Faculty of Engineering
Electrical Engineering Department
Communications and Electronics Program

Logic Circuit Design Class Notes EE242

SPRING 2015

LOGIC CIRCUIT DESIGN EE242

Term**:** SPRING 2015

Instructor: Dr. M. El-Banna, Room: EE-Building 4th floor

Classes : Group1 SUN 10:10 – 11:40

Group2 SUN 2:00 – 03:30

Topics to be covered:

Chapter 1: Implementation of Logic Functions

Chapter 2: Design of Combinational Circuits

Chapter 3: Sequential Circuits

Chapter 4: Design of Synchronous Sequential Circuits

Chapter 5: Introduction to Counters, Registers, and HDL language

Text Book: Fundamental Logic Design, Thomson, Charles Roth

Further Reading: Contemporary Digital Design, Johnson and Karim

<u>Grading Policy:</u>

Midterm	30
LAB	30
Final	90

ILOs of the Logic Circuit Design EE242

1. Comprehend and use the main blocks of combinational circuits, MUXs, ROMs, PLAs, PALs, Decoders and Encoders
2. Design combinational circuits using different blocks.
3. Carry out a project using the NI- LabVIEW software package to design and test combinational circuits, if time allows.
4. Differentiate between combinational and sequential circuits
5. Review all types of Flip Flops used in sequential circuits and represent their functions by state diagrams.
6. Convert verbally stated design problems into state diagrams and hence state tables.
7. Differentiate between Mealy and Moore finite state machines.
8. Follow up the design procedure of sequential circuits starting from implication tables through partition tables, state transition tables, excitations maps and eventually the hardware implementation.
9. Use the Verilog/VHDL language to design come known logic circuits, combinational and/or sequential
10. Comprehend and design synchronous and asynchronous counters

Implementation of Logic Functions

CHAPTER FOUR

4.1 Introduction

The design mechanism of combinational logic circuits is usually a multi-step process. The realization, and the subsequent minimization, of the logic function is not the end of the design. We are already familiar with the various schemes for coming up with the reduced logic function either in the SOP or in the POS format. These forms can be translated easily into either a familiar AND-OR or OR-AND pattern of logic circuits. However, we have also seen in the last chapter that digital ICs have several practical limitations that may affect the implementation of circuits. These include the fan-in and fan-out limitations and the fact that ICs are more frequently available in the NAND and NOR form than in the AND and OR form. NAND and NOR gates are easier to realize with electronic components and are, therefore, the basic ingredients used in all of the logic families. Consequently, it is important for the designer to be familiar with the techniques for translating the reduced function so that either NAND gates or NOR gates may be used.

Combinational circuits may be realized using a standardized combinational unit called a multiplexer (MUX). In the MUX some of the input variables are used as input selectors for the unit and the remaining variables are entered as data inputs. Two other devices, read-only memories (ROMs) and programmable logic arrays (PLAs), are also frequently used to implement combinational networks. This chapter will explore the possibilities of using only one type of gate or one of the modules—MUX, ROM, PAL, or PLA—for the realization of combinational circuits. Such exploration is extremely useful because most designers usually choose to use only one type of basic gate unless there is a particular reason for doing otherwise. After studying this chapter, you should be able to:

- Design combinational circuits using only NAND gates;

- ○ Design combinational circuits using only NOR gates;
- ○ Design combinational circuits using single- or multi-level multiplexers (MUXs);
- ○ Design combinational circuits using read-only memories (ROMs);
- ○ Design combinational circuits using programmable logic arrays (PLAs);
- ○ Construct a complex circuit using the outputs of a known functional unit.

4.2 Universal Logic Elements

In practice many logic circuits are built using only NAND and NOR gates because the basic gates in some of the logic families such as TTL and CMOS are NAND and NOR, respectively. NAND and NOR gates are considered universal logic elements since they both can be easily manipulated to obtain all possible logic functions. This simplification follows directly from Boolean theorems that we have discussed in the earlier chapters.

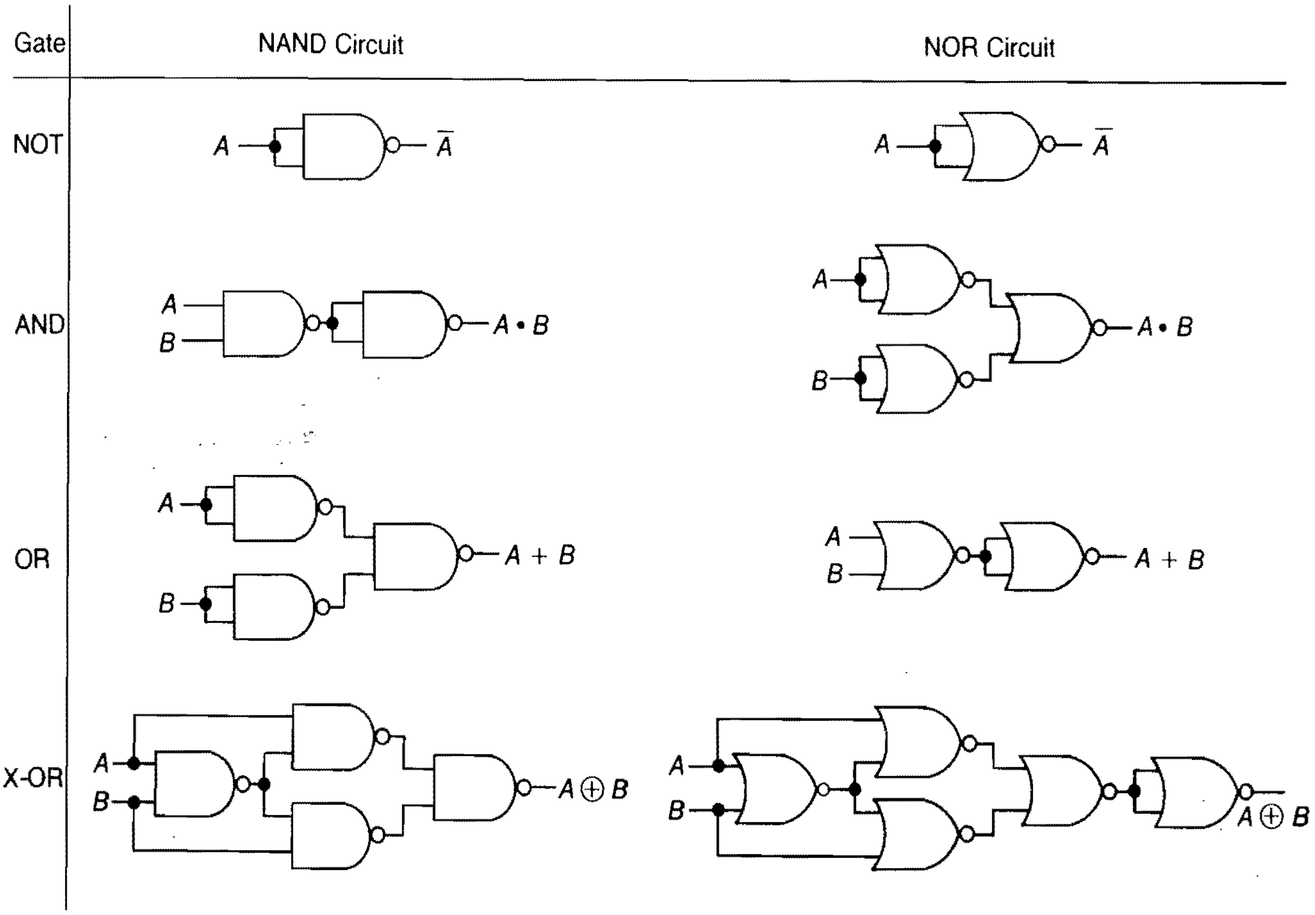
A close inspection of the truth table of these two functions, as described in Section 1.7, reveals that the NAND and NOR operators are duals of each other. Recall also from Chapter 1 that the dual of a Boolean expression is obtained by replacing every OR with AND, AND with OR, 0 with 1, and 1 with 0. Six of these dual properties are listed as follows:

NAND	NOR
1. $\overline{a \cdot 0} = 1$	$\overline{a + 1} = 0$
2. $\overline{a \cdot 1} = \bar{a}$	$\overline{a + 0} = \bar{a}$
3. $\overline{a \cdot a} = \bar{a}$	$\overline{a + a} = \bar{a}$
4. $\overline{a \cdot b} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \cdot \bar{b}$
5. $\overline{\bar{a} \cdot \bar{b}} = a + b$	$\overline{\bar{a} + \bar{b}} = a \cdot b$
6. $\overline{\overline{a \cdot b}} = a \cdot b$	$\overline{\overline{a + b}} = a + b$

All of the logic functions may be generated using these properties of NAND and NOR logic. The corresponding NAND and NOR logic circuits for various functions are shown in Figure 4.1.

The circuits of Figure 4.1 show how NAND and NOR gates may be cascaded to form each of the logic functions NOT, AND, OR, and X-OR. Since either NANDs or NORs may be used to implement all of the logic operations, designers may prefer to use only NANDs or only NORs in order to decrease the inventory of spare parts. One of the methods by which to realize this is the *brute force*

FIGURE 4.1 Logic Functions Using NANDs and NORs.



scheme, where each of the logic operations of the Boolean function is replaced by the corresponding NAND/NOR circuit. Note, however, that restricting the number of inputs to the gates will not cause any major problem if proper use of the involution and DeMorgan's laws are made.

EXAMPLE 4.1

Implement $\overline{A \oplus B}$ using only NAND gates.

SOLUTION

There are two different ways to implement this function: (a) using NAND equivalents of an X-OR gate and of a NOT gate, and (b) using the NAND equivalents of either the SOP or the POS terms.

a. The first possibility results in the circuit of Figure 4.2.
b. Otherwise, $\overline{A \oplus B}$ can be expressed in the SOP form as $AB + \bar{A}\bar{B}$. Consequently, the circuit appears as in Figure 4.3. The circuit of Figure 4.3 can be reduced further since $X = \bar{\bar{X}}$. Therefore, the circuit reduces to that of Figure 4.4. Note also that the function could be expressed in the POS form. Consequently, $\overline{A \oplus B} = (A + \bar{B})(\bar{A} + B)$, which leads to another variation of an X-NOR circuit as shown in Figure 4.5. The circuit of Figure 4.5

FIGURE 4.2

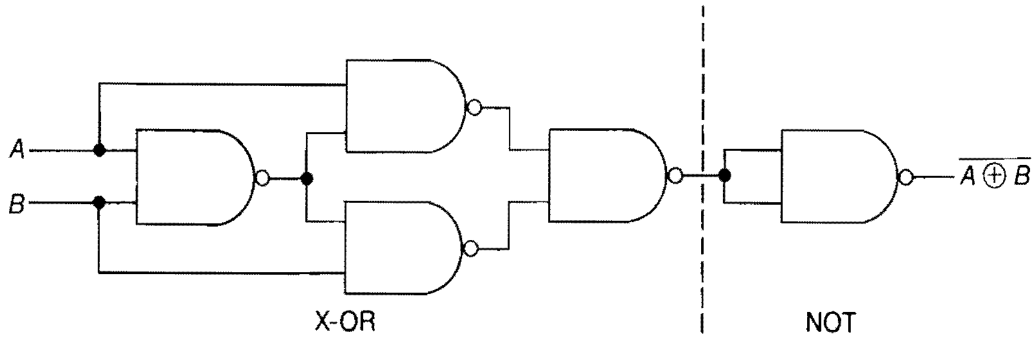


FIGURE 4.3

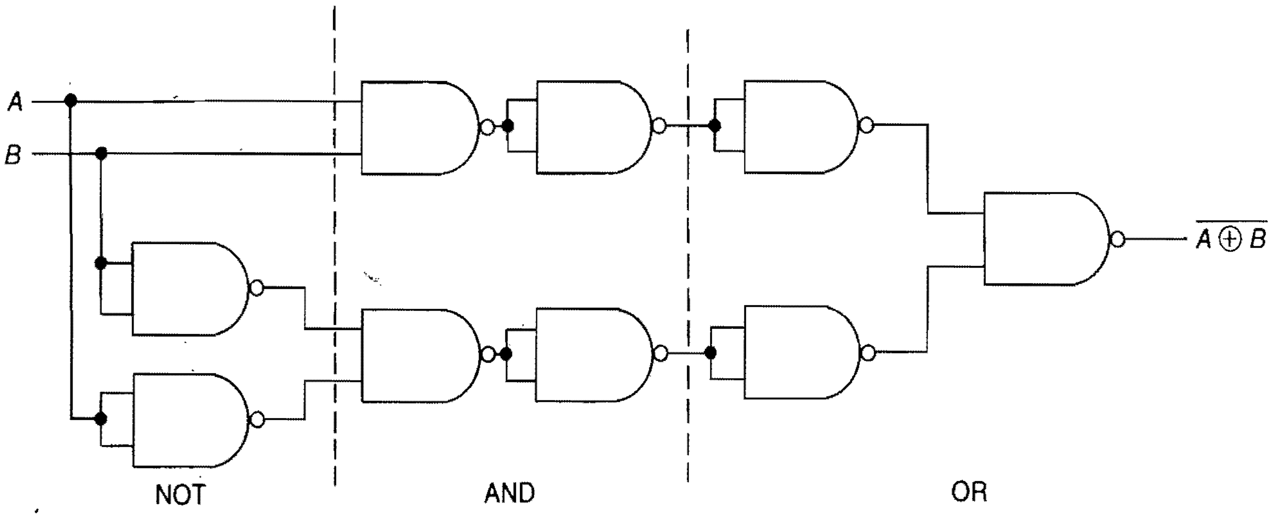
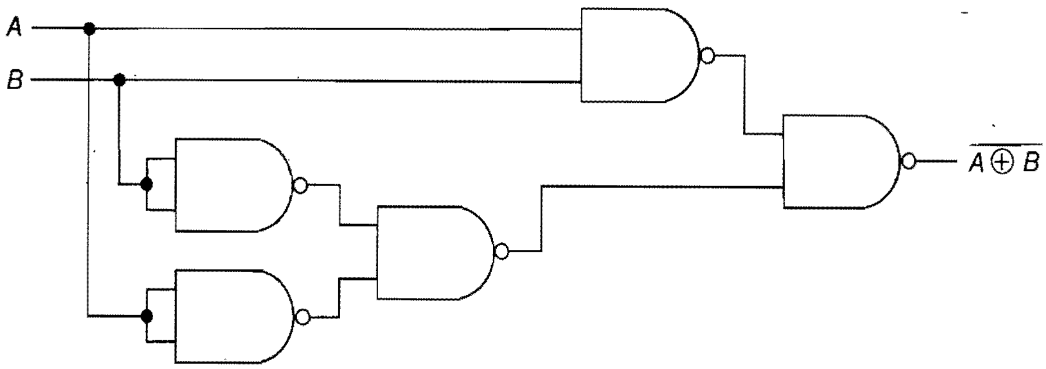


FIGURE 4.4



may be reduced further by making use of the law of involution. The resulting circuit is shown in Figure 4.6.

The first and the third forms, as shown in Figures 4.2 and 4.4, require five NAND gates each, and the fifth, as shown in Figure 4.6, requires a total of six NAND gates.

FIGURE 4.5

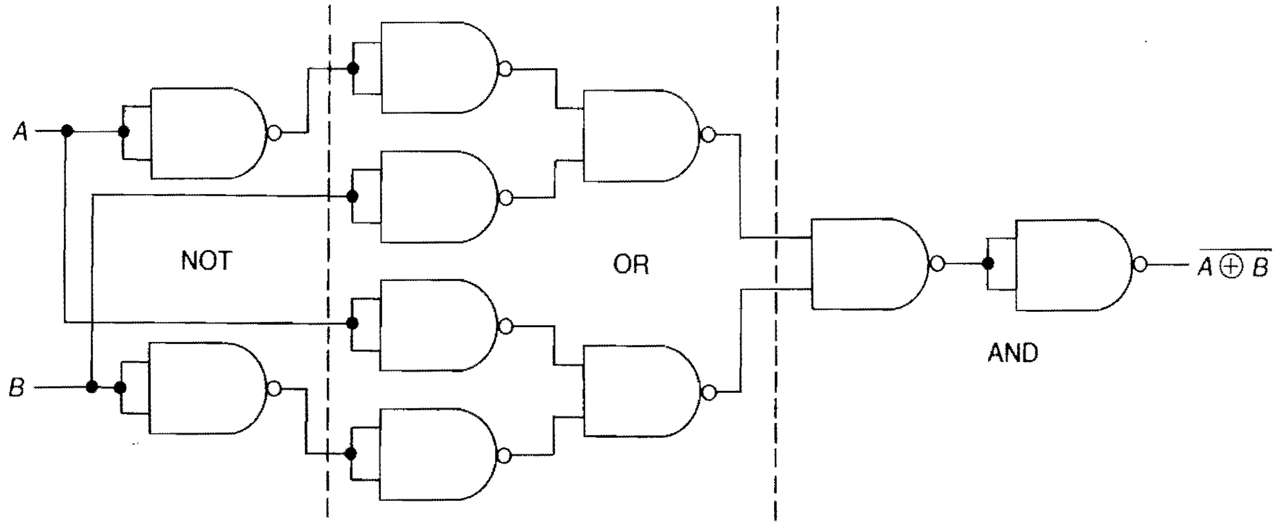
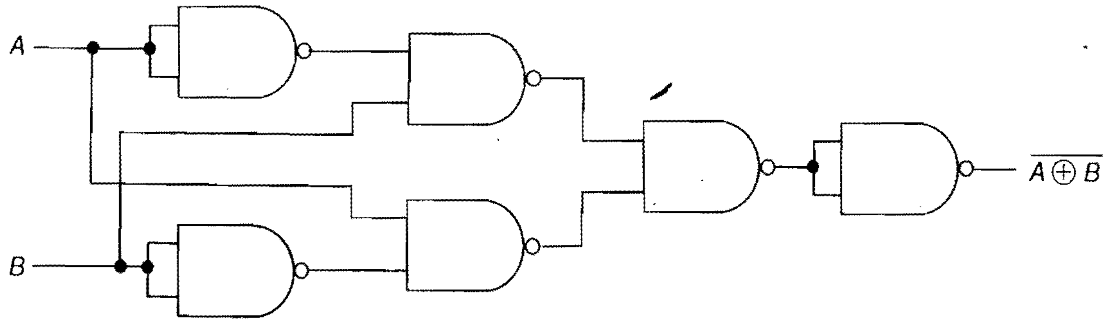


FIGURE 4.6



4.3 Function Implementation Using NANDs

It is quite easy to realize any SOP function using two levels of NAND gates. This method makes use of the fact that complementing a function twice returns the function to its original form. This result is achieved in two steps:

1. The function is complemented by complementing the ANDed terms and replacing the OR signs with AND signs.
2. The original function is then recovered by complementing the complement function.

It is not necessary to perform this operation each time a NAND realization is required. SOP forms always assume the same two-level NAND form.

The output of a NAND gate is also equivalent to the ORed output of the complements of the input. This statement follows directly from DeMorgan's theorem. Consequently, we are led to the following set of rules for obtaining the output function of a multi-level NAND circuit:

Rule 1. Consider the gate from which the output signal is derived as the first level, the preceding gate as the second level, and so on.

Rule 2. In odd-numbered levels the NAND gates perform OR operations. All ungated input variables entering the odd-level NAND gates will appear complemented in the final expression.

Rule 3. In even-numbered levels the NAND gates perform AND operations. All input variables entering the even-level NAND gates will appear uncomplemented in the final expression.

EXAMPLE 4.2

Using only NAND gates, implement the function given by

$f(A,B,C,D) = ABC + A\bar{B}\bar{C} + \bar{C}D$

SOLUTION

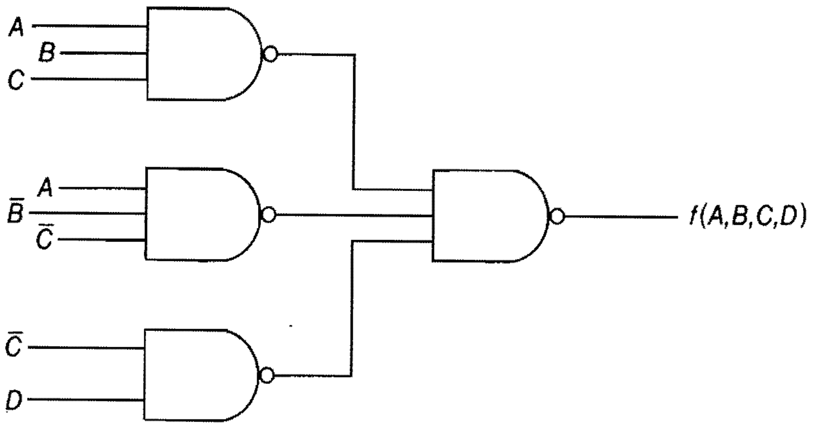
$f = ABC + A\bar{B}\bar{C} + \bar{C}D$

which yields

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

The final circuit, therefore, is obtained as shown in Figure 4.7. The circuit requires three three-input NAND gates and one two-input NAND gate provided B and C inputs are also available in the complemented form.

FIGURE 4.7



4.4 Function Implementation Using NORs

The implementation of an SOP function using only NOR gates is possible only if the function is first converted to the equivalent POS form. The process includes the following steps:

1. Plot the function on a K-map and obtain the complemented function by grouping all zeros.

2. Expand each of the ANDed terms by using DeMorgan's theorem.
3. Complement the whole Boolean expression.

NOR realizations of SOP functions always have the same two-level structure. Steps 1 through 3 should be followed until the designer is confident of the result.

DeMorgan's theorem may be used to interpret the NOR operation as well. The output of the NOR gate is equivalent to the ANDed output of the complements of the inputs. Rules for the interpretation of multi-level NOR circuits are listed as follows:

Rule 1. Consider the gate from which the output signal is derived as the first level, the preceding gate as the second level, and so on.

