3.4.1 Karnaugh Maps

To minimize a Boolean equation in the sum-of-products form, we need to reduce the number of product terms by applying the combining Boolean Theorem (Theorem 14) from Section 2.5.1. In so doing, we will also have

reduced the number of variables used in the product terms. For example, given the following 3-variable function $F = xy'z' + xyz'$ we can reduce it to

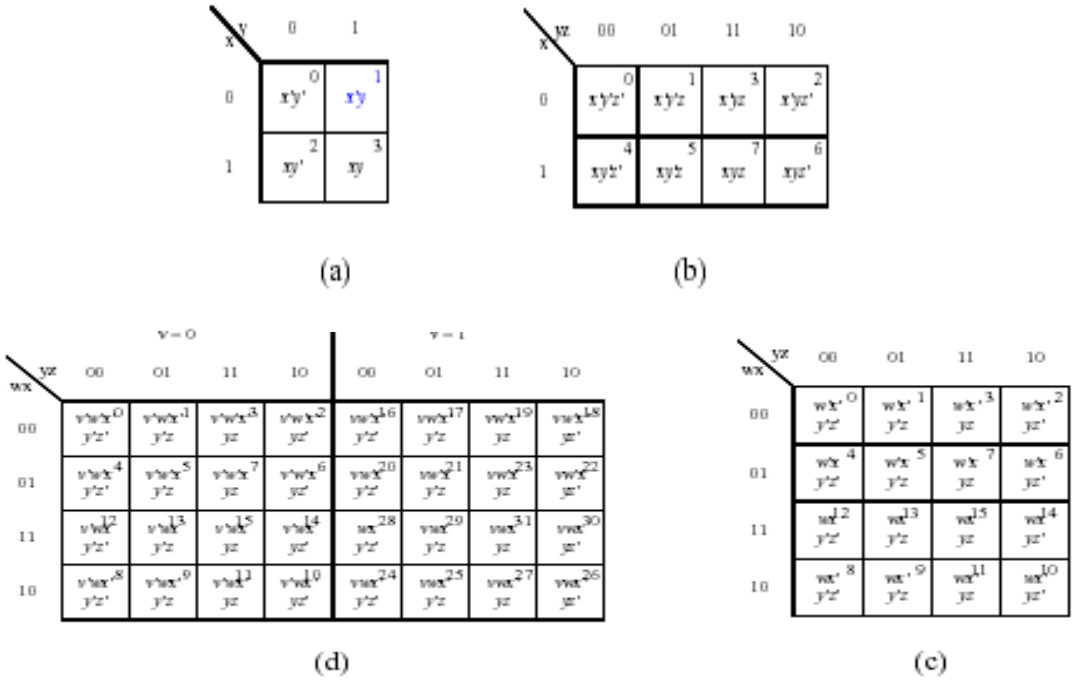
$F = xz' (y' + y) = xz' 1 = xz'$

In other words, two product terms that differ by only one variable whose value is a 0 (primed) in one term, and a 1 (unprimed) in the other term, can be combined together to form just one term with that variable omitted as shown in the example above. Thus, we have reduced the number of product terms and the resulting product term has one less variable. By reducing the number of product terms, we reduce the number of OR operators required, and by reducing the number of variables in a product term, we reduce the number of AND operators required. Looking at a logic function's truth table, it is sometimes difficult to see how the product terms can be combined and minimized. A Karnaugh map, or K-map for short, provides a simple and straightforward procedure for combining these product terms. A K-map is just a graphical representation of a logic function's truth table where the minterms are grouped in such a way that it allows one to easily see which of the minterms can be combined. It is a 2-dimensional array of squares, each of which represents one minterm in the Boolean function. Thus, the map for an n-variable function is an array with $2n$ squares.

Figure 3.5 shows the K-maps for functions with 2, 3, 4, and 5 variables. Notice the labeling of the columns and rows are such that any two adjacent columns or rows differ in only one bit change. This condition is required because we want minterms in adjacent squares to differ in the value of only one variable or one bit, and so these minterms can be combined together. This is why the labeling

for the third and fourth columns, and the third and fourth rows are always interchanged. When we read K-maps, we need to visualize it as such that the two end columns or rows wrap around so that the first and last columns, and the first and last rows are really adjacent to each other because they differ in only one bit also.

In Figure 3.5, the K-map squares are annotated with its minterm and its minterm number for easy reference only. For example, in Figure 3.5 (a) for a 2-variable K-map, the entry in the first row and second column is labeled *x'y*, and annotated with the number 1. This is because the first row is when the variable *x* is a 0, and the second column is when the variable *y* is a 1. Since for minterms, we need to prime a variable whose value is a 0, and not prime it if its value is a 1, therefore, this entry represents the minterm *x'y*, which is minterm



number 1.

Figure 3.5. Karnaugh maps for:

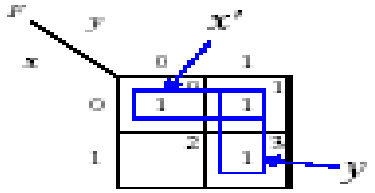
(a) 2 variables; (b) 3 variables; (c) 4 variables; (d) 5 variables.

Be careful that if we label the rows and columns differently, the minterms and the minterm numbers will be in different locations. When we are actually using K-maps to minimize an equation, we will not write these in the squares.

Instead, we will be putting 0's and 1's in the squares.

Given a Boolean function, we set the value for each K-map square to either a 0 or a 1 depending on whether that minterm for the function is a 0-minterm or a 1-minterm respectively. However, since we are only interested in the 1-minterms for a function, the 0's are sometimes not written in the 0-minterm squares.

For example, the K-map for the 2-variable function $F = x'y' + x'y + xy$ is



The 1-minterms $m0$ ($x'y'$) and $m1$ ($x'y$) are adjacent to each other, which means that they differ in the value of only one variable. In this case, x is 0 for both minterms, but for y, it is a 0 for one minterm and a 1 for the other minterm. Thus, variable y can be dropped and the two terms are combined together giving just x'. The prime in x' is because x is 0 for both minterms. This reasoning corresponds with the expression

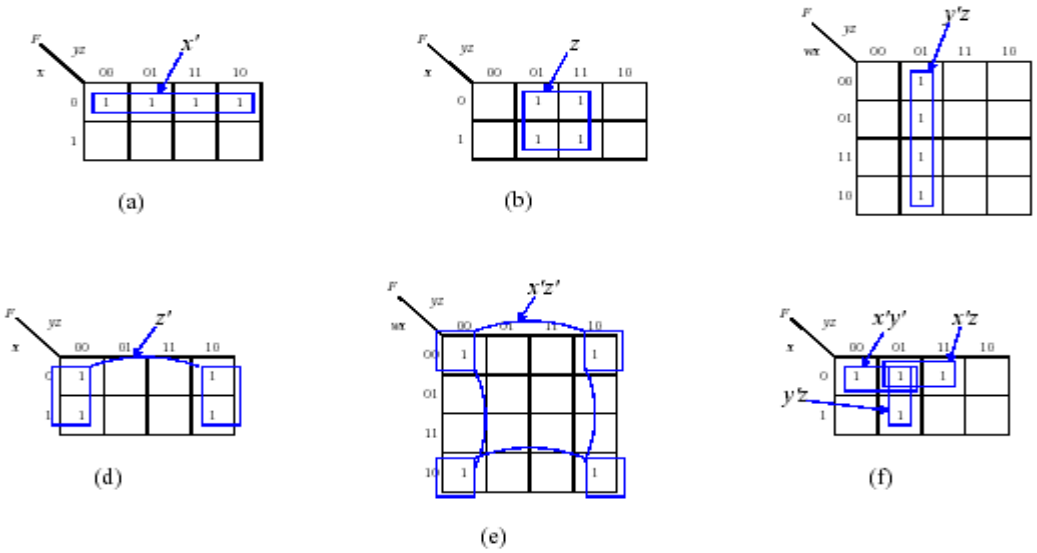
$$x'y' + x'y = x' (y'+y) = x'$$

Similarly, the 1-minterms $m1$ ($x'y$) and $M3$ (xy) are also adjacent and y is the variable having the same value for both minterms, and so they can be combined to give $x'y + xy = y$. We use the term **subcube** to refer to a rectangle of adjacent 1-minterms. These subcubes must be rectangular in shape and can only have sizes that are powers of two. Formally, for an n-variable K-map, an *m-subcube* is defined as that set of $2m$ minterms in which $n - m$ of the variables will have the same value in every minterm while the remaining variables will take on the $2m$ possible combinations of 0's and 1's. Thus, a 1-minterm all by itself is called a 0-subcube, and two adjacent 1-minterms is a 1-subcube. In the above 2-variable K-map, there are two 1-subcubes: one labeled with x' and one

with *y*.

A 2-subcube will have four adjacent 1-minterms and can be in the shape of any one of those shown in Figure 3.6 (a) to (e). Notice that Figure 3.6 (d) and (e) also form 2-subcubes even though the four 1-minterms are not physically adjacent to each other. They are considered to be adjacent because the first and last rows, and first and last columns wrap around in a K-map. In Figure 3.6 (f), the four 1-minterms cannot form a 2-subcube because even though they are physically adjacent to each other, they do not form a rectangle. However, they can form three 1- subcubes – *y'z*, *x'y'* and *x'z*. We say that a subcube is *characterized* by the variables having the same values for all the minterms in that subcube. In general, an *m*-subcube for an *n*-variable K-map will be characterized by $n - m$ variables. If the value that is similar for all the variables is a 1, that variable is unprimed, whereas, if the value that is similar for all the variables is a 0, that variable is primed.

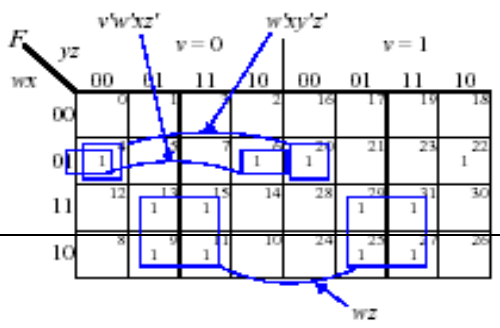
In an expression, this is equivalent to the resulting smaller product term when the minterms are combined together. For example, the 2-subcube in Figure 3.6 (d) is characterized by *z′* since the value of *z* is 0 for all the minterms, whereas



the values for *x* and *y* are not all the same for all the minterms. Similarly, the 2-subcube in Figure 3.6 (e) is characterized by *x′z′*.

Figure 3.6. Examples of K-maps with 2-subcubes: (a) and (b) 3-variable; (c) 4-variable; (d) 3-variable with wrap around subcube; (e) 4-variable with wrap around subcube; (f) cannot form one 2-subcube.

For a 5-variable K-map as shown in Figure 3.7, we need to visualize the right half of the array where *v* = 1 to be on top of the left half where *v* = 0. Thus, for example, minterm 20 is adjacent to minterm 4 since one is on top of the other, and they form the 1-subcube *w′xy′z′*. Minterms 9, 11, 13, 15, 25, 27, 29, and 31 are all adjacent, and together they form the subcube *wz*. Now, that we are viewing this 5-variable K-map in three dimensions, we also need to change the condition of the subcube shape to be a three dimensional rectangle. Even though minterm 6 is physically adjacent to minterm 20 in the map, they cannot be combined together because when you visualize the right half as being on top of the left half, then they are really not on top



of each other. Instead minterm 6 is adjacent to minterm 4 because the columns wrap around, and they form the subcube *v'w'xz'*. You can see that this visualization becomes almost impossible very quickly as we increase the number of variables.

Figure 3.7. A 5-variable K-map with wrap around subcubes.

The K-map method reduces a Boolean function from its canonical form to its standard form. The goal for the K-map method is to find as few subcubes as possible to cover all the 1-minterms in the given function. This naturally implies that the size of the subcube should be as big as possible. The reasoning for this is that each subcube corresponds to a product term, and all the subcubes (or product terms) must be ORed together to get the function. Larger subcubes require fewer AND gates because of fewer variables in the product term, and fewer subcubes will require fewer inputs to the OR gate.

The procedure for using the K-map method is as follows:

1. Draw the appropriate K-map for the given function and place a 1 in the squares that correspond to the function's 1-minterms.

2. For each 1-minterm, find the largest subcube that covers this 1-minterm. This largest subcube is known as aprime implicant (PI). By definition, a **prime implicant** is a subcube that is not contained within any other subcube. If there are more than one subcube that is the same size as the largest subcube, then they are all prime implicants.

3. Look for 1-minterms that are covered by only one prime implicant. Since this prime implicant is the only subcube that covers this particular 1-minterm, this prime implicant must be in the final solution. This prime

implicant is referred to as an *essential* prime implicant (EPI). By definition, an **essential prime implicant** is a subcube that includes a 1-minterm that is not included in any other subcube.

