UNIT II

COMBINATIONAL CIRCUITS

2. Combinational Circuits

A *combinational circuit* is one where the output at any time depends only on the present combination of inputs at that point of time with total disregard to the past state of the inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions. The other category of logic circuits, called *sequential logic circuits*, comprises both logic gates and memory elements such as flip-flops. Owing to the presence of memory elements, the output in a sequential circuit depends upon not only the present but also the past state of inputs.

Figure shows the block schematic representation of a generalized combinational circuit having n input variables and m output variables or simply outputs. Since the number of input variables is

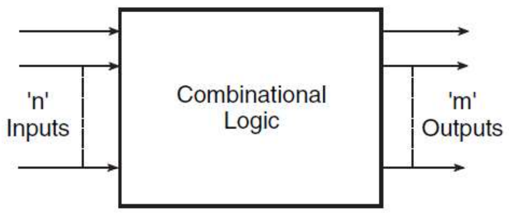


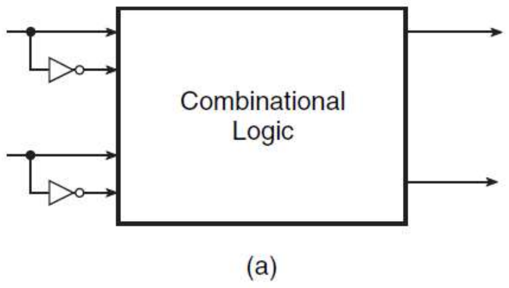
Figure Generalized combinational circuit.

n, there are 2^n possible combinations of bits at the input. Each output can be expressed in terms of input variables by a Boolean expression, with the result that the generalized system of Fig.can be expressed by m Boolean expressions. As an illustration, Boolean expressions describing the function of a four-input OR/NOR gate are given as

$$Y_1 \text{ (OR output)} = A+B+C+D \quad \text{and} \quad Y_2 \text{ (NOR output)} = \overline{A+B+C+D}$$

Also, each of the input variables may be available as only the normal input on the input line designated for the purpose. In that case, the complemented input, if desired, can be generated by using an inverter, as shown in Fig.(a), which illustrates the case of a four-input, two-output combinational function. Also, each of the input variables may appear in two wires, one representing the normal literal and the other representing the complemented one, as shown in Fig.(b).

In combinational circuits, input variables come from an external source and output variables feed an external destination. Both source and destination in the majority of cases are storage registers, and these



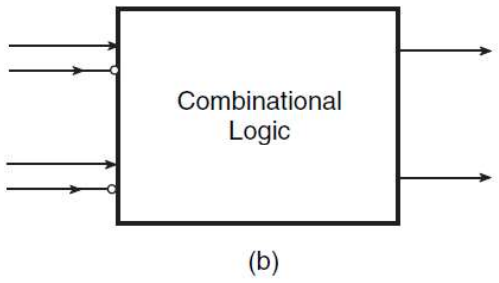


Figure Combinational circuit with normal and complemented inputs.

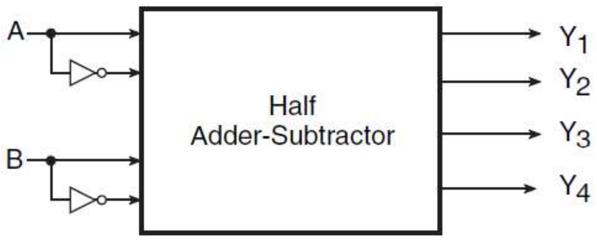


Figure Two-input, four-output combinational circuit.

storage devices provide both normal as well as complemented outputs of the stored binary variable. As an illustration, Fig. shows a simple two-input (A, B), four-output (Y_1, Y_2, Y_3, Y_4) combinational logic circuit described by the following Boolean expressions

$$Y_1 = A.\overline{B} + \overline{A}.B$$

$$Y_2 = A.\overline{B} + \overline{A}.B$$

$$Y_3 = A.B$$

$$Y_4 = \overline{A}.B$$

The implementation of these Boolean expressions needs both normal as well as complemented inputs. Incidentally, the combinational circuit shown is that of a half-adder–subtractor, with A and B representing the two bits to be added or subtracted and Y_1,Y_2,Y_3,Y_4 representing SUM, DIFFERENCE, CARRY and BORROW outputs respectively.

2.1 Implementing Combinational Logic

The different steps involved in the design of a combinational logic circuit are as follows:

1. Statement of the problem.

2. Identification of input and output variables.

3. Expressing the relationship between the input and output variables.

4. Construction of a truth table to meet input–output requirements.

5. Writing Boolean expressions for various output variables in terms of input variables.

6. Minimization of Boolean expressions.

7. Implementation of minimized Boolean expressions.

These different steps are self-explanatory. One or two points, however, are worth mentioning here. There are various simplification techniques available for minimizing Boolean expressions. These include the use of theorems and identities, Karnaugh mapping, the Quinne–McCluskey tabulation method and so on. Also, there are various possible minimized forms of Boolean expressions. The following guidelines should be followed while choosing the preferred form for hardware implementation:

1. The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.

2. There should be a minimum number of interconnections, and the propagation time should be the shortest.

3. Limitation on the driving capability of the gates should not be ignored.

It is difficult to generalize as to what constitutes an acceptable simplified Boolean expression. The importance of each of the above-mentioned aspects is governed by the nature of application.

2.2 Arithmetic Circuits – Basic Building Blocks

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction. We will begin with the basic building blocks that form the basis of all hardware used to perform the aforesaid arithmetic operations on binary numbers. These include half-adder, full adder, half-subtractor, full subtractor and controlled inverter.

2.3 Half-Adder

A *half-adder* is an arithmetic circuit block that can be used to add two bits. Such a circuit thus has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY. Figure shows the truth table of a half-adder, showing all possible input combinations and the corresponding outputs.

The Boolean expressions for the SUM and CARRY outputs are given by the equations

$$\text{SUM } S = A.\overline{B} + \overline{A}.B$$
$$\text{CARRY } C = A.B$$

An examination of the two expressions tells that there is no scope for further simplification. While the first one representing the SUM output is that of an EX-OR gate, the second one representing the

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

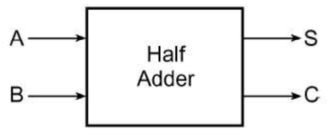


Figure Truth table of a half-adder.

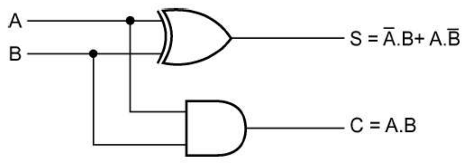


Figure Logic implementation of a half-adder.

CARRY output is that of an AND gate. However, these two expressions can certainly be represented in different forms using various laws and theorems of Boolean algebra to illustrate the flexibility that the designer has in hardware-implementing as simple a combinational function as that of a half-adder. Although the simplest way to hardware-implement a half-adder would be to use a two-input EX-OR gate for the SUM output and a two-input AND gate for the CARRY output, as shown in Fig., it could also be implemented by using an appropriate arrangement of either NAND or NOR gates. Figure shows the implementation of a half-adder with NAND gates only.

A close look at the logic diagram of Fig. reveals that one part of the circuit implements a two-input EX-OR gate with two-input NAND gates. The AND gate required to generate CARRY output is implemented by complementing an already available NAND output of the input variables.

2.4 Full Adder

A *full adder* circuit is an arithmetic circuit block that can be used to add three bits to produce a SUM and a CARRY output. Such a building block becomes a necessity when it comes to adding binary numbers with a large number of bits. The full adder circuit overcomes the limitation of the

half-adder, which can be used to add two bits only. Let us recall the procedure for adding larger binary numbers. We begin with the addition of LSBs of the two numbers. We record the sum under the LSB column and take the carry, if any, forward to the next higher column bits. As a result, when we add the next adjacent higher column bits, we would be required to add three bits if there were a carry from the previous addition. We have a similar situation for the other higher column bits

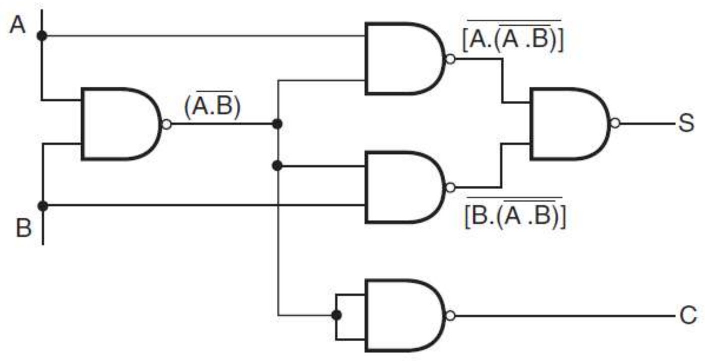
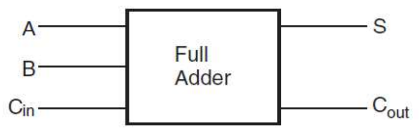

A
B
$\overline{(A.B)}$
$[\overline{A.\overline{(A.B)}}]$
$[\overline{B.\overline{(A.B)}}]$
S
C

Figure Half-adder implementation using NAND gates.



A	B	C_{in}	SUM (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

Figure Truth table of a full adder.

also until we reach the MSB. A full adder is therefore essential for the hardware implementation of an adder circuit capable of adding larger binary numbers. A half-adder can be used for addition of LSBs only.

Figure shows the truth table of a full adder circuit showing all possible input combinations and corresponding outputs. In order to arrive at the logic circuit for hardware implementation of a full adder, we will firstly write the Boolean expressions for the two output variables, that is, the SUM and CARRY outputs, in terms of input variables.

$$S = \overline{A}.\overline{B}.C_{\text{in}} + \overline{A}.B.\overline{C}_{\text{in}} + A.\overline{B}.\overline{C}_{\text{in}} + A.B.C_{\text{in}}$$
$$C_{\text{out}} = \overline{A}.B.C_{\text{in}} + A.\overline{B}.C_{\text{in}} + A.B.\overline{C}_{\text{in}} + A.B.C_{\text{in}}$$

The next step is to simplify the two expressions. We will do so with the help of the Karnaugh mapping technique. Karnaugh maps for the two expressions are given in Fig. As is clear from the two maps, the expression for the SUM (S) output cannot be simplified any further, whereas the simplified Boolean expression for C_{out} is given out by the equation

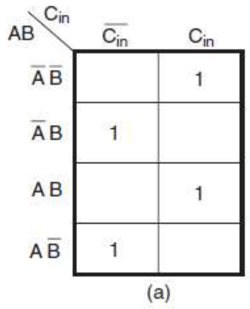
$$C_{\text{out}} = B.C_{\text{in}} + A.B + A.C_{\text{in}}$$

Figure shows the logic circuit diagram of the full adder. A full adder can also be seen to comprise two half-adders and an OR gate. The expressions for SUM and CARRY outputs can be rewritten as follows:

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

Similarly, the expression for CARRY output can be rewritten as follows:

$$\begin{aligned}C_{out} &= B.C_{in}.(A+\overline{A})+A.B+A.C_{in}.(B+\overline{B})\\ &= A.B+A.B.C_{in}+\overline{A}.B.C_{in}+A.B.C_{in}+A.\overline{B}.C_{in} = A.B+A.B.C_{in}+\overline{A}.B.C_{in}+A.\overline{B}.C_{in}\\ &= A.B.(1+C_{in})+C_{in}.(\overline{A}.B+A.\overline{B})\end{aligned}$$



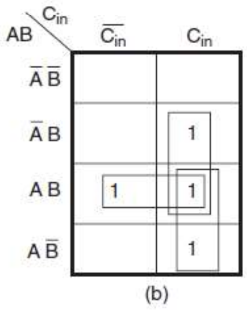


Figure Karnaugh maps for the sum and carry-out of a full adder.

$$C_{out} = A.B+C_{in}.(\overline{A}.B+A.\overline{B})$$

Boolean expression can be implemented with a two-input EX-OR gate provided that one of the inputs is C and the other input is the output of another two-input EX-OR gate with A and Bin as its inputs. Similarly, Boolean expression can be implemented by ORing two minterms. One of them is the AND output of A and B. The other is also the output of an AND gate whose inputs are Cand the output of an EX-OR operation on A and B. The whole idea of writing the Boolean in expressions in this modified form was to demonstrate the use of a half-adder circuit in building a full adder.

