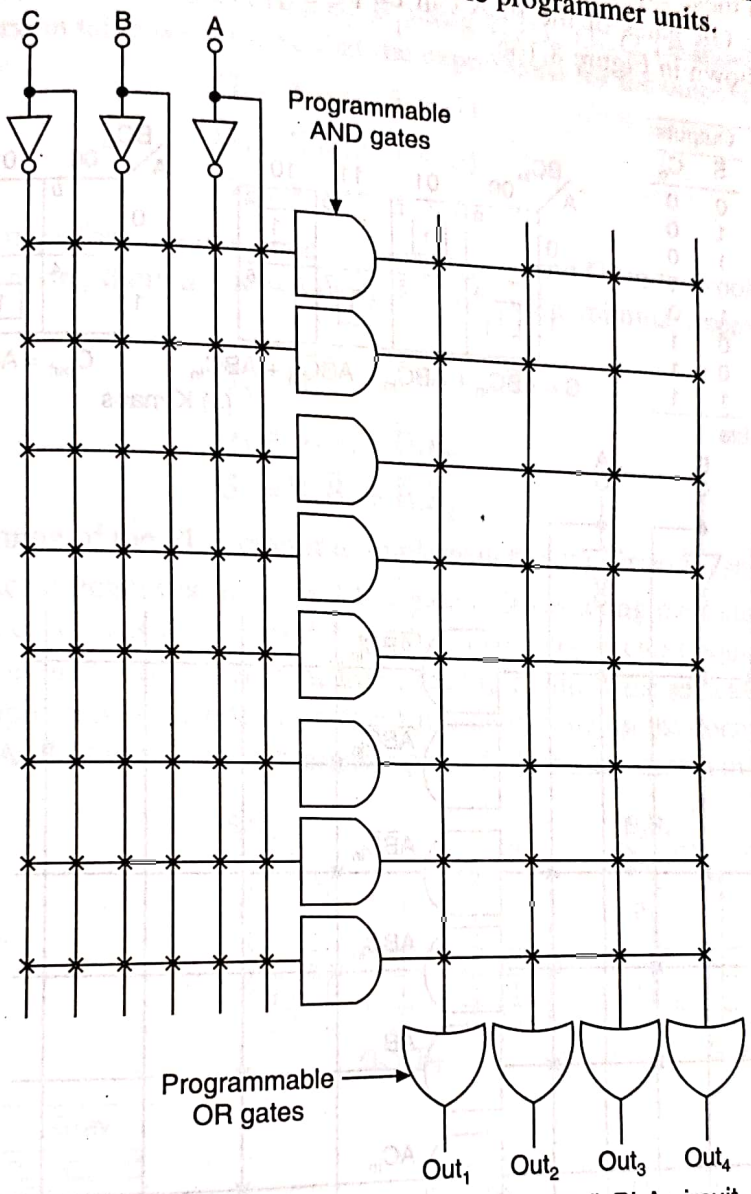


Figure 8.15 Structure of (an unprogrammed) PLA circuit.

EXAMPLE 8.7 Show how the PLA circuit in Figure 8.15 would be programmed to implement the sum and carry outputs of a full adder.

Solution

The truth table of a full-adder is shown in Figure 8.16a . Drawing the K-maps for the sum and carry-out terms and minimizing them, the minimal expressions for the sum and carry-out terms are:

The sum is

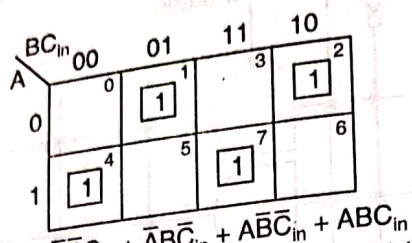
$$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$$

and the carry-out is $C_{out} = AB + AC_{in} + BC_{in}$

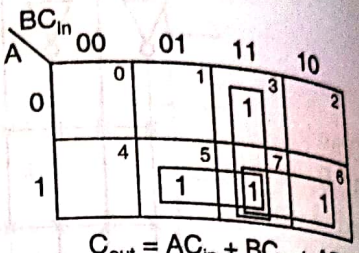
To implement these expressions, we need a 4-input OR gate and a 3-input OR gate. Since the inputs to the OR gates of the PLA can be programmed, we can implement the given expressions as shown in Figure 8.16c.