Rule 2. In odd-numbered levels the NOR gates perform AND operations. All ungated input variables entering the odd-level NOR gates will appear complemented in the final expression.

Rule 3. In even-numbered levels the NOR gates perform OR operations. Input variables entering the even-level NOR gates will appear uncomplemented in the final Boolean expression.

EXAMPLE 4.3

Using only NOR gates, implement the function given by

$f(A,B,C,D) = \Sigma m(0,2,4,5,8,10,13)$

SOLUTION

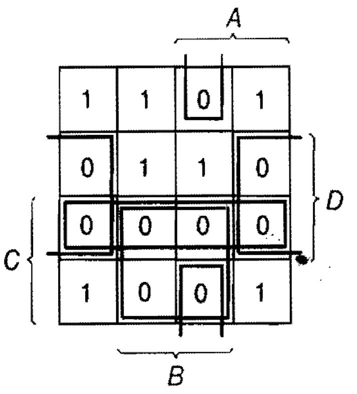
The minterms are plotted in a four-variable K-map and the corresponding zeros are grouped as shown in Figure 4.8. This gives

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

Therefore,

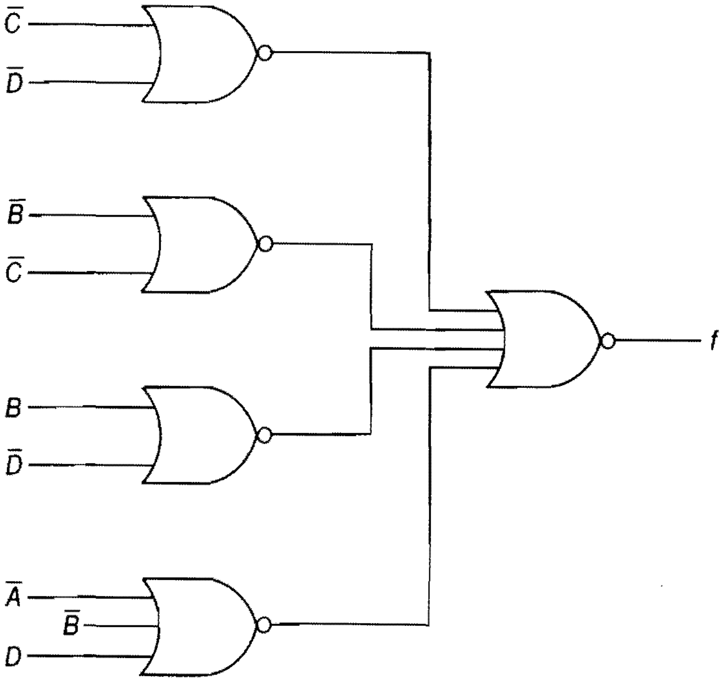
$$f = \overline{\overline{\overline{\bar{C} + \bar{D}} + \overline{\bar{B} + \bar{C}} + \overline{B + \bar{D}} + \overline{\bar{A} + \bar{B} + D}}}$$

FIGURE 4.8



The equivalent NOR circuit, therefore, may be obtained as shown in Figure 4.9. It requires three two-input NOR gates, one three-input NOR gate, and one four-input NOR gate provided the inputs are also available in the complemented form.

FIGURE 4.9

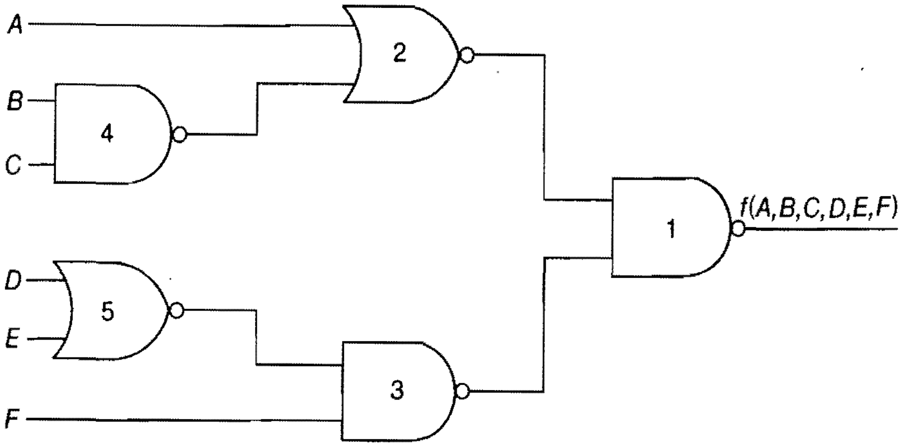


EXAMPLE 4.4

Write the output expression for the multi-level circuit shown in Figure 4.10.

SOLUTION

FIGURE 4.10



This is a three-level circuit. Therefore, the involved operations may be summarized as follows:

a. Gate 1 performs the OR operation;

b. Gate 2 performs the OR operation, and *A* must appear uncomplemented;

c. Gate 3 performs the AND operation, and *F* must appear uncomplemented;

d. Gate 4 performs the OR operation, and both B and C must appear complemented;

e. Gate 5 performs the AND operation, and both D and E must appear complemented.

Therefore,

$$f(A,B,C,D) = [(\bar{D} \cdot \bar{E}) \cdot F] + [A + (\bar{B} + \bar{C})] = \bar{D}\bar{E}F + A + \bar{B} + \bar{C}$$

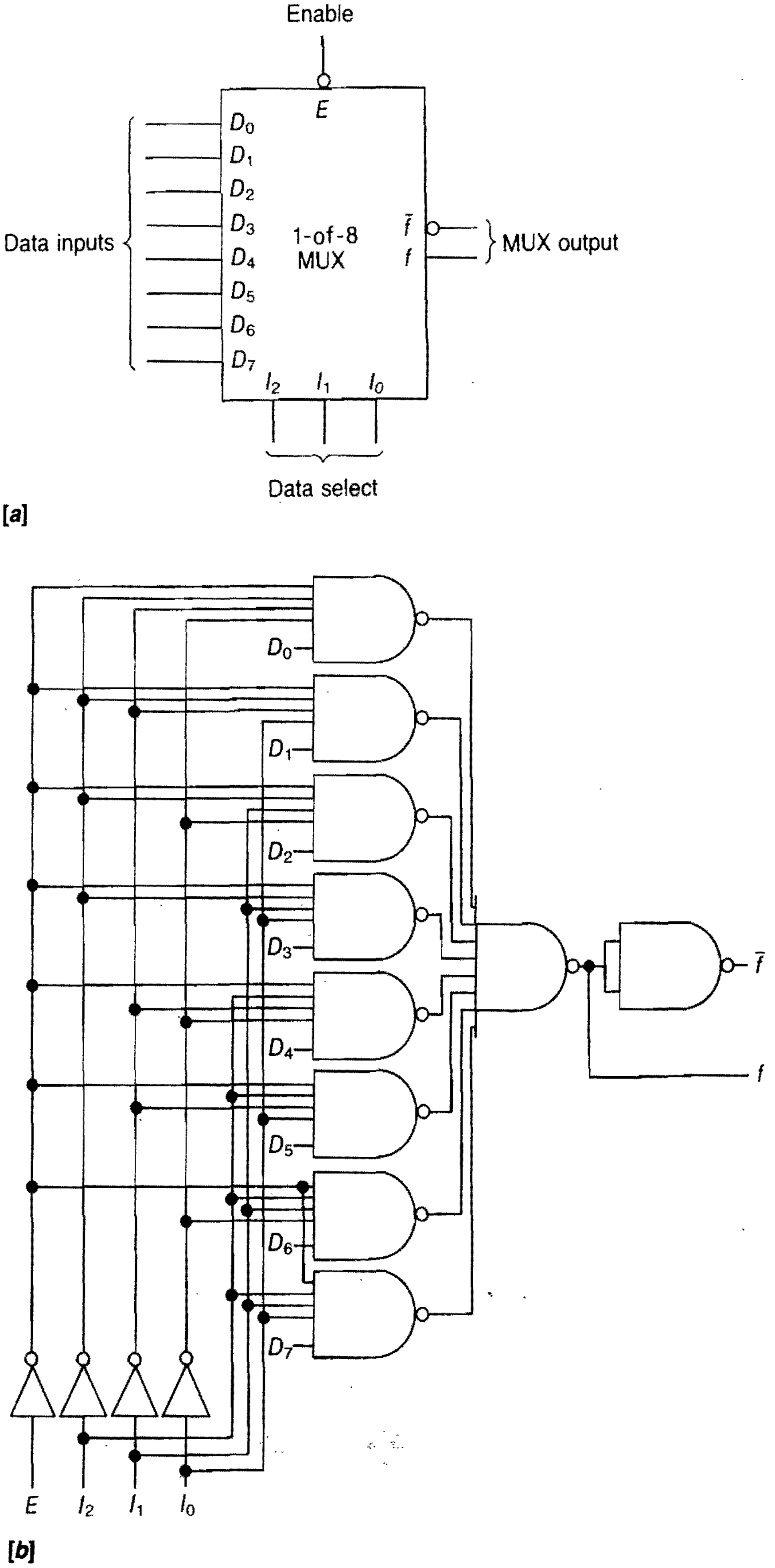
Function output expressions may also be determined by starting at the inputs and making repeated use of DeMorgan's theorem. This latter technique is best when there is a mix of NOR and NAND gates in the circuit.

4.5 Function Implementation Using MUXs

A *multiplexer* (*MUX*), known also as a *data selector*, is a combinational network that has up to 2^n data inputs, n control inputs, and an output line. Commercial MUXs are limited to values of n of 1 through 4. An additional input is available that allows cascading of multiplexers to obtain higher-order devices. The MUX allows the selection of one of the 2^n data inputs as the device output. This selection is made by the control lines. A block diagram of a MUX with eight data input lines, D_0, D_1, D_2, D_3, D_4, D_5, D_6, and D_7, is shown in Figure 4.11[*a*]. Most MUXs are provided with at least two additional lines: $\bar{f}$ for the complemented output and E for enabling the device. The internal circuit configuration of the corresponding MUX is shown in Figure 4.11[*b*]. For every 2^n inputs the MUX has exactly n control lines. By applying appropriate signals to the control lines, any one of the data lines may be selected. For example, when $I_2I_1I_0 = 011$, the D_3 input is routed to the output provided $E = 0$. Note that whenever $E = 1$, the MUX is completely disabled; that is, regardless of the control variables or the data inputs, the output is 0. The enable input allows several of these devices to be cascaded together.

A MUX with 2^n input lines and n selection lines (such a device is also referred to as a 1-of-2^n MUX) may be wired to realize any Boolean function of $n + 1$ variables. This fact will be illustrated by implementing the function $f(A,B,C,D) = \Sigma m(0,2,4,5,6,8,10,13)$ using a 1-of-8 MUX. A 1-of-8 MUX has three control inputs where all but any one of the input variables may be entered. For example, we may consider A, B, and C inputs as the three control inputs, I_2, I_1, and I_0, respectively. The technique, therefore, consists of determining the function output in terms of the fourth input, D, for every possible combination of the control inputs. The values so obtained are then entered at the respective data inputs of the MUX. The truth table for the said function may then be reorganized as shown

FIGURE 4.11 Eight-Input MUX: [*a*] Block Diagram and [*b*] Circuit.



in Figure 4.12. The entries under the $f(D)$ column are determined by comparing the entries of the column D with that of the column f. For any combination of A, B, and C, the following are true:

1. If $f = 1$ irrespective of D, then $f(D) = 1$.
2. If $f = 0$ irrespective of D, then $f(D) = 0$.
3. If $f = x$ when $D = x$, then $f(D) = D$.
4. If $f = \bar{x}$ when $D = x$, then $f(D) = \bar{D}$.

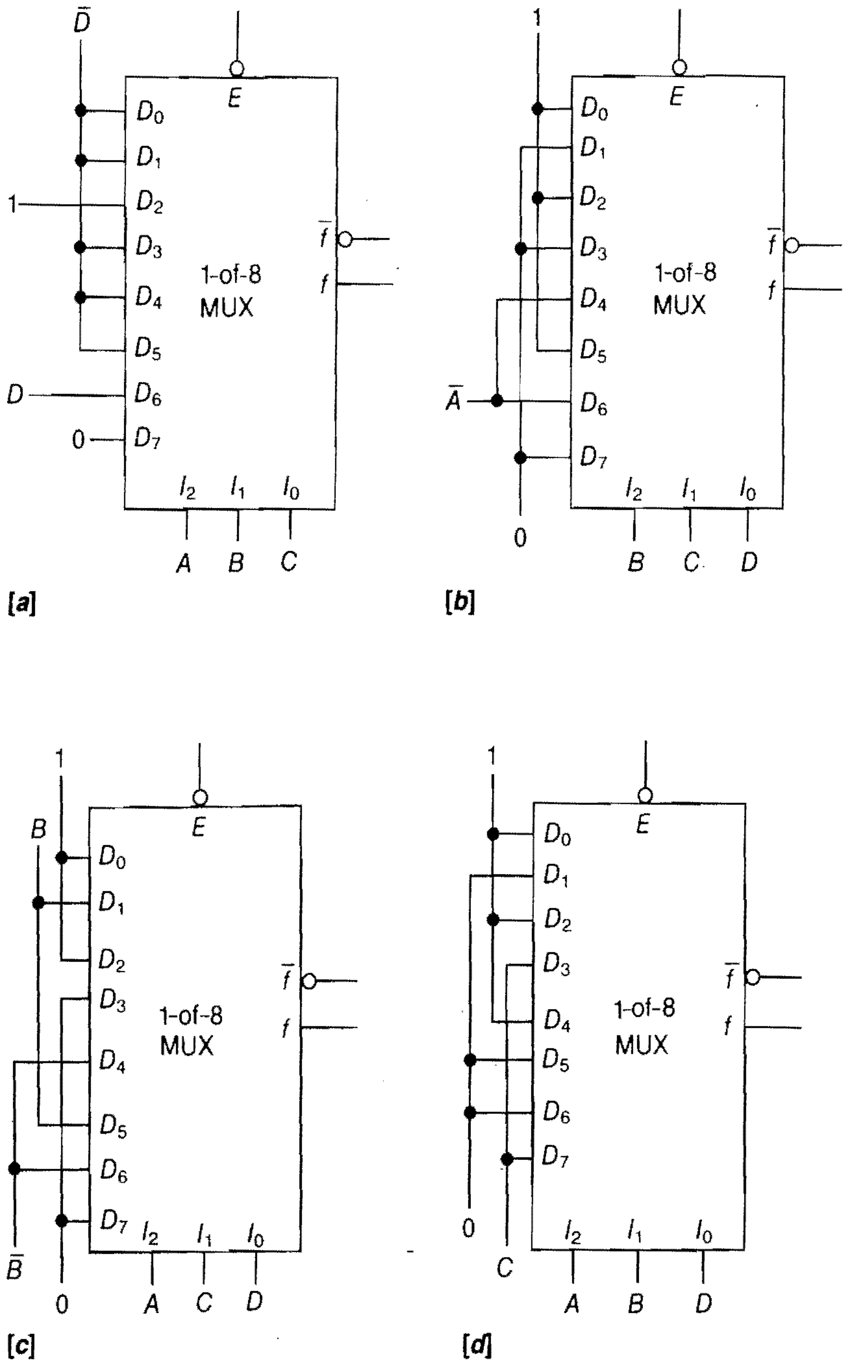
FIGURE 4.12 **Truth Table for $f(A,B,C,D) = \Sigma m(0,2,4,5,6,8,10,13)$.**

A	B	C	D	f	f(D)		
0	0	0	0	1	$\bar{D}$	0	D
			1	0		1	
0	0	1	0	1	$\bar{D}$	0	0
			1	0		0	
0	1	0	0	1	1	1	$\bar{D}$
			1	1		0	
0	1	1	0	1	$\bar{D}$	1	$\bar{D}$
			1	0		0	
1	0	0	0	1	$\bar{D}$	0	0
			1	0		0	
1	0	1	0	1	$\bar{D}$	0	0
			1	0		0	
1	1	0	0	0	D	1	1
			1	1		1	
1	1	1	0	0	0	1	1
			1	0		1	

The function, therefore, is implemented as shown in Figure 4.13[*a*]. The eight values of $f(D)$ are fed respectively into data inputs D_0 through D_7. Consequently, for any combination of the three control inputs, $f(D)$ would actually appear at the MUX output. The implementations of the same function for the other three combinations of the control inputs–B, C, and D; A, C, and D; and A, B, and D–can also be obtained in like manner. The corresponding MUX configurations are shown in Figures 4.13[*b*–*d*]. The designer might even change the order of the control inputs, resulting in a total of 24 different circuit configurations. In addition, if one of the control variables is available only in complemented form, the variable may be used without inverting and the inputs rearranged accordingly (see Problem 3b).

It is quite obvious that the use of MUXs provides the designer with numerous choices. Consequently, it is necessary to consider each of the solutions to determine which is the optimum. In Figure

FIGURE 4.13 MUX Implementation of $f(A,B,C,D) = \Sigma m(0,2,4\text{–}6,8,10,13)$: [*a*] $I_2I_1I_0 = ABC$, [*b*] $I_2I_1I_0 = BCD$, [*c*] $I_2I_1I_0 = ACD$, and [*d*] $I_2I_1I_0 = ABD$.



4.13[*a*], $\bar{D}$ is seen to be tied to five of the data inputs. The gate that provides $\bar{D}$ must then have a fan-out of at least five. If we limit ourselves to the four choices of Figure 4.13, it is apparent that Figure 4.13[*d*] provides the most preferable circuit, because the C variable needs to be fed directly to only two of the data inputs. Example 4.5 illustrates the mechanism of obtaining a multi-level multiplexer circuit.

EXAMPLE 4.5

Implement the function

$f(A,B,C,D,) = \sum m(3,4,8\text{–}10,13\text{–}15)$

a. by using a 1-of-4 MUX and a few assorted gates,

b. by using 1-of-2 and 1-of-4 MUXs in two levels.

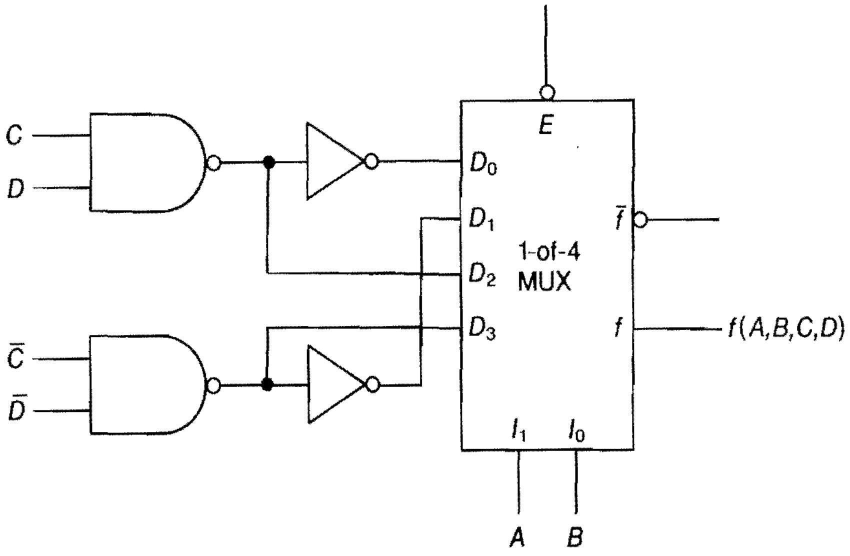
SOLUTION

a. The function may be expressed in the SOP form and then regrouped as

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

Accordingly, the resultant circuit is obtained as shown in Figure 4.14.

FIGURE 4.14



b. Also,

$$\begin{aligned} f(A,B,C,D) &= \bar{A}\bar{B}[\bar{C}(0) + C(D)] + \bar{A}B[\bar{C}(\bar{D}) + C(0)] \\ &\quad + A\bar{B}[\bar{C}(1) + C(\bar{D})] + AB[\bar{C}(D) + C(1)] \end{aligned}$$

This implies that a two-level MUX circuit would be able to generate this function. The first level of a 1-of-2 MUX essentially eliminates the need for discrete gates. The resultant circuit is obtained as shown in Figure 4.15. However it can be shown that

$$\overline{\bar{C}(0) + C(D)} = \bar{C}(1) + C(\bar{D})$$

and

$$\overline{\bar{C}(\bar{D}) + C(0)} = \bar{C}(D) + C(1)$$

Note also that the MUXs usually are provided with an additional output for providing the complemented result. Consequently, two of the first-level 1-of-2 MUXs may be removed. The resulting reduced multi-level MUX circuit is obtained as shown in Figure 4.16.