The full adder of the type described above forms the basic building block of binary adders. However, a single full adder circuit can be used to add one-bit binary numbers only. A cascade arrangement of these adders can be used to construct adders capable of adding binary numbers with a larger number of bits. For example, a four-bit binary adder would require four full adders of the type shown in Fig. to be connected in cascade. Figure shows such an arrangement. (A_3, A2, A1, A0) and (B_3, B2, B1, B0) are the two binary numbers to be added, with A_0 and B_0 representing LSBs and A_3 and B_3 representing MSBs of the two numbers.

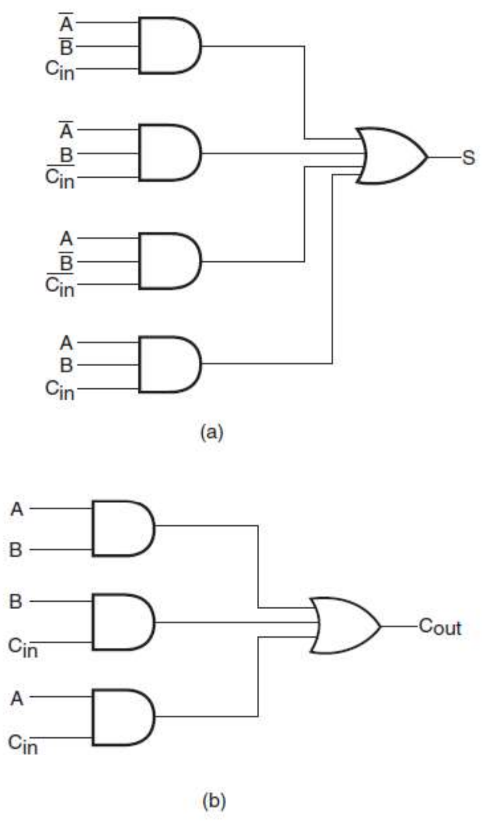
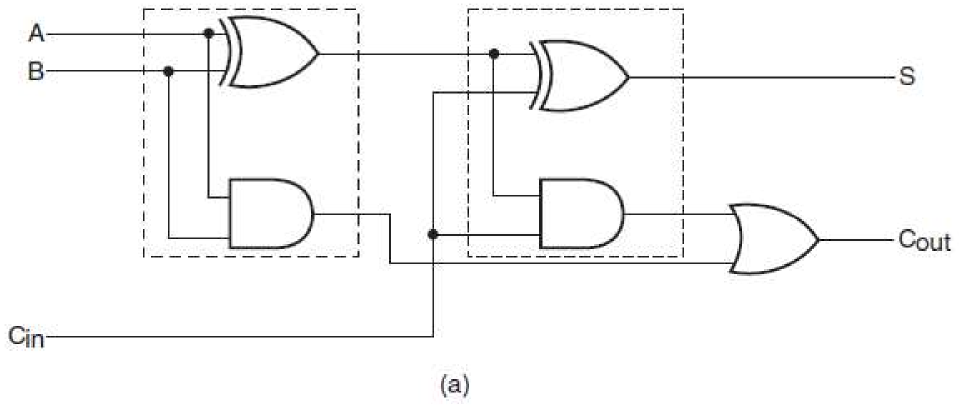


Figure Logic circuit diagram of a full adder.

2.5 Half-Subtractor

A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a '1' has been borrowed to perform the subtraction. The truth table of a half-subtractor, as shown in Fig., explains this further. The Boolean expressions for the two outputs are given by the equations



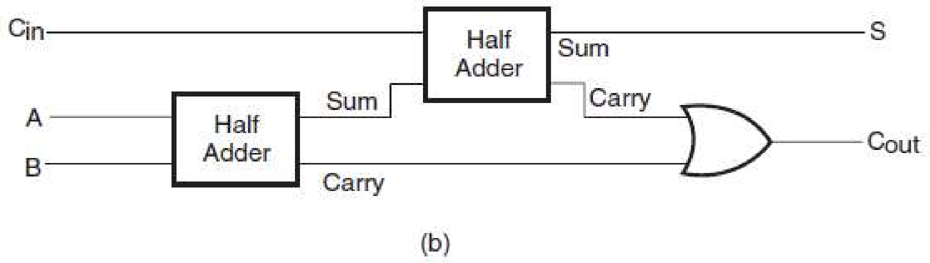


Figure Logic implementation of a full adder with half-adders.

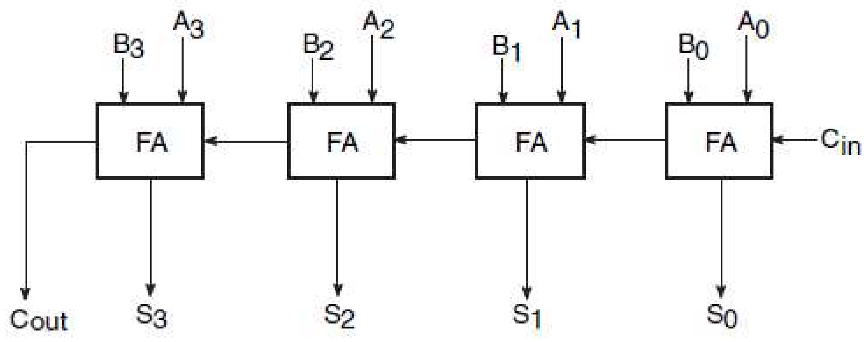
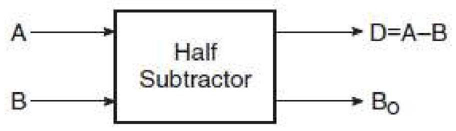


Figure Four-bit binary adder.

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

$$B_o = \overline{A}.B$$

It is obvious that there is no further scope for any simplification of the Boolean expressions given by above Equations. While the expression for the DIFFERENCE (D) output is that of



A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Figure Half-subtractor.

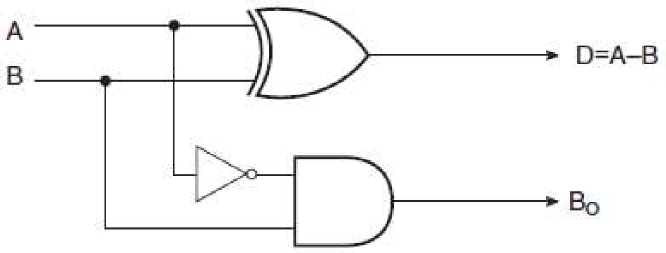


Figure Logic diagram of a half-subtractor.

an EX-OR gate, the expression for the BORROW output (B_0) is that of an AND gate with input A complemented before it is fed to the gate. Figure shows the logic implementation of a half-subtractor. Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, that is, the minuend, is complemented, an AND gate can be used to implement the BORROW output.

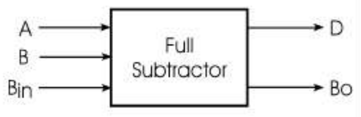
2.6 Full Subtractor

A *full subtractor* performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a '1' has already been borrowed by the previous adjacent lower minuend bit or not. As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_0. The BORROW output bit tells whether the minuend bit needs to borrow a '1' from the next possible higher minuend bit.

The Boolean expressions for the two output variables are given by the equations

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

$$B_o = \overline{A}.\overline{B}.B_{in} + \overline{A}.B.\overline{B}_{in} + \overline{A}.B.B_{in} + A.B.B_{in}$$



Minuend (A)	Subtrahend (B)	Borrow In (B_{in})	Difference (D)	Borrow Out (B_o)
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

Figure Truth table of a full subtractor.

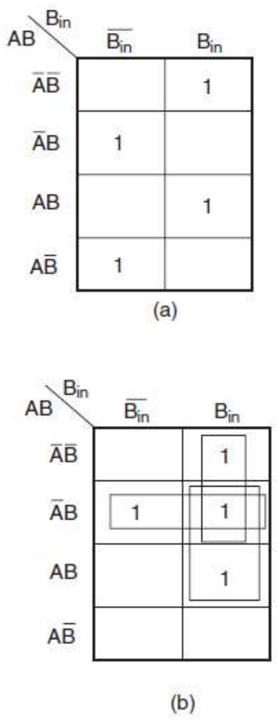
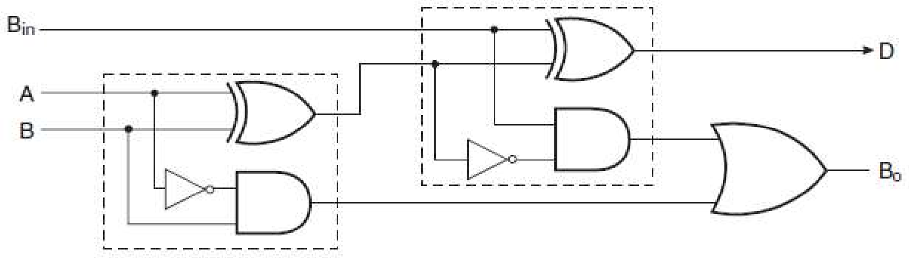


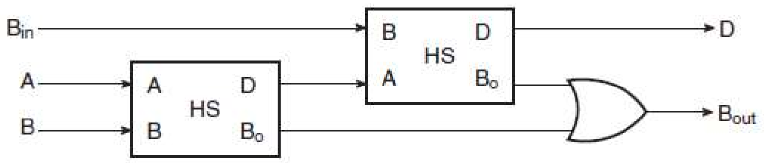
Figure Karnaugh maps for difference and borrow outputs.