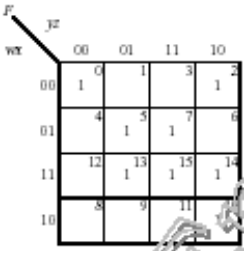
4. Create a minimal cover list by selecting the smallest possible number of

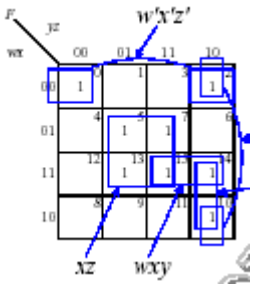
prime implicants such that every 1- minterm is contained in at least one prime implicant. This cover list must include all the essential prime implicants plus zero or more of the remaining prime implicants. It is acceptable that a particular 1-minterm is covered in more than one prime implicant, but all 1-minterms must be covered.

5. The final minimized function is obtained by ORing all the prime implicants from the minimal cover list. Note that the final minimized function obtained by the K-map method may not be in its most reduced form. It is only in its most reduced *standard* form. Sometimes, it is possible to reduce the standard form further into a nonstandard form.

Example 3.4

Use the K-map method to minimize a 4-variable (*w*, *x*, *y*, and *z*) function *F* with





1-minterm	Prime Implicant
m_0	*w'x'z'*
m_2	*w'x'z', x'yz'*
m_5	*xz*
m_7	*xz*
m_{10}	*x'yz', wyz'*
m_{13}	*xz*
m_{14}	*wyz', wxy*
m_{15}	*xz*

the 1-minterms: *m*0, *m*2, *m*5, *m*7, *m*10, *m*13, *m*14, and *m*15. We start with the following 4-variable K-map:

Thus, there are five prime implicants: *w'x'z'*, *x'yz'*, *xz*, *wyz'*, and *wxy*. Of these five prime implicants, *w'x'z'* and *xz* are essential prime implicants since *m*0 is covered only by *w'x'z'*, and *m*5, *m*7, and *m*13 are covered only by *xz*. We start the cover list by including the two essential prime implicants *w'x'z'* and *xz*.

These two subcubes will have covered minterms *m*0, *m*2, *m*5, *m*7, *m*13, and *m*15. To cover the remaining two uncovered minterms *m*10 and *m*14, we want to use as few prime implicants as possible. Hence, we select the prime implicant *wyz'* which covers both of

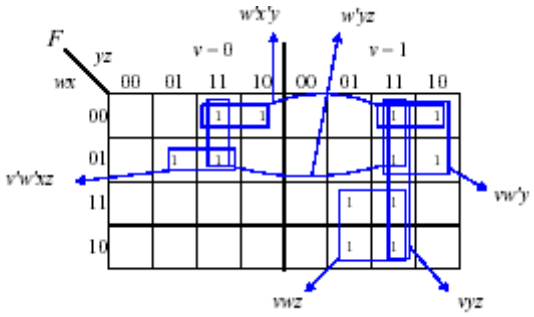
them. Finally, our reduced standard form equation is obtained by ORing these three prime implicants

$F = w'x'z' + xz + wyz'$

Notice that we can reduce this standard form equation even further by factoring out the z' from the first and last term to get the non-standard form equation $F = z'(w'x' + wy) + xz$ □

Example 3.5

Use the K-map method to minimize a 5-variable function *F* (*v*, *w*, *x*, *y* and *z*) with the 1-minterms: *v'w'x'yz'*, *v'w'x'yz*, *v'w'xy'z*, *v'w'xyz*, *vw'x'yz'*, *vw'x'yz*,



vw'xyz', *vw'xyz*, *vwx'y'z*, *vwx'yz*, *vwxy'z*, and *vwxyz*.

The list of prime implicants is: *v'w'xz*, *w'x'y*, *w'yz*, *vw'y*, *vyz*, and *vwz*. From this list of prime implicants, *w'yz* and *vyz* are not essential. The four remaining essential prime implicants are able to cover all the 1-minterms. Hence, the solution in standard form is

$F = v'w'xz + w'x'y + vw'y + vwz$

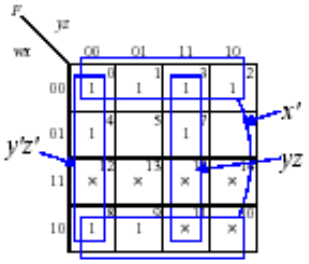
3.4.2 Don't-cares

There are times when a function is not fully specified. In other words, there are some minterms for the function where we do not care whether their values are a 0 or a 1. When drawing the K-map for these "don't-care" minterms, we assign an "Ø" in that square instead of a 0 or a 1. Usually, a function can be reduced even further if we remember that these Ø's can be either a 0 or a 1. As you recall when drawing K-maps, enlarging a subcube reduces the number of variables for that term. Thus, in drawing subcubes, some of them may be enlarged if we treat some of these x's as 1's. On the other hand, if some of these Ø's will not enlarge a subcube, then we want to treat them as 0's so that we do not need to cover them. It is not necessary to treat all Ø's to be all 1's or all 0's. We can assign some Ø's to be 0's and some to be 1's.

For example, given a function having the following 1-minterms and don't-care minterms:

1-minterms: *m*0, *m*1, *m*2, *M3*, *m*4, *m*7, *m*8, and *m*9

x-minterms: *m*10, *m*11, *m*12, *m*13, *m*14, and *m*15



we obtain the following K-map with the prime implicants *x'*, *yz*, and *y'z'*.

Notice that in order to get the 4-subcube characterized by *x'* the two don't-care minterms *m*10 and *m*11 are taken to have the value 1. Similarly, the don't-care minterms *m*12 and *m*15 are assigned a 1 for the subcubes *y'z'* and *yz* respectively. On the other hand, the don't-care minterms *m*13 and *m*14 are taken to have the value 0 so that they do not need to be covered in the solution. The reduced standard form function as obtained from the K-map is, therefore,

$F = x' + yz + y'z'$. Again, this equation can be reduced further by recognizing that $yz + y'z' = y \_ z$. Thus, $F = x' + (y \_ z)$.

4.3 Adder

4.3.1 Half Adder

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The Input variables designats the augend and addend bits; the output variables produce the sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs. The truth table for the half adder is listed in Table 4.3.

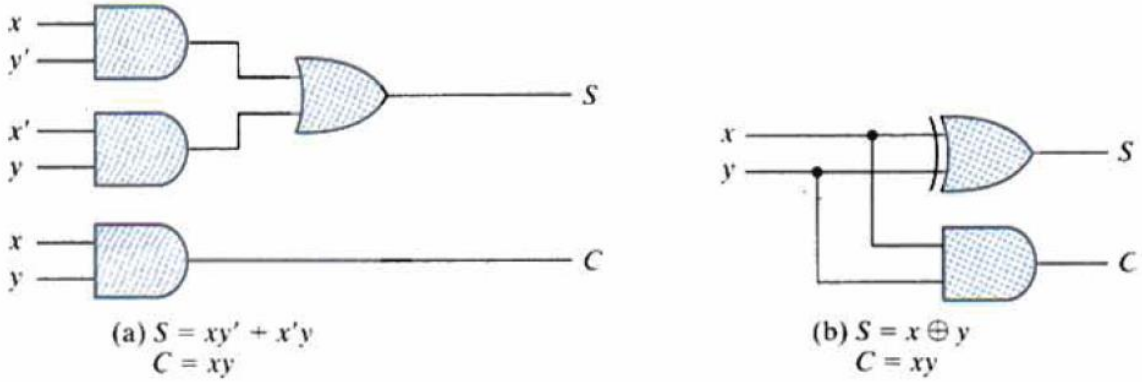
Table 4.3
Half Adder

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The C output is 1 only when both inputs are 1, The S output represent the least significant bit of the sum.

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

$$S = x'y + xy'$$
$$C = xy$$



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

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

FIGURE 4.5
Implementation of half adder

4.3.2 Full Adder

A full adder is combinational circiut that forms the arithmatic sum of three bits. It consists of three inputs and two outputs. Two of the input variabies denoted by x and y, represent the two significant bits to be added. The third input ,z, repesents the carry from the previous lower significant position. Two output are necessory because the arithmatic sum of two bits range from 0 to 3. and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S for

sum and C for carr. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full adder is listed in Table 4.4.

Table 4.4
Full Adder

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The eight rows under the input variables designate all possible combinations of the three variables. The output variables are determined from he arithmatic sum of the input bits.

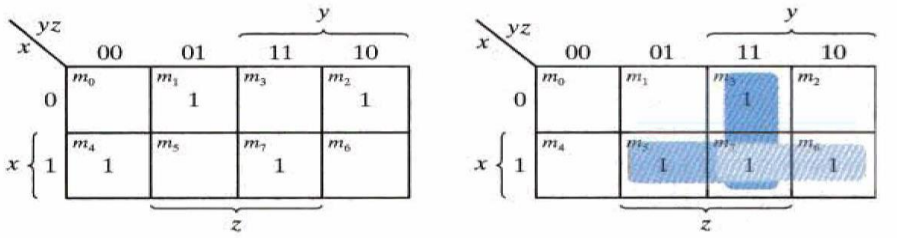


FIGURE 4.6
Maps for full adder

$$S = x'y'z + x'yz' + xy'z' + xyz$$
$$C = xy + xz + yz$$

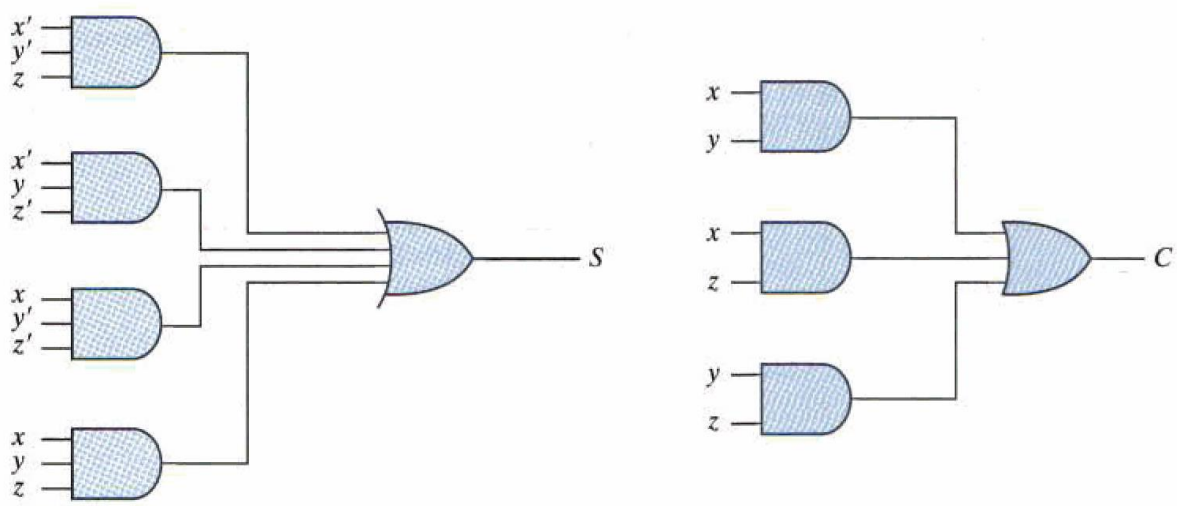


FIGURE 4.7
Implementation of full adder in sum-of-products form

$$
\begin{aligned}
S &= z \oplus (x \oplus y) \\
&= z'(xy' + x'y) + z(xy' + x'y)' \\
&= z'(xy' + x'y) + z(xy + x'y') \\
&= xy'z' + x'yz' + xyz + x'y'z
\end{aligned}
$$