FIGURE 4.15

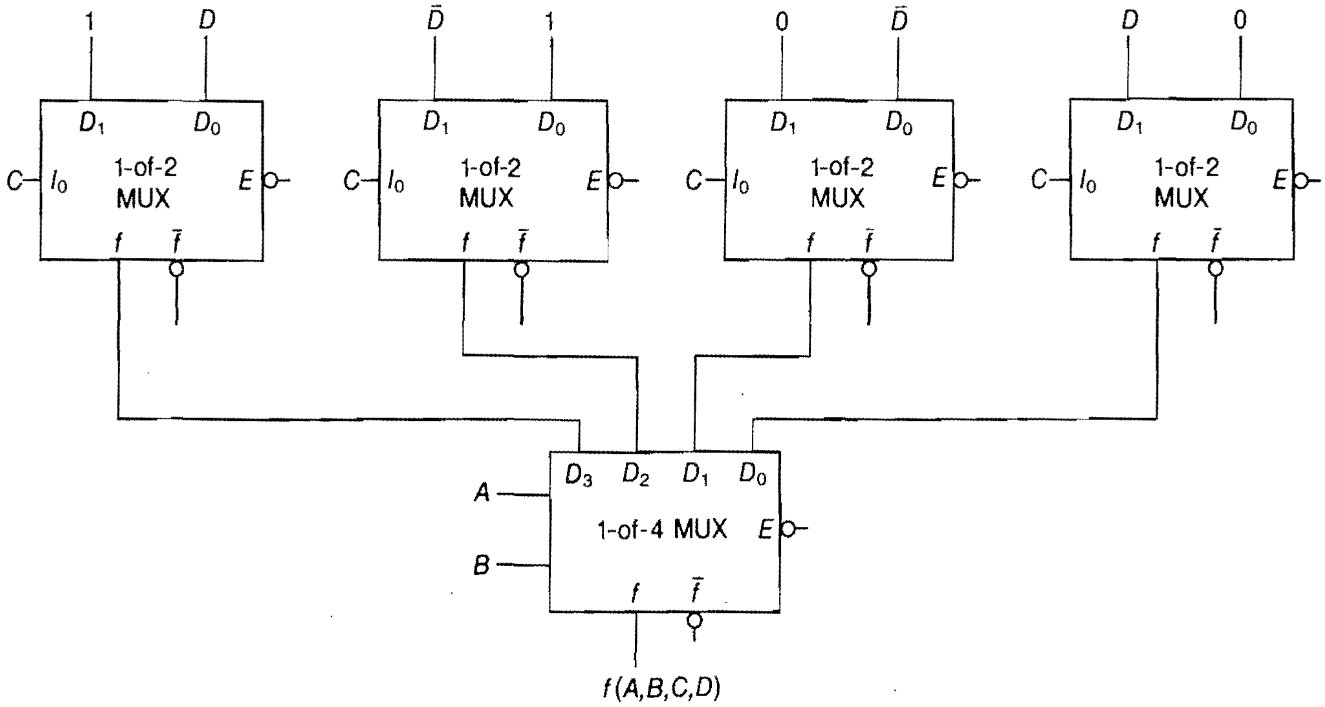
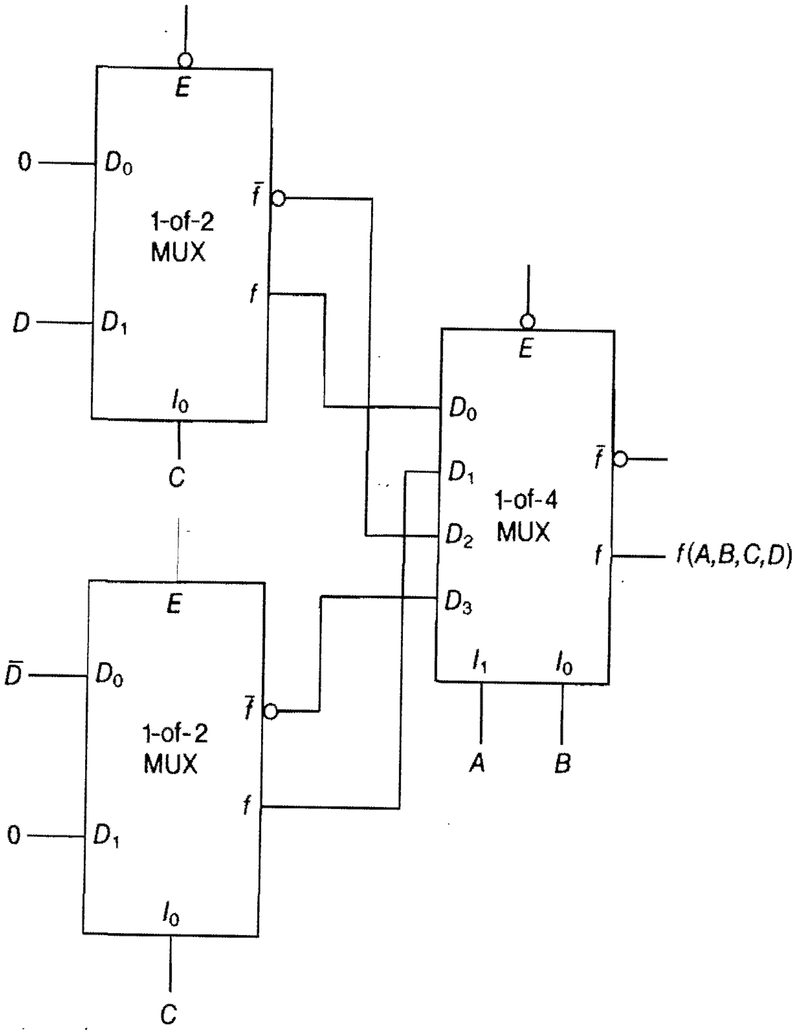


FIGURE 4.16



MUXs can also be applied in a more brute force manner. This technique involves 1-of-2^n MUXs for functions of n variables, while Example 4.5 used 1-of-2^{n-1} MUXs to implement a function. The function values from a function's truth table are transformed directly to the inputs of the MUX. The n variables are treated as control inputs to the MUX. Figure 4.17 shows the truth table for a three-variable function and the corresponding 1-of-8 MUX implementation. For two functions of the same variables, two MUXs may be used as shown in Figure 4.18.

FIGURE 4.17 **Realization of $f(A,B,C) = \Sigma m(1\text{–}3,5,6)$ Using 1-of-8 MUX.**

A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

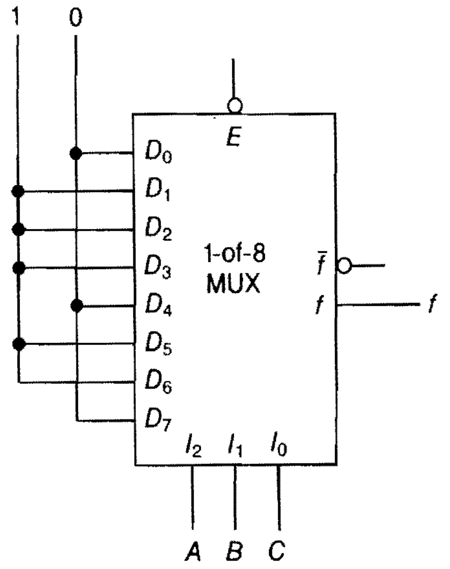
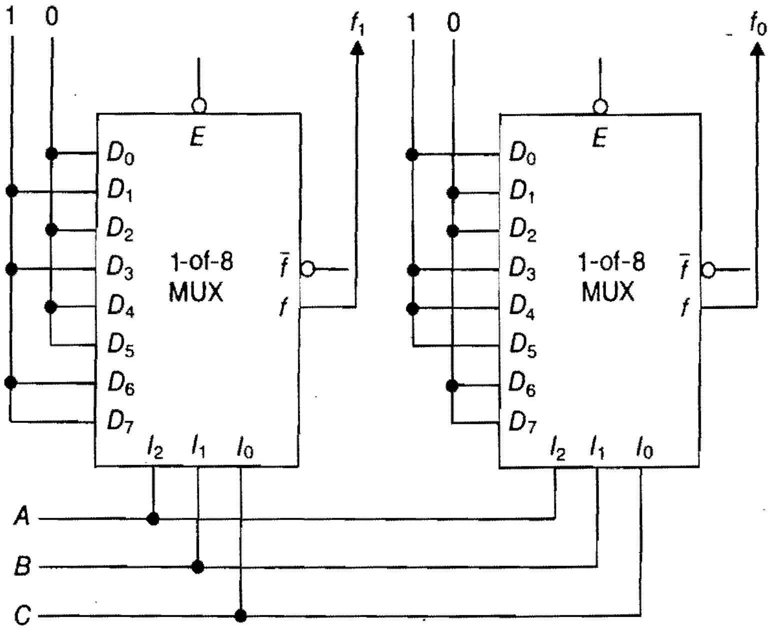


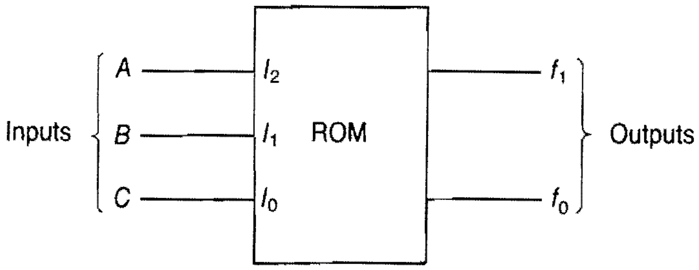
FIGURE 4.18 **Realization of $f_1(A,B,C) = \Sigma m(1,3,6,7)$ and $f_0(A,B,C) = \Sigma m(0,3\text{–}5)$ Using Two 1-of-8 MUXs.**

A	B	C	f_1	f_0
0	0	0	0	1
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	1	0



The technique just discussed is not as efficient as the one before in terms of MUX size, but it is certainly easy to implement and is a good way to introduce another device that may be used to implement combinational logic functions. If we put the circuit implementation of Figure 4.18 in a box, we could show the circuit symbolically as illustrated in Figure 4.19. Such a representation is called a *black-box* representation since the user is not required to know about the details of the internal logic. Not knowing what is in the box in Figure 4.19, we might explain the circuit action by simply saying that ABC forms an *address*, and f_1 and f_0 are what is stored there. This is precisely the explanation of the operation of a device with which we can implement multiple functions of a set of variables. This device is called a *read-only memory* (*ROM*). It can be considered as a set of storage cells with every cell having a value for each of the multiple functions. These ROMs can be programmed (1s and 0s applied to the inputs of each of the function's MUX inputs) by the manufacturer or by the designer if he or she has available blank ROM chips and an appropriate programmer. The information *stored* in the ROM remains there permanently. The ROM equivalent of the logic of Figure 4.18 is shown by the block diagram of Figure 4.19. When power is applied, f_1 and f_0 have the same values as before power was removed.

FIGURE 4.19 **Black-Box Representation of the Circuit of Figure 4.18.**



ROMs come in many sizes. The sizes are determined by the number of storage cells (corresponding to the number of MUX inputs) and the number of bits stored in each cell (the number of functions that can be implemented). In Figures 4.17 and 4.18 there are three variables that allow addressing 2^3 storage cells in our MUX-implemented ROM. In general, n variables would require 2^n storage cells and correspond to n address lines. Each storage cell would have one bit (1 or 0) for each function of the n variables.

Commercially available ROMs come in many sizes. They may be listed as 2K × 1 or 2K × 8, meaning 2048 storage locations one bit wide in the first case and 2048 storage locations eight bits wide in the second case. The first would allow implementing a function of up to 11 variables, the second up to eight functions of 11 variables. The designer selects a ROM of adequate size to implement the desired function.

The following section discusses ROMs in more detail. For most function generation applications, ROMs can be visualized as a set of MUXs—one for each function of the input variables (addresses).

EXAMPLE 4.6

Obtain a multi-bit shifter (see Example 2.4 for definition) that has an $(n + 2)$-bit input, x, an n-bit output, y, and three control inputs: s, d, and E. You may use only MUXs for the design.

SOLUTION

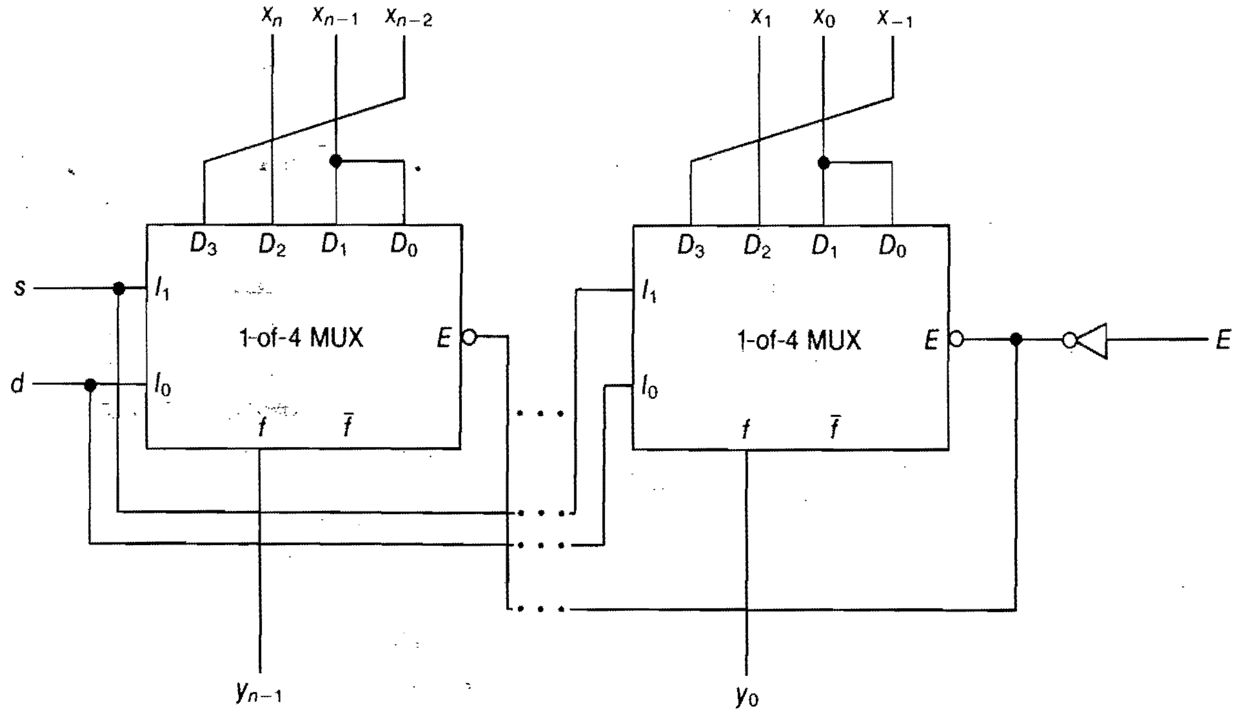
The characteristics of the shifter could be summarized as follows:

$$y_i = \begin{cases} x_{i-1} & \text{if } d = 1, s = 1, \text{and } E = 1 \text{ (left-shift)} \\ x_{i+1} & \text{if } d = 0, s = 1, \text{and } E = 1 \text{ (right-shift)} \\ x_i & \text{if } s = 0 \text{ and } E = 1 \text{ (no-shift)} \\ 0 & \text{if } E = 0 \end{cases}$$

where $0 \le i \le n - 1$.

A 1-of 4 MUX could be used corresponding to each bit of y. The variables s and d can be fed as its selectors, and E as the enable input. The inputs x_i, x_{i-1}, and x_{i+1} could be introduced at the MUX data inputs, and they could be suitably selected as the shifter output. The multi-bit shifter is obtained accordingly as shown in Figure 4.20. Whenever $E = 0$, the MUXs are disabled and the output becomes zero.

FIGURE 4.20



4.6 Function Implementation Using ROMs

A *read-only memory* (*ROM*), as the name implies, is intended to hold fixed information that can only be read, not altered. The primary use of the ROM is to provide a means for storing binary information. The storing is done during the fabrication of the ROM and may not be altered without undergoing a significantly involved process. The same is true for a combinational network that has been designed, fabricated, tested, and encapsulated with only the inputs and the outputs available. The ROM has become an important part of many digital systems because of the ease with which complex functions such as code conversion, program storage, and character generation can be implemented. The chip count of circuits, for which the access time of the ROM is not a restriction, may be greatly reduced by using ROMs.

A $2^m \times n$ ROM is an array of memory cells organized into 2^m words of n bits each, as shown by the block diagram of Figure 4.21. Such a ROM is accessed by means of m address lines and the stored information is retrieved via a total of n data-out lines, one for each bit of the word. The ROM corresponds to a combinational network with n outputs, where each of the outputs is associated with up to 2^m different minterms. A ROM may be provided with one or more chip-select lines to permit cascading smaller ROMs to form a ROM with more words (allowing implementation of functions of more variables).

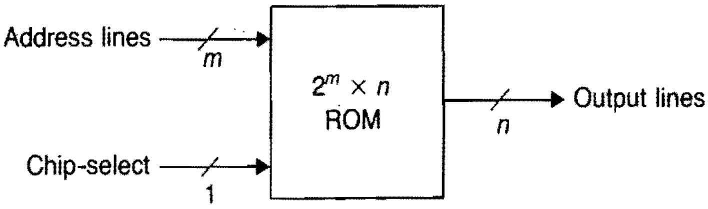
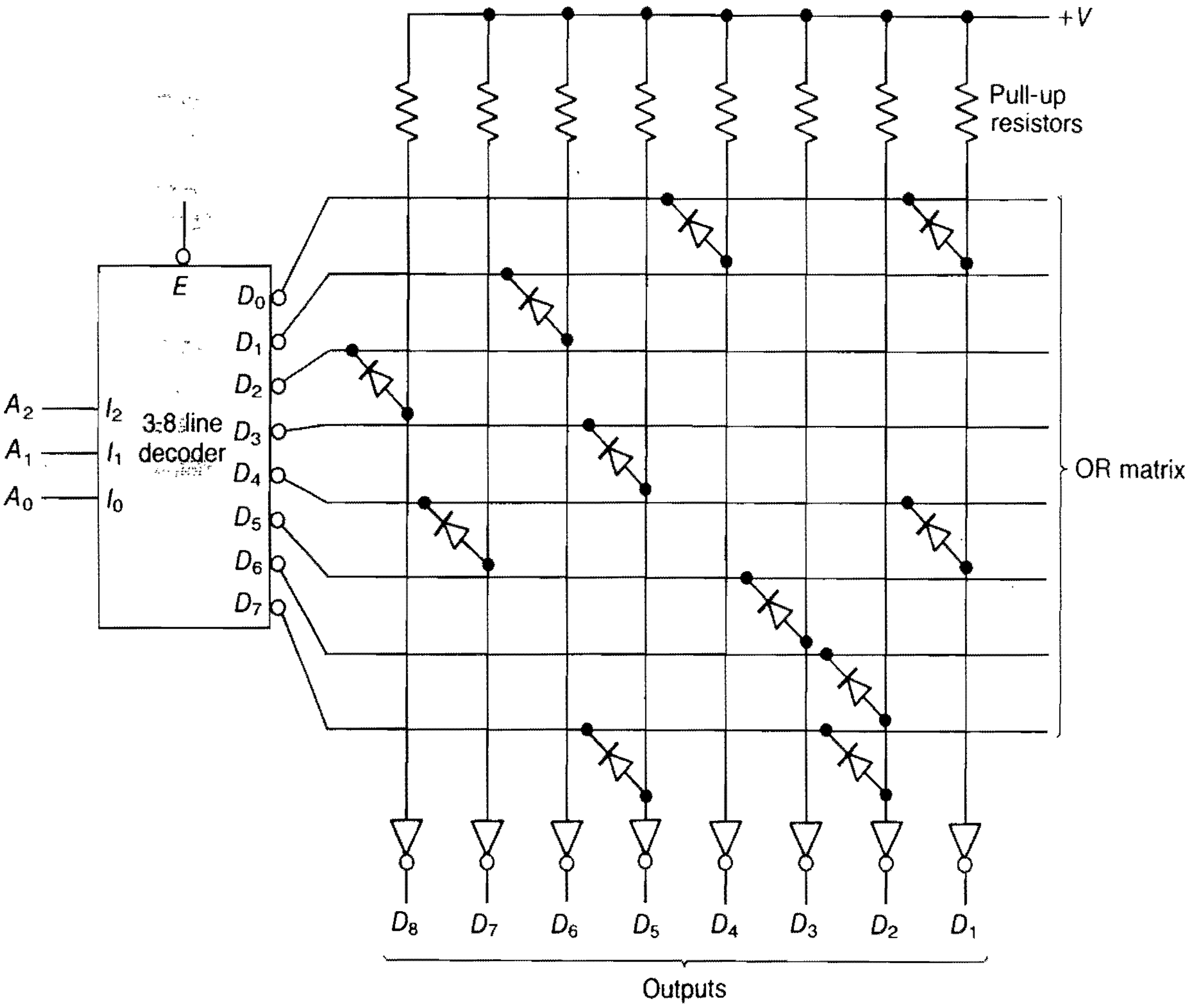


FIGURE 4.21 **Block Diagram of a ROM.**

As indicated in the previous section, a ROM is a combinational circuit. A ROM can be implemented by using only diodes, bipolar transistors, or MOS transistors. Although the diode matrix ROM no longer represents the current ROM technology, it serves as a simple model to show the basic concept. Figure 4.22 shows a simple diode ROM, where the row and column lines are interconnected via diodes placed at the respective intersections. The absence or presence of a diode indicates that the corresponding row and column intersection is programmed with a 1 or a 0. If the output is buffered with inverters, the converse is true. The output for the *i*th address depends on the ORing diodes connected to that line. For example, if $A_2A_1A_0 = 100$, the output will be $D_8D_7 \dots D_1 = 01000001$. For $A_2A_1A_0 = 100$ input, a low is produced on the decoder output numbered 4. This low and the pull-up resistor forward biases each of the diodes connected to this row and pulls down

FIGURE 4.22 **8 × 8 Diode ROM.**



the corresponding column outputs to a low. In the absence of a diode, a high is maintained at the output due to the pull-up resistors. The capacity of a ROM usually is quoted as the number of possible intersections in the matrix, for example, 8 × 8 = 64 bits = (1/16)K bits in this case. A total of 1024 intersections is usually referred to as 1K bits. Examples 4.7 and 4.8 illustrate the use of diode ROMs in combinational problems.