The Karnaugh maps for the two expressions are given in Fig. As is clear from the two Karnaugh maps, no simplificationis possible for the difference output D. The simplified expression for B_0 is given by the equation

$$B_0 = \overline{A}.B + \overline{A}.B_{in} + B.B_{in}$$



(a)



(b)

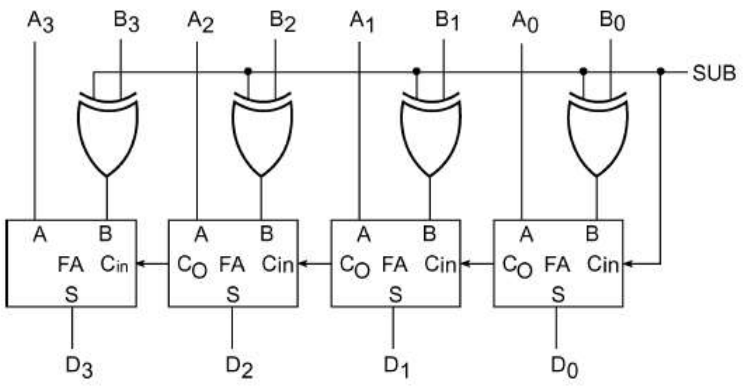
Figure Logic implementation of a full subtractor with half-subtractors.

If we compare these expressions with those derived earlier in the case of a full adder, we find that the expression for DIFFERENCE output D is the same as that for the SUM output. Also, the expression for BORROW output B_0 is similar to the expression for CARRY-OUT C_0. In the case of a half-subtractor, the A input is complemented. By a similar analysis it can be shown that a full subtractor can be implemented with half-subtractors in the same way as a full adder was constructed using half-adders.

Again, more than one full subtractor can be connected in cascade to perform subtraction on two larger binary numbers.

2.7 Parallel Adder–Subtractor

Subtraction of two binary numbers can be accomplished by adding 2's complement of the subtrahend to the minuend and disregarding the final carry, if any. If the MSB bit in the result of addition is



a ‘0’, then the result of addition is the correct answer. If the MSB bit is a ‘1’, this implies that the answer has a negative sign. The true magnitude in this case is given by 2’s complement of the result of addition.

Full adders can be used to perform subtraction provided we have the necessary additional hardware to generate 2’s complement of the subtrahend and disregard the final carry or overflow. Figure shows one such hardware arrangement. Let us see how it can be used to perform subtraction of two four-bit binary numbers. A close look at the diagram would reveal that it is the hardware arrangement for a four-bit binary adder, with the exception that the bits of one of the binary numbers are fed through controlled inverters. The control input here is referred to as the SUB input. When the SUB input is in logic ‘0’ state, the four bits of the binary number (B_3 B_2 B_1 B_0) are passed on as such to the B inputs of the corresponding full adders. The outputs of the full adders in this case give the result of addition of the two numbers. When the SUB input is in logic ‘1’ state, four bits of one of the numbers, (B_3 B_2 B_1 B_0) in the present case, get complemented. If the same ‘1’ is also fed to the CARRY-IN of the LSB full adder, what we finally achieve is the addition of 2’s complement and not 1’s complement. Thus, in the adder arrangement of Fig., we are basically adding 2’s complement of (B_3 B_2 B_1 B_0) to (A_3 A_2 A_1 A_0). The outputs of the full adders in this case give the result of subtraction of the two numbers. The arrangement shown achieves $A - B$. The final carry (the CARRY-OUT of the MSB full adder) is ignored if it is not displayed.

For implementing an eight-bit adder–subtractor, we will require eight full adders and eight two-input EX-OR gates. Four-bit full adders and quad two-input EX-OR gates are individually available in integrated circuit form. A commonly used four-bit adder in the TTL family is the type number 7483. Also, type number 7486 is a quad two-input EX-OR gate in the TTL family.

2.8 Carry Propagation–Look-Ahead Carry Generator

The four-bit binary adder described in the previous pages can be used to add two four-bit binary numbers. Multiple numbers of such adders are used to perform addition operations on larger-bit binary numbers. Each of the adders is composed of four full adders (FAs) connected in cascade. The block schematic arrangement of a four-bit adder is reproduced in Fig. for reference and further discussion. This type of adder is also called a parallel binary adder because all the bits of the augend and addend are present and are fed to the full adder blocks simultaneously. Theoretically, the addition operation in various full adders takes place simultaneously. What is of importance and interest to users, more so when they are using a large number of such adders in their overall computation system, is whether the result of addition and carry-out are available to them at the same time. In other words, we need to see if this addition operation is truly parallel in nature. We will soon see that it is not. It is in fact limited by what is known as *carry propagation time*. Here, C_i and $C_{i+1}$are the input and output CARRY; P_i andG_i are two new binary variables called CARRY PROPAGATE and CARRY GENERATE and will beaddressed a little later.

For i=1, the diagram in Fig. (b) is that of the LSB full adder of Fig.(a). We can see here that C_2, which is the output CARRY of FA (1) and the input CARRY for FA (2), will appear at the output aftera minimum of two gate delays plus delay due to the half adder after application of A_i, B_i and C_i inputs.

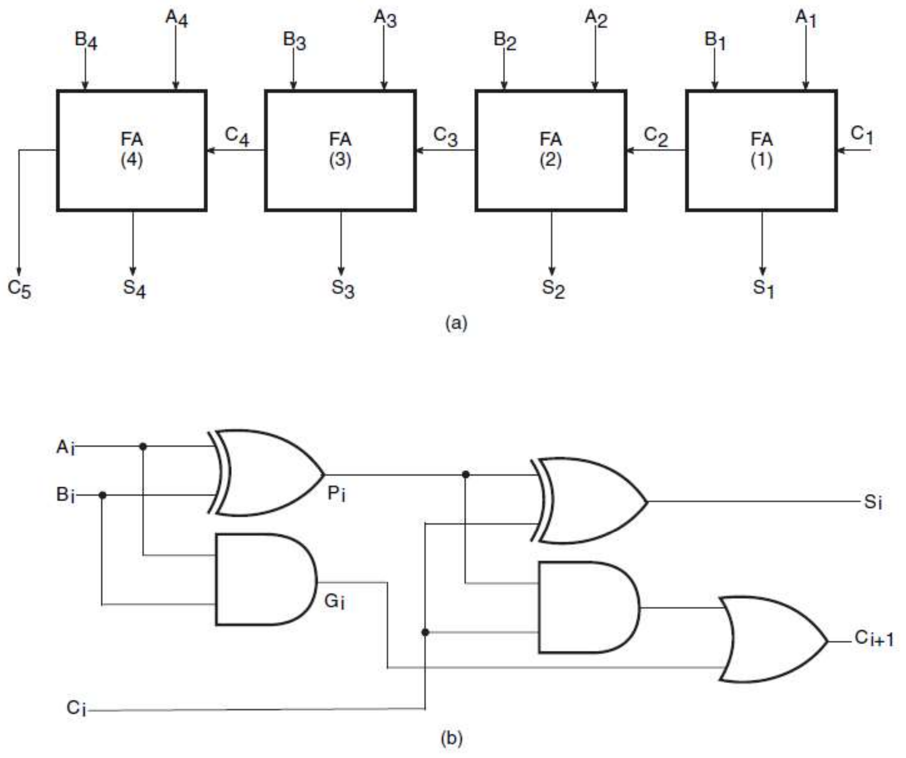


Figure Four-bit binary adder.

The steady state of C_2 will be delayed by two gate delays after the appearance of C_1. Similarly, C_3 and C_4 steady state will be four and six gate delays respectively after C_1. And final carry C_5 will appear after eight gate delays.

Extending it a little further, let us assume that we are having a cascade arrangement of two four-bit adders to be able to handle eight-bit numbers. Now, C_5 will form the input CARRY to the second four-bit adder. The final output CARRY C_9 will now appear after 16 gate delays. This carry propagation delay limits the speed with which two numbers are added. The outputs of any such adder arrangement will be correct only if signals are given enough time to propagate through gates connected between input and output. Since subtraction is also an addition process and operations like multiplication and division are also processes involving successive addition and subtraction, the time taken by an addition process is very critical.

One of the possible methods for reducing carry propagation delay time is to use faster logic gates.

But then there is a limit below which the gate delay cannot be reduced. There are other hardware- related techniques, the most widely used of which is the concept of look-ahead carry. This concept attempts to look ahead and generate the carry for a certain given addition operation that would otherwise have resulted from some previous operation. In order to explain the concept, let us define two new binary variables: P_i called CARRY PROPAGATE and G_i called CARRY GENERATE. Binary variable G_i is so called as it generates a carry whenever A_i and B_i are '1'. Binary variable P_i is called CARRY PROPAGATE as it is instrumental in propagation of C_i to C_{i+1}. CARRY, SUM, CARRY GENERATE and CARRY PROPAGATE parameters are given by the following expressions:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i.B_i$$

$$S_i = P_i \oplus C_i$$

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

In the next step, we write Boolean expressions for the CARRY output of each full adder stage in the four-bit binary adder. We obtain the following expressions:

$$C_2 = G_1 + P_1.C_1$$

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

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

$$C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_1.P_2.P_3.C_1$$

From the expressions for C_2, C_3 and C_4 it is clear that C_4 need not wait for C_3 and C_2 to propagate. Similarly, C_3 does not wait for C_2 to propagate. Hardware implementation of these expressions gives us a kind of look-ahead carry generator. A look-ahead carry generator that implements the above expressions using AND-OR logic is shown in Fig.

Figure shows the four-bit adder with the look-ahead carry concept incorporated. The block labelled *look-ahead carry generator* is similar to that shown in Fig. The logic gates shown to the left of the block represent the input half-adder portion of various full adders constituting the four-bit adder. The EX-OR gates shown on the right are a portion of the output half-adders of various fulladders.

All sum outputs in this case will be available at the output after a delay of two levels of logic gates. 74182 is a typical look-ahead carry generator IC of the TTL logic family. This IC can be used to generate relevant carry inputs for four four-bit binary adders connected in cascade to perform operation on two 16-bit numbers. Of course, the four-bit adders should be of the type so as to produce CARRY GENERATE and CARRY PROPAGATE outputs. Figure shows the arrangement. In the figure shown, C_n is the CARRY input, G_0, G_1, G_2 and G_3 are CARRY GENERATE inputs for 74182 and P_0, P_1, P_2 and P_3 are CARRY PROPAGATE inputs for 74182. C_{n+x}, $C_{n+y}$and$C_{n+z}$are the CARRY outputs generated by 74182 for the four-bit adders. The G and P outputs of 74182 need to be cascaded.

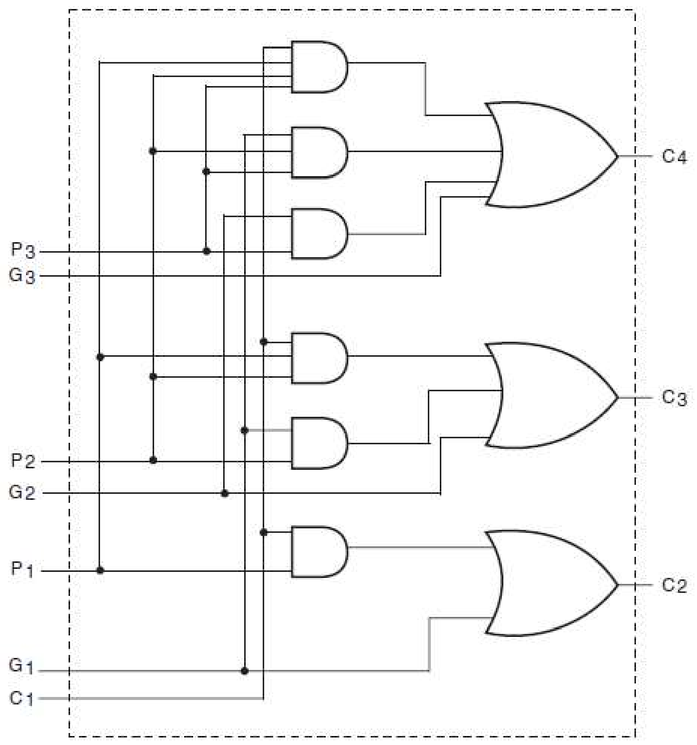


Figure Look-ahead carry generator.