$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$

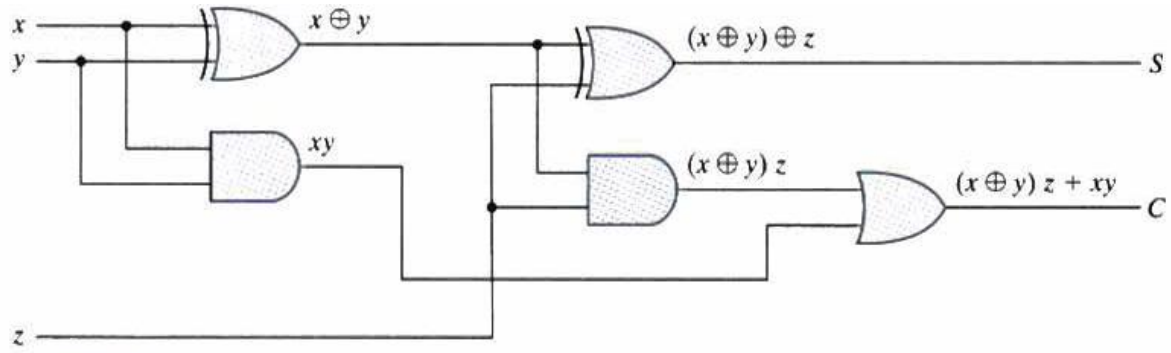


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

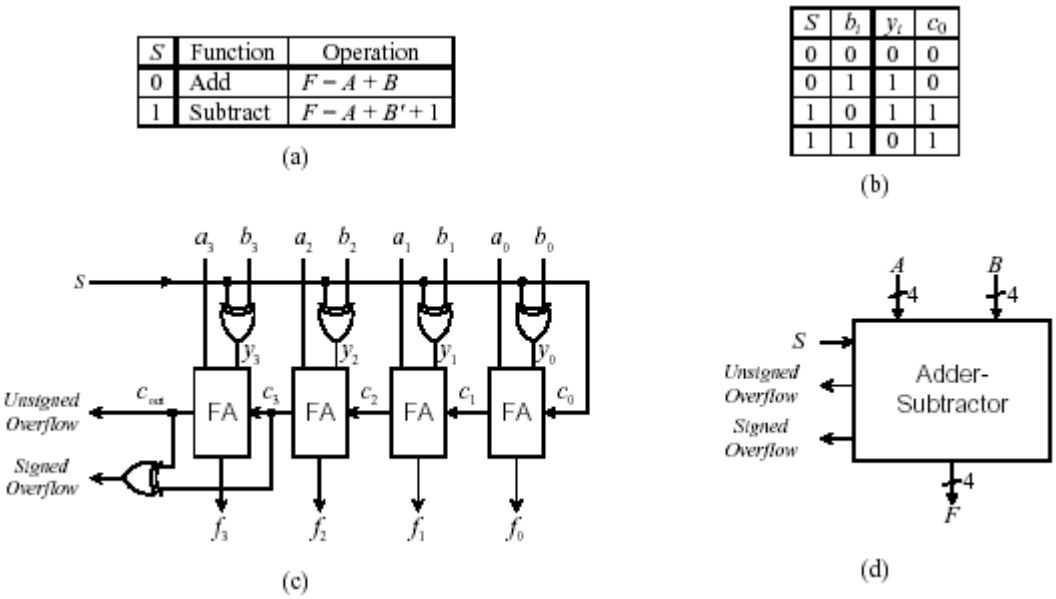
4.5 Adder-Subtractor Combination

It turns out that instead of having to build a separate adder and subtractor units, we can modify the ripple-carry adder (or the carry-lookahead adder) slightly to perform both operations. The modified circuit performs subtraction by adding the negated value of the second operand. In other words, instead of performing the subtraction $A - B$, the addition operation $A + (-B)$ is performed. Recall that in two's complement representation, to negate a value involves inverting all the bits from 0 to 1 or vice versa, and then adding a 1. Hence, we need to modify the adder circuit so that we can selectively do either one of two things: 1) flip the bits of the B operand, and then add an extra 1 for the subtraction operation, or 2) not flip the

bits, and not add an extra 1 for the addition operation.

For this adder-subtractor combination circuit, in addition to the two input operands A and B, a select signal S is needed to select which operation to perform. The assignment of the two operations to the select signal S is shown in Figure 4.8 (a). When $S = 0$, we want to perform an addition, and when $S = 1$, we want to perform a subtraction. When $S = 0$, B does not need to be modified, and like the adder circuit from Section 4.2.2, the initial carry-in signal c_0 needs to be set to a 0. On the other hand, when $S = 1$, we need to invert the bits in B and add a 1. The addition of a 1 is accomplished by setting the initial carry-in signal c_0 to a 1. Two circuits are needed for handling the above situations: one for inverting the bits in B, and one for setting c_0. Both of these circuits are dependent on S.

The truth table for these two circuits is shown in Figure 4.8 (b). The input variable b_i is the *i*th bit of the *B* operand. The output variable y_i is the output from the circuit that either inverts or not inverts the bits in *B*. From this truth table, we can conclude that the circuit for y_i is just a 2-input XOR gate, while

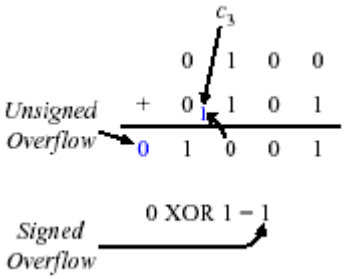


the circuit for *c*0 is just a direct connection from *S*. Putting everything together, we obtain the adder-subtractor combination circuit for four bits as shown in Figure 4.8 (c). The logic symbol for the circuit is shown in Figure 4.8 (d).

Figure 4.8. Adder-subtractor combination: (a) operation table; (b) truth table for *yi* and *c*0; (c) circuit; (d) logic symbol.

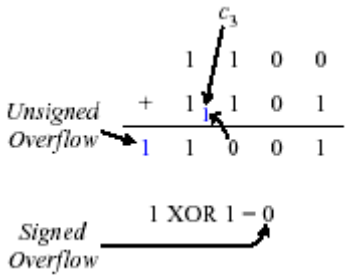
Notice the adder-subtractor circuit in Figure 4.8 (c) has two different overflow signals, *Unsigned_Overflow* and *Signed_Overflow*. This is because the circuit can deal with both signed and unsigned numbers. When working with unsigned numbers only, the output signal *Unsigned_Overflow* is sufficient to determine whether there is an overflow or not. However, for signed numbers, we need to perform the XOR of *Unsigned_Overflow* with c_3 producing the *Signed_Overflow* signal in order to determine whether there is an overflow or not. For example, the valid range for a 4-bit *signed* number goes from –23 to 23–1, i.e., from – 8 to 7. Adding the two signed numbers 4 + 5 = 9 should result in a signed number overflow since 9 is outside the range. However, the valid

range for a 4-bit *unsigned* number goes from 0 to 24–1, i.e., 0 to 15. If we treat the two numbers 4 and 5 as unsigned numbers, then the result of adding these two unsigned numbers, 9, is inside the range. So when adding the Chapter 4 Combinational Components Page 12 of 46 two numbers 4 and 5, the *Unsigned_Overflow* signal should be de-asserted, while the *Signed_Overflow* signal should be asserted. Performing the addition of 4 + 5 in binary as shown below



we get 0100 + 0101 = 1001, which produces a 0 for the *Unsigned_Overflow* signal. However, the addition produces a 1 for c_3, and XORing these two values, 0 for *Unsigned_Overflow* and 1 for c_3, results in a 1 for the *Signed_Overflow* signal.

In another example, adding the two 4-bit signed numbers – 4 + (– 3) = – 7 should not result in a signed overflow. Performing the arithmetic in binary, –4



= 1100 and –3 = 1101, as shown below

we get 1100 + 1101 = 11001, which produces a 1 for both *Unsigned_Overflow* and c_3. XORing these two values together gives a 0 for the *Signed_Overflow* signal. On the other hand, if we treat the two binary numbers, 1100 and 1101, as unsigned numbers, then we are adding 12 + 13 = 25. 25 is outside the

unsigned number range, and so the *Unsigned_Overflow* signal should be asserted.

4.4 Decimal Adder

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an output carry.

Suppose we apply two BCD digits to a four-bit binmy adder. The adder will form the sum in binary and produce a result that ranges from 0 through 19. These binary numbers are listed in Table 4.5 and are labled by symbols K, Z_8, Z_4, Z_2, and Z_1, K is the carry, and the subscripts under the letter Z represent the weights 8,4,2, and 1 that can be assigned to the four bits in the BCD code. The columns under the binary sum list the binary value that appears in the outputs of the four-bit binary adder. The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum." The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum.

Table 4.5
Derivation of BCD Adder

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

$$C = K + Z_8Z_4 + Z_8Z_2$$

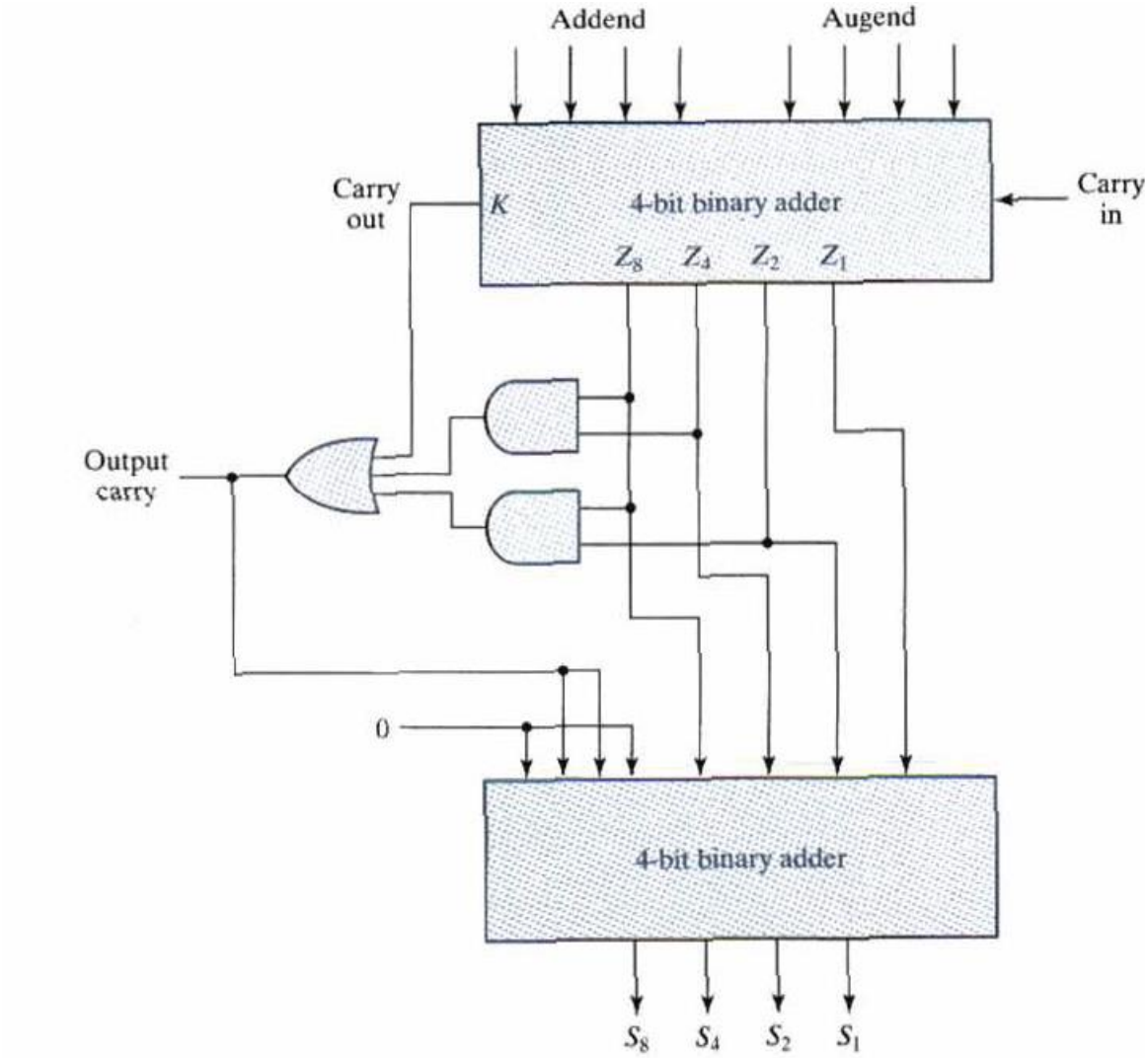


FIGURE 4.14
Block diagram of a BCD adder

4.5 Multiplier

Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the least significant bit. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

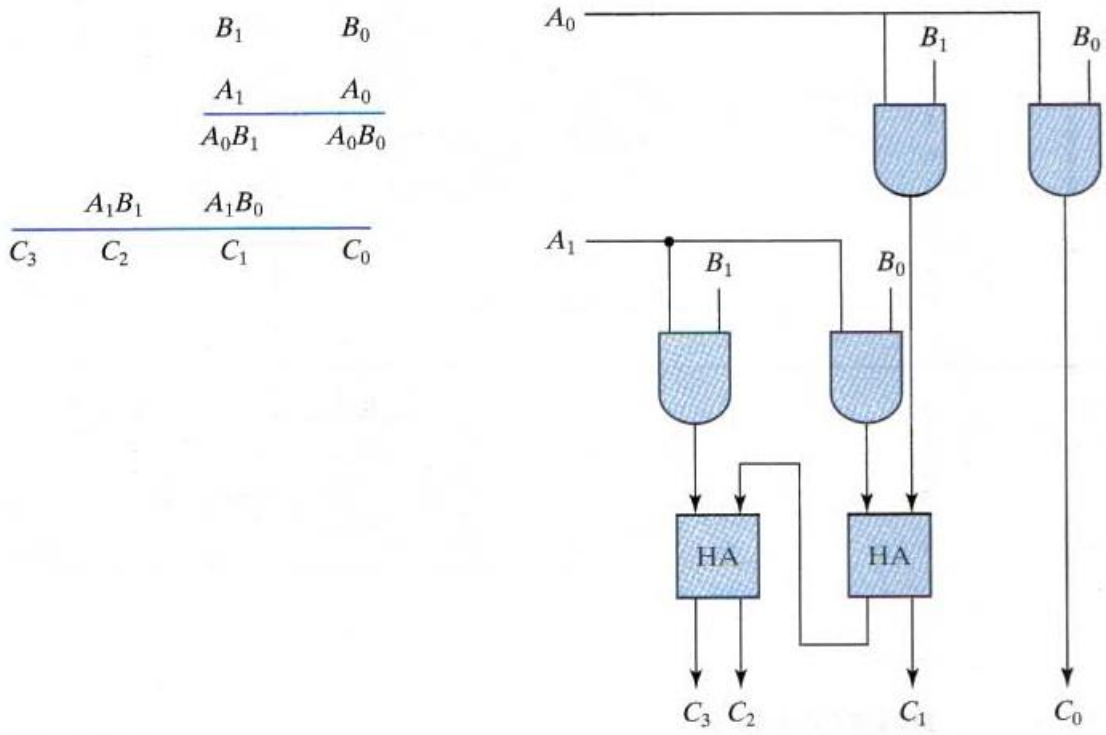


FIGURE 4.15
Two-bit by two-bit binary multiplier

As a second example. consider a multiplier circuit mat multiplies a binary number represented by four bits by a number represnted by three bits. Let the multiplicand be represented by B B B B and the muliplier by A A A A . Since k= 4 and J = 3. We need 12 AND gates and 2 four-bit adder to produce the product of seven bits. The logic diagram of the multiplier s shown in Fig. below