EXAMPLE 4.7

Use ROM to realize the implementation of the integer function

$f(x) = x^3 \quad \text{for } 0 \leq x \leq 3$

SOLUTION

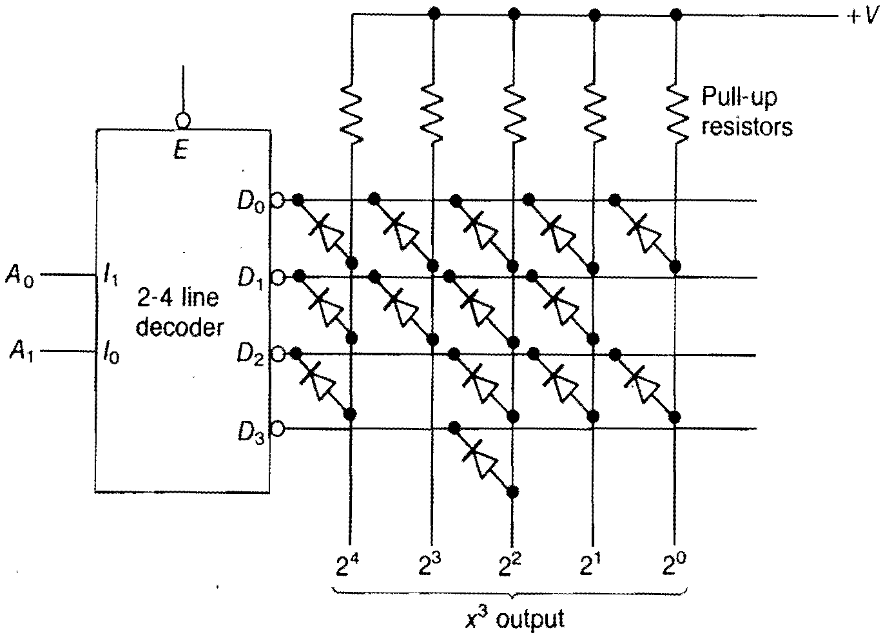
The function truth table is obtained as shown in Figure 4.23. A 2-4 line decoder would be sufficient to decode the numbers 0, 1, 2, and 3, and it is apparent that a maximum of five output lines are needed to represent the cube of the largest number. The two select lines, A_0 and A_1, can be used to select any one of these four outputs.

FIGURE 4.23

x	$f(x)$	$f(x)$ in Binary
0	0	00000
1	1	00001
2	8	01000
3	27	11011

The resultant diode matrix implementation of x^3 ROM, therefore, is obtained as shown in Figure 4.24.

FIGURE 4.24

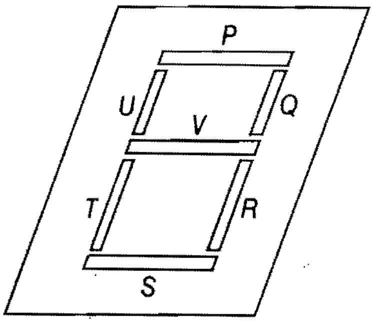


EXAMPLE 4.8

Using a minimal ROM implementation, design a seven-segment-to-BCD code converter. The seven-segment display device consists of seven LEDs, as shown in Figure 4.25, arranged in such a way that they could be used for displaying data.

SOLUTION

FIGURE 4.25



The seven-segment-to-BCD truth table is obtained as shown in Figure 4.26. The input entries are selected in such a way that the output would be displayed only if the corresponding LED segments are lighted. For example, when all of the seven segments are turned on, the display will be an 8.

FIGURE 4.26

Inputs							Outputs			
P	*Q*	*R*	*S*	*T*	*U*	*V*	*D*	*C*	*B*	*A*
1	1	1	1	1	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1
1	1	0	1	1	0	1	0	0	1	0
1	1	1	1	0	0	1	0	0	1	1
0	1	1	0	0	1	1	0	1	0	0
1	0	1	1	0	1	1	0	1	0	1
0	0	1	1	1	1	1	0	1	1	0
1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	0	0	0
1	1	1	0	0	1	1	1	0	0	1

Note, however, that there could be up to two choices for displaying a 1: either *U* and *T* or *Q* and *R*.

A direct implementation of this table requires a $2^7 \times 4 = 512$-bit ROM. Note that the partitioned table (with inputs *P, Q, R, S,* and *T*), as shown in Figure 4.27, would require only seven of the possible $2^5 = 32$ combinations. This situation suggests that a reasonable improvement is possible if the designer is willing to cascade at least two smaller ROMs. One of these ROMs should have at the least three outputs–*Z, Y,* and *X*–since $2^3 > 7$. The output of the first ROM–*Z, Y,* and *X*–can be fed along with the other two inputs–*U* and *V*–into the second ROM. However, care must be taken in organizing the second ROM so that its output becomes equivalent to that of Figure 4.26. Accordingly, the compressed truth table of Figure 4.28 is obtained such that it incorporates the same logic as that of Figure 4.26. The seven inputs have now been replaced by only five inputs, where the first three are functions of *P, Q, R, S,* and *T* and the other two are *U* and *V* themselves.

Outputs *Z, Y,* and *X* are realizable using a 96-bit (3×2^5) ROM. A second ROM can be used where *Z, Y, X, U,* and *V* are the inputs. The second ROM size is $4 \times 2^5 = 128$ bits. Therefore, the total ROM size requirement is reduced to only $96 + 128 = 224$ bits. This solution reduces

FIGURE 4.27

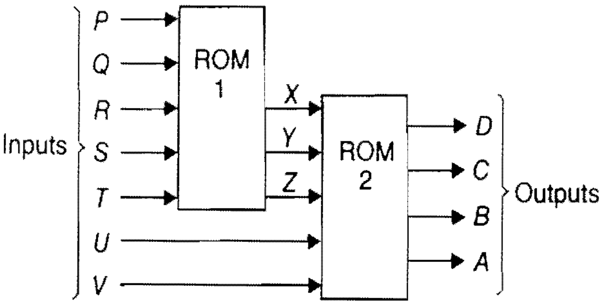
P	*Q*	*R*	*S*	*T*	*Z*	*Y*	*X*
1	1	1	1	1	0	0	0
0	1	1	0	0	0	0	1
1	1	0	1	1	0	1	0
1	1	1	1	0	0	1	1
1	0	1	1	0	1	0	0
0	0	1	1	1	1	0	1
1	1	1	0	0	1	1	0

FIGURE 4.28

Z	*Y*	*X*	*U*	*V*	*D*	*C*	*B*	*A*
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	1
0	1	0	0	1	0	0	1	0
0	1	1	0	1	0	0	1	1
0	0	1	1	1	0	1	0	0
1	0	0	1	1	0	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	0	0	0	1	1	1
0	0	0	1	1	1	0	0	0
1	1	0	1	1	1	0	0	1

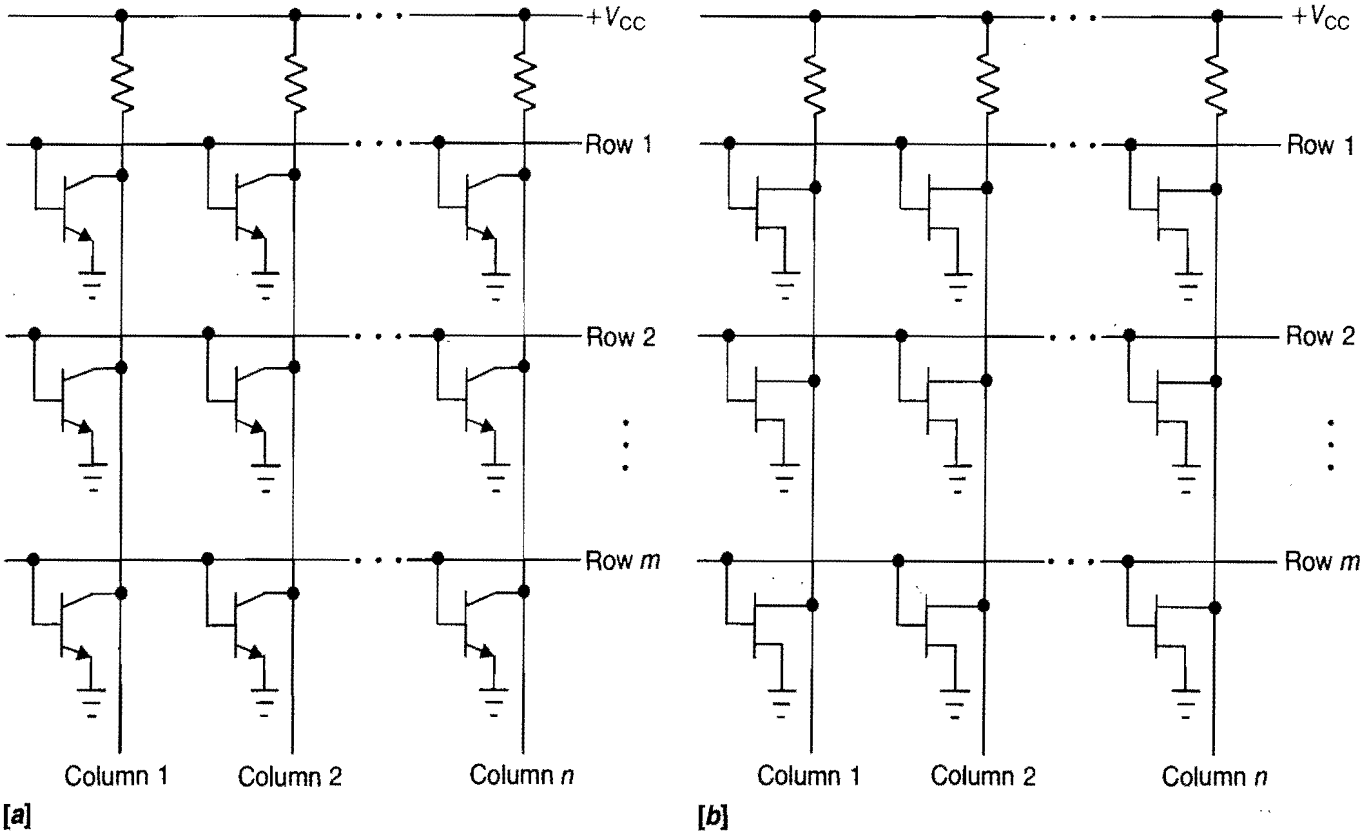
the ROM size to about half of the original. The ROM implementation of the circuit, therefore, is given by the multi-level circuit of Figure 4.29.

FIGURE 4.29



Although the diode matrix serves to demonstrate the ROM concept, ROMs are presently manufactured using bipolar and MOS transistors, as shown in Figure 4.30. The presence of a connection from a row line to either a transistor base or a MOSFET gate represents a logic 0, and the absence of such a connection represents a logic 1. For economical reasons MOS ROM is preferred to bipolar ROM for large numbers of bits. Access time for bipolar ROM, however, is much less than that of MOS ROM.

FIGURE 4.30 **[*a*] Bipolar *m* × *n* ROM and [*b*] MOS *m* × *n* ROM.**

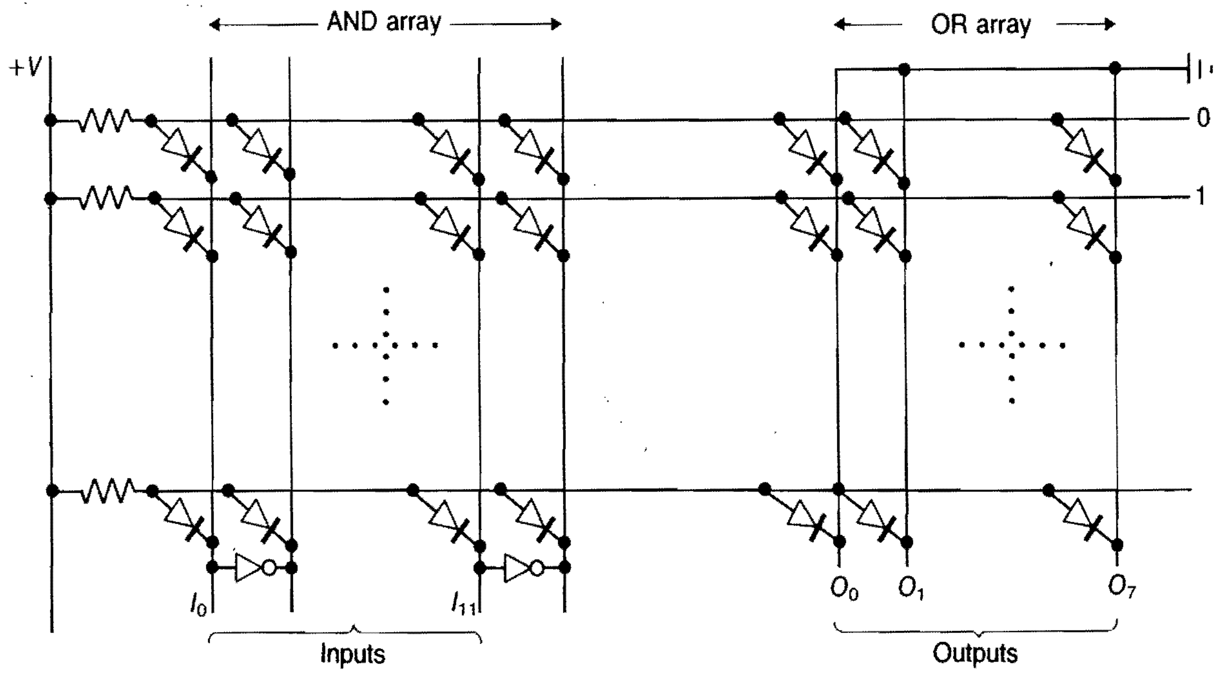


4.7 Function Implementation Using PLAs and PALs

Examples 4.7 and 4.8 illustrated how a ROM may be used to implement SOP logic expressions. There is another aspect of the ROM that deserves attention, however. A ROM consists of a level of AND gates, which constitute the decoder part, followed by a second level of OR gates (made up of diodes or transistors), which constitute the encoder section. A ROM may be thought of as a programmable array of logic gates. With this array of AND and OR gates, every combination of minterms of the input variables (addresses) can be formed. This flexibility is costly in that, when implementing complex functions, not all minterms are necessary to realize a given expression. For example, a ROM that processes 12 variables requires a total of 4K byte (eight bits are called a *byte*) memory. For example, such an arrangement is needed for the Hollerith code conversion circuit that has up to 12 input variables, but has only 96 eight-bit output combinations of these variables. This situation implies that 4000 out of 4096 bytes will remain unused. Such waste can be eliminated by the use of a *programmable logic array* (*PLA*).

A PLA consists of an array of AND-OR logic along with *inverters* that may be programmed to realize the desired output. In essence a PLA may be regarded as being made up of two separate ROMs: an AND ROM and an OR ROM. A typical PLA configuration is shown in Figure 4.31 in a 12 × 32 × 8 format. The circuit consists

FIGURE 4.31 12 × 32 × 8 PLA Using Diodes.



of an initial AND array, which can implement any one or more of the 32 product terms of up to 12 variables. The inclusion of an inverter with each input variable allows any minterm to be formed.

A PLA may be used as a Boolean function generator in much the same way as a ROM. As a simple example, Figure 4.32 shows a small PLA layout with six inputs, twelve product terms, and four outputs. The dots in the matrix of the top section can be thought of as AND inputs and those on the bottom part can be interpreted as OR inputs to generate the outputs. The output functions are easily determined as follows:

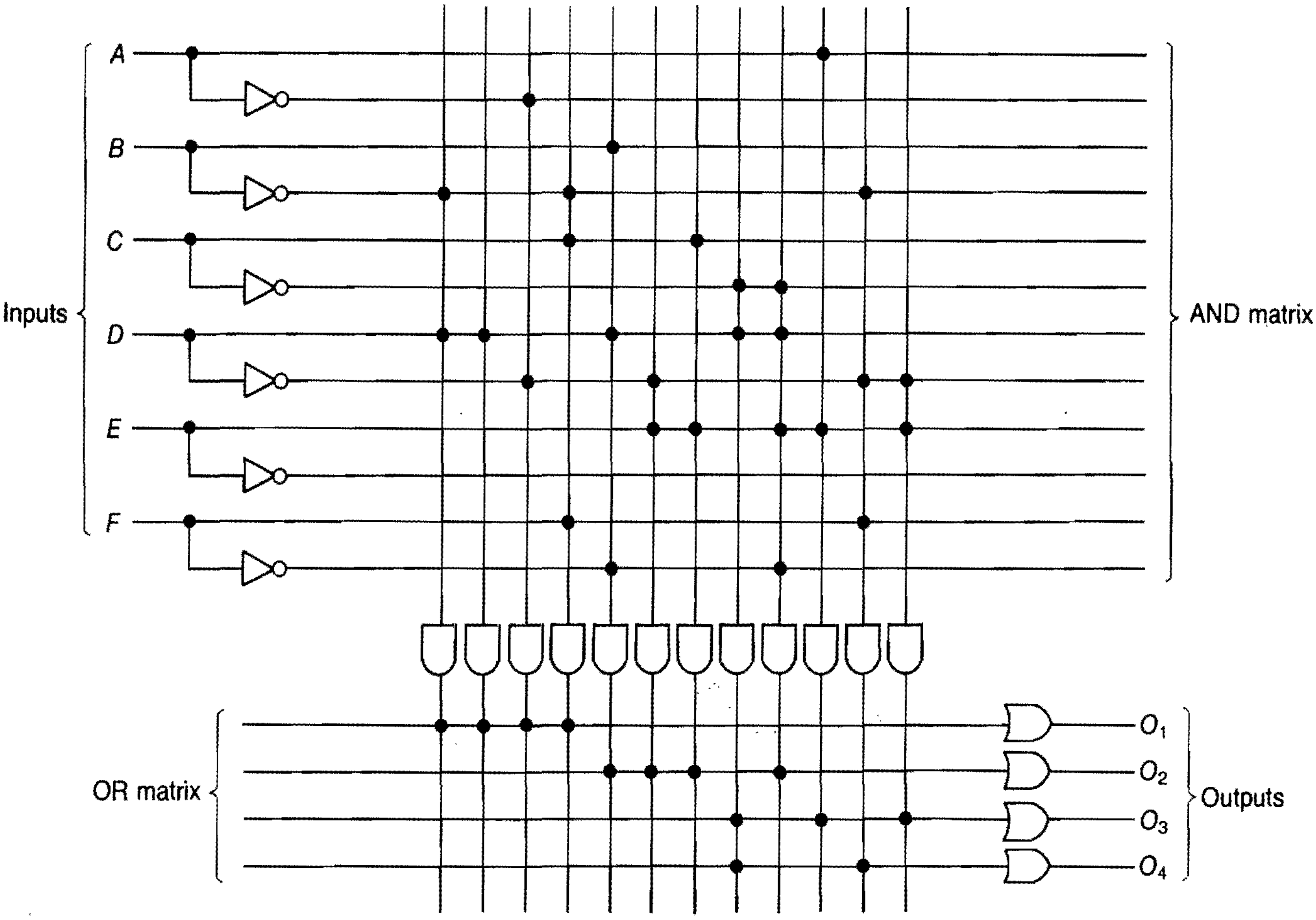
$$O_1 = \bar{B}D + D + \bar{A}\bar{D} + \bar{B}CF = \bar{A} + D + \bar{B}CF$$

$$O_2 = BD\bar{F} + \bar{D}E + CE + \bar{C}DE\bar{F}$$

$$O_3 = \bar{C}D + AE + \bar{D}E$$

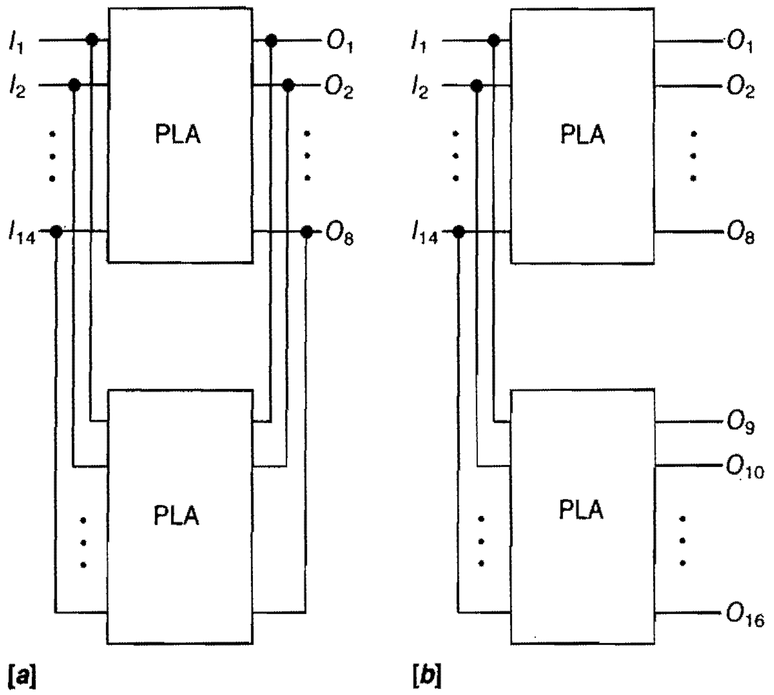
$$O_4 = \bar{C}D + \bar{B}\bar{D}F$$

FIGURE 4.32 **PLA Implementation.**



Larger numbers of product terms and/or outputs may be obtained when more than one PLA is cascaded. Some of these expansion schemes are shown in Figure 4.33. The product terms essentially are increased by tying the outputs in parallel. This configuration resembles the wiring of open-collector outputs. Correspondingly, an increase in the word size could be accomplished by unhooking the outputs.