Inputs			Outputs	
A	B	C_{in}	S	C_o
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

(a) Truth table

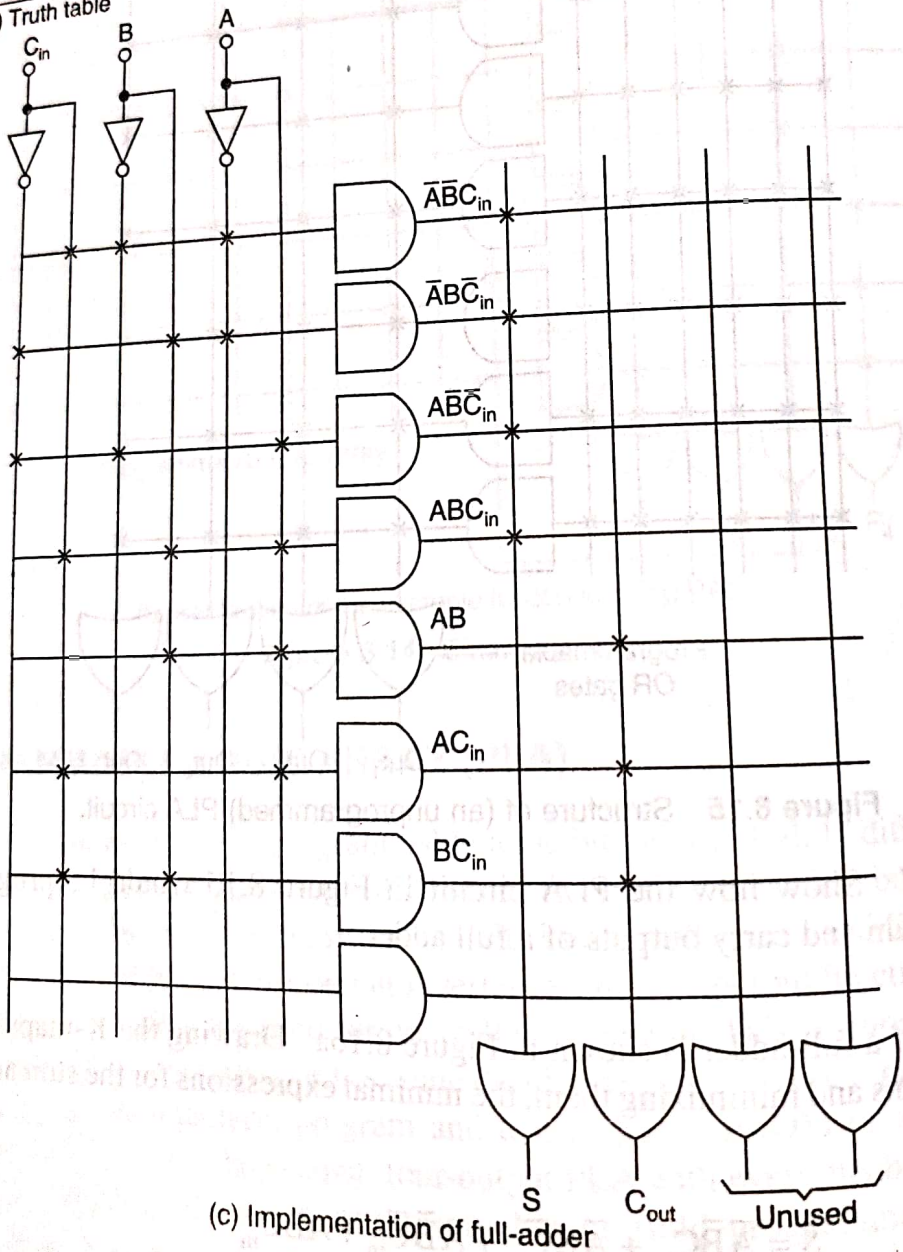


$S = \bar{A}\bar{B}C_{in} + \bar{A}B\bar{C}_{in} + A\bar{B}\bar{C}_{in} + ABC_{in}$



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

(b) K-maps



(c) Implementation of full-adder

Figure 8.16 Example 8.7.