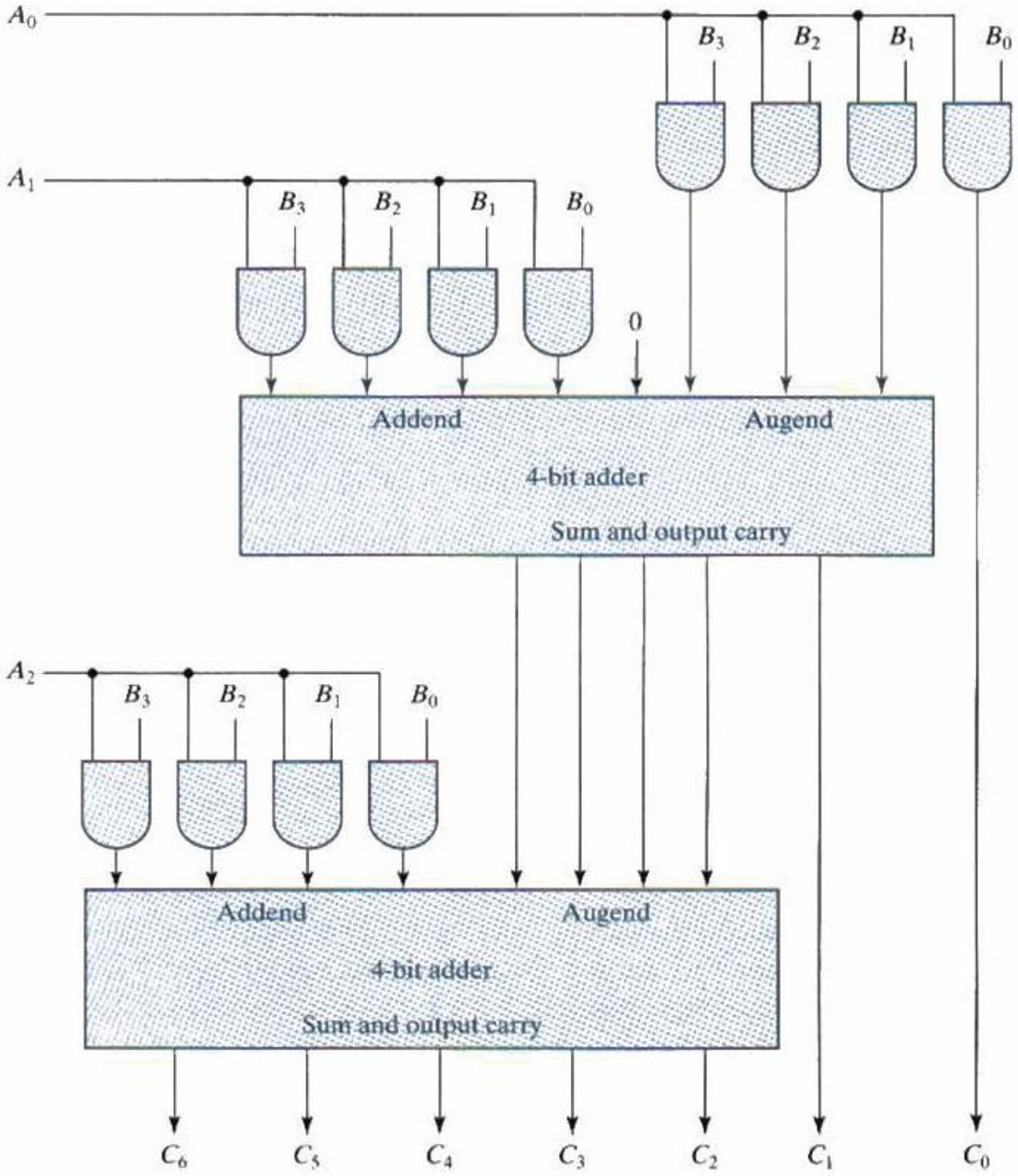


FIGURE 4.16
Four-bit by three-bit binary multiplier

4.5 Magnitude Comparator

The comparison of two numbers is an operation that determined whether one number is greater than, less than, or equel to the other number. A comparator is a combinational circuit that compares two numbers A and B and detemims their relative magnitude. The outcome of the comparison is spacified by three binary variables that indicate whether A > B, A = B, or A < B.

$$A = A_3A_2A_1A_0$$
$$B = B_3B_2B_1B_0$$

$$x_i = A_iB_i + A_i'B_i' \quad \text{for } i = 0, 1, 2, 3$$

$$(A = B) = x_3x_2x_1x_0$$

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$
$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1' + x_3x_2x_1A_0'B_0'$$

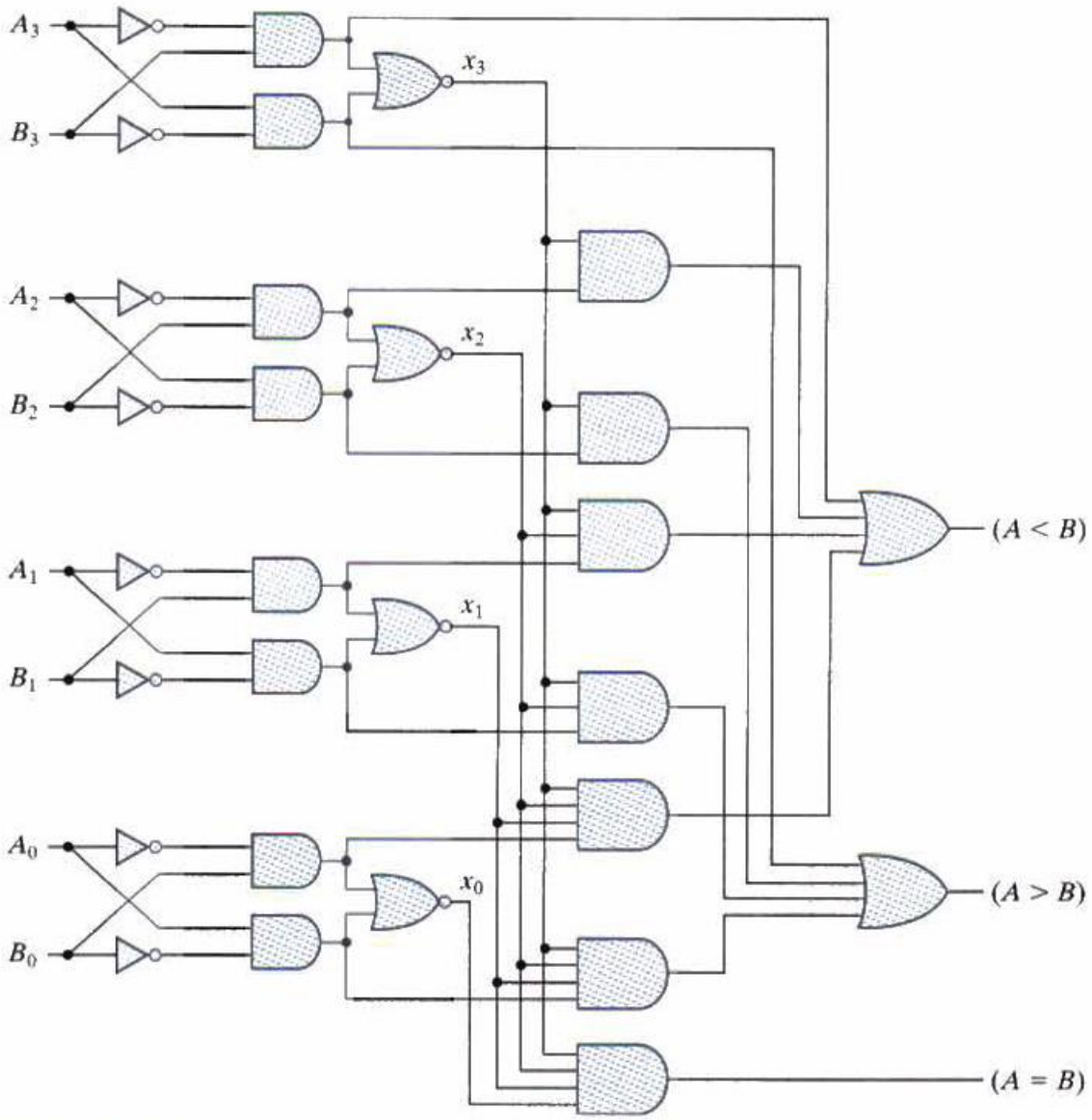


FIGURE 4.17
Four-bit magnitude comparator

4.7 Decoder

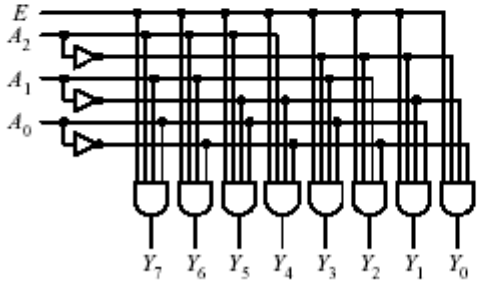
A **decoder**, also known as a **demultiplexer**, asserts one out of *n* output lines depending on the value of an *m*-bit binary input data. In general, an *m*-to-*n* decoder has *m* input lines, A_{m-1}, …, A_0, and *n* output lines, Y_{n-1}, …, Y_0, where $n = 2m$. In addition, it has an enable line *E* for enabling the decoder. When the decoder is disabled with *E* set to 0, all the output lines are de-asserted. When the decoder is enabled, then the output line whose index is equal to the value

of the input binary data is asserted. For example, for a 3-to-8 decoder, if the input address is 101, then the output line Y_5 is asserted (set to 1 for active high) while the rest of the output lines are de-asserted (set to 0 for active high).

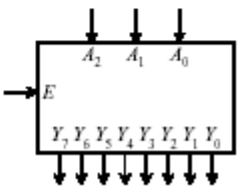
A decoder is used in a system having multiple components, and we want only one component to be selected or enabled at any one time. For example, in a large memory system with multiple memory chips, only one memory chip is enabled at a time. One output line from the decoder is connected to the enable input on each memory chip. Thus, an address presented to the decoder will enable that corresponding memory chip. The truth table, circuit, and logic symbol for a 3-to-8 decoder are shown in Figure 4.15. A larger size decoder can be implemented using several smaller decoders. For example, Figure 4.16 uses seven 1-to-2 decoders to implement a 3-to-8 decoder. The correct operation of this circuit is left as an exercise for the reader.

E	A_2	A_1	A_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	×	×	×	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

(a)



(b)



(c)

Figure 4.15. A 3-to-8 decoder: (a) truth table; (b) circuit; (c) logic symbol.