FIGURE 4.33 **PLA Expansion Scheme: [*a*] Product Term Expansion and [*b*] Output Expansion.**

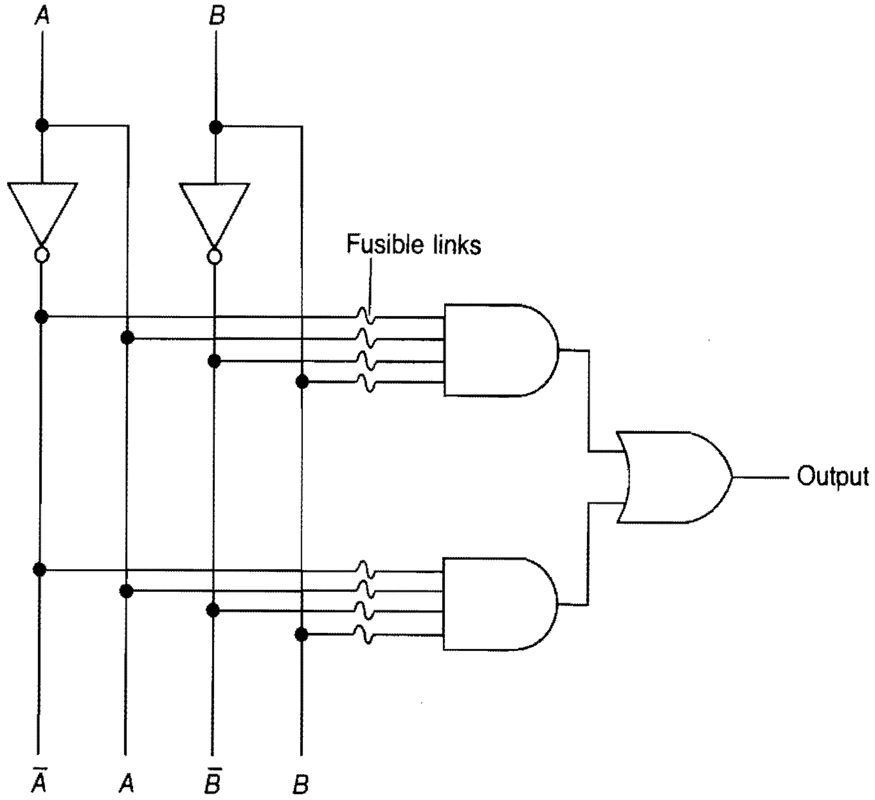


The designer must use care in choosing the minterms to be formed in the AND section of the PLA. In order to use PLAs optimally, it is necessary to have as many output functions as possible that have common minterms. It is not necessary for each function to be minimized; the goal is to minimize the total number of minterms required to implement the set of functions.

It would be appropriate now to discuss an additional programmable device known as *programmable array logic* (*PAL*). It also allows the systems engineer to design his or her "own chip" by fusible links to configure AND and OR gates to perform the desired logic functions. The PAL is basically a programmable AND array driving a fixed OR array. In comparison, both of the arrays of PLA are programmable, while the programmable version of ROM, known commonly as PROM (to be discussed in detail in Chapter 12), has a fixed AND matrix and programmable OR array.

In the PAL circuit, as shown in Figure 4.34, an AND array allows the designer to specify the product terms required and connect them to perform the required SOP logic functions. The PALs are available in a number of different part types, however, that vary the OR gate format. Specifying the OR gate connection, therefore, becomes a task of device selection rather than of programming.

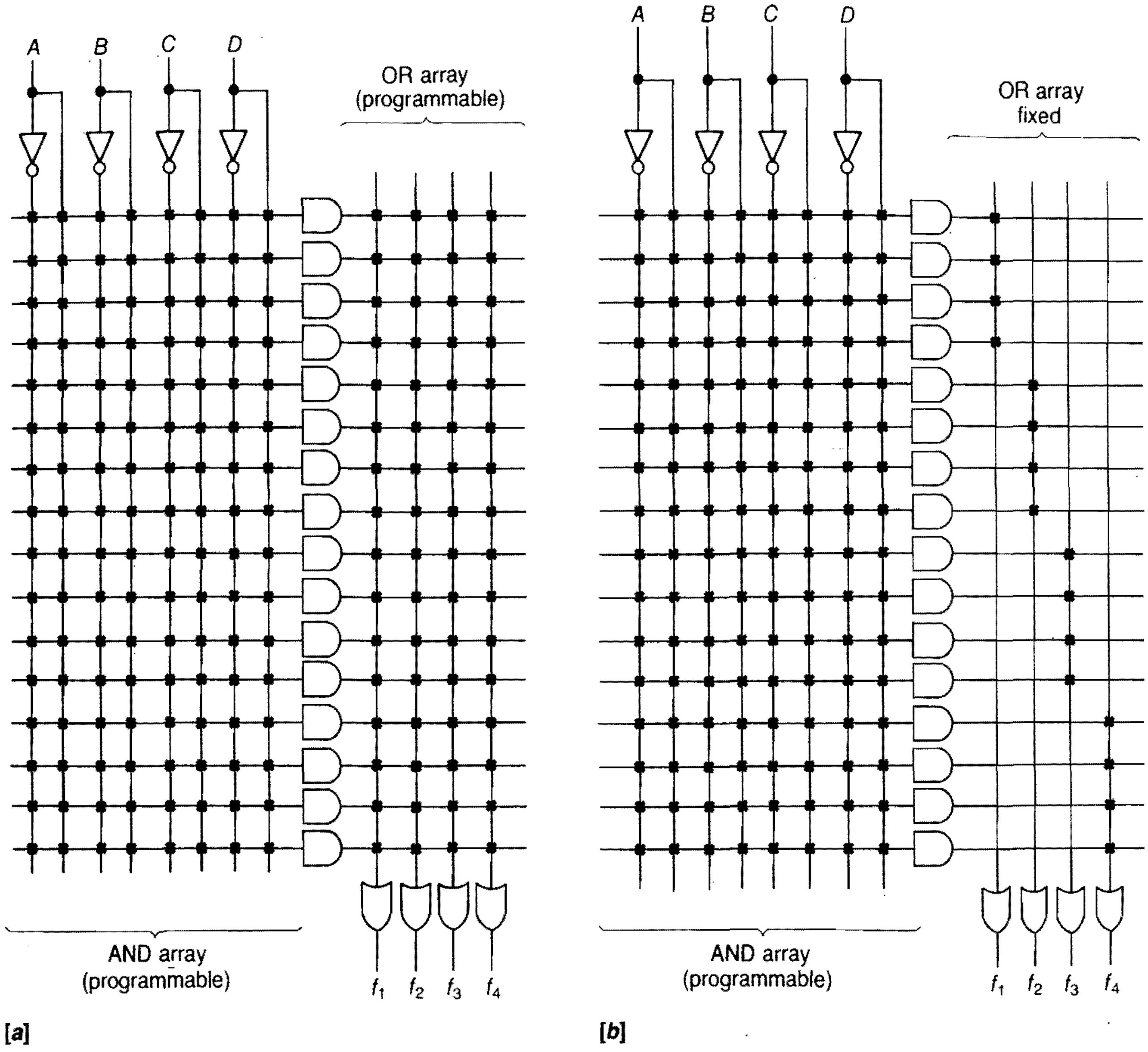
FIGURE 4.34 **Logic Diagram of a Simple PAL.**



Consequently, PALs totally eliminate the need for a second matrix without any significant loss of flexibility.

In general PALs offer cost-effective capabilities for improving the effectiveness of existing logic designs by expediting and simplifying prototypes and board layouts. Figures 4.35[*a–b*] respectively show the PLA and PAL configurations of a four-input–four-output AND-OR circuit. The PLA provides the most flexibility for implementing logic functions since the designer is equipped with complete control over all inputs and outputs. However, this flexibility makes PLAs expensive and somewhat formidable to comprehend. In comparison, the PAL combines much of the flexibility of the PLA with the low cost and easy programmability of the PROM.

FIGURE 4.35 **A Four-Input, Four-Output AND-OR Circuit Using [*a*] PLA and [*b*] PAL.**



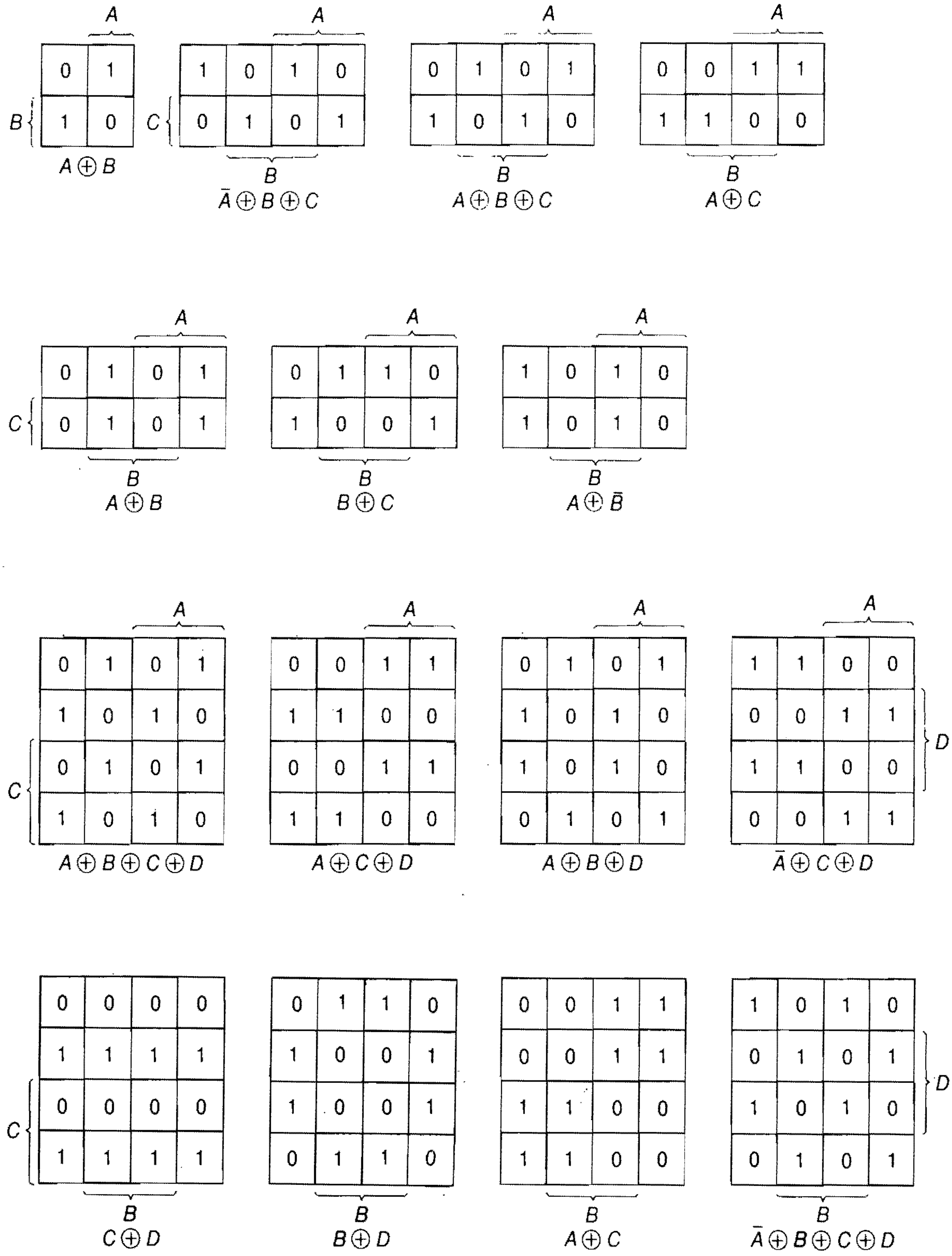
✱ indicates a fusible link. Links are removed where no connection is desired.

4.8 Bridging Technique

The bridging technique is not so much a self-contained design algorithm as it is a way to bend the characteristics of a Boolean function that cannot be reduced further. If after using K-maps or the Q-M technique, the function is still large and unwieldy due to the minterms being logically separated, the function might be *bridged* by using known functions that exhibit similar patterns of logically separated minterms. The X-OR function frequently is used in the bridging process.

Consider the K-maps of several X-OR functions, shown in Figure 4.36. It is obvious that such K-maps are not reducible without the X-OR function. A close examination of these maps reveals that there are equal numbers of 1s and 0s on each half. In addition, the

FIGURE 4.36 Examples of Several X-OR Functions.



complement of the X-OR function may be obtained either by complementing the whole function or by complementing odd numbers of variables. An X-OR function can be changed to another equivalent X-OR function as long as either (a) an even number of variables have been complemented, or (b) the entire function and an odd number of variables have been complemented. For example,

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

The X-OR functions are often very useful in implementing functions that have logically isolated minterms.

As long as the function to be implemented bears the characteristic of an X-OR function, it can be realized using one or more X-OR gates. However, the problem becomes more difficult when the K-map closely resembles that of an X-OR function but is not one. The bridging technique then is used to connect the desired function and a closely resembling X-OR function. The technique consists of the following steps:

1. Match the function K-map, F, as closely as possible to a known X-OR K-map, f.
2. Realize the function F using bridging such that $F = f \cdot X + Y$, where X and Y are two separate functions of the same input variables. X and Y are determined by closely comparing K-maps of F and f.

Example 4.9 illustrates the idea behind the bridging scheme.

EXAMPLE 4.9

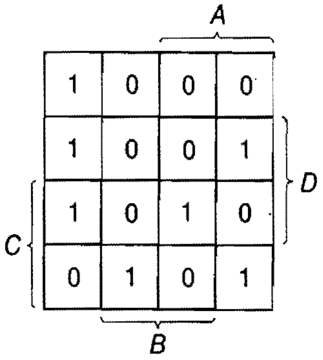
Using X-OR and other assorted gates, implement the function

$F(A,B,C,D) = \Sigma m(0,1,3,6,9,10,15)$

SOLUTION

The function K-map is obtained as shown in Figure 4.37. By comparing the K-map of Figure 4.37 with those of Figure 4.36, it would become obvious that the closest match occurs with $f = \bar{A} \oplus B \oplus C \oplus D$. However,

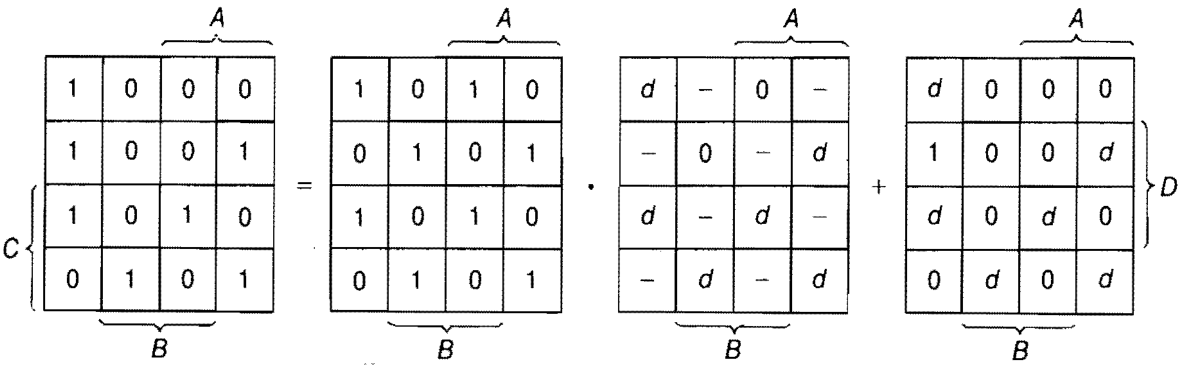
FIGURE 4.37



1	0	0	0
1	0	0	1
1	0	1	0
0	1	0	1

they are not exactly alike. They differ at three minterm locations: 1, 5, and 12. The two functions may be bridged, therefore, as shown in Figure 4.38. The bridging between the two functions F and f required that certain constraints, as listed in Figure 4.39, be met in determining X and Y functions. These constraints follow directly from the equation $F = f \cdot X + Y$.

FIGURE 4.38



1	0	0	0
1	0	0	1
1	0	1	0
0	1	0	1

=

1	0	1	0
0	1	0	1
1	0	1	0
0	1	0	1

·

d	–	0	–
–	0	–	d
d	–	d	–
–	d	–	d

+

d	0	0	0
1	0	0	d
d	0	d	0
0	d	0	d

FIGURE 4.39

F	f	X	Y
0	0	–	0
0	1	0	0
1	0	–	1
1	1	d	d

The d's in the table of Figure 4.39 indicate that either X or Y or both must equal 1. In other words, X and Y cannot simultaneously be 0 when $F = f = 1$. This use of d, however, permits many choices for the selection of X and Y. It can be seen that if all d's in X are set equal to 0, then $F = Y$, which is contrary to what is expected in bridging. When $F = Y$ no bridging is needed. On the other hand, if all d's in X are set equal to 1, then

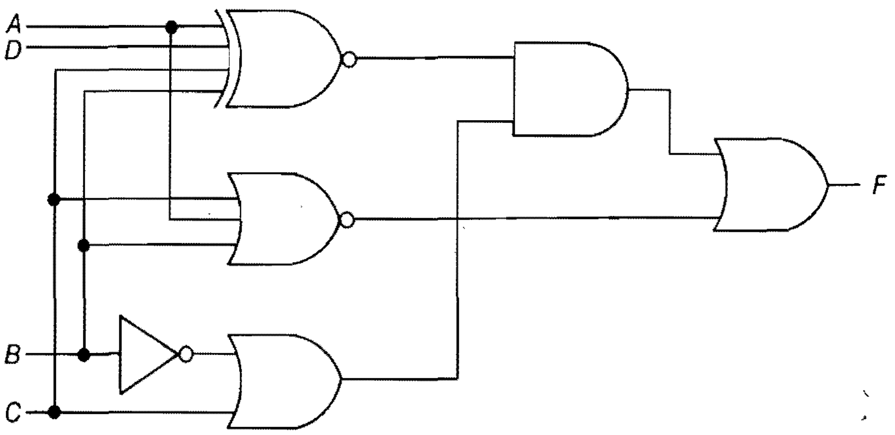
$$Y = \bar{A}\bar{B}\bar{C}D$$
$$X = \bar{B} + C$$

Therefore,

$$F = (\bar{A} \oplus B \oplus C \oplus D) \cdot (\bar{B} + C) + \bar{A}\bar{B}\bar{C}$$

The resultant circuit is obtained as shown in Figure 4.40. This bridged circuit is certainly better than the circuit that could be obtained by using only

FIGURE 4.40



a K-map. If a designer was limited to using only a K-map, the function would have reduced instead to

$$F(A,B,C,D) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}D + \bar{B}\bar{C}D + ABCD + \bar{A}BC\bar{D} + A\bar{B}C\bar{D}$$

The bridge scheme is very general and it does not have to involve only X-OR functions. This technique can be used for generating a complex function when a like function of the same variables already exists (see Problem 16).

4.9 Summary

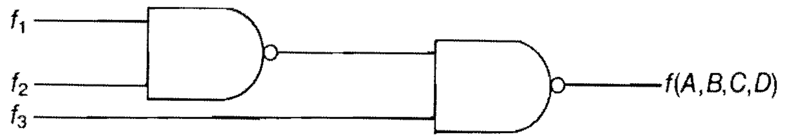
In this chapter various practical techniques were introduced for realizing combinational circuits. In particular, circuits using only NAND gates, only NOR gates, only MUXs, only ROMs, only PLAs, and only PALs were discussed. In addition, the bridging technique was introduced to handle functions that are otherwise not reducible.

Problems

1. Obtain the circuit for the following functions using only NAND gates:
   a. $f(A,B,C,D) = \Sigma m(1,4,10,11,13,15)$
   b. $f(A,B,C,D) = \Sigma m(1,3,4,9,10,13)$
   c. $f(A,B,C,D) = \Sigma m(1,8\text{–}10,15)$
   d. $f(A,B,C,D,E) = \Sigma m(1,3\text{–}7,11,14\text{–}17,22,24\text{–}27,30)$
   e. $f(A,B,C,D,E) = \Sigma m(1,8\text{–}10,13\text{–}17,21,25\text{–}27,30,31)$
2. Obtain NOR circuits for the functions of Problem 1.
3. a. Use a single level of 1-of-8 MUXs and a few assorted gates (if needed) to obtain a combinational circuit for each of the functions of Problem 1.
   b. For each of your solutions, complement one variable and rearrange the inputs so that the function is still correct.
4. Using 1-of-4 MUXs, obtain a two-level MUX circuit for each of the functions of Problem 1.
5. Obtain the circuit for the function $f(X,Y,Z,U,V) = \Sigma m(0,1,6,7,9,12,13,15,18,20,22,24\text{–}26,28)$ using two levels of 1-of-4 MUXs and a few assorted gates.
6. Use bridging to implement the following functions using X-OR gates:
   a. $f(A,B,C,D) = \Sigma m(1,2,5,7,8,10,13,14)$
   b. $f(A,B,C,D) = \Sigma m(2,3,6,7,9,11,12,13)$
   c. $f(W,X,Y,Z) = \Sigma m(0,2,3,6\text{–}8,10,13)$
   d. $f(W,X,Y,Z) = \Sigma m(0,6,9,10,15)$
7. Implement the functions of Problem 6 using ROMs.
8. Implement the functions of Problem 6 using PLAs.

9. Given the function $f(A,B,C,D) = \Sigma m(0,4,9,10,11,12)$ and $f_1 = B \oplus D$, determine f_2 and f_3 for the circuit of Figure 4.P1.

FIGURE 4.P1



10. Use a ROM to design a binary-to-Gray code converter.
11. Draw the logic diagram of an 8 × 2 ROM that produces the full adder function as described in Chapter 1 (Table 1.4).
12. Use a ROM to achieve four-bit by four-bit binary multiplication.
13. Design a four-input network that squares each of the binary inputs using (a) a ROM, (b) a PLA, and (c) a PAL. Assume that both input and output are unsigned.
14. Design a five-input logic network that finds the 2's complement of a positive number using (a) a ROM, (b) a PLA, and (c) a PAL.
15. Design a network that accepts trigonometric angles in degrees (between 0° and 10° in steps of 1°) and gives out the corresponding tangent value correct up to five significant places. Use (a) a ROM and (b) a PLA.
16. Consider the truth table for the full adder of Table 1.4. Bridge the carry-out function with $A_i \oplus B_i$, where A_i and B_i are the augend and addend, respectively.

Suggested Readings

Bartee, T. C. "Computer design of multiple output logical networks." *IRE Trans. Elect. Comp.* vol. EC-10 (1961): 21.

Cerny, E., and Marin, M. A. "A computer algorithm for the synthesis of memoryless logic circuits." *IEEE Trans. Comp.* vol. C-23 (1974): 455.

Cerny, E., and Marin, M. A. "An approach to unified methodology of combinational switching circuits," *IEEE Trans. Comp.* vol. C-26 (1977): 745.

Culliney, J. N.; Young, M. H.; Nakagawa, T.; and Muroga, S. "Results of the synthesis of optimal networks of AND and OR gates for four-variable switching functions." *IEEE Trans. Comp.* vol. C-27 (1979): 76.