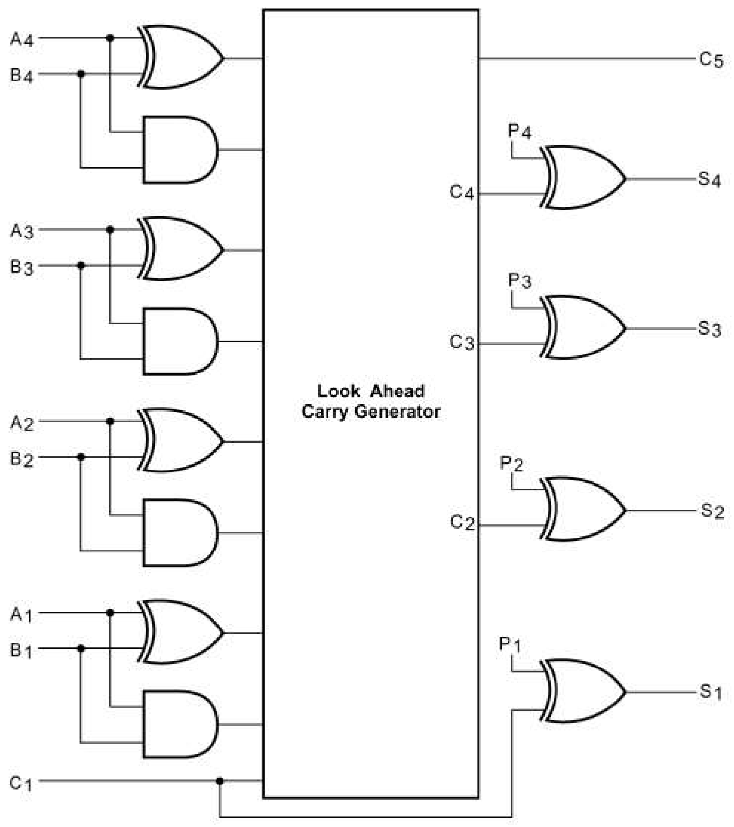


Figure Four-bit full adder with a look-ahead carry generator.

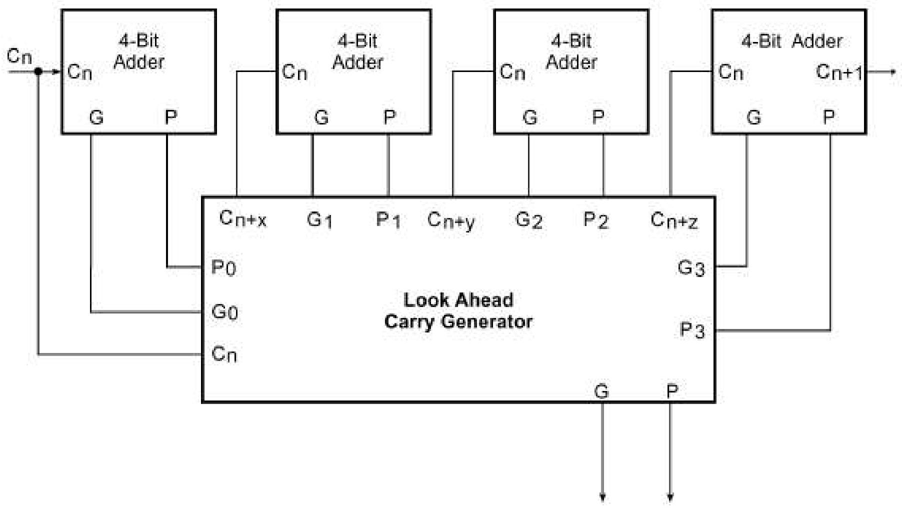


Figure IC 74182 interfaced with four four-bit adders.

2.9 Magnitude Comparator

A *magnitude comparator* is a combinational circuit that compares two given numbers and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions $A = B$ $A > B$ and $A < B$, if A and B are the two numbers being compared. Depending upon the relative magnitude of the two numbers, the relevant output changes state. If the two numbers, let us say, are four-bit binary numbers and are designated as (A_3 A_2 A_1 A_0) and (B_3 B_2 B_1 B_0), the two numbers will be equal if all pairs of significant digits are equal, that is, $A_3 = B_3$, $A_2 = B_2$ $A_1 = B_1$ and $A_0 = B_0$. In order to determine whether A is greater than or less than B we inspect the relative magnitude of pairs of significant digits, starting from the most significant position. The comparison is done by successively comparing the next adjacent lower pair of digits if the digits of the pair under examination are equal. The comparison continues until a pair of unequal digits is reached. In the pair of unequal digits, if $A_i = 1$ and $B_i = 0$, then $A > B$, and if $A_i = 0$, $B_i = 1$ then $A < B$. If X, Y and Z are three variables respectively representing the $A = B$, $A >$ Band $A < B$ conditions, then the Boolean expression representing these conditions are given by the equations

$$X = x_3.x_2.x_1.x_0 \quad \text{where } x_i = A_i.B_i + \overline{A_i}.\overline{B_i}$$

$$Y = A_3.\overline{B_3} + x_3.A_2.\overline{B_2} + x_3.x_2.A_1.\overline{B_1} + x_3.x_2.x_1.A_0.\overline{B_0}$$

$$Z = \overline{A_3}.B_3 + x_3.\overline{A_2}.B_2 + x_3.x_2.\overline{A_1}.B_1 + x_3.x_2.x_1.\overline{A_0}.B_0$$

Let us examine equation. x_3 will be '1' only when both A_3 and B_3 are equal. Similarly, conditions for x_2 , x_1 and x_0 to be '1' respectively are equal A_2 and B_2 , equal A_1 and B_1 and equal A_0 and B_0ANDing of x_3 , x_2 , x_1 and x_0 ensures that X will be '1' when x_3 , x_2 , x_1 and x_0 are in the logic '1'state. Thus, $X = 1$ means that $A = B$.

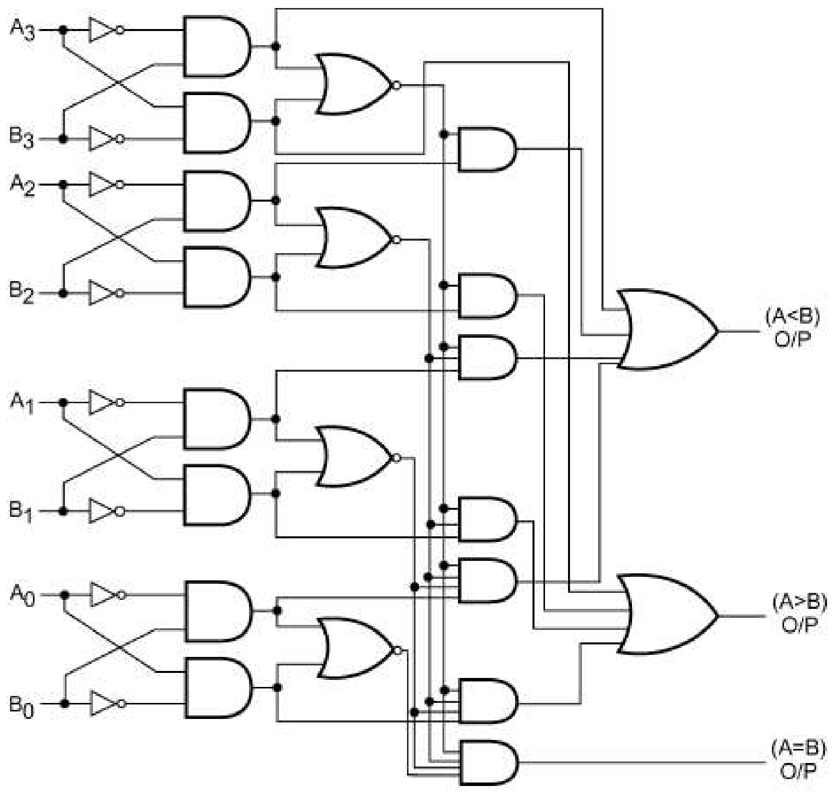


Figure Four-bit magnitude comparator.

Magnitude comparators are available in IC form. For example, 7485 is a four-bit magnitude comparator of the TTL logic family. IC 4585 is a similar device in the CMOS family. 7485 and 4585 have the same pin connection diagram and functional table. The logic circuit inside these devices determines whether one four-bit number, binary or BCD, is *less than*, *equal to* or *greater than* a second four-bit number. It can perform comparison of straight binary and straight BCD (8-4-2-1) codes. These devices can be cascaded together to perform operations on larger bit numbers without the help of any external gates. This is facilitated by three additional inputs called cascading or expansion inputs available on the IC. These cascading inputs are also designated as A = B, A > B and A < B inputs. Cascading of individual magnitude comparators of the type 7485 or 4585 is discussed in the following paragraphs. IC 74AS885 is another common magnitude comparator. The device is an eight- bit magnitude comparator belonging to the advanced Schottky TTL family. It can perform high-speed arithmetic or logic comparisons on two eight-bit binary or 2's complement numbers and produces two fully decoded decisions at the output about one number being either greater than or less than the other. More than one of these devices can also be connected in a cascade arrangement to perform comparison of numbers of longer lengths.

2.10 BCD Adder

A *BCD adder* is used to perform the addition of BCD numbers. A BCD digit can have any of the ten possible four-bit binary representations, that is, 0000, 0001,…, 1001, the equivalent of decimal numbers 0, 1,…, 9. When we set out to add two BCD digits and we assume that there is an input carry too, the highest binary number that we can get is the equivalent of decimal number 19 (9 + 9 + 1).