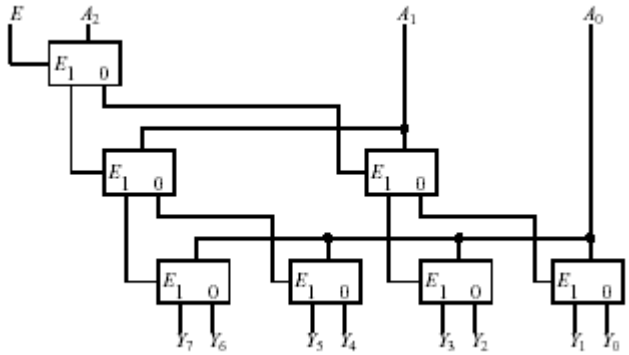


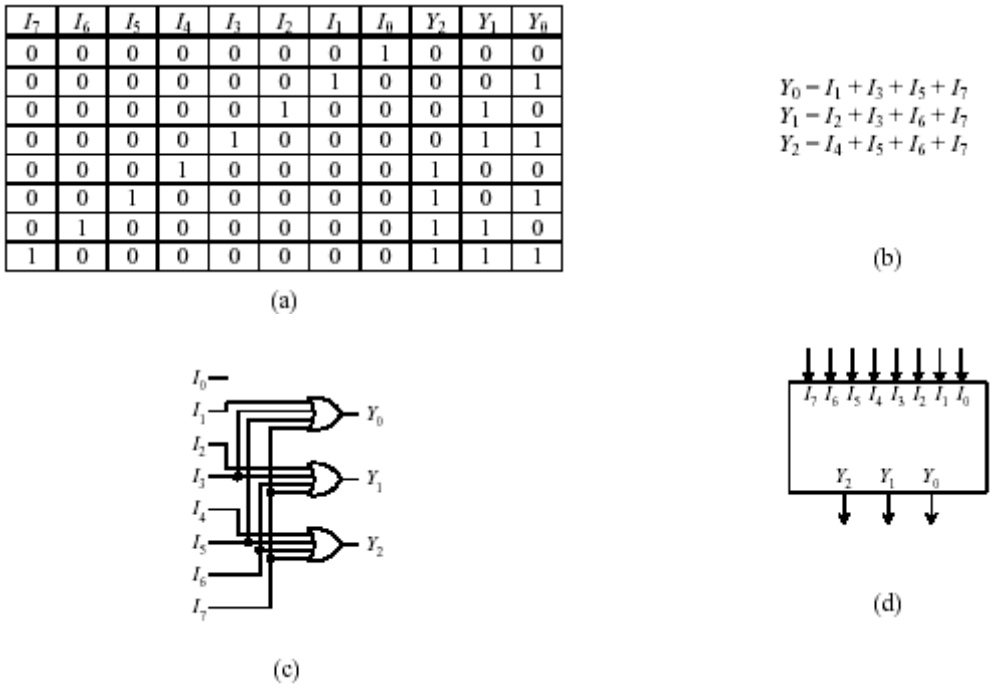
Figure 4.16. A 3-to-8 decoder implemented with seven 1-to-2 decoders

4.8 Encoder

An **encoder** is almost like the inverse of a decoder where it encodes a 2*n*-bit input data into an *n*-bit code. The encoder has 2*n* input lines and *n* output lines as shown by the logic symbol in Figure 4.18 (c) for $n = 3$. The operation of the encoder is such that exactly one of the input lines should have a 1 while the remaining input lines should have a 0. The output is the binary value of the input line index that has the 1. The truth table for an 8-to-3 encoder is shown in Figure 4.18 (a). For example, when input I_3 is a 1, the three output bits Y_2, Y_1, and Y_0, are set to 011, which is the binary number for the index 3. Entries having multiple 1's in the truth table inputs are ignored since we are assuming that only one input line can be a 1. Looking at the three output columns in the truth table, we obtain the three equations shown in Figure 4.18 (b), and the resulting circuit in (c). The logic symbol is shown in (d).

Encoders are used to reduce the number of bits needed to represent some given data either in data storage or in data transmission. Encoders are also used in a system with 2*n* input devices, each of which may need to request for service. One input line is connected to one input device. The input device requesting for service will assert the input line that is connected to it. The corresponding *n*-bit output value will indicate to the system which of the 2*n* devices is requesting for service. For example, if device 5 requests for service, it will assert the I_5 input line. The system will know that device 5 is requesting for service since the output will be 101 = 5. However, this only works correctly if it is guaranteed that only one of the 2*n* devices will request for service at any one time.

If two or more devices request for service at the same time, then the output will be incorrect. For example, if devices 1 and 4 of the 8-to-3 encoder request for service at the same time, then the output will also be 101 because I_4 will assert the Y_2 signal, and I_1 will assert the $Y0$ signal. To resolve this problem, a priority is assigned to each of the input lines so that when multiple requests are made, the encoder outputs the index value of the input line with the highest priority. This modified encoder is known as a **priority encoder**.



I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

(a)

$$Y_0 = I_1 + I_3 + I_5 + I_7$$
$$Y_1 = I_2 + I_3 + I_6 + I_7$$
$$Y_2 = I_4 + I_5 + I_6 + I_7$$

(b)

(c)

(d)

Figure 4.18. An 8-to-3 encoder: (a) truth table; (b) equations; (c) circuit; (d) logic symbol.

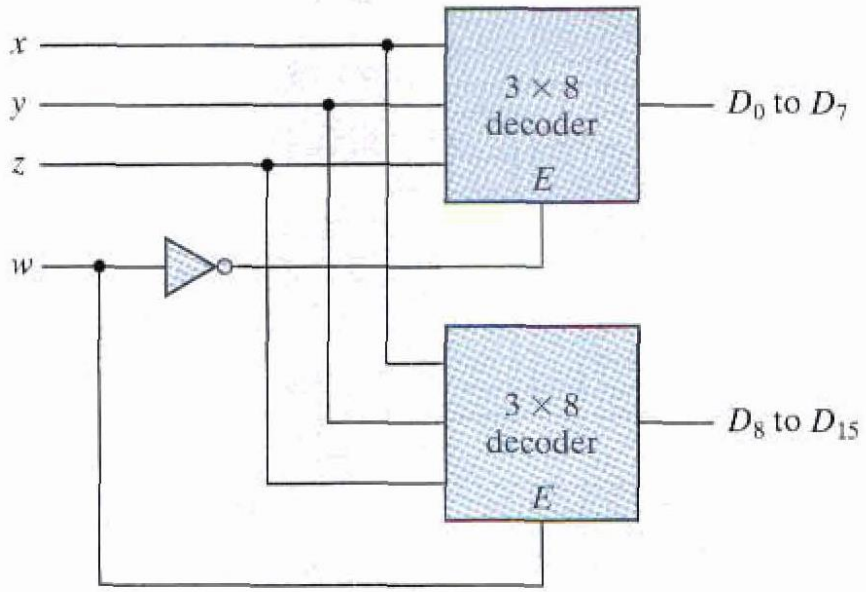


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

Example

Design Full Adder using 3X8 decoder

Sol.

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$
$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

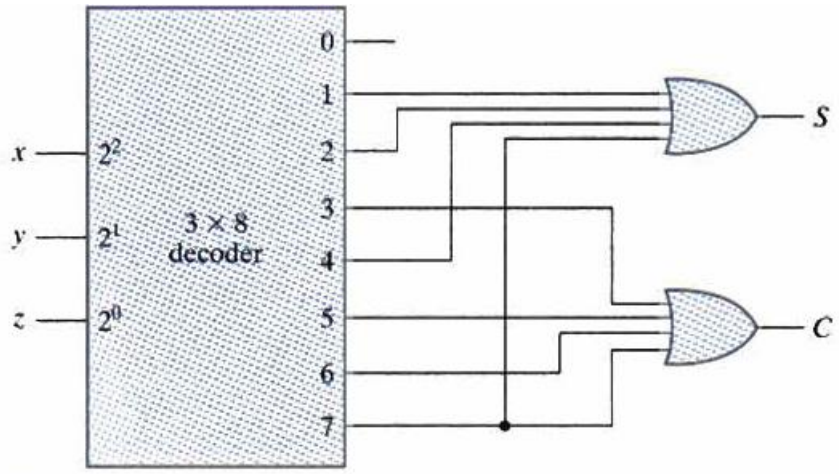


FIGURE 4.21
Implementation of a full adder with a decoder

4.8 Encoder

An encoder is a digital circuit that outperforms the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value, An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 4,7. It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

Table 4.7
Truth Table of an Octal-to-Binary Encoder

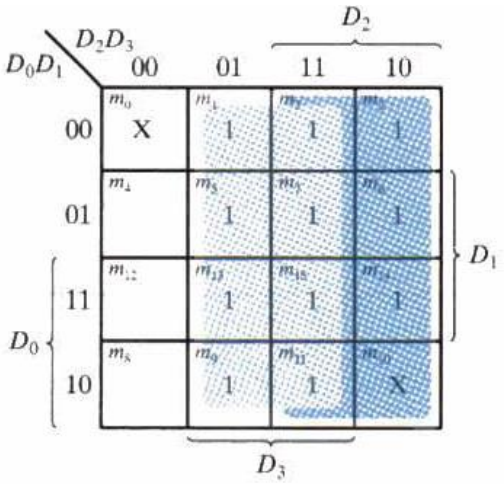
Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$z = D_1 + D_3 + D_5 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$x = D_4 + D_5 + D_6 + D_7$$

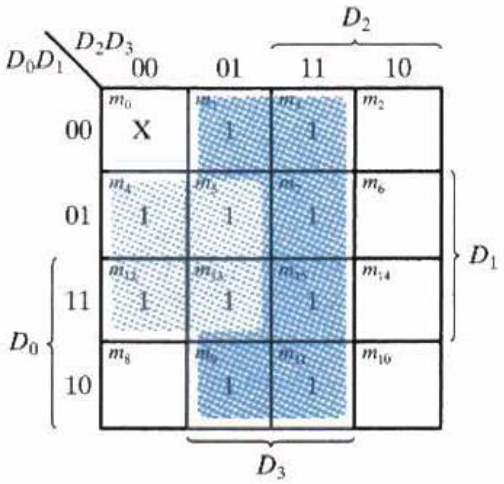
4.8 Periority encoder

Table 4.8
Truth Table of a Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1



$x = D_2 + D_3$



$y = D_3 + D_1D'_2$

FIGURE 4.22
Maps for a priority encoder

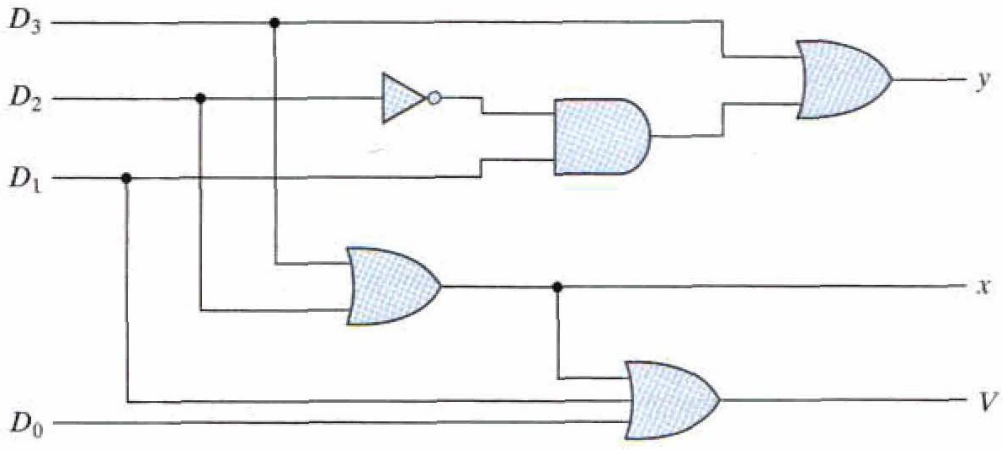


FIGURE 4.23
Four-input priority encoder

$$x = D_2 + D_3$$

$$y = D_3 + D_1 D_2'$$

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

4.9 Multiplixer

A multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

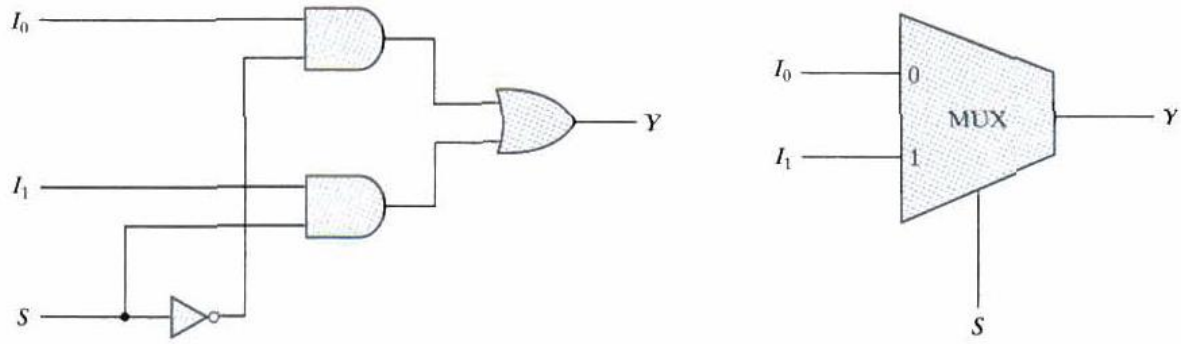
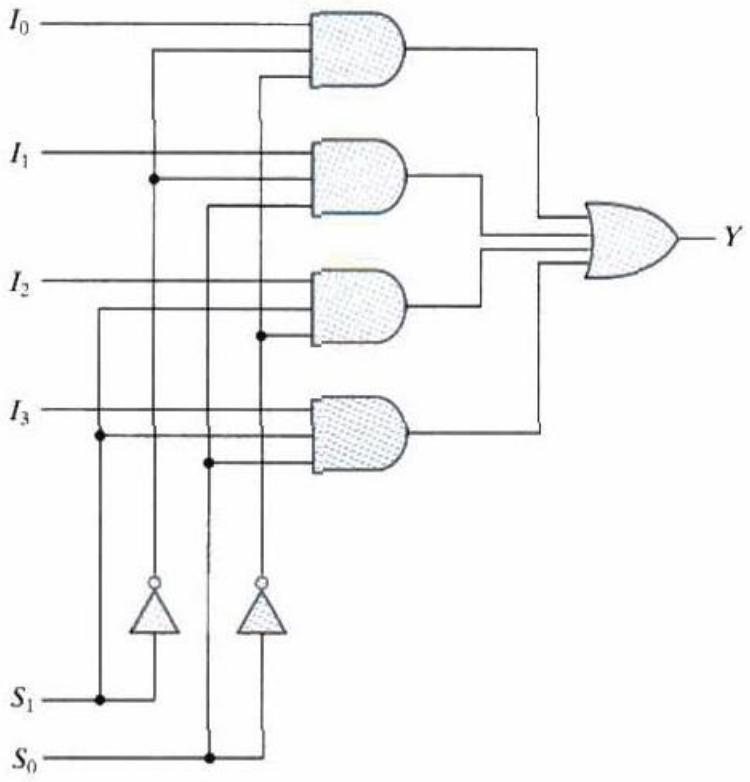


FIGURE 4.24
Two-to-one-line multiplexer



(a) Logic diagram

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

(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

Example:

Implement the following function using 4x1 Mux.