Curtis, H. A. "Short-cut method of deriving nearly optimal arrays of NAND trees." *IEEE Trans. Comp.* vol. C-28 (1979): 521.

Davio, M. "Read-only memory implementation of discrete functions." *IEEE Trans. Comp.* vol. C-29 (1980): 931.

Dietmeyer, D. L., and Su, Y. H. "Logic design automation of fan-in limited NAND networks." *IEEE Trans. Comp.* vol. C-18 (1969): 11.

Ektare, A. B., and Mital, D. P. "Multiplexer logic circuit design using cubical complexes." *Elect. Lett.* vol. 16 (1980): 495.

Ektare, A. B., and Mital, D. P., "Probabilistic approach to multiplexer logic circuit design," *Elect. Lett.* vol. 16 (1980): 686.

Ellis, D. T. "A synthesis of combinational logic with NAND or NOR elements." *IEEE Trans. Elect. Comp.* vol. EC-14 (1965): 701.

Fleisher, H., and Maissel, L. "An introduction to array logic." *IBM J. Res. & Dev.* vol. 19 (1975): 98.

Jones, J. W. "Array logic macros." *IBM J. Res. & Dev.* vol. 19 (1975): 120.

Jullien, G. A. "Residue number scaling and other operations using ROM arrays." *IEEE Trans. Comp.* vol. C-27 (1978): 325.

Kambayashi, Y. "Logic design of programmable logic arrays." *IEEE Trans. Comp.* vol. C-28 (1979): 609.

Lai, H. C., and Muroga, S. "Minimum parallel binary adders with NOR(NAND) gates." *IEEE Trans. Comp.* vol. C-28 (1979): 648.

Li, H. F. "Variable selection in logic synthesis using multiplexers." *Int. J. Electron.* vol. 49 (1980): 185.

Liu, T. K.; Hohulin, K. R.; Shiau, L. E.; and Muroga, S. "Optimal one-bit full adders with different types of gates." *IEEE Trans. Comp.* vol. C-23 (1974): 63.

Logue, J. C.; Brickman, N. F.; Howley, F.; Jones, J. W.; and Wu, W. W. "Hardware implementation of a small system in programmable logic arrays." *IBM J. RES. & Dev.* vol. 19 (1975): 110

Lotfi, Z. M., and Tosser, A. J. "Systematic search for minimum synthesis of logical functions with multiplexers." *Int. J. Electron.* vol. 47 (1980): 569.

Nagle, H. T., Jr.; Carroll, B. D.; and Irwin, J. D. *An Introduction to Computer Logic.* Englewood Cliffs, N.J.: Prentice-Hall, 1975.

Nakamura, K. "Synthesis of gate-minimum multi-output two-level negative gate networks." *IEEE Trans. Comp.* vol. C-28 (1979): 768.

Papachristou, C. A. "An algorithm for optimal NAND cascade logic synthesis." *IEEE Trans. Comp.* vol. C-27 (1978): 1099.

Peatman, J. B. *Digital Hardware Design.* New York: McGraw-Hill, 1980.

Preparta, F. P. "On the design of universal Boolean functions." *IEEE Trans. Comp.* vol. C-20 (1971): 418.

Sasao, T. "Input variable assignment and output phase optimization of PLA's." *IEEE Trans. Comp.* vol. C-33 (1984): 879.

Sasao, T. "An algorithm to derive the complement of a binary function with multiple-valued inputs." *IEEE Trans. Comp.* vol. C-34 (1985): 131.

Shannon, C. E. "A symbolic analysis of relay and switching circuits." *Trans. AIEE.* vol. 57 (1938): 713.

Weinberger, A. "Device sharing in array logic." *IBM Tech. Disc. Bull.* vol. 19 (1976): 1357.

Weinberger, A. "High-speed programmable logic array adders." *IBM J. Res. & Dev.* vol. 23 (1979): 163.

Whitehead, D. G. "Algorithm for logic circuit synthesis by using multiplexers." *Elect. Lett.* vol. 13 (1977): 355.

Wood, R. "A high density programmable logic array chip." *IEEE Trans. Comp.* vol. C-28 (1979): 602.

Design of Combinational Circuits

CHAPTER FIVE

5.1 Introduction

A *combinational logic circuit*, as shown by the block diagram of Figure 5.1, is defined as a combination of logic devices whose output is a function of the present values of the input variables and independent of the past values. After propagation time through the circuit, input variable changes cause output changes that are dependent only on the present input values.

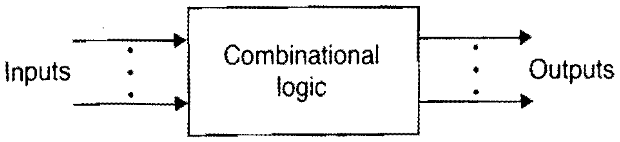


FIGURE 5.1 **Block Diagram of a Combinational Network.**

In Chapters 1, 2, and 4 we introduced the necessary tools to design combinational logic circuits. The design algorithm leading to the realization of a complex combinational circuit consists of the following essential steps:

1. The complex logic problem is intuitively analyzed and decomposed into a set of smaller but nontrivial functional units.
2. The number and characteristics of both input and output variables for each of the functional units are identified.
3. A truth table for each of the functional units is determined.
4. The output functions for each of the functional units are simplified using one of the minimization schemes covered in previous chapters.
5. The circuits corresponding to each of the functional units are assembled and tested individually and then connected to form the desired complex function.

In practice, the designer would have to consider various practical limitations such as the number of logic gates, interconnections, gate inputs, fan-out, and the length of propagation delay. In the not too distant past the only option available to the designer was to assemble the entire logic circuit with a sack full of SSI chips. However, at this time we have better alternatives because many more complex logic circuits are available in IC form. The only limitation to the use of either the MSI or the LSI is that we may not be able to locate a device that exactly meets our requirements. In that event the designer must modify the standard device by externally combining it with other SSI or MSI chips.

In this chapter applications of these combinational logic design tools will be considered. The design and application of MSI devices, including adders, subtracters, decoders, encoders, and error-control logic, is presented. These devices play an important part in the development of more advanced digital systems. The application of the MSI devices considered is not governed by a set of design procedures as well defined as those procedures for individual logic gates. Experience and intuition (horse sense) become important. There is no substitute for understanding exactly what the MSI devices can do. After studying this chapter, you should be able to:

- ○ Break a complex design into manageable subunits;
- ○ Design individual subunits and be able to cascade them together;
- ○ Understand the working principles and design process of various combinational binary adders and/or subtracters;
- ○ Understand the working principles and design process of various code converters;
- ○ Understand the working principles and design of BCD arithmetic circuits;
- ○ Understand the working principles and design of various decoders and encoders;
- ○ Understand the working principles and design of various error-correcting circuits.

5.2 Binary Adders

Most arithmetic operations are reducible to simple addition or subtraction processes that can be performed repetitively for more complex operations such as multiplication and division. If the addition circuit has no carry-in, the addition of the least significant bits involves only two operands, the addend and augend. The addition of the remaining bits, however, requires the carry-in from the addition of the previous column.

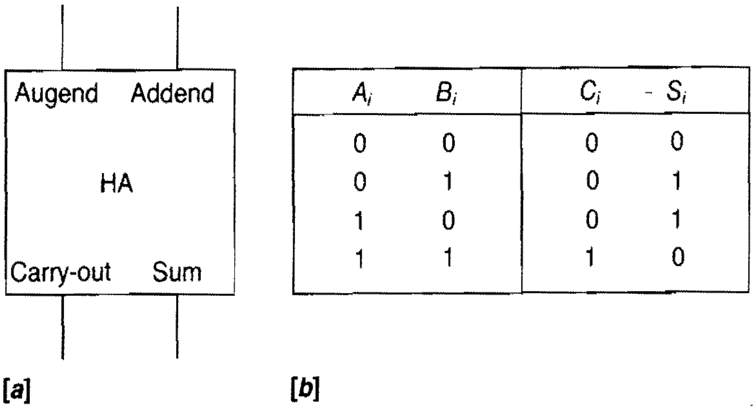
A multi-bit adder can be realized in various ways, each having different speed and cost characteristics. A two-level network would obviously prove to be the fastest. However, this network would require a large number of gates and gate inputs. It would be necessary to have 2^{2n} NAND gates of $2n + 1$ inputs and one NAND gate of 2^{2n} inputs to add two n-bit numbers. This number of gates and inputs is quite significant for even small values of n. The alternative to this expensive design is quite straightforward. It follows directly from our observation of the algorithm of an n-bit add operation. Irrespective of the number of bits, the process of adding augend and addend is identical at each of the columns except at the least significant position. The design of a parallel n-bit addition circuit, therefore, is accomplished by designing a total of n single-bit addition circuits. In order to allow n single-bit adders to be connected together to form an n-bit adder, the single-bit adder stages need to be full adders (adders with three inputs). The least significant carry-in is tied to a 0, and each of the remaining carry-in inputs is tied to the carry-out of the previous single-bit addition.

As an introduction to adder design, we shall first consider a half adder (adder with only two inputs, addend and augend). The resultant circuit will have application in the design of a full adder. In fact, an n-bit adder circuit using $n-1$ full adders and one half adder also can be designed.

5.2.1 Half Adder (HA)

The *half adder* (*HA*) unit is a simple multiple-output combinational circuit used for adding two bits without a carry-in. The truth table for the two inputs, A_i and B_i, and the output sum, S_i, and carry-out, C_i, are shown in Figure 5.2.

FIGURE 5.2 **Half Adder: [*a*] Block Diagram and [*b*] Truth Table.**



A_i	B_i	C_i	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Using a Karnaugh map, the equations for the sum, S_i, and carry, C_i, are as follows:

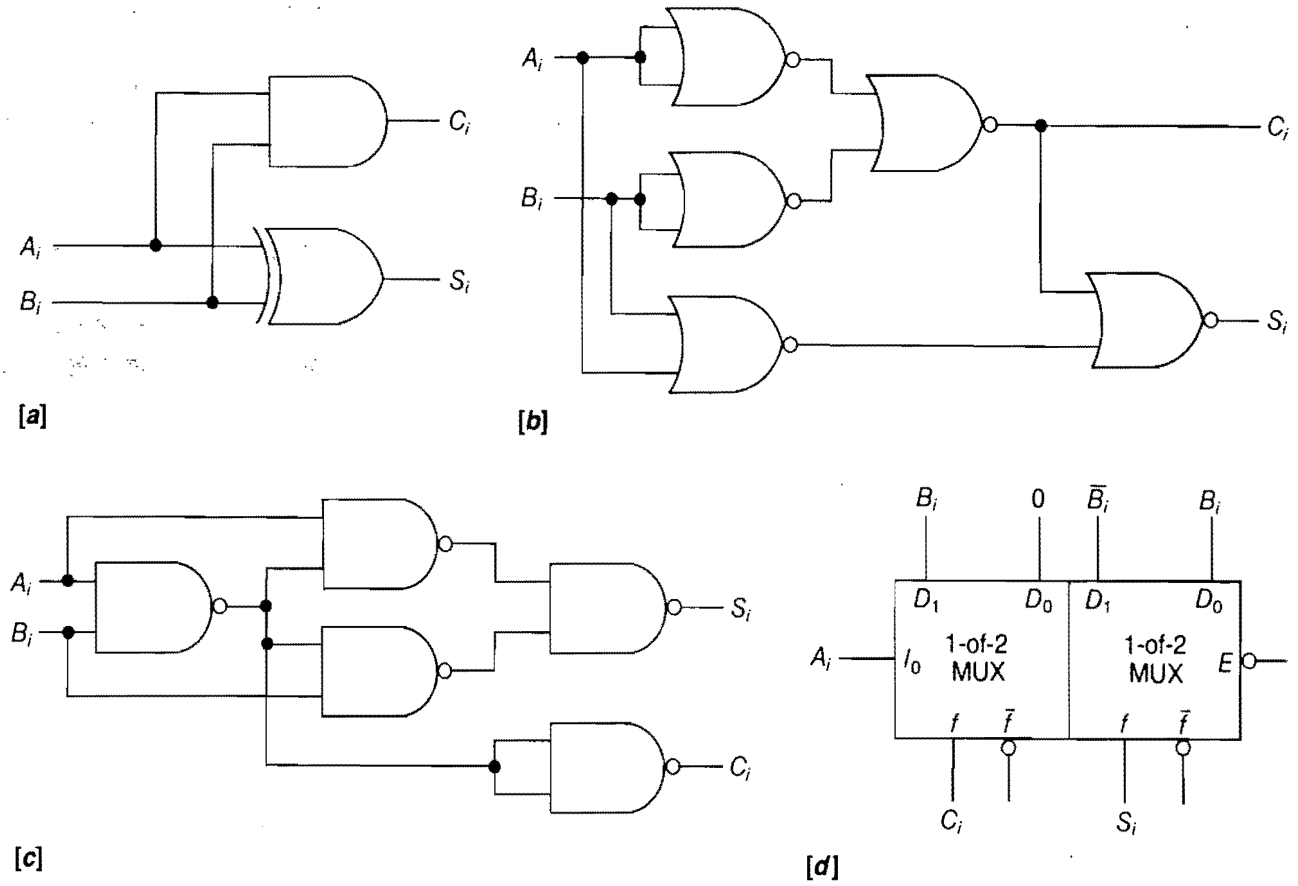
$$S_i = A_i\bar{B}_i + \bar{A}_iB_i = A_i \oplus B_i \qquad [5.1]$$

$$C_i = A_iB_i \qquad [5.2]$$

There are several ways to implement these functions. Figure 5.3

FIGURE 5.3 **HA Circuit: [*a*] Using AND and X-OR Gates, [*b*] Using NOR Gates, [*c*] Using NAND Gates, and [*d*] Using MUXs.**

illustrates four ways to implement the HA functions. Circuits using [*a*] an AND and an X-OR gate, [*b*] NOR gates, [*c*] NAND gates, and [*d*] multiplexers are shown.



5.2.2 Full Adder (FA)

A *full adder* (*FA*) is a three-input, two-output logic circuit that adds two binary digits, A_i and B_i, and a carry-in from the $i - 1$ bit position, C_{i-1}. The block diagram and the corresponding truth table are shown in Figures 5.4[*a–b*]. The K-maps for the sum bit, S_i, and the carry-out, C_i, are constructed from the truth table and shown in Figure 5.4[*c*]. The equations for the sum and carry-out can then be obtained from the K-maps as follows:

$$\begin{aligned} S_i(A_i,B_i,C_{i-1}) &= \overline{A}_i\overline{B}_iC_{i-1} + \overline{A}_iB_i\overline{C}_{i-1} + A_i\overline{B}_i\overline{C}_{i-1} \\ &\quad + A_iB_iC_{i-1} \end{aligned} \qquad [5.3]$$

$$= \overline{\overline{\overline{A}_i + \overline{B}_i + C_{i-1}} + \overline{\overline{A}_i + B_i + \overline{C}_{i-1}} + \overline{A_i + \overline{B}_i + \overline{C}_{i-1}} + \overline{A_i + B_i + C_{i-1}}} \qquad [5.4]$$

and

$$C_i(A_i,B_i,C_{i-1}) = A_iB_i + A_iC_{i-1} + B_iC_{i-1} \qquad [5.5]$$

$$= \overline{\overline{A_i + B_i} + \overline{A_i + C_{i-1}} + \overline{B_i + C_{i-1}}} \qquad [5.6]$$

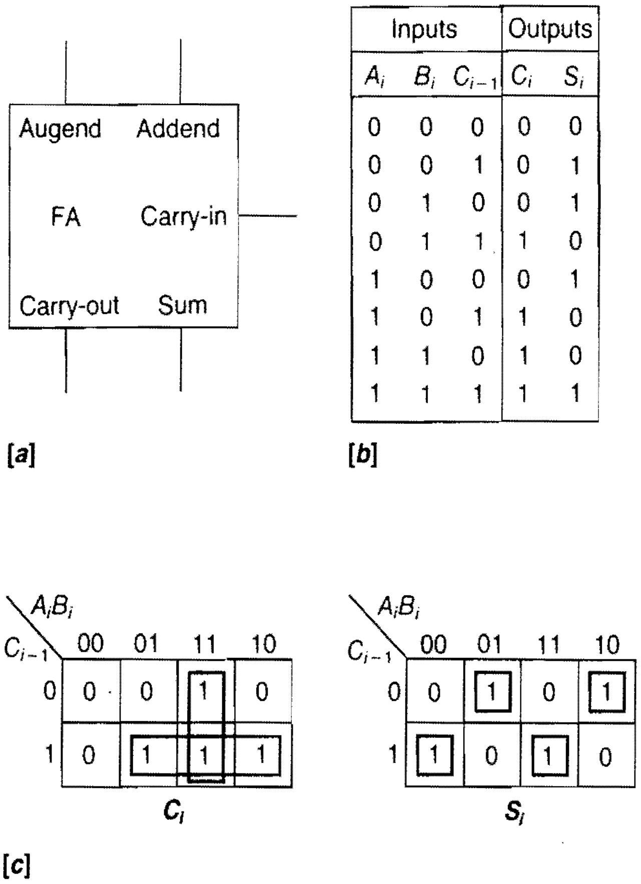


FIGURE 5.4 **Full Adder: [*a*] Block Diagram, [*b*] Truth Table, and [*c*] K-Maps for the Carry-Out and the Sum.**

Inputs			Outputs	
A_i	B_i	C_{i-1}	C_i	S_i
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

[*b*]

$C_{i-1} \backslash A_iB_i$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

C_i

$C_{i-1} \backslash A_iB_i$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

S_i

[*c*]

There are several ways to implement the FA equations. The direct implementation of Equations [5.3] and [5.5] leads to FA circuits using only NAND gates. If Equations [5.4] and [5.6] are used, the equivalent NOR circuits may be obtained. Figures 5.5[*a–b*] respectively show the typical FA circuits using only NAND and only NOR gates. Each of these circuits requires a total of 12 gates and 31 gate inputs. However, by making use of the bridging technique, the number of gates can be reduced.

Note that the K-map for the sum output is irreducible, producing a cumbersome circuit. Note also that the K-map for S_i is that of a three-input X-OR function. A review of Figure 4.36 indicates that the sum equation may be reduced to

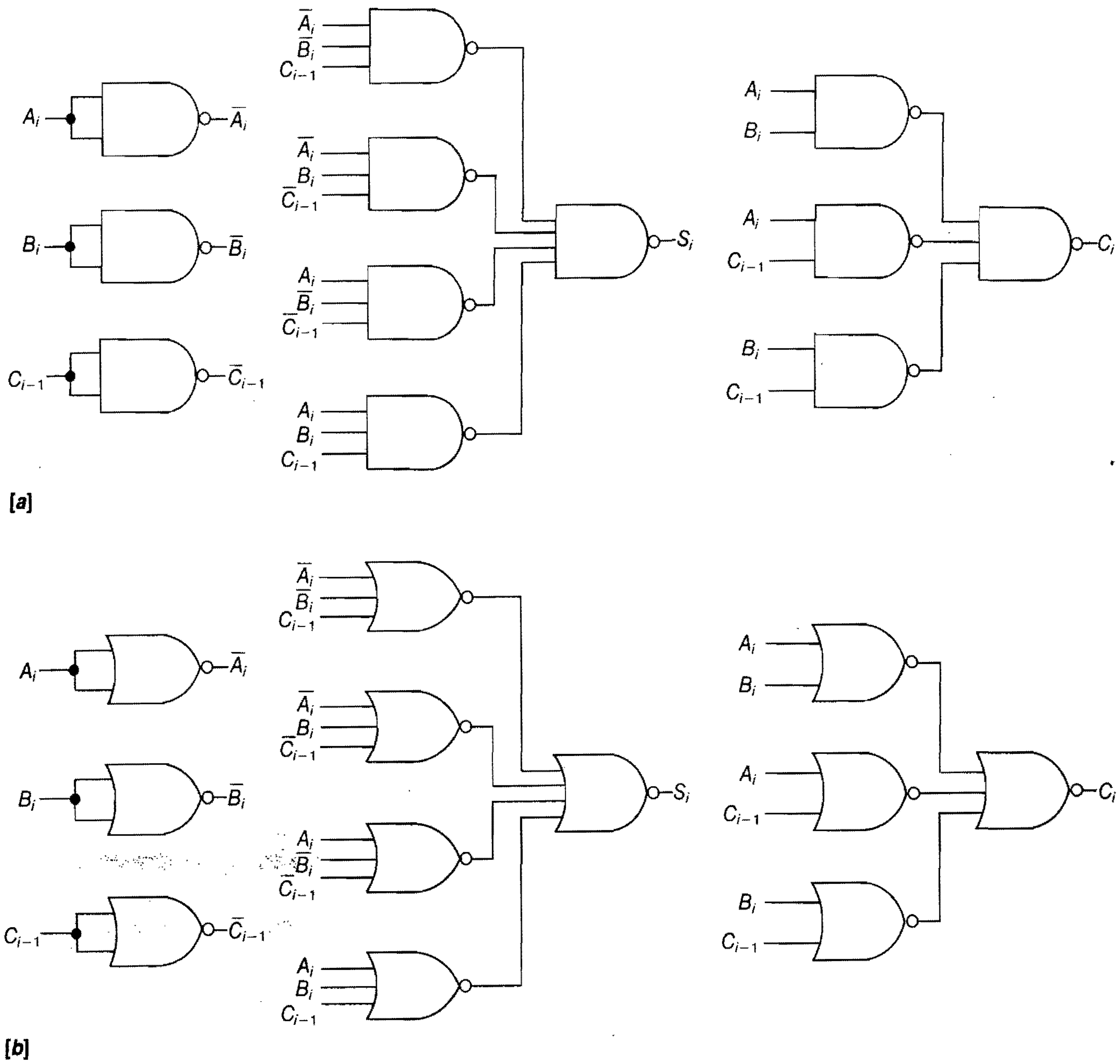
$$S_i = A_i \oplus B_i \oplus C_{i-1} \qquad [5.7]$$