EXAMPLE 8.8 Show how the PLA circuit in Figure 8.15 can be programmed to implement the 3-bit binary-to-gray conversion.

Solution

The conversion table of 3-bit binary (B_3, B_2, B_1)-to-gray (G_3, G_2, G_1) is shown in Figure 8.17a. From the conversion table we observe that the expressions for the outputs are:

$$G_3 = \Sigma\, m(4, 5, 6, 7)$$
$$G_2 = \Sigma\, m(2, 3, 4, 5)$$
$$G_1 = \Sigma\, m(1, 2, 5, 6)$$

Drawing the K-maps for the Gray code outputs G_3, G_2, and G_1 in terms of binary inputs B_3, B_2, B_1 and minimizing them as shown in Figure 8.17b, the minimal expressions for G_3, G_2, and G_1 are:

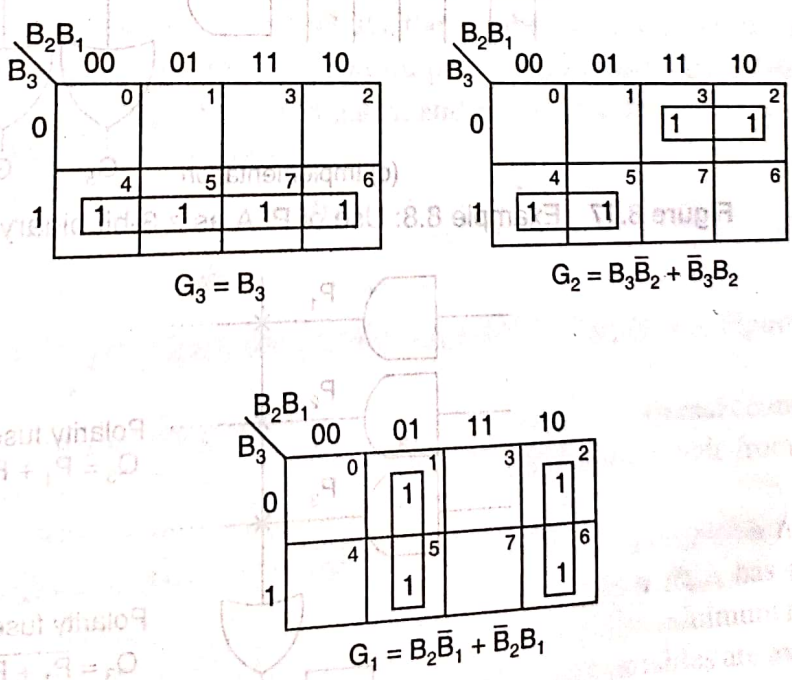
$$G_3 = B_3$$
$$G_2 = B_3\bar{B}_2 + \bar{B}_3B_2$$
$$G_1 = B_2\bar{B}_1 + \bar{B}_2B_1$$

The programming of the PLA circuit to implement the conversion is shown in Figure 8.17c.

The product term generated in each AND gate is listed along the output of the gate in the diagram. The product term is determined from the inputs whose cross points are connected and marked with a ×. The output of an OR gate gives the logic sum of the selected product terms. The output may be complemented or left in its true form depending on the connection for one of the X-OR gate inputs. A PLD with a programmable polarity feature is shown in Figure 8.18.