F=∑(1,2,6,7)

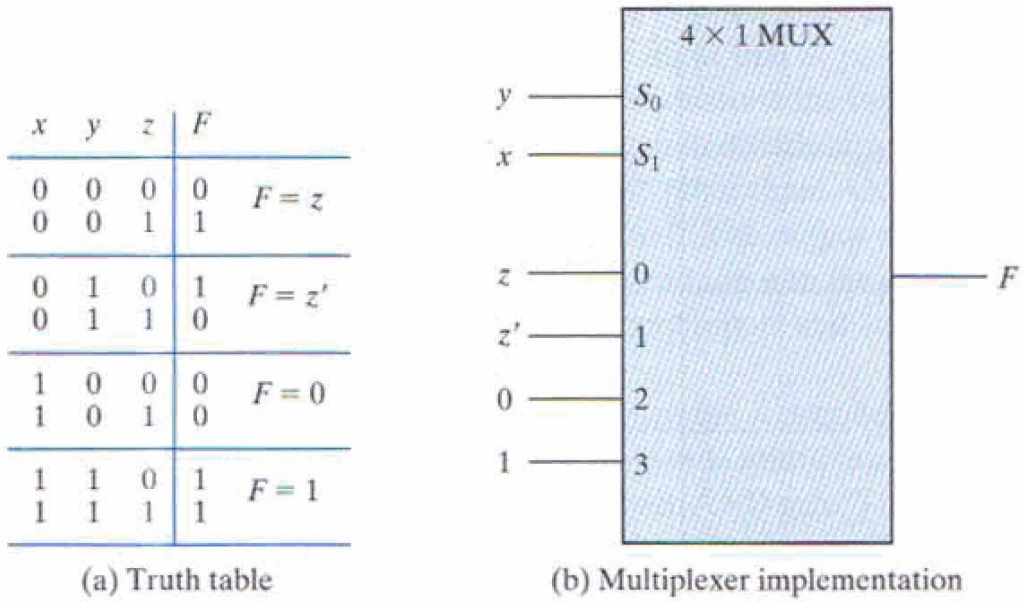


FIGURE 4.27
Implementing a Boolean function with a multiplexer

Chapter Five

Synchronous Sequential Logic

5.1 Introduction

The digital circuits considered thus far have been combinational; that is, the outputs are entirely dependent on the current inputs. Although every digital system is likely to have some combinational circuits, most systems encountered in practice also include storage elements, which required that the system be considered in terms of ***sequential logic***.

5.2 Sequential Circuits

A block diagram of sequential circuit is shown below

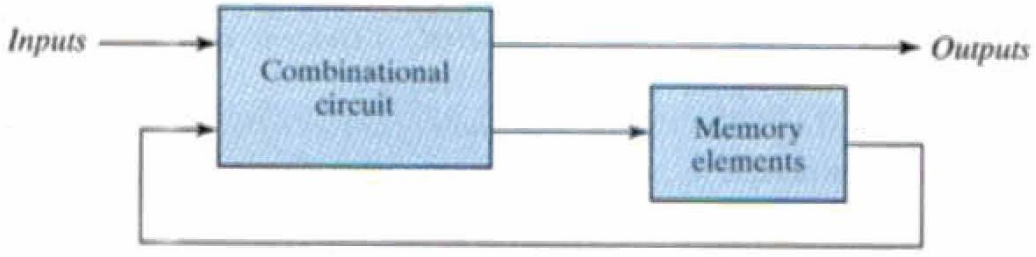


FIGURE 5.1
Block diagram of sequential circuit

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A ***synchronous*** sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an ***asynchronous*** sequential circuit depends upon the input signals at any instant of time and the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. The block diagram of synchronous clocked sequential circuit is shown:

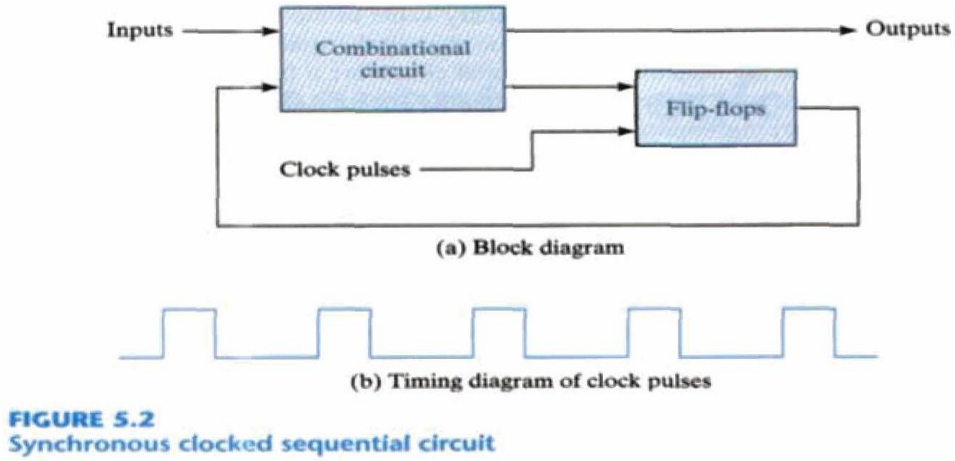


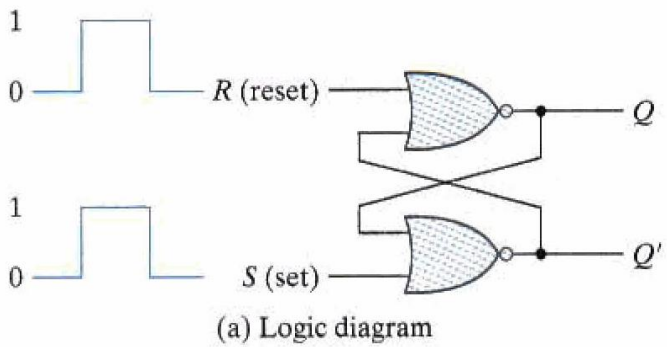
FIGURE 5.2
Synchronous clocked sequential circuit

The ***outputs*** are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both).

5.3 Storage Elements: Latches

SR Latch

The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled S for set and R for reset. The SR latch constructed with two cross coupled NOR gates is shown



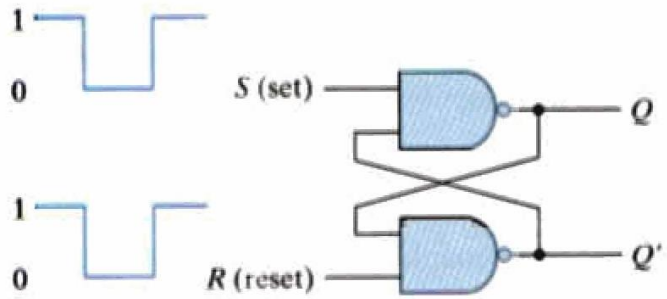
(a) Logic diagram

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

FIGURE 5.3
SR latch with NOR gates

The SR latch with two cross-coupled NAND gates is shown



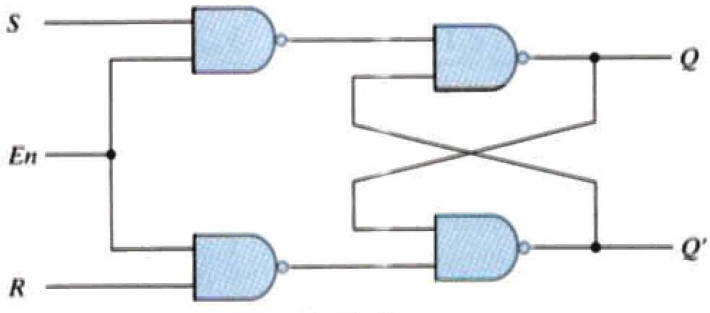
(a) Logic diagram

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table

FIGURE 5.4
SR latch with NAND gates

The operation of basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of latch can be changed. An SR latch with a control input is shown



(a) Logic diagram

En	S	R	Next state of Q
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

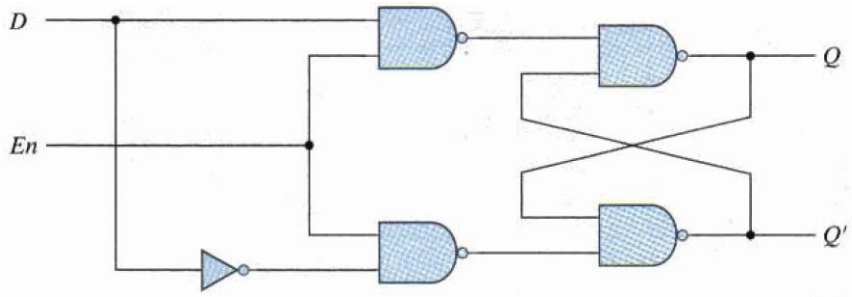
(b) Function table

FIGURE 5.5
SR latch with control input

D latch

One way to eliminate the undesirable condition of the indeterminate state in the

SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown below



(a) Logic diagram

En	D	Next state of Q
0	X	No change
1	0	Q = 0; reset state
1	1	Q = 1; set state

(b) Function table

FIGURE 5.6
D latch

The graphic symbols for the various latches are shown below:

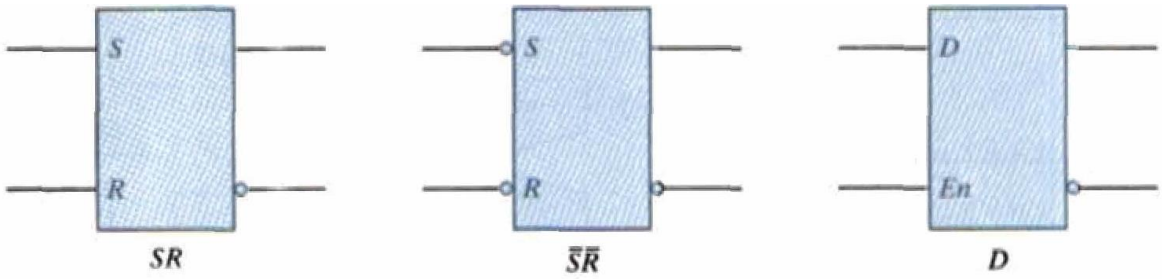


FIGURE 5.7
Graphic symbols for latches

5.4 Storgae Elements: Flip-Flops

The state of a latch or flip-flop is switched by a change in the control input. This momnetry change is called a trigger, and the transition it causes is said to trigger the flip-flop.

Flip-flop cicuits are constructed in such a way as to make them operate properly when they are part of a sequential cicuit that employs a common clock. The problem with the latch is that it responds to a change in the level of a clock pulse. As shwon below, apositive level response in the enable input allows changes in the output when the D input changes while clock pulse stay at logic 1.

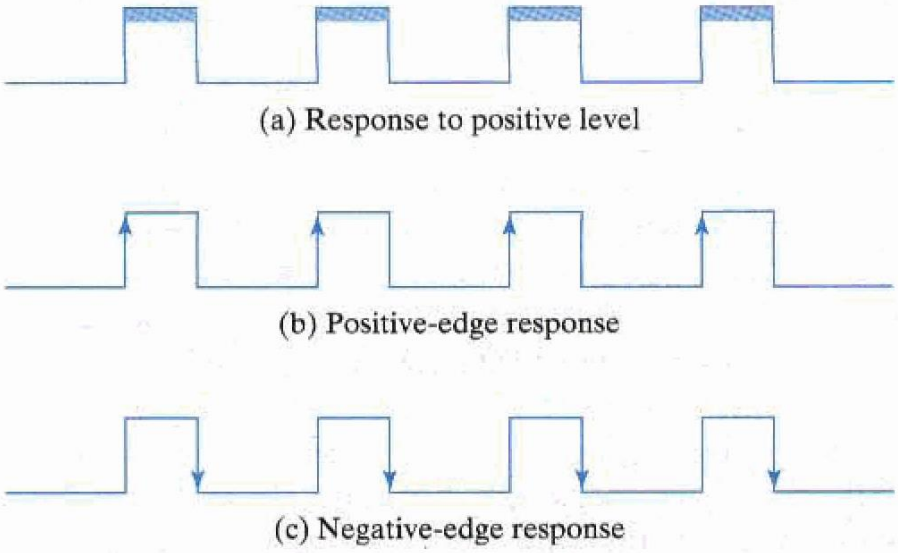


FIGURE 5.8
Clock response in latch and flip-flop

Egded Trigger D Flip Flop

The construction of a D flip-flop with two D latches and an inverter is shown below. The first latch called the ***master*** and the second the ***slave***.