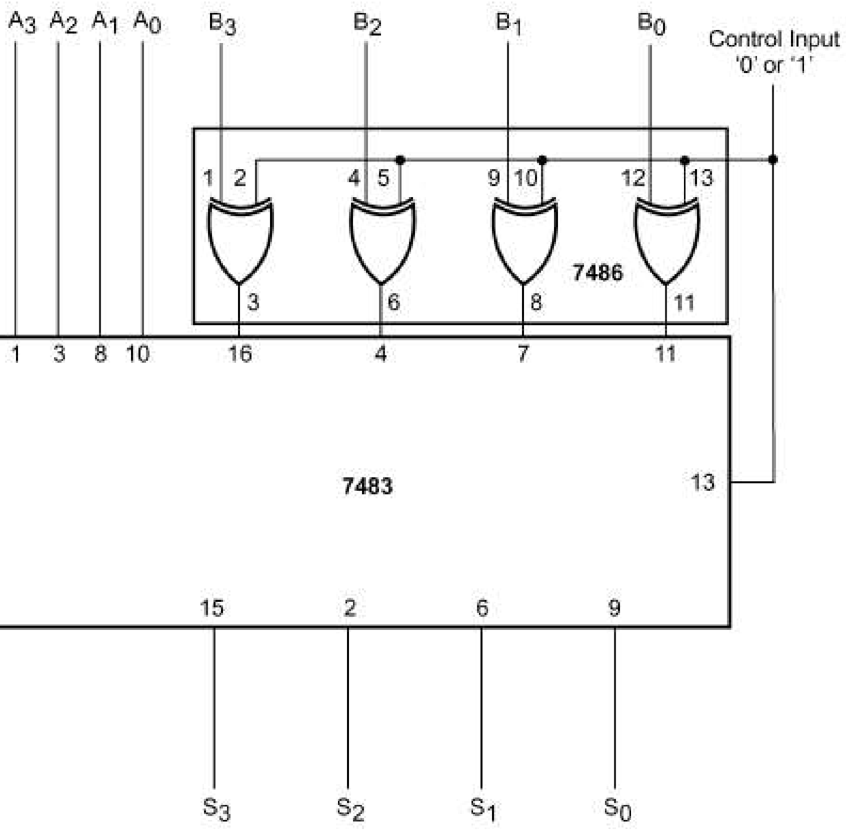


Figure Four-bit adder-subtractor.

This binary number is going to be $(10011)_2$. On the other hand, if we do BCD addition, we would expect the answer to be $(0001\ 1001)_{BCD}$. And if we restrict the output bits to the minimum required, the answer in BCD would be $(1\ 1001)_{BCD}$. Table lists the possible results in binary and the expected results in BCD when we use a four-bit binary adder to perform the addition of two BCD digits. It is clear from the table that, as long as the sum of the two BCD digits remains equal to or less than 9, the four-bit adder produces the correct BCD output.

The binary sum and the BCD sum in this case are the same. It is only when the sum is greater than 9 that the two results are different. It can also be seen from the table that, for a decimal sum greater than 9 (or the equivalent binary sum greater than 1001), if we add 0110 to the binary sum, we can get the correct BCD sum and the desired carry output too. The Boolean expression that can apply the necessary correction is written as

$$C = K + Z_3.Z_2 + Z_3.Z_1$$

Equation implies the following. A correction needs to be applied whenever K = 1. This takes care of the last four entries. Also, a correction needs to be applied whenever both Z_3 and Z_2 are '1'. This takes care of the next four entries from the bottom, corresponding to a decimal sum equal to

Table Results in binary and the expected results in BCD using a four-bit binary adder to perform the addition of two BCD digits.

Decimal sum	Binary sum					BCD sum				
	K	Z_3	Z_2	Z_1	Z_0	C	S_3	S_2	S_1	S_0
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0
9	0	1	0	0	1	0	1	0	0	1
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

12, 13, 14 and 15. For the remaining two entries corresponding to a decimal sum equal to 10 and11, a correction is applied for both Z_3 and Z_1 , being '1'. While hardware-implementing, 0110 can be added to the binary sum output with the help of a second four-bit binary adder. The correction logic as dictated by the Boolean expression should ensure that (0110) gets added only when the above expression is satisfied. Otherwise, the sum output of the first binary adder should be passed on as such to the final output, which can be accomplished by adding (0000) in the second adder. Figure shows the logic arrangement of a BCD adder capable of adding two BCD digits with the help of two four-bit binary adders and some additional combinational logic.

The BCD adder described in the preceding paragraphs can be used to add two single-digit BCD numbers only. However, a cascade arrangement of single-digit BCD adder hardware can be used to perform the addition of multiple-digit BCD numbers. For example, an n-digit BCD adder would require n such stages in cascade. As an illustration, Fig. shows the block diagram of a circuit for the addition of two three-digit BCD numbers. The first BCD adder, labelled LSD (Least Significant Digit), handles the least significant BCD digits. It produces the sum output (S_3 S_2 S_1 S_0), which is the BCD code for the least significant digit of the sum. It also produces an output carry that is fed as an input carry to the next higher adjacent BCD adder. This BCD adder produces the sum output (S_7 S_6 S_5 S_4), which is the BCD code for the second digit of the sum, and a carry output. This output carry serves as an input carry for the BCD adder representing the most significant digits. The sum outputs (S_{11} S_{10} S_9 S_8) represent the BCD code for the MSD of the sum.

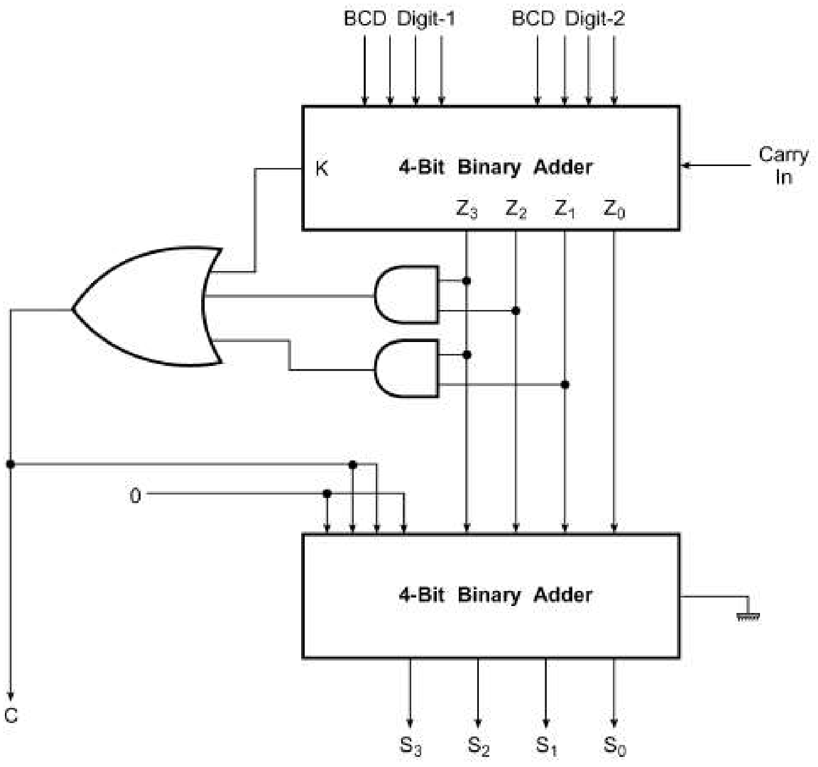


Figure Single-digit BCD adder.

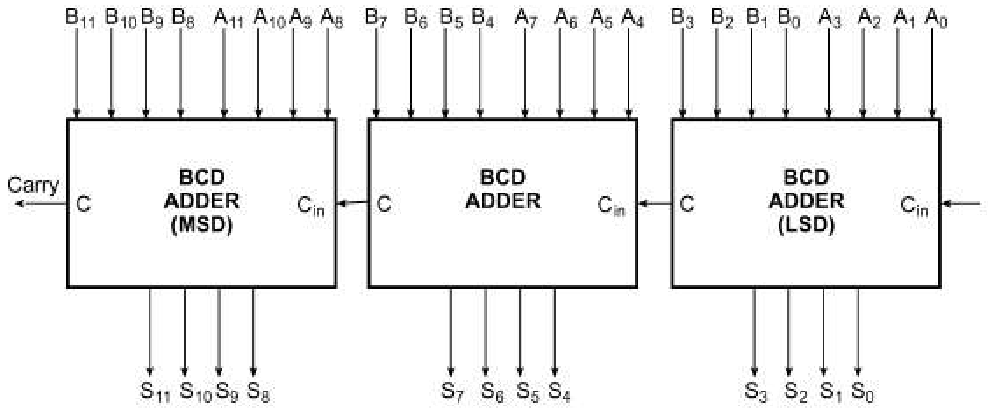
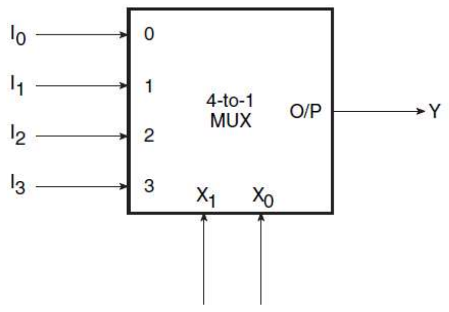


Figure Three-digit BCD adder.

2.11 Multiplexer

A *multiplexer* or *MUX*, also called a *data selector*, is a combinational circuit with more than one input line, one output line and more than one selection line. There are some multiplexer ICs that provide complementary outputs. Also, multiplexers in IC form almost invariably have an ENABLE or STROBE input, which needs to be active for the multiplexer to be able to perform its intended function. A multiplexer selects binary information present on any one of the input lines, depending upon the logic status of the selection inputs, and routes it to the output line. If there are n selection lines, then the number of maximum possible input lines is 2^n and the multiplexer is referred to as a 2^n-to-1 multiplexer or $2^n \times 1$ multiplexer. Figures (a) and (b) respectively show the circuit representation and truth table of a basic 4-to-1 multiplexer.

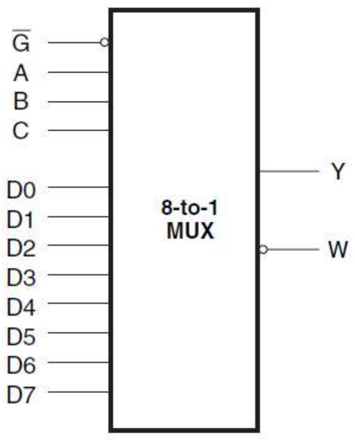


(a)

X_1	X_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b)

Figure (a) 4-to-1 multiplexer circuit representation and (b) 4-to-1 multiplexer truth table.



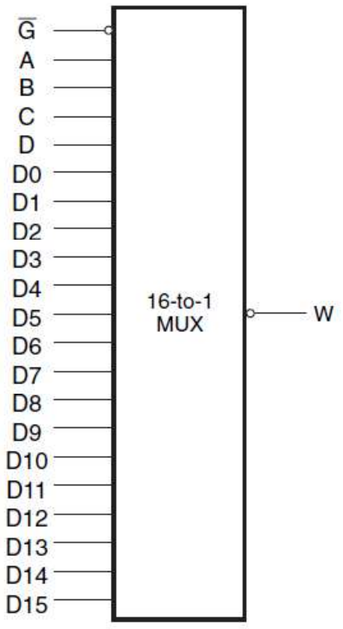
(a)

Inputs				Output	
Select			Enable		
C	B	A	$\overline{G}$	Y	W
X	X	X	H	L	H
L	L	L	L	D0	$\overline{D0}$
L	L	H	L	D1	$\overline{D1}$
L	H	L	L	D2	$\overline{D2}$
L	H	H	L	D3	$\overline{D3}$
H	L	L	L	D4	$\overline{D4}$
H	L	H	L	D5	$\overline{D5}$
H	H	L	L	D6	$\overline{D6}$
H	H	H	L	D7	$\overline{D7}$

$\overline{G}$: ENABLE input
A, B, C : Select inputs
D0-D7 : Data inputs
Y,W : outputs

(b)

Figure (a) 8-to-1 multiplexer circuit representation and (b) 8-to-1 multiplexer truth table.