Examining the carry-out K-map reveals that it could be bridged with an X-OR function as follows (see Chapter 4, Problem 16):

$$C_i = (A_i \oplus B_i)C_{i-1} + A_iB_i \qquad [5.8]$$

Now recall that $A_i \oplus B_i$ is the sum output and A_iB_i is the carry-

FIGURE 5.5 FA Circuit: [*a*] Using Only NAND Gates and [*b*] Using Only NOR Gates.



out for an HA. Thus the HAs designed in the last section may be used for realizing an FA. Expressing the FA equations in terms of the HA equations, Equations [5.1] and [5.2], gives

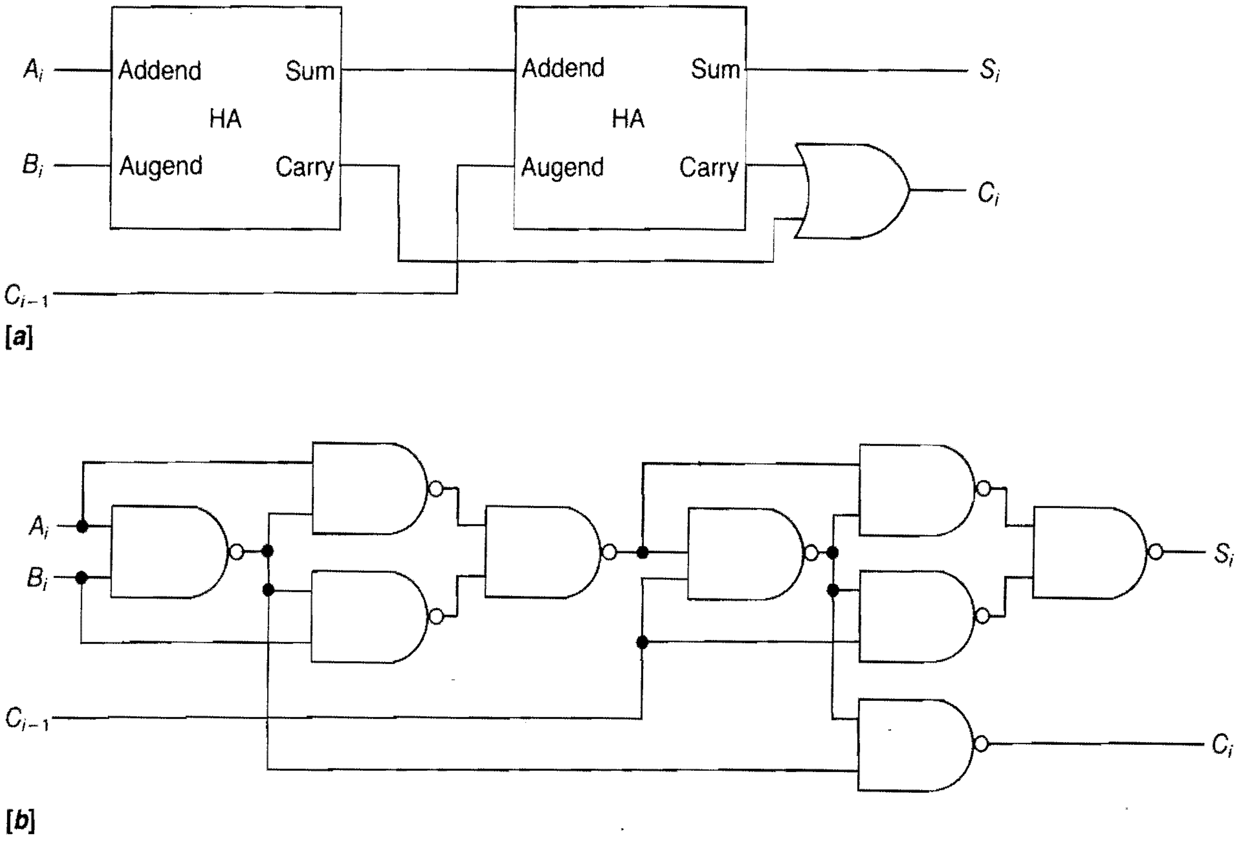
$$S_i = S_{i(\mathrm{HA})} \oplus C_{i-1} \qquad [5.9]$$

$$C_i = S_{i(\mathrm{HA})} \cdot C_{i-1} + C_{i(\mathrm{HA})} \qquad [5.10]$$

where $S_{i(\mathrm{HA})}$ and $C_{i(\mathrm{HA})}$ are the sum and carry-out of the HA. Note also that Equation [5.9] involves an X-OR operation between the carry-in and the HA sum. If the sum output of the HA and the carry-in are fed into a second HA, the final sum output will be the

FA sum, S_i. In addition, if the carry-outs from both of the HAs are ORed together, the FA carry-out, C_i, is obtained. Figure 5.6[*a*] shows the FA circuit using two HAs and an OR gate. The circuit corresponding to Equations [5.7] and [5.8] also may be implemented using only NAND gates or only NOR gates. The resultant NAND equivalent circuit is shown in Figure 5.6[*b*]. This NAND circuit requires only nine NAND gates and a total of 18 gate inputs. We have designed an FA circuit using three fewer gates and 13 fewer gate inputs than would be necessary had we not made use of the X-OR K-map structure and bridging.

FIGURE 5.6 **FA Circuits: [*a*] Using HAs and [*b*] Using Only NAND Gates.**



We have completed the design of half and full adders. These devices are also commercially available in the form of MSI devices. Four FAs usually are connected and are commercially available in an MSI four-bit adder IC. Using a commercially available four-bit adder, two quantities, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, can be added. The resultant sum is $S_3S_2S_1S_0$, and the carry-out is from the A_3, B_3, and C_2 addition. Figure 5.7 demonstrates the connection of *n* FAs to

FIGURE 5.7 **n-Bit Parallel Adder Circuit: [*a*] Using n FAs and [*b*] Using $n-1$ FAs and One HA.**

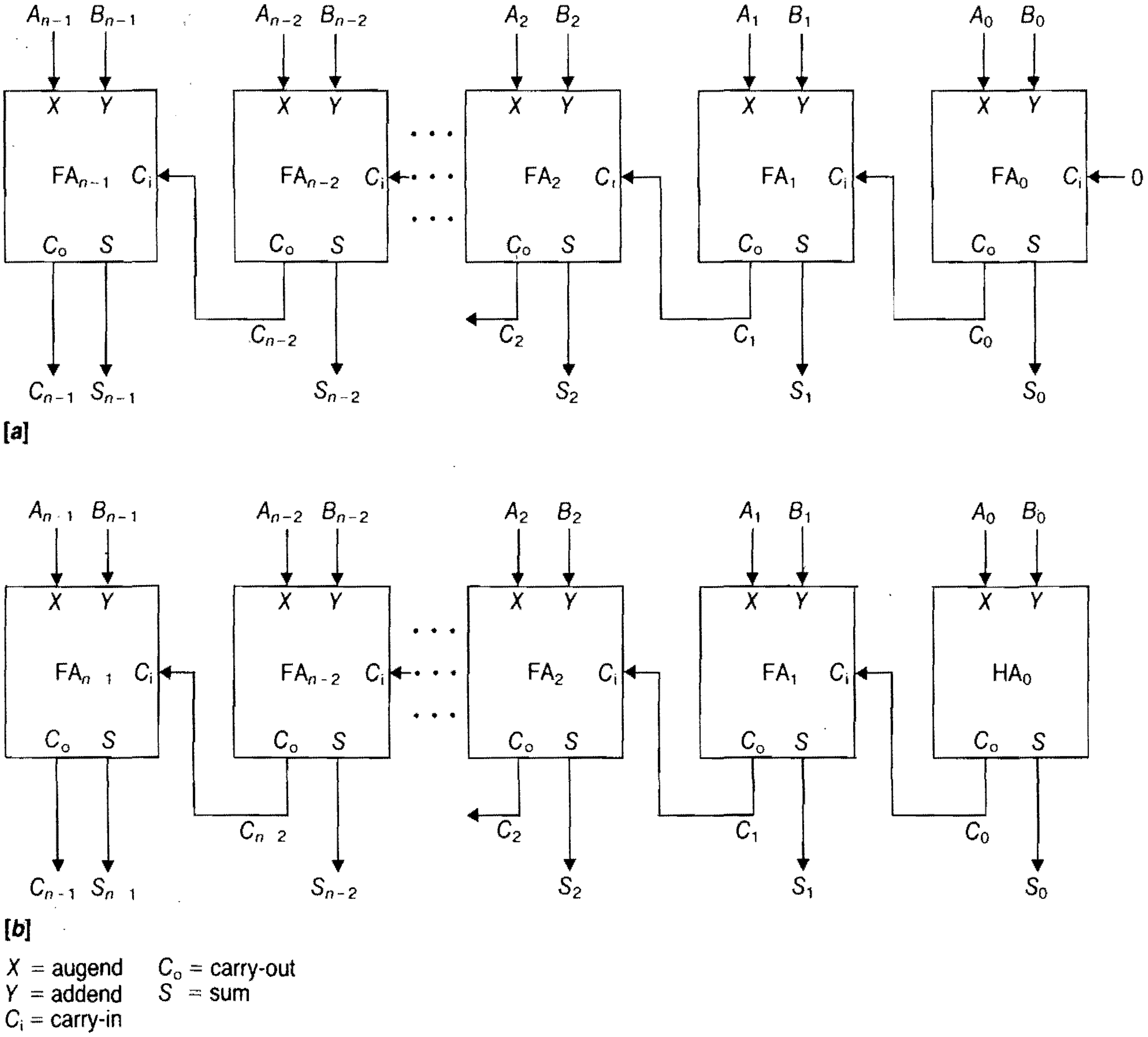


FIGURE 5.8 **Multiplication of Two Three-Bit Numbers.**

$$
\begin{array}{cccccc}
 & & & B_2 & B_1 & B_0 \\
 & & & A_2 & A_1 & A_0 \\
\cline{4-6}
 & & & A_0B_2 & A_0B_1 & A_0B_0 \\
 & & A_1B_2 & A_1B_1 & A_1B_0 & \\
 & A_2B_2 & A_2B_1 & A_2B_0 & & \\
\hline
P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

make an n-bit adder. The delay of this n-bit ripple adder is $n\Delta$ where Δ is the propagation delay of the carry-out of a single FA. This delay is accumulated when the carry into the multi-bit adder has to propagate through all of the FAs to get to the final carry-out. Consequently, this delay becomes more and more significant as n becomes larger.

Multiplication of binary numbers makes use of addition just as multiplication of decimal numbers does. The multiplication of two three-bit numbers, $A = A_2A_1A_0$ and $B = B_2B_1B_0$, is symbolically obtained as shown in Figure 5.8, where $P_5P_4P_3P_2P_1P_0$ forms the product. Note that for the multiplication of two n-bit numbers, the product has the possibility of $2n$ bits.

Later in the text sequential design techniques will be presented that will allow designing a multiplier that uses a repetitive algorithm for multiplication. It is possible, however, to design a combinational circuit using FAs that will perform the multiplication of two binary numbers by performing the sum of the three partial products of two numbers. This combinational circuit must be able to add columns of bits. Example 5.1 will demonstrate how FAs can be used to do similar functions.

EXAMPLE 5.1

Design a circuit using FAs that may be used for adding a column of six single-bit numbers.

SOLUTION

Let the column of numbers be a_0, a_1, a_2, a_3, a_4, and a_5. The sum if all were 1 would add up to 110, so there will be three outputs: S_2, S_1, and S_0. However, an FA may add up to only three single bits. Two FAs may be used to obtain two partial sums of the six bits as follows:

$$\begin{array}{rr} & a_0 \\ & a_1 \\ & a_2 \\ \hline x_1 & y_1 \end{array} \qquad \begin{array}{rr} & a_3 \\ & a_4 \\ & a_5 \\ \hline x_2 & y_2 \end{array}$$

where x_1 and x_2 are the carry-outs and y_1 and y_2 are the respective sums. As a next step the least significant bits, y_1 and y_2, may be added as follows:

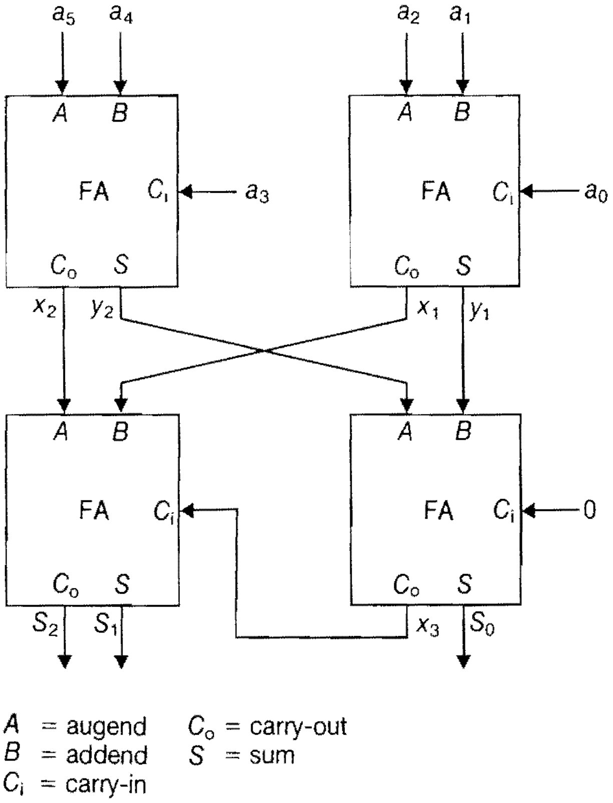
$$\begin{array}{rr} & y_1 \\ & y_2 \\ & 0 \\ \hline x_3 & S_0 \end{array}$$

where x_3 and S_0 are the carry-out and the sum, respectively. Finally, x_1, x_2, and x_3 could be fed into a fourth FA to yield the carry-out, S_2, and the sum, S_1, as follows:

$$\begin{array}{rr} & x_1 \\ & x_2 \\ & x_3 \\ \hline S_2 & S_1 \end{array}$$

The complete circuit, therefore, would require four FAs. The resulting circuit is obtained as shown in Figure 5.9.

FIGURE 5.9



Another example of the application of adders in digital systems is given in Example 5.2. This example also demonstrates how a computer that is designed to handle binary quantities of *n* bits can perform operations on 2*n*-bit quantities. In programming, such an operation is commonly called a ***multiple-precision operation.***

EXAMPLE 5.2

Use four-bit multipliers and four-bit binary adders to design a circuit for multiplying two eight-bit numbers. The block diagrams of the multiplier and adder units are provided in Figure 5.10. The four-bit multipliers are assumed to be ROMs. The two four-bit quantities to be multiplied make up an eight-bit address. Each ROM storage location is the eight-bit product of the two four-bit quantities that make up its address.

SOLUTION

FIGURE 5.10

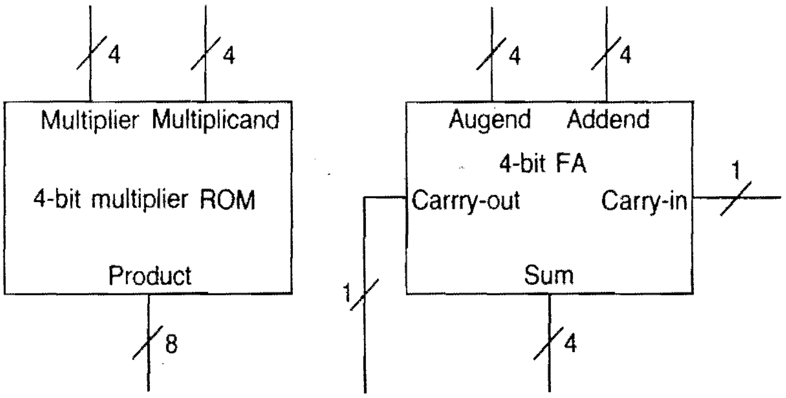
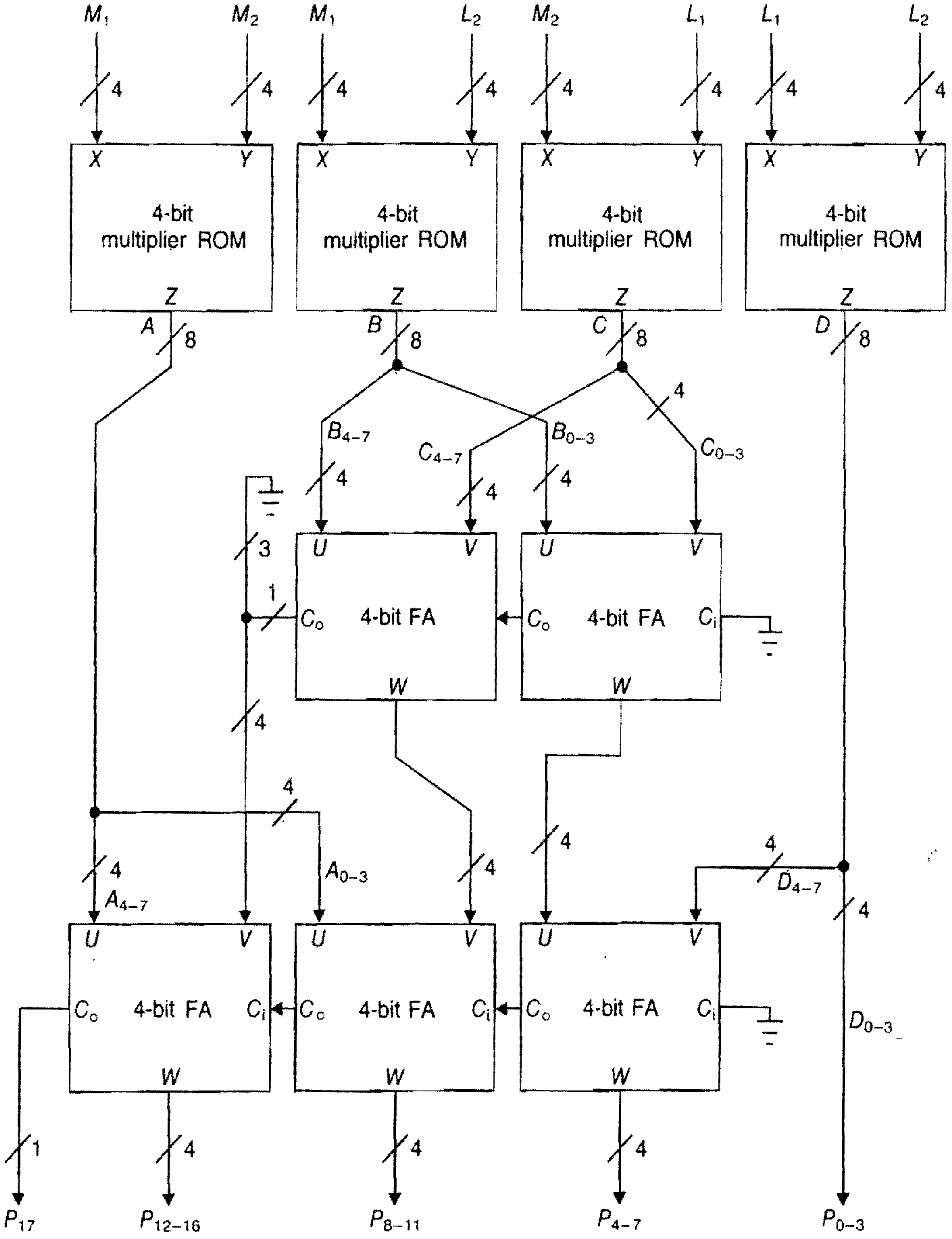


FIGURE 5.11



X = multiplier V = addend
Y = multiplicand W = sum
Z = product C_i = carry-in
U = augend C_o = carry-out

A good approach to any design problem is to break the given problem into several simpler problems. This procedure is necessary in this example in order to make the problem fit the devices that are provided. Consider the multiplication of two eight-bit numbers, X_1 and X_2, each consisting of a least significant four bits, L_i, and a most significant four bits, M_i. The eight-bit number can then be expressed as the sum of the two four-bit parts:

$$X_i = 2^4(M_i) + L_i$$

where M_i is shifted to the left four places (multiplied by 2^4) before being added to L_i. The product, P, may now be expressed as

$$\begin{aligned} P &= [2^4(M_1) + L_1][2^4(M_2) + L_2] \\ &= 2^8(M_1M_2) + 2^4(M_1L_2 + M_2L_1) + (L_1L_2) \end{aligned}$$

Each of these four partial products may be obtained using four four-bit multiplier units. The four multiplier units would respectively have (a) M_1 and M_2, (b) M_1 and L_2, (c) M_2 and L_1, and (d) L_1 and L_2 as inputs. This configuration would result in a total of four eight-bit outputs: A, B, C, and D, respectively. Since two eight-bit quantities are being multiplied, a 16-bit product is expected. Note also that D should be added to the sum of B and C that have been shifted to the left four places, and this in turn should be added to A that have been shifted to the left eight places. A network of six four-bit FAs may be employed, as shown in Figure 5.11, to obtain the 16-bit sum of the shifted partial products. The final product is obtained by adding the partial products. Care must be taken to connect the partial products at the correct bit positions relative to their power of 2. The shifting is implicit in the interconnection pattern of the adder modules.

Note that a single ROM for multiplying two eight-bit numbers would require a total of 1,048,576 bits, that is, 16 bits of address and 16 bits in each location, to store the 16-bit product, or $2^{16} \times 16$ bits. Each ROM that we used in this example had a size of only 2048 bits, giving a total of 8192 bits. This design would probably be less expensive but would be slower due to the time required to perform the additions. Such time and money trade-offs will be a typical design decision that must be made by every engineer.