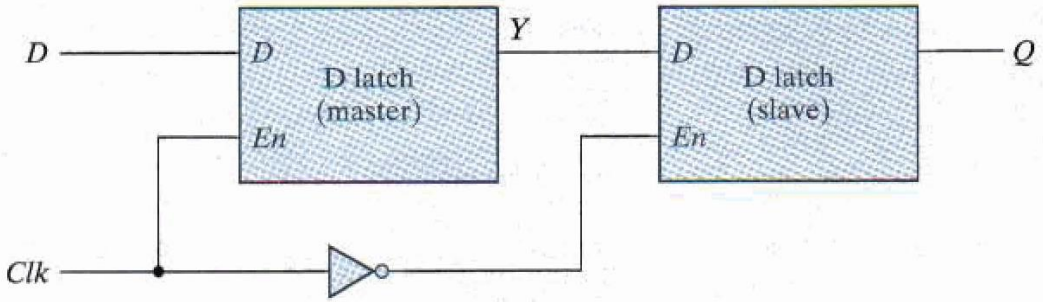
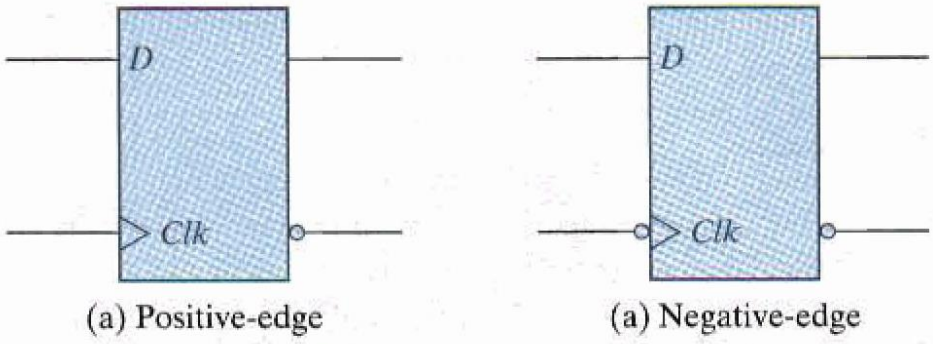


FIGURE 5.9
Master–slave *D* flip-flop

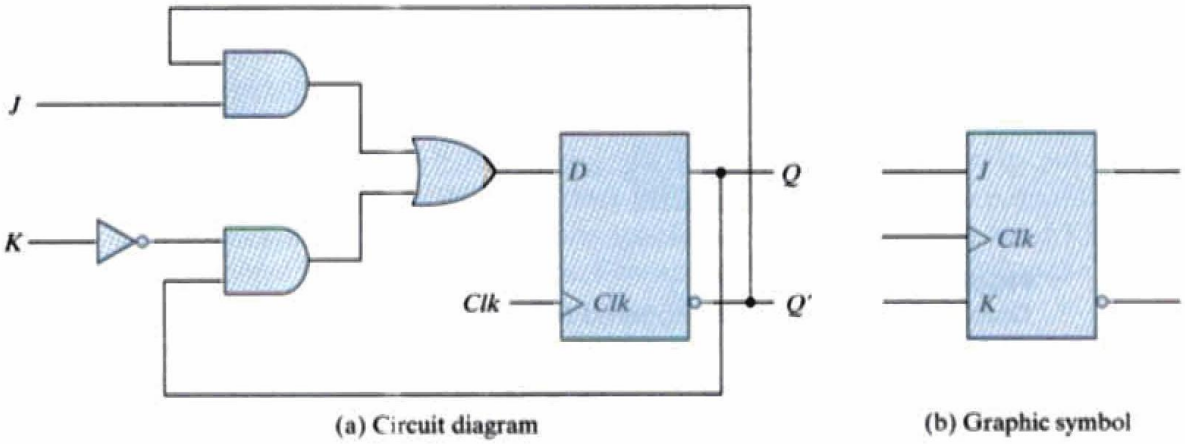


(a) Positive-edge (a) Negative-edge

FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

Other Flip Flops

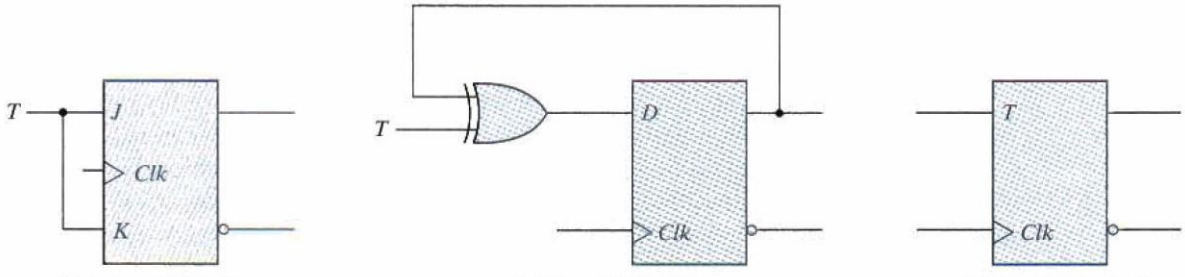
Two flip-flops less widely used in the design of digital systems are the JK and T flip-flops. The circuit diagram of a JK flip-flop constructed with a D flip-flop and gates is shown below:



(a) Circuit diagram (b) Graphic symbol

FIGURE 5.12
***JK* flip-flop**

The T (toggle) flip-flop is a complementing flip-flop and can be obtianed from a JK flip-flop when input J and K are tied together. This is shown below:



(a) From *JK* flip-flop (b) From *D* flip-flop (c) Graphic symbol

FIGURE 5.13
***T* flip-flop**

Characteristic tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in next table:

Table 5.1
Flip-Flop Characteristic Tables

JK Flip-Flop

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

D Flip-Flop

D	Q(t + 1)	
0	0	Reset
1	1	Set

T Flip-Flop

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

5.5 Analysis of Clocked Sequential Circuits

Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is detrmined from the inputs, the outputs, and the state of its flip-flops. The outputs and the next state are both a function of the inputs and the present state.

The behavior of a clocked sequential cicuit can be described algebraically by

means of state equetions. A *state equation* (also called a *transition equation*) specifies the next state as a function of the present state and inputs.

Example:

Consider the sequential cicuit shown below:

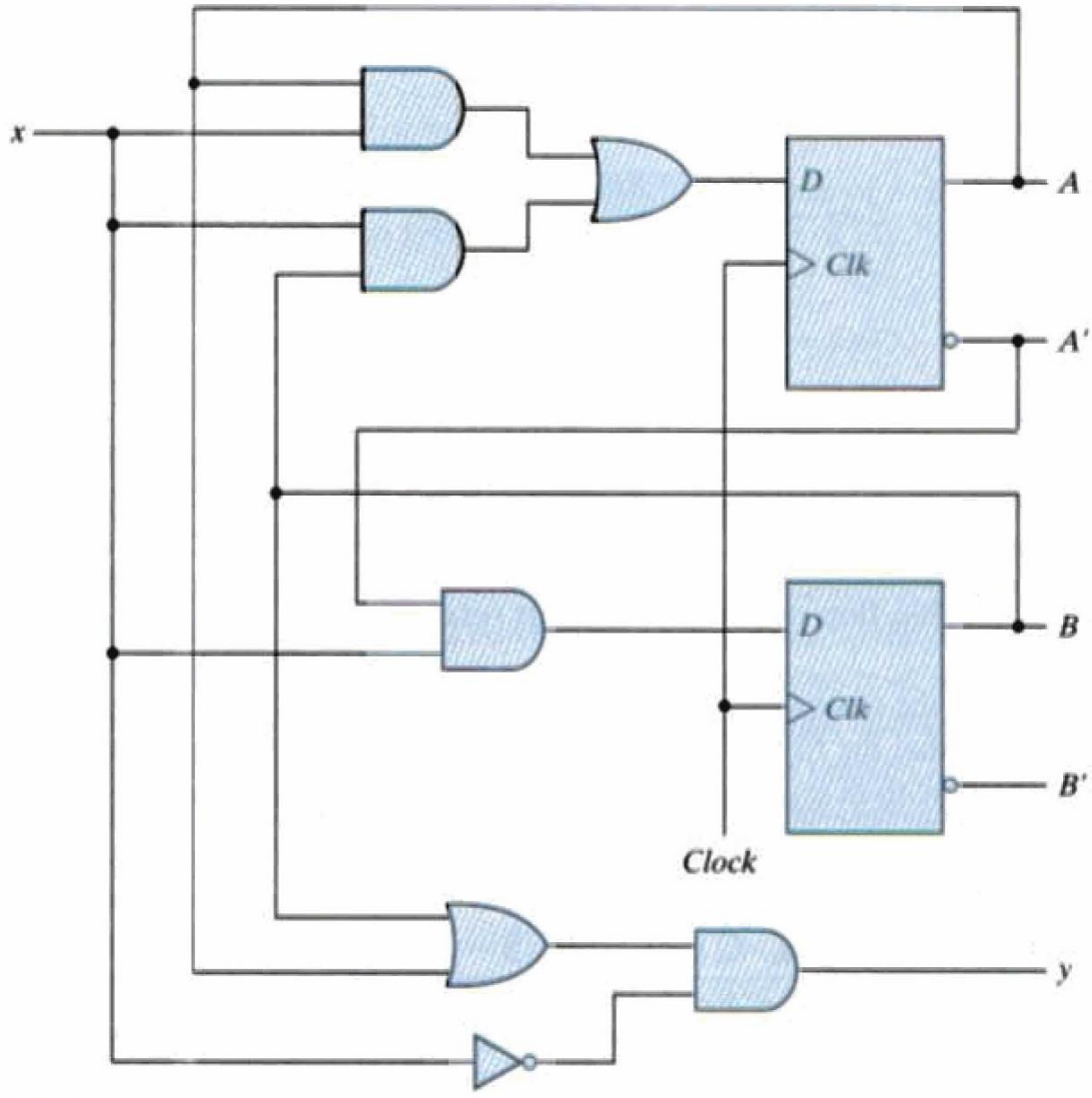


FIGURE 5.15
Example of sequential circuit

$$A(t+1) = A(t)x(t) + B(t)x(t)$$
$$B(t+1) = A'(t)x(t)$$

or

$$A(t+1) = Ax + Bx$$

$$B(t+1) = A'x$$

Then

$$y(t) = [A(t) + B(t)]x'(t)$$

Or

$$y = (A + B)x'$$

The sequence of inputs, outputs, and flip-flop states can be enumerated in a ***state table*** (sometimes called a ***transition table***). The state table for the pervious circuit is shown below:

Table 5.2
State Table for the Circuit of Fig. 5.15

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Or in other form like below

Table 5.3
Second Form of the State Table

Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

State Diagram

The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles. The state diagram of the example sequential circuit is shown below:

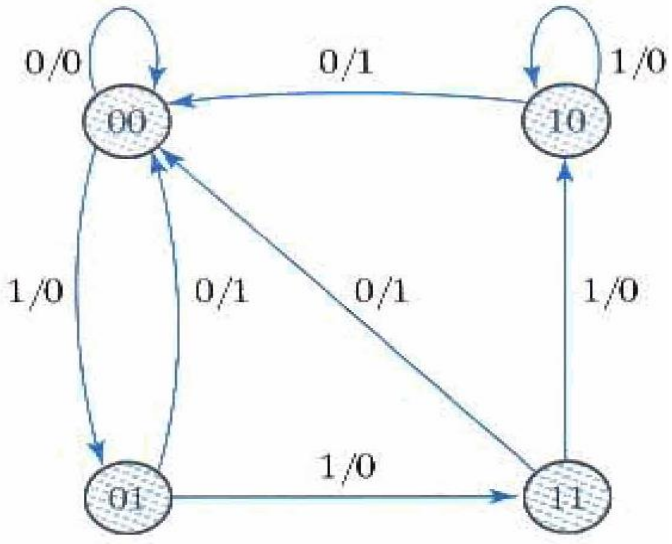
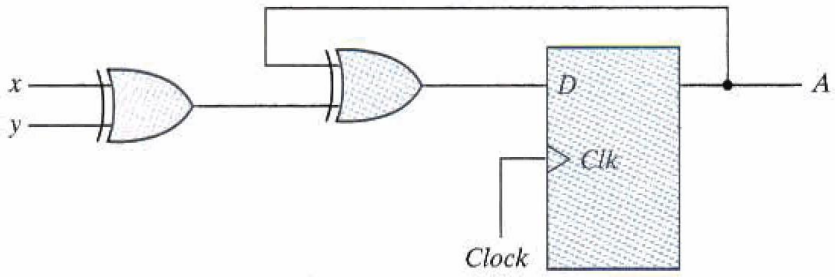