(a)

Inputs					Output W
Select				Enable	
D	C	B	A	$\overline{G}$	
X	X	X	X	H	H
L	L	L	L	L	$\overline{D0}$
L	L	L	H	L	$\overline{D1}$
L	L	H	L	L	$\overline{D2}$
L	L	H	H	L	$\overline{D3}$
L	H	L	L	L	$\overline{D4}$
L	H	L	H	L	$\overline{D5}$
L	H	H	L	L	$\overline{D6}$
L	H	H	H	L	$\overline{D7}$
H	L	L	L	L	$\overline{D8}$
H	L	L	H	L	$\overline{D9}$
H	L	H	L	L	$\overline{D10}$
H	L	H	H	L	$\overline{D11}$
H	H	L	L	L	$\overline{D12}$
H	H	L	H	L	$\overline{D13}$
H	H	H	L	L	$\overline{D14}$
H	H	H	H	L	$\overline{D15}$

(b)

Figure (a) 16-to-1 multiplexer circuit representation and (b) 16-to-1 multiplexer truth table.

2.11.1 Inside the Multiplexer

We will briefly describe the type of combinational logic circuit found inside a multiplexer by considering the 2-to-1 multiplexer in Fig. (a), the functional table of which is shown in Fig. (b). Figure (c) shows the possible logic diagram of this multiplexer. The circuit functions as follows:

- For $S=0$, the Boolean expression for the output becomes $Y=I_0$.
- For $S=1$, the Boolean expression for the output becomes $Y=I_1$.

Thus, inputs I_0 and I_1 are respectively switched to the output for $S=0$ and $S=1$. Extending the concept further, Fig. shows the logic diagram of a 4-to-1 multiplexer. The input combinations 00, 01, 10 and 11 on the select lines respectively switch I_0, I_1, I_2 and I_3 to the output.

$$Y = I_0.\overline{S_1}.\overline{S_0} + I_1.\overline{S_1}.S_0 + I_2.S_1.\overline{S_0} + I_3.S_1.S_0$$

$$\begin{aligned} Y = I_0.\overline{S_2}.\overline{S_1}.\overline{S_0} + I_1.\overline{S_2}.\overline{S_1}.S_0 + I_2.\overline{S_2}.S_1.\overline{S_0} + I_3.\overline{S_2}.S_1.S_0 + I_4.S_2.\overline{S_1}.\overline{S_0} \\ + I_5.S_2.\overline{S_1}.S_0 + I_6.S_2.S_1.\overline{S_0} + I_7.S_2.S_1.S_0 \end{aligned}$$

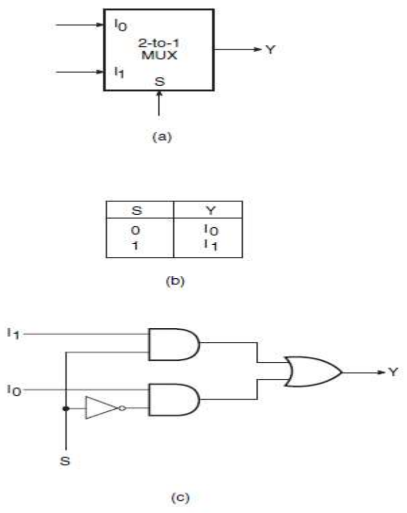
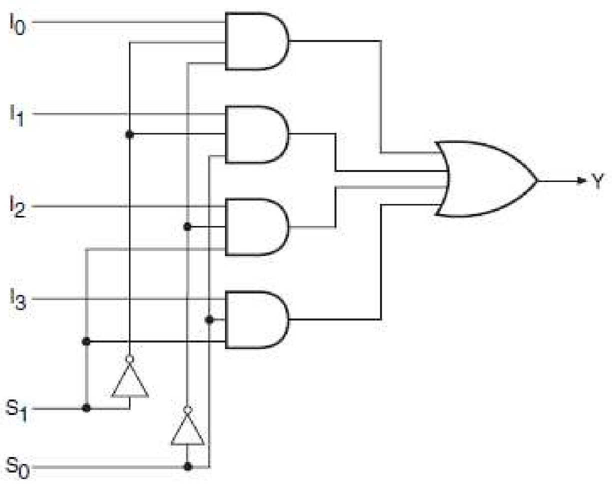


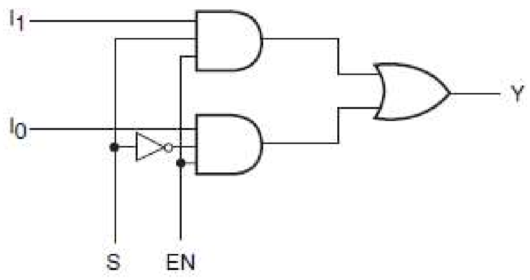
Figure (a) 2-to-1 multiplexer circuit representation, (b) 2-to-1 multiplexer truth table and (c) 2-to-1 multiplexer logic diagram.



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Figure Logic diagram of a 4-to-1 multiplexer.

As outlined earlier, multiplexers usually have an ENABLE input that can be used to control the multiplexing function. When this input is enabled, that is, when it is in logic '1' or logic '0' state, depending upon whether the ENABLE input is active HIGH or active LOW respectively, the output is enabled. The multiplexer functions normally. When the ENABLE input is inactive, the output is disabled and permanently goes to either logic '0' or logic '1' state, depending upon whether the output is uncomplemented or complemented. Figure 8.6 shows how the 2-to-1 multiplexer of Fig. can be modified to include an ENABLE input. The functional table of this modified multiplexer is also shown in Fig. The ENABLE input here is active when HIGH. Some IC packages have more than one multiplexer. In that case, the ENABLE input and selection inputs are common to all multiplexers within the same IC package.



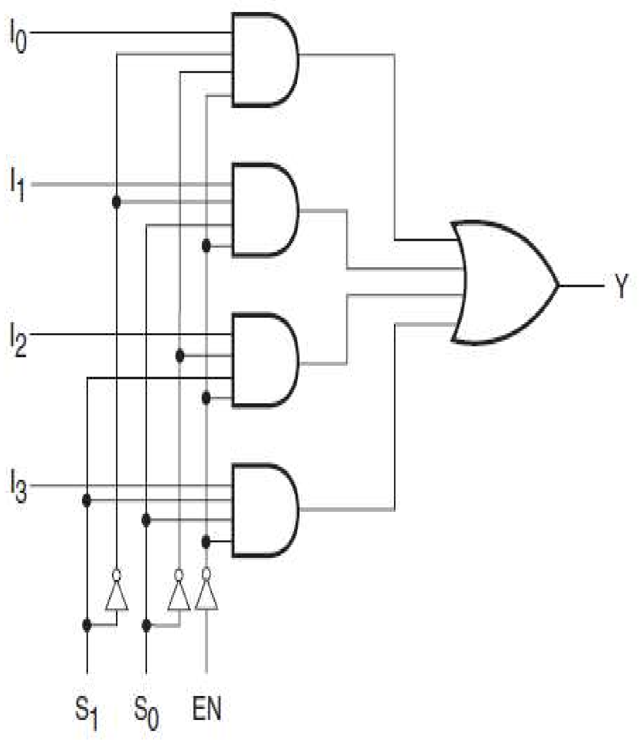
S	EN	Y
X	0	0
0	1	I_0
1	1	I_1

Figure 2-to-1 multiplexer with an ENABLE input.

2.11.2 Implementing Boolean Functions with Multiplexers

One of the most common applications of a multiplexer is its use for implementation of combinational logic Boolean functions. The simplest technique for doing so is to employ a 2^n-to-1 MUX to implement an n-variable Boolean function. The input lines corresponding to each of the minterms present in the Boolean function are made equal to logic '1' state. The remaining minterms that are absent in the Boolean function are disabled by making their corresponding input lines equal to logic '0'. As an example, Fig. (a) shows the use of an 8-to-1 MUX for implementing the Boolean function given by the equation

$$f(A, B, C) = \sum 2, 4, 7$$



S_1	S_0	EN	Y
X	X	1	0
0	0	0	I_0
0	1	0	I_1
1	0	0	I_2
1	1	0	I_3

Figure 4-to-1 multiplexer with an ENABLE input.

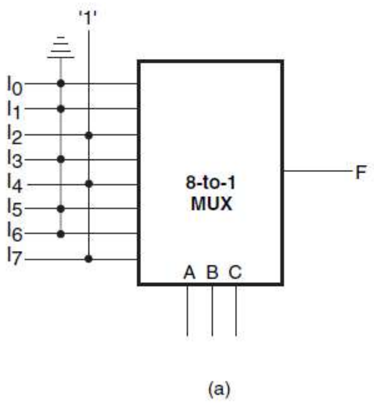
As shown in Fig. the input lines corresponding to the three minterms present in the given Boolean function are tied to logic '1'. The remaining five possible minterms absent in the Boolean function are tied to logic '0'.

However, there is a better technique available for doing the same. In this, a 2^n-to-1 MUX can be used to implement a Boolean function with $n + 1$ variables. The procedure is as follows. Out of $n + 1$ variables, n are connected to the n selection lines of the 2^n-to-1 multiplexer. The left-over variable is used with the input lines. Various input lines are tied to one of the following: '0', '1', the left-over variable and the complement of the left-over variable. Which line is given what logic status can be easily determined with the help of a simple procedure.

It is a three-variable Boolean function. Conventionally, we will need to use an 8-to-1 multiplexer to implement this function. We will now see how this can be implemented with a 4-to-1 multiplexer. The chosen multiplexer has two selection lines.

In the next step, two of the three variables are connected to the two selection lines, with the higher-order variable connected to the higher-order selection line. For instance, in the present case, variables B and C are the chosen variables for the selection lines and are respectively connected to selection lines S_1 and S_0. In the third step, a table of the type shown in Table is constructed. Under the inputs to the multiplexer, minterms are listed in two rows, as shown. The first row lists those terms where remaining variable A is complemented, and second row lists those terms where A is uncomplemented. This is easily done with the help of the truth table.

The required minterms are identified or marked in some manner in this table. In the given table, these entries have been highlighted. Each column is inspected individually. If neither minterm of a certain column is highlighted, a '0' is written below that. If both are highlighted, a '1' is



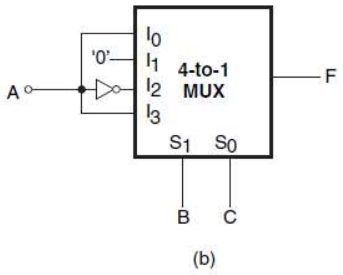


Figure Hardware implementation of the Boolean function given by equation

Truth table.