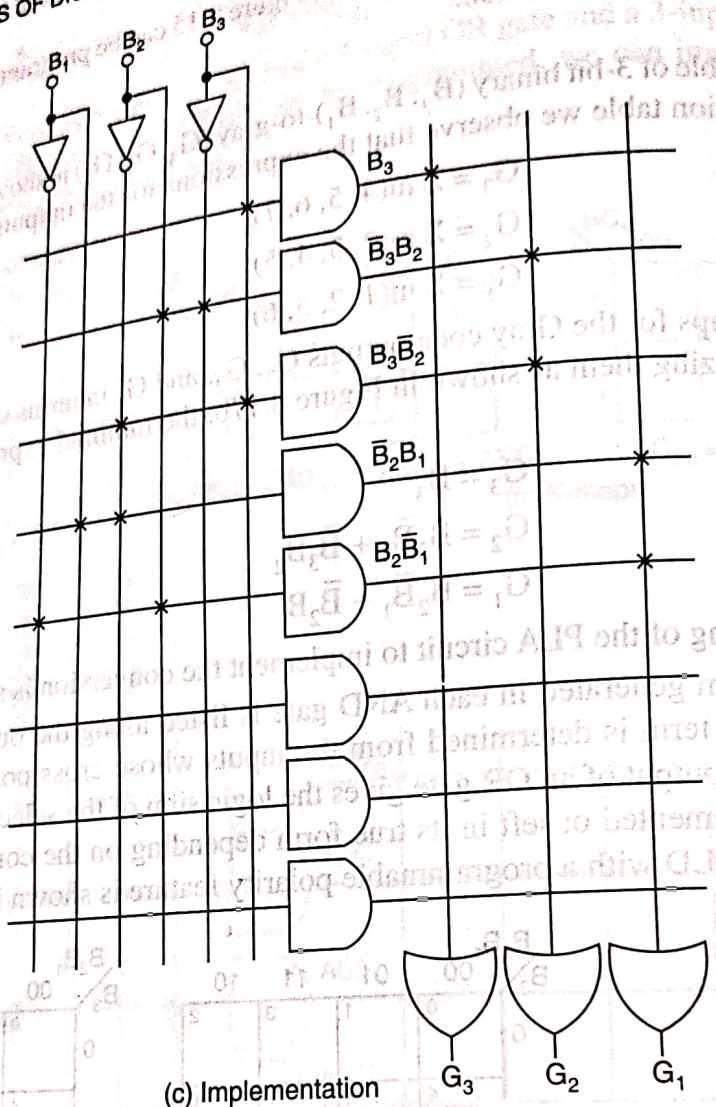
Binary			Gray		
B_3	B_2	B_1	G_3	G_2	G_1
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

(a) Conversion table



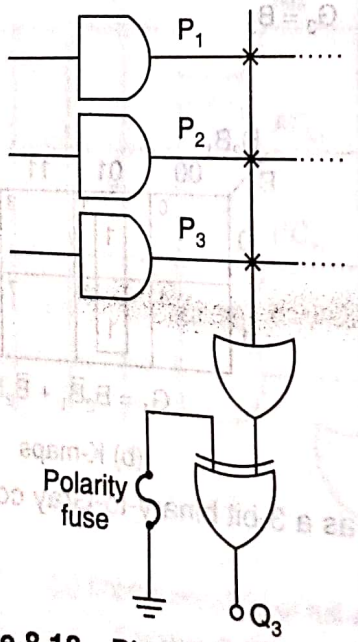
(b) K-maps

Figure 8.17 Example 8.8: Use of PLA as a 3-bit binary-to-Gray code converter *(Contd.)*



(c) Implementation

Figure 8.17 Example 8.8: Use of PLA as a 3-bit binary-to-Gray code converter.



Polarity fuse intact

$Q_3 = P_1 + P_2 + P_3$

Polarity fuse blown

$Q_3 = \overline{\bar{P}_1 + P_2 + P_3}$

$= \bar{P}_1 \cdot \bar{P}_2 \cdot \bar{P}_3$

Figure 8.18 PLD with a programmable polarity feature.

8.8.1 PLA Programming Table

The fuse map of a PLA can be specified in a tabular form. The PLA programming table consists of three columns. The first column lists the product term numerically. The second column specifies the required paths between inputs and AND gates. The third column specifies the paths between the AND and OR gates. For each output variable, we may have a T (for true) or C (for complement) for programming the X-OR gate. The product terms listed on the left are not part of the table; they are included for reference only. For each product term the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its true form, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked as a – (dash).

The paths between the inputs and the AND gates are specified under the column heading *inputs* in the programming table. A 1 in the input column specifies a connection from the input variable to the AND gate. A 0 in the input column specifies a connection from the complement of the variable to the input of the AND gate. A – (dash) specifies a blown fuse in both the input variable and its complement. It is assumed that an open terminal in the input of an AND gate behaves like a 1.