FIGURE 5.16
State diagram of the circuit of Fig. 5.15

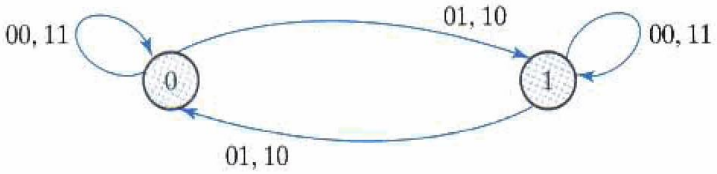
Example 2:



(a) Circuit diagram

Present state	Inputs		Next state
A	*x*	*y*	*A*
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

(b) State table



(c) State diagram

FIGURE 5.17
Sequential circuit with *D* flip-flop

Example 3:

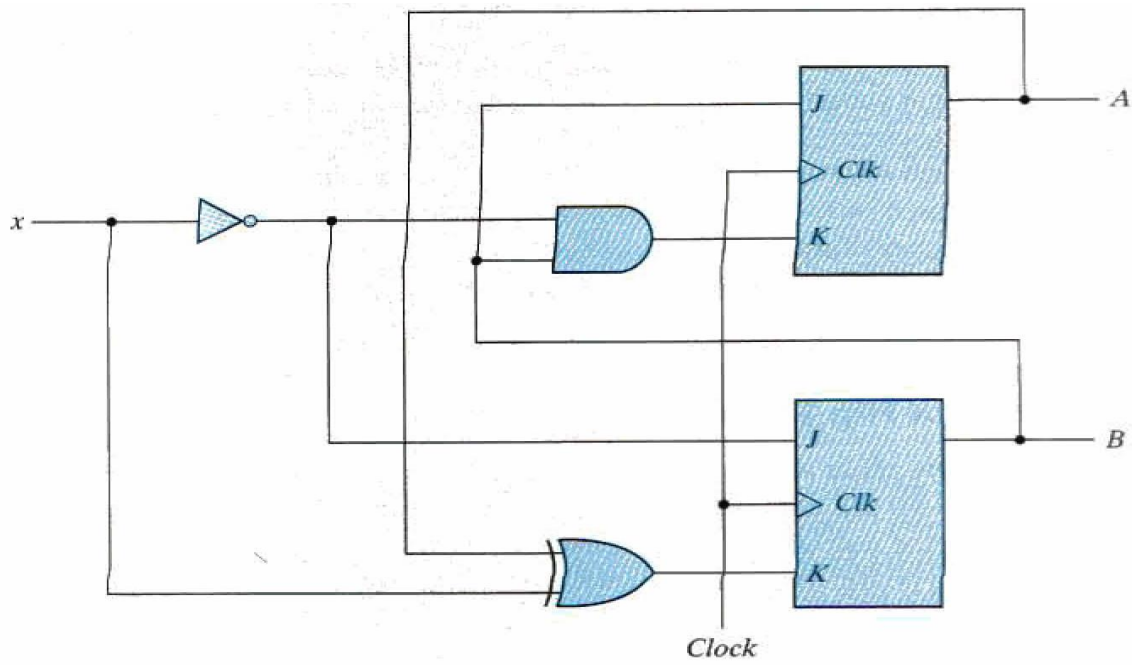
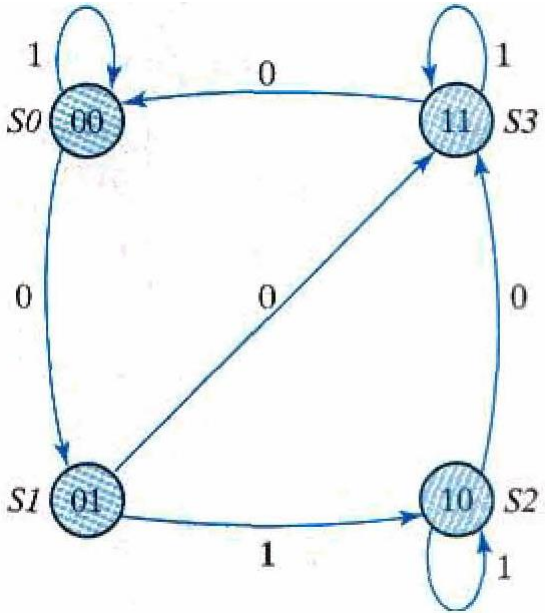


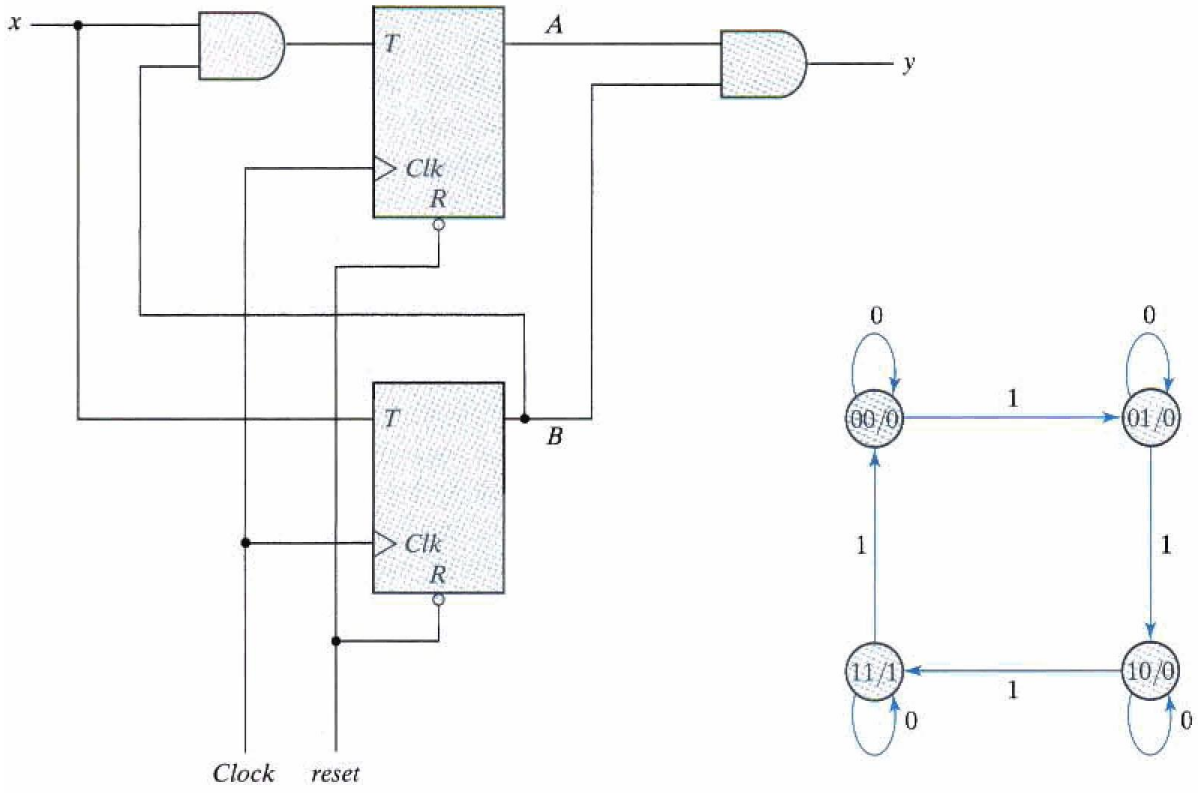
FIGURE 5.18
Sequential circuit with *JK* flip-flop

Table 5.4
State Table for Sequential Circuit with JK Flip-Flops

Present State		Input	Next State		Flip-Flop Inputs			
A	B	x	A	B	J_A	K_A	J_B	K_B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0



Example 4:



(a) Circuit diagram (b) State diagram

FIGURE 5.20
Sequential circuit with *T* flip-flops

Table 5.5
State Table for Sequential Circuit with T Flip-Flops

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

5.6 State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the *state reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the

external, input-output requirements unchanged.

Since ***m*** flip- flops produce 2^{m} states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates.

Example 1:

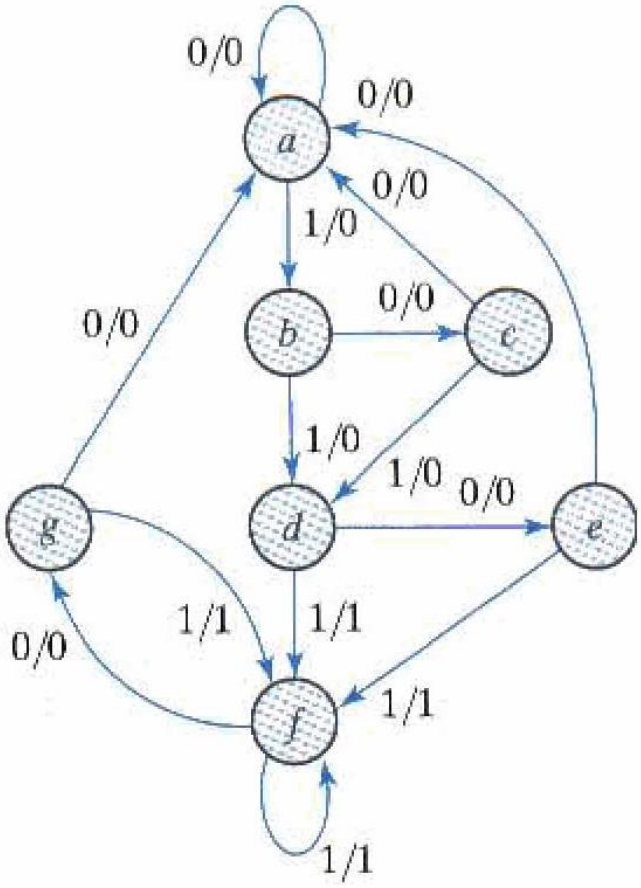


FIGURE 5.25
State diagram

As an example, consider the input sequence 01010110100 starting from the initial state a. Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state a, an input of 0 produce an output of 0 and the circuit remains in state a. With present state a and an output of 1, the output is 0 and the next state is b. With present state b and an input of 0, the output is 0 and the next state is

c. Continuing this process, we find the complete sequence to be as follows:

state	*a*	*a*	*b*	*c*	*d*	*e*	*f*	*f*	*g*	*f*	*g*	*a*
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

We now proceed to reduce the number of states for this example. First. we need the state table; it is more convenient to apply procedures for state reduction with the use of a table rather than a diagram. The state table of the circuit is listed in Table 5.6 and is obtained directly from the state diagram.

Table 5.6
State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	*a*	*b*	0	0
b	*c*	*d*	0	0
c	*a*	*d*	0	0
d	*e*	*f*	0	1
e	*a*	*f*	0	1
f	*g*	*f*	0	1
g	*a*	*f*	0	1

The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7.

Table 5.7
Reducing the State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f	0	1

The same next states and outputs appear in the row with present state ***d***. Therefore, states ***f*** and ***d*** are equivalent, and state ***f*** can be removed ***d*** replaced by ***d*** The final reduced table is shown in Table 5.8.

state	***a***	***a***	***b***	***c***	***d***	***e***	***d***	***d***	***e***	***d***	***e***	***a***
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

Table 5.8
Reduced State Table

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

The state diagram for the reduced table consists of only five states and is shown in Fig. 5.26

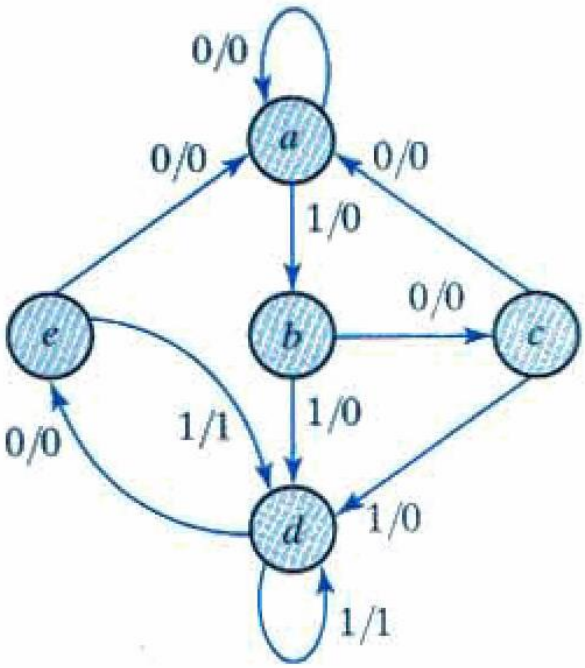


FIGURE 5.26
Reduced state diagram

5.7 State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m state the codes must contain n bits, where 2n>= m. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111.

The simplest way to code five states is to use the 6rst five integers in binary counting order, as shown in the first assignment of table 5.9. Another similar assignment is the Gray code shown in assignment 2.

Table 5.9
Three Possible Binary State Assignments

State	Assignment 1, Binary	Assignment 2, Gray Code	Assignment 3, One-Hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

Table 5.10
Reduced State Table with Binary Assignment 1

	Next State		Output	
Present State	$x = 0$	$x = 1$	$x = 0$	$x = 1$
000	000	001	0	0
001	010	011	0	0
010	000	011	0	0
011	100	011	0	1
100	000	011	0	1