Minterm	A	B	C	$f(A,B,C)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

written. If only one is highlighted, the corresponding variable (complemented or uncomplemented) is written. The input lines are then given appropriate logic status. In the present case, I_0, I_1, I_2 and I_3 would be connected to A, 0, $\overline{A}$ and A respectively. Figure (b) shows the logic implementation.

Table Implementation table for multiplexers.

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

Table Implementation table for multiplexers.

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

It is not necessary to choose only the leftmost variable in the sequence to be used as input to the multiplexer. Any of the variables can be used provided the implementation table is constructed accordingly. In the problem illustrated above, A was chosen as the variable for the input lines, and accordingly the first row of the implementation table contained those entries where 'A' was complemented and the second row contained those entries where A was uncomplemented. If we consider C as the left-out variable, the implementation table will be as shown in Table

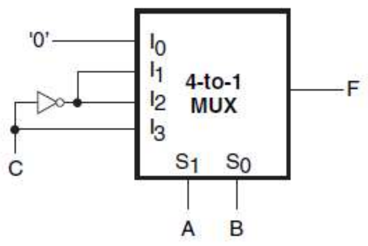


Figure Hardware implementation using a 4-to-1 multiplexer.

Table Implementation table for multiplexers.

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

Example

Implement the product-of-sums Boolean function expressed by $\Pi 1,2,5$ by a suitable multiplexer.

Solution

- Let the Boolean function be $f(A, B, C) = \prod 1, 2, 5$.
- The equivalent sum-of-products expression can be written as $f(A, B, C) = \sum 0, 3, 4, 6, 7$.

The truth table for the given Boolean function is given in Table The given function can be implemented with a 4-to-1 multiplexer with two selection lines. Variables A and B are chosen for the selection lines. The implementation table as drawn with the help of the truth table is given in Table

Table Truth table.

C	B	A	$f(A,B,C)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table Implementation table.

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

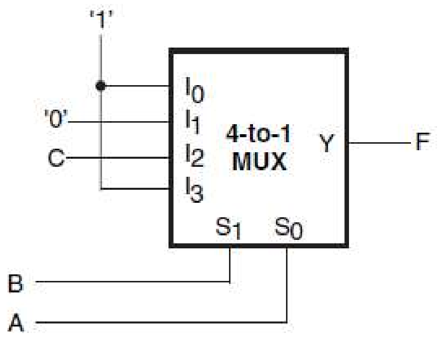
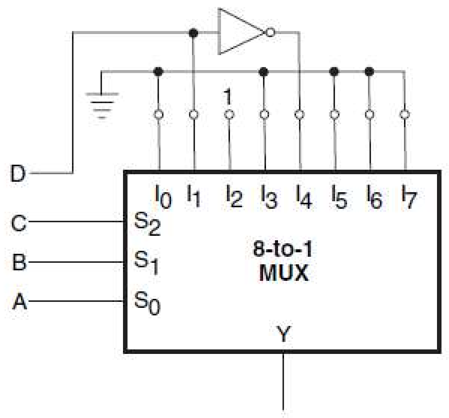
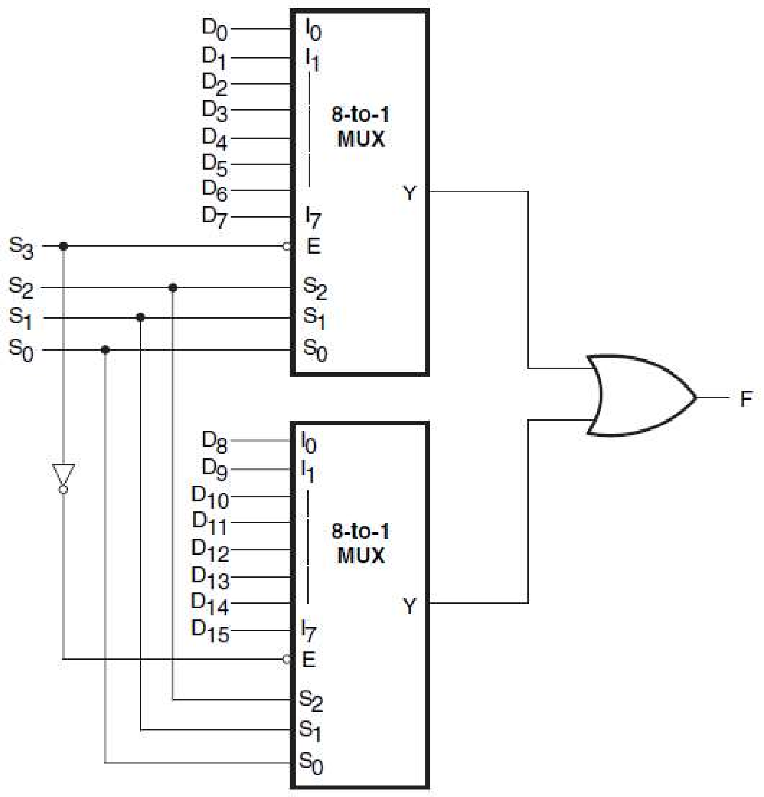


Figure Example



2.12 Encoders

An *encoder* is a multiplexer without its single output line. It is a combinational logic function that has 2^n (or fewer) input lines and n output lines, which correspond to n selection lines in a multiplexer. The n output lines generate the binary code for the possible 2^n input lines. Let us take the case of an octal-to-binary encoder. Such an encoder would have eight input lines, each representing an octal digit, and three output lines representing the three-bit binary equivalent. The truth table of such an encoder is given in Table In the truth table, D_0 to D_7 represent octal digits 0 to 7. A, B and C represent the binary digits.



The eight input lines would have $2^8 = 256$ possible combinations. However, in the case of an octal-to-binary encoder, only eight of these 256 combinations would have any meaning. The remaining combinations of input variables are 'don't care' input combinations. Also, only one of the input lines at a time is in logic '1' state. Figure shows the hardware implementation of the octal-to-binary encoder described by the truth table in Table . This circuit has the shortcoming that it produces an all 0s output sequence when all input lines are in logic '0' state. This can be overcome by having an additional line to indicate an all 0s input sequence.

2.12.1 Priority Encoder

A *priority encoder* is a practical form of an encoder. The encoders available in IC form are all priority encoders. In this type of encoder, a priority is assigned to each input so that, when more than one input is simultaneously active, the input with the highest priority is encoded. We will illustrate the concept of priority encoding with the help of an example. Let us assume that the octal-to-binary encoder described in the previous paragraph has an input priority for higher-order digits. Let us also assume that input lines D_2, D_4 and D_7 are all simultaneously in logic '1' state. In that case, only D_7 will be encoded and the output will be 111. The truth table of such a priority

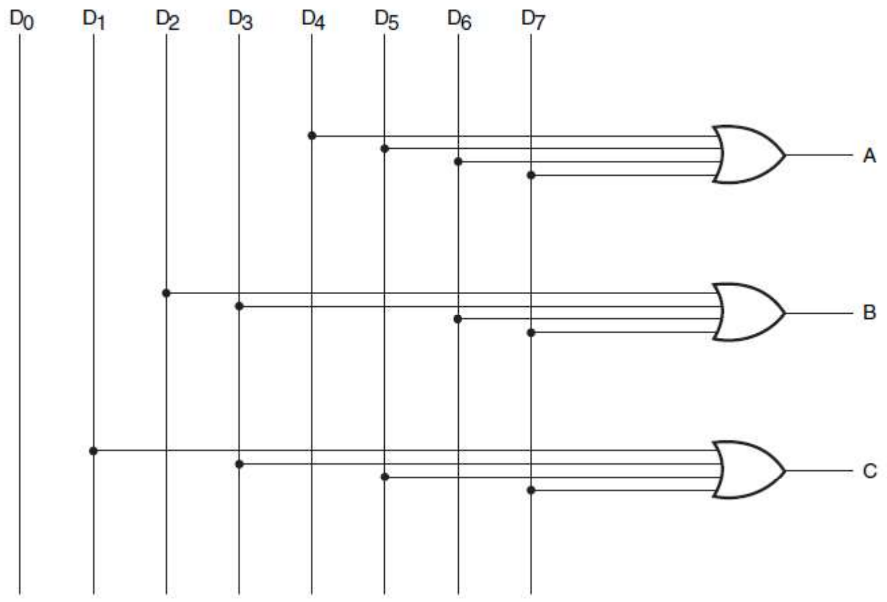


Figure Octal-to-binary encoder.

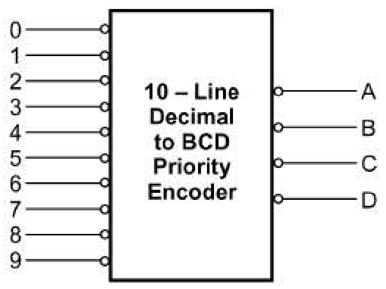
Table Truth table of an encoder.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

encoder will then be modified to what is shown in Table Looking at the last row of the table, it implies that, if $D_7 = 1$, then, irrespective of the logic status of other inputs, the output is 111 as D_7 will only be encoded. As another example, Fig. shows the logic symbol and truth table of a 10-line decimal to four-line BCD encoder providing priority encoding for higher-order digits, with digit 9 having the highest priority. In the functional table shown, the input line with highest priority having a LOW on it is encoded irrespective of the logic status of the other input lines.

Table Priority encoder.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
X	1	0	0	0	0	0	0	0	0	1
X	X	1	0	0	0	0	0	0	1	0
X	X	X	1	0	0	0	0	0	1	1
X	X	X	X	1	0	0	0	1	0	0
X	X	X	X	X	1	0	0	1	0	1
X	X	X	X	X	X	1	0	1	1	0
X	X	X	X	X	X	X	1	1	1	1



Inputs										Outputs			
0	1	2	3	4	5	6	7	8	9	D	C	B	A
X	X	X	X	X	X	X	X	X	0	0	1	1	0
X	X	X	X	X	X	X	X	0	1	0	1	1	1
X	X	X	X	X	X	X	0	1	1	1	0	0	0
X	X	X	X	X	X	0	1	1	1	1	0	0	1
X	X	X	X	X	0	1	1	1	1	1	0	1	0
X	X	X	X	0	1	1	1	1	1	1	0	1	1
X	X	X	0	1	1	1	1	1	1	1	1	0	0
X	X	0	1	1	1	1	1	1	1	1	1	0	1
X	0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1

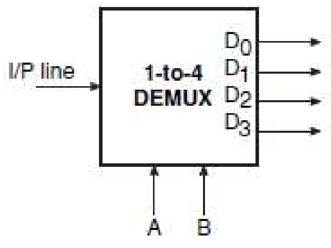
Figure 10-line decimal to four-line BCD priority encoder.

Some of the encoders available in IC form provide additional inputs and outputs to allow expansion. IC 74148, which is an eight-line to three -line priority encoder, is an example. ENABLE-IN (EI) and ENABLE-OUT (EO) terminals on this IC allow expansion. For instance, two 74148s can be cascaded to build a 16-line to four-line priority encoder.