The paths between the AND and OR gates are specified under the column heading *outputs*. The output variables are marked with 1s for those product terms that are included in the function. Each product term that has a 1 in the output column requires a path from the output of the AND gate to the input of the OR gate. Those marked with a – (dash) specify a blown fuse. It is assumed that an open terminal in the input of an OR gate behaves like a 0. Finally, a T (true) output dictates that the other input of the corresponding X-OR gate be connected to 0, and a C (complement) specifies a connection to 1.

The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. For n inputs, k product terms, and m outputs the internal logic of the PLA consists of n buffer inverter gates, k AND gates, m OR gates, and m X-OR gates.

The implementation of the Boolean functions

$$F_1 = \bar{A}B + A\bar{C} + \bar{A}B\bar{C}$$

$$F_2 = \overline{A\bar{C} + \bar{B}C}$$

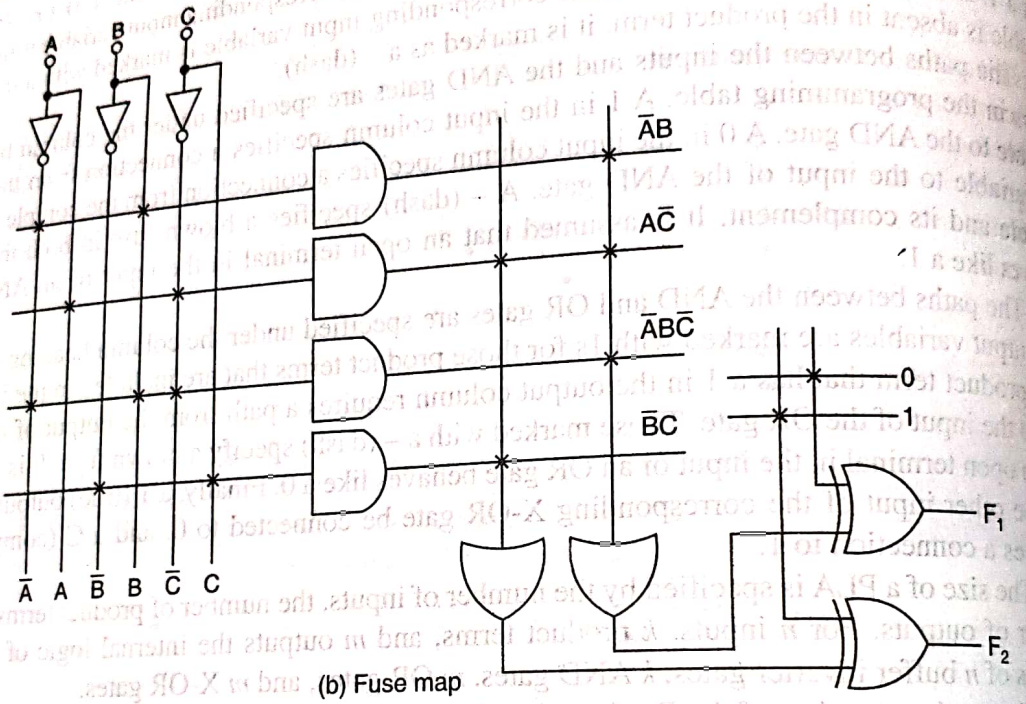
using PLA is shown in Figure 8.19b. The programming table that specifies the PLA of Figure 8.19b is listed in Figure 8.19a.

When designing a digital system with a PLA, there is no need to show the internal connections of the unit as was done in Figure 8.19b. All that is needed is a PLA programming table from which the PLA can be programmed to supply the required logic.

When implementing a combinational circuit with a PLA, careful investigation must be undertaken in order to reduce the number of distinct product terms. Since a PLA has a finite number of AND gates this can be done by simplifying each Boolean function to a minimum number of terms. The number of literals in a term is not important since all the input variables are available any way. Both the true and the complement of each function should be simplified to see which one can be expressed with a fewer product terms and which one provides product terms that are common to other functions.

Product term		Inputs A	B	C	Outputs (T) F	(C) F
1	$\bar{A}B$	0	1	–	1	–
2	$A\bar{C}$	1	–	0	1	1
3	$\bar{A}B\bar{C}$	0	1	0	1	–
4	$\bar{B}C$	–	0	1	–	1

(a) PLA programming table



(b) Fuse map

Figure 8.19 Programmable logic array.