2.13 Demultiplexers and Decoders

A *demultiplexer* is a combinational logic circuit with an input line, 2^n output lines and n select lines. It routes the information present on the input line to any of the output lines. The output line that gets the information present on the input line is decided by the bit status of the selection lines. A *decoder* is a special case of a demultiplexer without the input line. Figure (a) shows the circuit representation of a 1-to-4 demultiplexer. Figure (b) shows the truth table of the demultiplexer when the input line is held HIGH.

A decoder, as mentioned earlier, is a combinational circuit that decodes the information on n input lines to a maximum of 2^n unique output lines. Figure shows the circuit representation of 2-to-4, 3-to-8 and 4-to-16 line decoders. If there are some unused or 'don't care' combinations in the n-bit code, then there will be fewer than 2^n output lines. As an illustration, if there are three input lines, it



(a)

I/P	Select		O/P			
	A	B	D_0	D_1	D_2	D_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

(b)

Figure 1-to-4 demultiplexer.

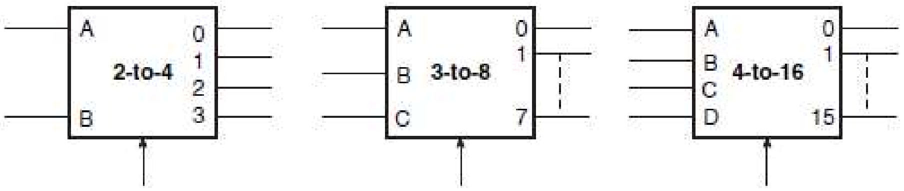


Figure Circuit representation of 2-to-4, 3-to-8 and 4-to-16 line decoders.

can have a maximum of eight unique output lines. If, in the three-bit input code, the only used three-bit combinations are 000, 001, 010, 100, 110 and 111 (011 and 101 being either unused or don't care combinations), then this decoder will have only six output lines. In general, if n and m are respectively the numbers of input and output lines, then $m \leq 2^n$.

A decoder can generate a maximum of 2^n possible minterms with an n-bit binary code. In order to illustrate further the operation of a decoder, consider the logic circuit diagram in Fig. This logic circuit, as we will see, implements a 3-to-8 line decoder function. This decoder has three inputs designated as A, B and C and eight outputs designated as D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7. From the truth table given along with the logic diagram it is clear that, for any given input combination, only one of the eight outputs is in logic '1' state. Thus, each output produces a certain minterm that corresponds to the binary number currently present at the input. In the present case, D_0, D_1, D_2, D_3, D_4, D_5, D_6 and D_7 respectively represent the following minterms:

$$D_0 \rightarrow \overline{A}.\overline{B}.\overline{C},\ D_1 \rightarrow \overline{A}.\overline{B}.C,\ D_2 \rightarrow \overline{A}.B.\overline{C},\ D_3 \rightarrow \overline{A}.B.C$$

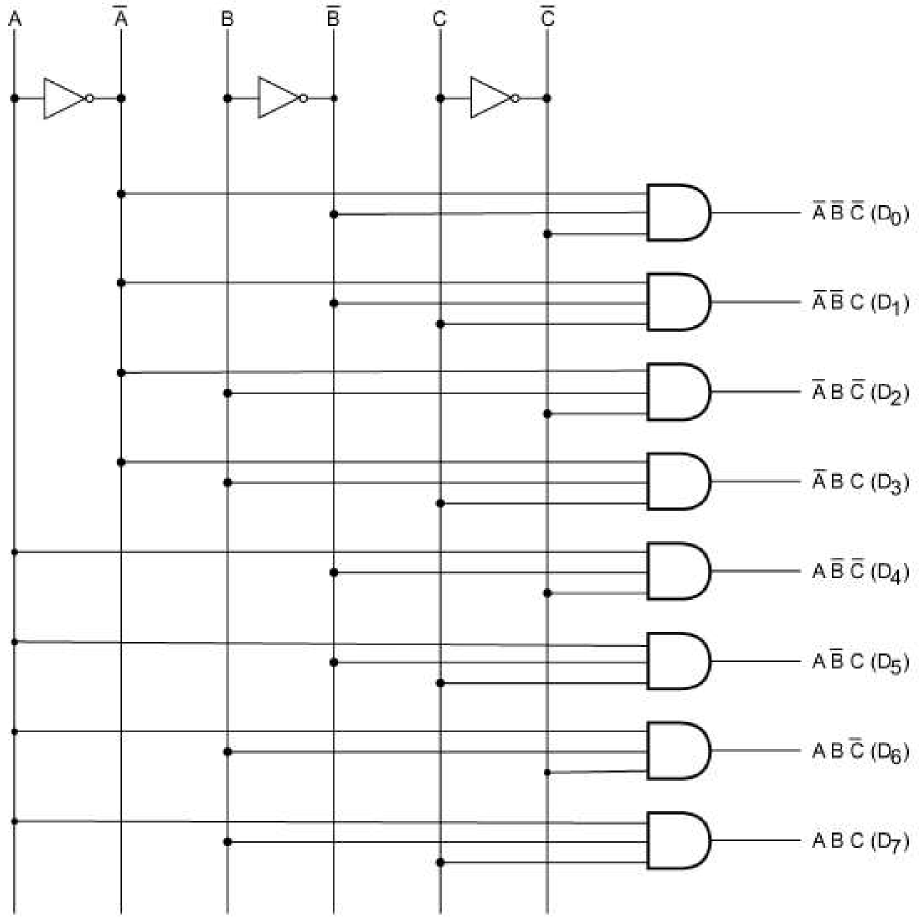
$$D_4 \rightarrow \overline{A}.\overline{B}.\overline{C},\ D_5 \rightarrow A.\overline{B}.C,\ D_6 \rightarrow A.B.\overline{C},\ D_7 \rightarrow A.B.C$$

2.13.1 Implementing Boolean Functions with Decoders

A decoder can be conveniently used to implement a given Boolean function. The decoder generates the required minterms and an external OR gate is used to produce the sum of minterms. Figure shows the logic diagram where a 3-to-8 line decoder is used to generate the Boolean function given by the equation

$$Y = A.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.B.C + \overline{A}.\overline{B}.\overline{C}$$

In general, an n-to-2^n decoder and m external OR gates can be used to implement any combinational circuit with n inputs and m outputs. We can appreciate that a Boolean function with a large number of minterms, if implemented with a decoder and an external OR gate, would require an OR gate with an equally large number of inputs. Let us consider the case of implementing a four-variable Boolean function with 12 minterms using a 4-to-16 line decoder and an external OR gate. The OR gate here needs to be a 12-input gate. In all such cases, where the number of minterms in a given Boolean function with n variables is greater than $2^n/2$ (or 2^{n-1}), the complement Boolean function will have fewer minterms. In that case it would be more advantageous to do NORing of minterms of the complement Boolean function using a NOR gate rather than doing ORing of the given function using an OR gate. The output will be nothing but the given Boolean function.


A
$\bar{A}$
B
$\bar{B}$
C
$\bar{C}$
$\bar{A}\bar{B}\bar{C}$ (D_0)
$\bar{A}\bar{B}C$ (D_1)
$\bar{A}B\bar{C}$ (D_2)
$\bar{A}BC$ (D_3)
$A\bar{B}\bar{C}$ (D_4)
$A\bar{B}C$ (D_5)
$AB\bar{C}$ (D_6)
ABC (D_7)

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

Figure Logic diagram of a 3-to-8 line decoder.

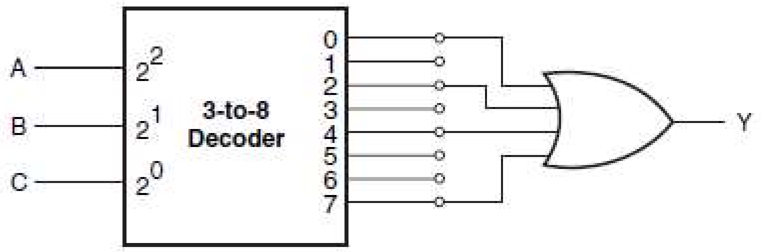


Figure Implementing Boolean functions with decoders.

2.13.2 Cascading Decoder Circuits

There can possibly be a situation where the desired number of input and output lines is not available in IC decoders. More than one of these devices of a given size may be used to construct a decoder that can handle a larger number of input and output lines. For instance, 3-to-8 line decoders can be used to construct 4-to-16 or 5-to-32 or even larger decoder circuits. The basic steps to be followed to carry out the design are as follows:

1. If n is the number of input lines in the available decoder and N is the number of input lines in the desired decoder, then the number of individual decoders required to construct the desired decoder circuit would be 2^{N-n}.
2. Connect the less significant bits of the input lines of the desired decoder to the input lines of the available decoder.
3. The left-over bits of the input lines of the desired decoder circuit are used to enable or disable the individual decoders.
4. The output lines of the individual decoders together constitute the output lines, with the outputs of the less significant decoder constituting the less significant output lines and those of the higher–order decoders constituting the more significant output lines. The concept is further illustrated in solved example 8.8, which gives the design of a 4-to-16 decoder using 3-to-8 decoders.

Example

Implement a full adder circuit using a 3-to-8 line decoder.

Solution

A decoder with an OR gate at the output can be used to implement the given Boolean function. The decoder should at least have as many input lines as the number of variables in the Boolean function to be implemented. The truth table of the full adder is given in Table , and Fig. shows the hardware implementation.

From the truth table, Boolean functions for SUM and CARRY outputs are given by the following equations:

$$\text{Sum output } S = \Sigma\ 1, 2, 4, 7$$

$$\text{Carry output } C_o = \Sigma\ 3, 5, 6, 7$$

Table Example

A	B	C	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

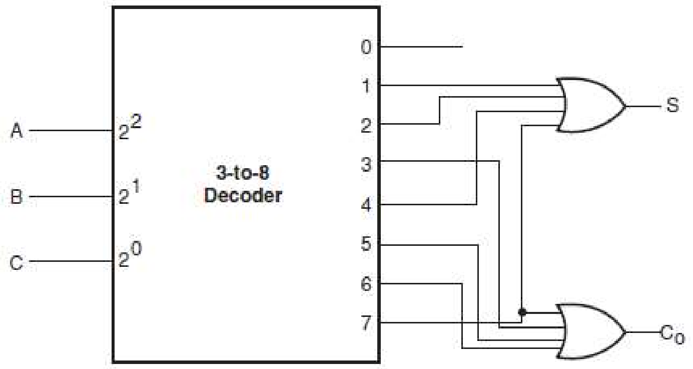


Figure Example

1. **Define combinational logic.**
   When logic gates are connected together to produce a specified output for certain specified combinations of input variables, with no storage involved, the resulting circuit is called combinational logic.
2. **Explain the design procedure for combinational circuits.**
   The problem definition
   Determine the number of available input variables & required O/P variables.
   Truth Table Construction
   Obtain simplified Boolean expression for each O/P (using K-Map).
   Obtain the logic diagram.
3. **Define Half adder and full adder**
   Half Adder: The logic circuit that performs the addition of two bits is a half adder.
   Full Adder: The circuit that performs the addition of three bits is a full adder.
4. **Define Decoder?**
   A decoder is a multiple - input multiple output logic circuit that converts coded inputs into coded outputs where the input and output codes are different.
5. **What is binary decoder?**