EXAMPLE 8.9 Implement the following two Boolean functions with a PLA:

$$F_1(A, B, C) = \Sigma\, m(0, 1, 2, 4)$$
$$F_2(A, B, C) = \Sigma\, m(0, 5, 6, 7)$$

Solution

The K-maps for the functions F_1 and F_2, their minimization, and the minimal expressions for both the true and complement forms of those in sum of products are shown in Figure 8.20. For finding the minimal in true form, consider the 1s on the map and for finding the minimal in complement form consider the 0s on the map.

Considering the 1s of F_1

$$F_1(T) = \bar{A}\bar{C} + \bar{B}\bar{C} + \bar{A}\bar{B}$$

Considering the 0s of F_1

$$\bar{F}_1 = AB + AC + BC$$

Therefore,

$$F_1(C) = \overline{(AB + AC + BC)}$$

Considering the 1s of F_2

$$F_2(T) = \bar{A}\bar{B}\bar{C} + AB + AC$$

Considering the 0s of F_2

$$\bar{F}_2 = A\bar{B}\bar{C} + \bar{A}B + \bar{A}C$$

Therefore,

$$F_2(C) = \overline{A\bar{B}\bar{C} + \bar{A}B + \bar{A}C}$$

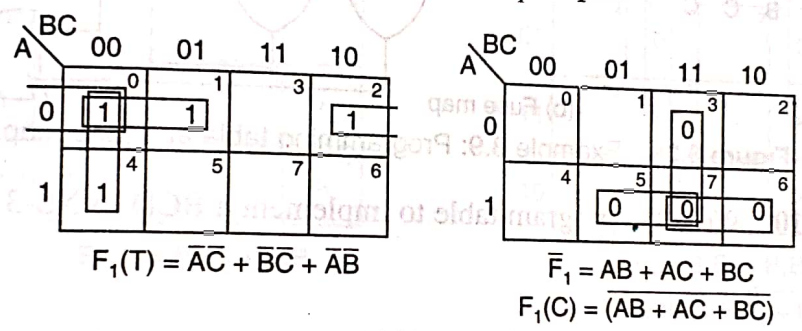
Out of $F_1(T)$, $F_1(C)$, $F_2(T)$, $F_2(C)$, the combination that gives the minimum number of product terms is

$$F_1(C) = \overline{(AB + AC + BC)}$$

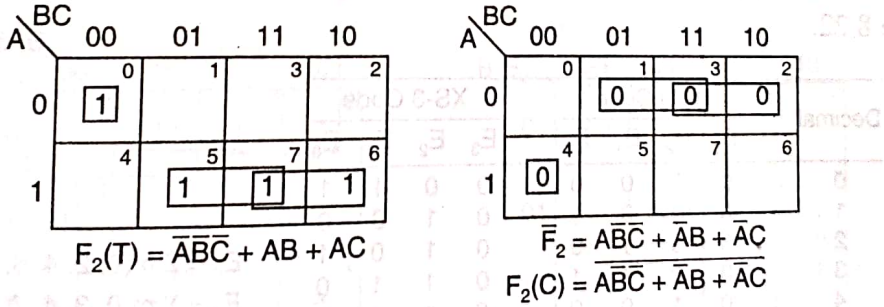
$$F_2(T) = AB + AC + \bar{A}\bar{B}\bar{C}$$

This gives four distinct terms: AB, AC, and BC and $\bar{A}\bar{B}\bar{C}$. The PLA programming table for this combination is shown in Figure 8.21a. The implementation using a PLA is shown in Figure 8.21b.

F_1 is the true output even though a C is marked over it in the table. This is because F_1 is generated with an AND-OR circuit and is available at the output of the OR gate. The X-OR gate complements the function to produce the true F_1 output.



(a) K-map for F_1



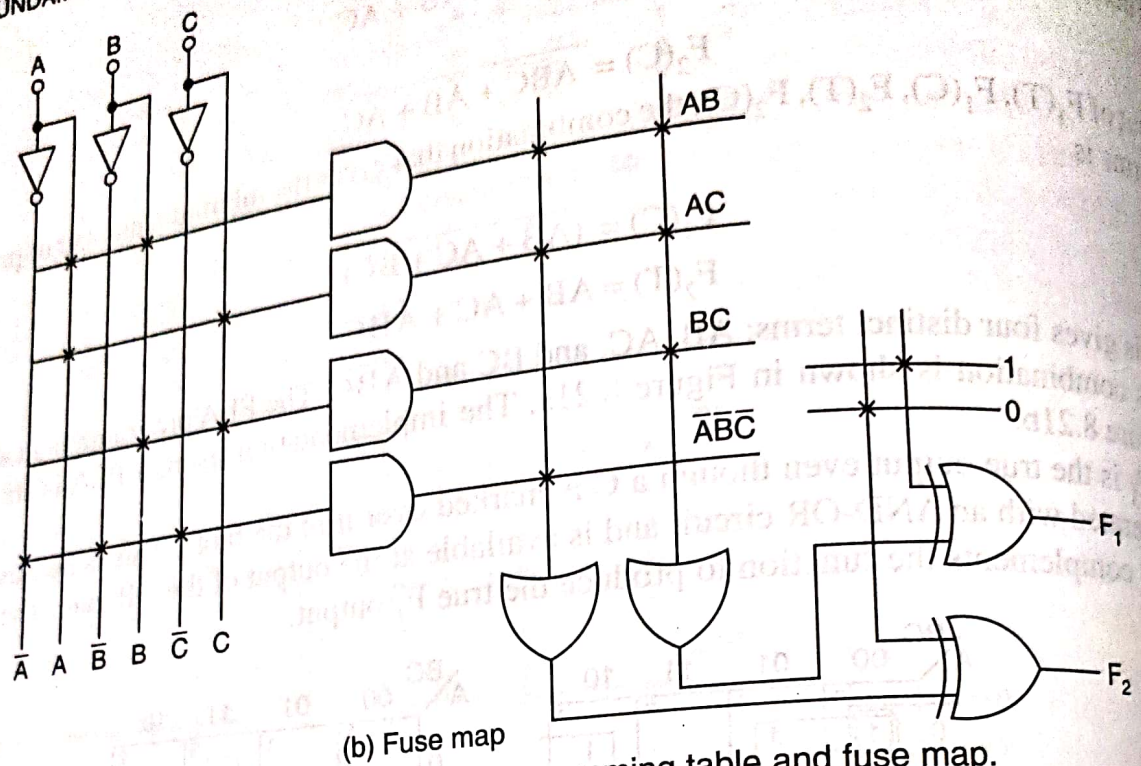
(b) K-map for F_2

Figure 8.20 Example 8.9: K-maps.

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
1	AB	1	1	–	1	1
2	AC	1	–	1	1	–
3	BC	–	1	1	–	1
4	$\bar{A}\bar{B}\bar{C}$	0	0	0		

(a) PLA programming table

Figure 8.21 Example 8.9: Programming table and fuse map *(Contd.)*



(b) Fuse map

Figure 8.21 Example 8.9: Programming table and fuse map.

EXAMPLE 8.10 Write the program table to implement a BCD to XS-3 code conversion using a PLA.

Solution

The BCD to XS-3 code conversion table and the expressions for the XS-3 outputs are shown in Figure 8.22.

Decimal	BCD Code				XS-3 Code			
	B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

(a) BCD to XS-3 conversion table

$E_0 = \Sigma\, m(0, 2, 4, 6, 8)$
$E_1 = \Sigma\, m(0, 3, 4, 7, 8)$
$E_2 = \Sigma\, m(1, 2, 3, 4, 9)$
$E_3 = \Sigma\, m(5, 6, 7, 8, 9)$

The don't cares are
$d = \Sigma\, d(10, 11, 12, 13, 14, 15)$

(b) Expressions for outputs

Figure 8.22 Example 8.10: Conversion table and expressions for outputs.

To write the PLA program table, obtain the true and complement form of the minimal expressions for outputs using K-maps as shown in Figure 8.23. For true form consider the 1s on the K-maps, and for complement form consider the 0s on the K-maps and obtain the expressions in SOP form in both cases.