5.3 Binary Subtracters

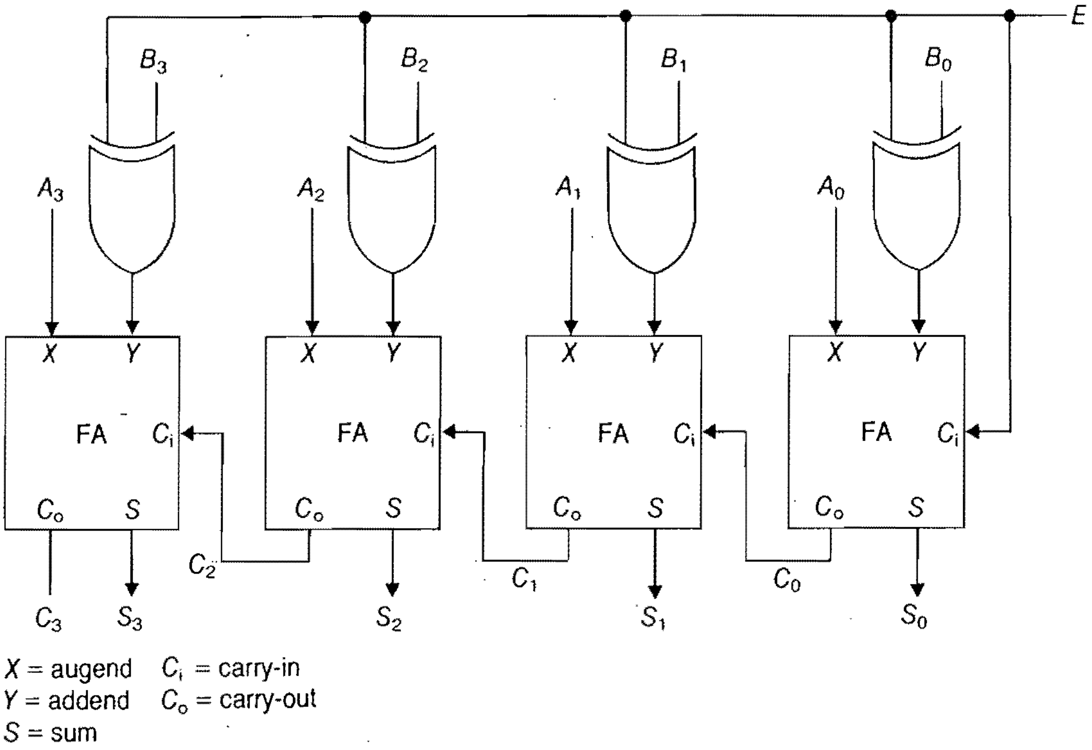
Many arithmetic circuits also require a unit for subtraction. A subtracter circuit could be designed from scratch. However, recall from Chapter 1 that subtraction also is possible by adding the complement of the subtrahend to the minuend. Consequently, rather than designing a straightforward subtracter, a multi-bit subtracter can be made using complement arithmetic. This design would involve the use of a multi-bit parallel adder circuit that is fed with the complemented subtrahend and the minuend.

In order to accommodate the multi-bit parallel adder to the requirement of our present design, certain modification is necessary for complementing the subtrahend. If each bit of the subtrahend is individually complemented, the corresponding 1's complement will be obtained. A 2's complement could be formed by adding a one to the LSB of the corresponding 1's complement, that is, by making the carry-in to the LSB position of the adder a 1. To perform the complementing of each bit, we can use an X-OR gate. An X-OR has a useful characteristic that makes it perform as a programmable inverter. Consider a two-input X-OR gate for which one input is a bit of the subtrahend and the other input is tied to a select line, E. When $E = 1$, the output will be the complement of the subtrahend bit. When $E = 0$, the output will be the uncomplemented bit, since from the X-OR truth table,

$$1 \oplus Y = \overline{Y}$$
$$0 \oplus Y = Y$$

The property of an X-OR function provides the possibility of having both an adder and a subtracter out of the same circuit. Such a circuit is shown in Figure 5.12 where the enable, E, allows the circuit to add or to subtract by taking the 1's complement and adding a one to the LSB. When $E = 0$, the circuit adds with a carry-in of 0, and when $E = 1$, the circuit complements B and adds that to A with a carry-in of 1.

FIGURE 5.12 **Four-Bit Adder/ Subtracter Unit.**



EXAMPLE 5.3

Obtain a combinational circuit for realizing the 2's complement of a five-bit binary number.

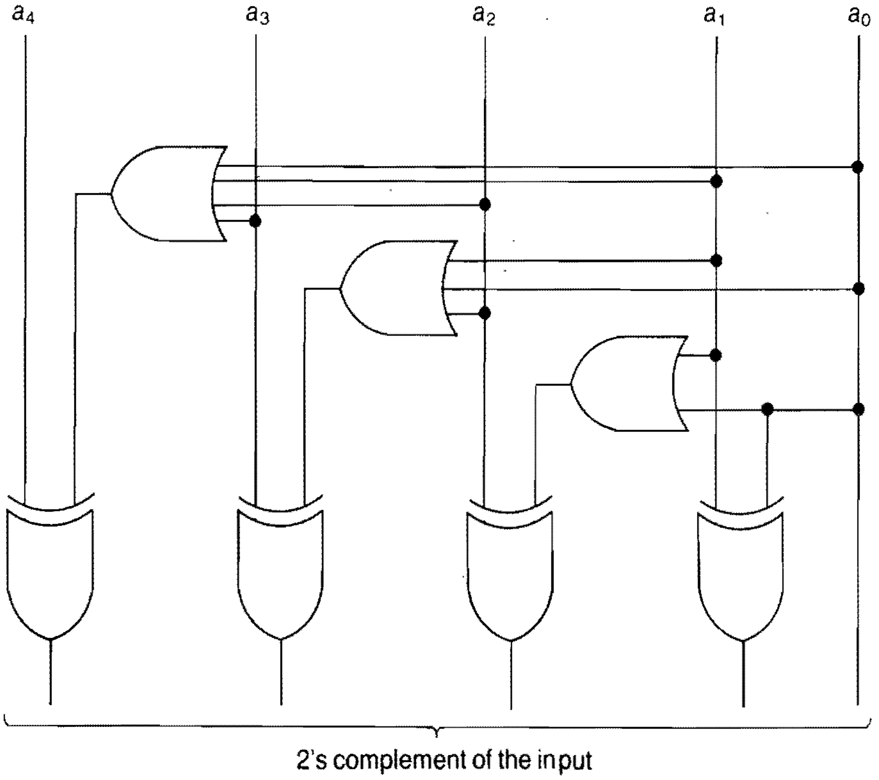
SOLUTION

From Section 1.4 we know that the LSB of a number always remains unchanged when obtaining the 2's complement. If the LSB, a_0, is 1, then the next higher bit, a_1, is inverted. An X-OR gate in the form of $a_0 \oplus a_1$

may be used to accomplish this. The next higher bit, a_2, is complemented if either a_1 or a_0 is 1. This line of argument could be carried out for all of the remaining bits. Such logical tests can be implemented by performing $a_0 + a_1 + \cdots + a_{m-1}$, which would indicate whether or not a 1 is present in the least significant m bits.

It is apparent, therefore, that X-OR and OR gates can be used to realize the necessary circuit for performing 2's complement of a number. The OR gates would perform tests and the X-OR gates would complement bits if necessary. The resulting circuit for a five-bit number $a_4a_3a_2a_1a_0$ is obtained as shown in Figure 5.13.

FIGURE 5.13

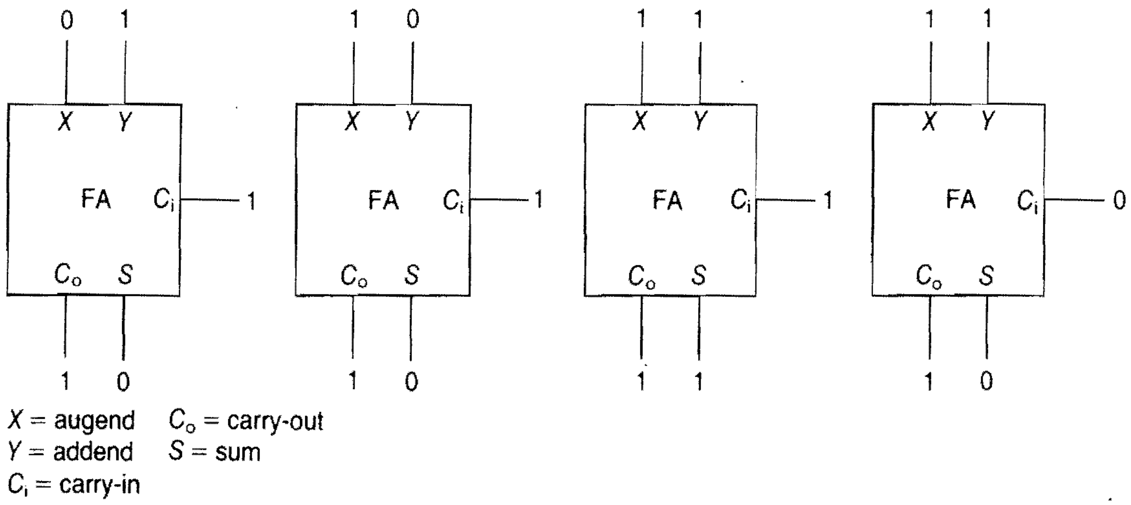


5.4 Carry Look-Ahead (CLA) Adders

The particular multi-bit parallel adder circuit developed in the previous section is sometimes referred to as a *ripple adder* because a carry from one unit of the adder may have to ripple through several units before the sum is obtained. Such ripple adders have also been used to form either 2's complement, or 1's complement sign-and-magnitude binary adder/subtracters. The performance of a ripple adder/subtracter, however, is limited by the time required for the carries to ripple through all of the stages of the circuit. For such devices the maximum delay is directly proportional to the number of FA units.

One particular method of speeding up the combinational addition process is known as *carry look-ahead* (*CLA*). In Figure 5.14 it may be seen that the carry-out is the same as the carry-in as long as one

FIGURE 5.14 **FA Configurations Resulting in a Carry-Out.**



of the other two inputs is a 1. Also, the carry-out is always a 1 independent of the carry-in when both of the other inputs are 1s, and 0 if both are 0. Consequently, two useful functions can be defined: the carry-propagate, P_i, and the carry-generate, G_i

$$P_i = A_i \oplus B_i \qquad [5.11]$$

$$G_i = A_i \cdot B_i \qquad [5.12]$$

where A_i and B_i are the addend and augend, respectively, of the *i*th full adder.

The FA equations, Equations [5.7] and [5.8], can then be rewritten as

$$S_i = P_i \oplus C_{i-1} \qquad [5.13]$$

$$C_i = G_i + P_iC_{i-1} \qquad [5.14]$$

For a four-bit adder the carries for the various stages are as follows:

$$C_0 = G_0 + P_0\mathrm{C}_{-1} \qquad [5.15]$$

$$\begin{aligned} C_1 &= G_1 + P_1C_0 \\ &= G_1 + P_1G_0 + P_1P_0C_{-1} \qquad [5.16] \end{aligned}$$

$$\begin{aligned} C_2 &= G_2 + P_2C_1 \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{-1} \qquad [5.17] \end{aligned}$$

and

$$\begin{aligned} C_3 &= G_3 + P_3C_2 \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_{-1} \qquad [5.18] \end{aligned}$$

while the sum bits are

$$S_0 = P_0 \oplus C_{-1} \tag{5.19}$$

$$S_1 = P_1 \oplus C_0 \tag{5.20}$$

$$S_2 = P_2 \oplus C_1 \tag{5.21}$$

and

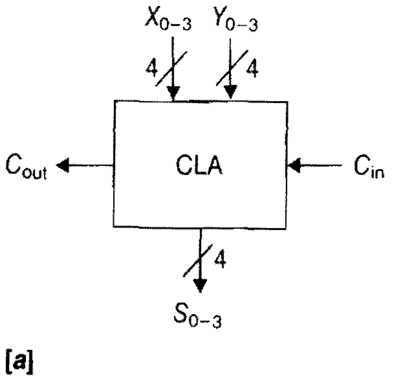
$$S_3 = P_3 \oplus C_2 \tag{5.22}$$

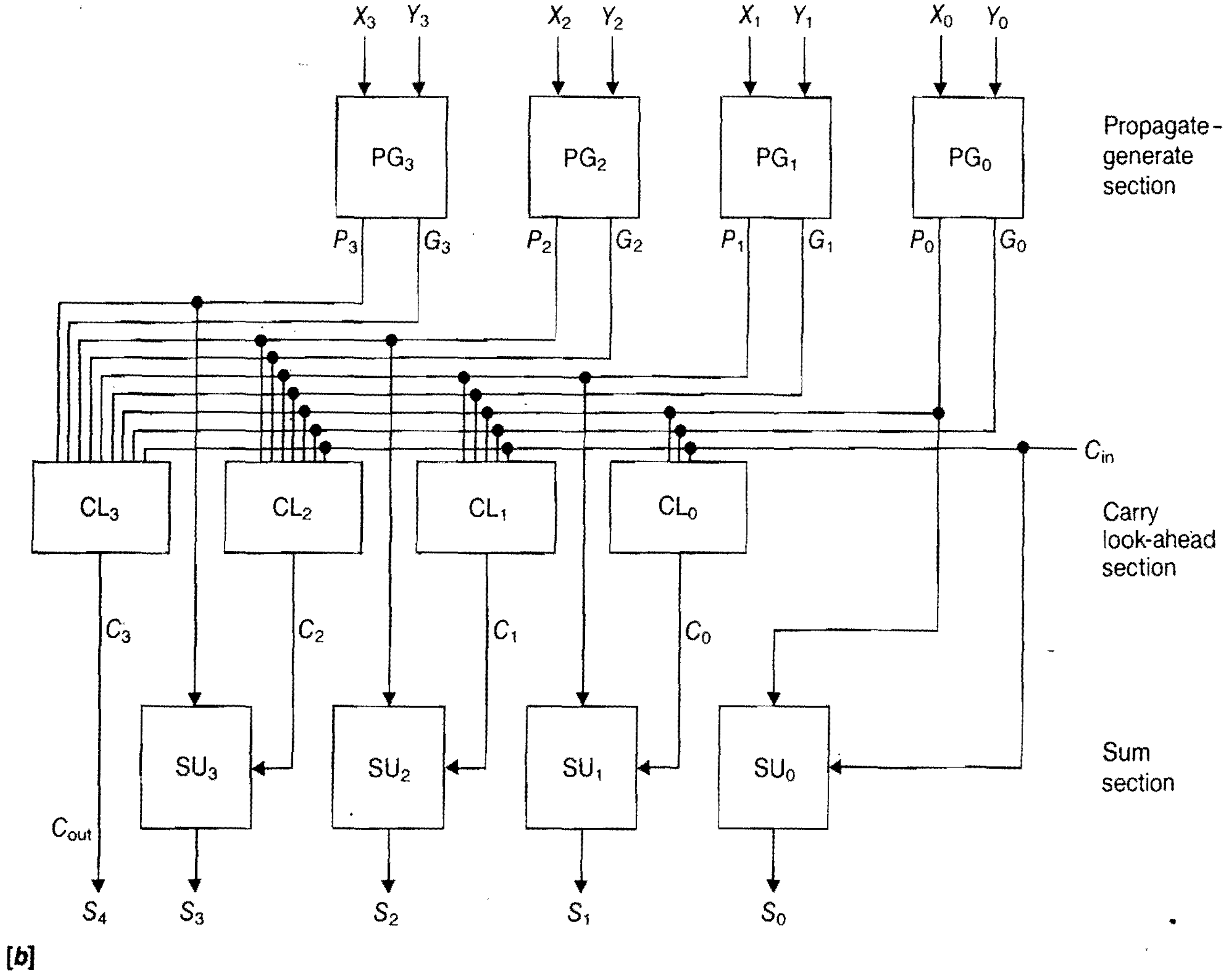
In Equations [5.15–18] the carry-out for each of the stages is dependent only on the initial carry, C_{-1}, and the corresponding propagate and generate functions. The equation for C_3 can be implemented with a two-gate level circuit. Since each of the generate and propagate functions can be expressed in terms of the two data bits, C_3 is available after two gate delays, resulting in a fast addition process. The block diagram of a four-bit CLA adder is shown in Figure 5.15. It consists of three sections, each having four subunits. The PG$_i$ section generates the carry-propagate and carry-generate functions. The CL$_i$ section intakes the outputs of the previous section and generates the carries. Finally, the SU$_i$ section generates the sum bits. However, the carry units are different for every bit, and their complexity increases as they move further away from the LSB.

The internal hardware for the four-bit CLA adder is shown in Figure 5.16. Each propagation-generation unit requires five NAND gates, each sum unit requires four NAND gates, and the n-bit carry section requires a total of $(n^2 + 5n)/2$ NAND gates. A four-bit CLA adder, therefore, requires a total of 54 NAND gates and involves a total of eight units of gate delay. In comparison, addition in a four-bit ripple adder requires 12 units of gate delay. The four-bit CLA, therefore, cuts down the time factor by about one-third. Similarly, a 64-bit CLA adder requires almost five times as many NAND gates as a 64-bit ripple adder, but reduces the propagation delay by a factor of 17. It follows, therefore, that the CLA adder will provide a faster addition time, especially when the number of bits is higher.

An inventory of the n-bit CLA adder reveals that the sum and the carry subunits require a total of $9n$ two-input NAND gates. The carry section, however, requires $(2n + 1)$ two-input NAND gates and $(n + 3 - m)$ m-input NAND gates for $3 \leq m \leq n + 1$. Therefore, from a practical standpoint CLA for too large n turns out to be quite problematic. It was seen in Chapter 3 that NAND gates with too many inputs are hard to come by. This limitation is compounded by the fact that the carry-in, C_{-1}, must drive a total of $n + 1$ gates. In addition, the propagate functions will be subjected to a fan-out requirement on the order of $(n + 1)^2/4$. All of these together result in a serious fan-in problem for n too large.

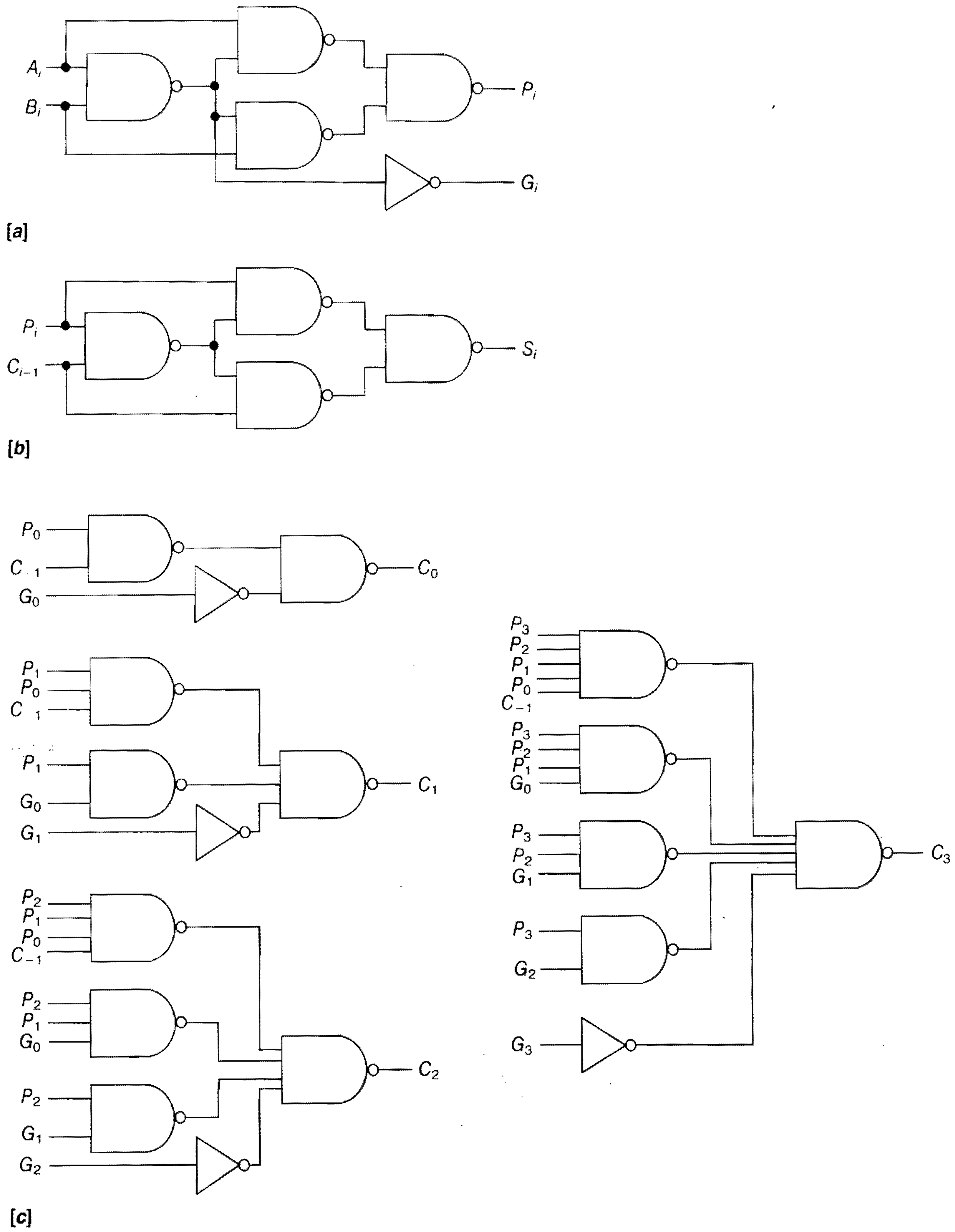
FIGURE 5.15 Four-Bit CLA Adder: [*a*] Block Diagram and [*b*] Internal Circuitry.





Commercially available four-bit adders perform an internal CLA for a four-bit add. These outputs are available and labeled as *P* for propagate and *G* for generate. These outputs when used with another MSI device called a *look-ahead carry generator* provide significant acceleration in the add operation, particularly for large num-

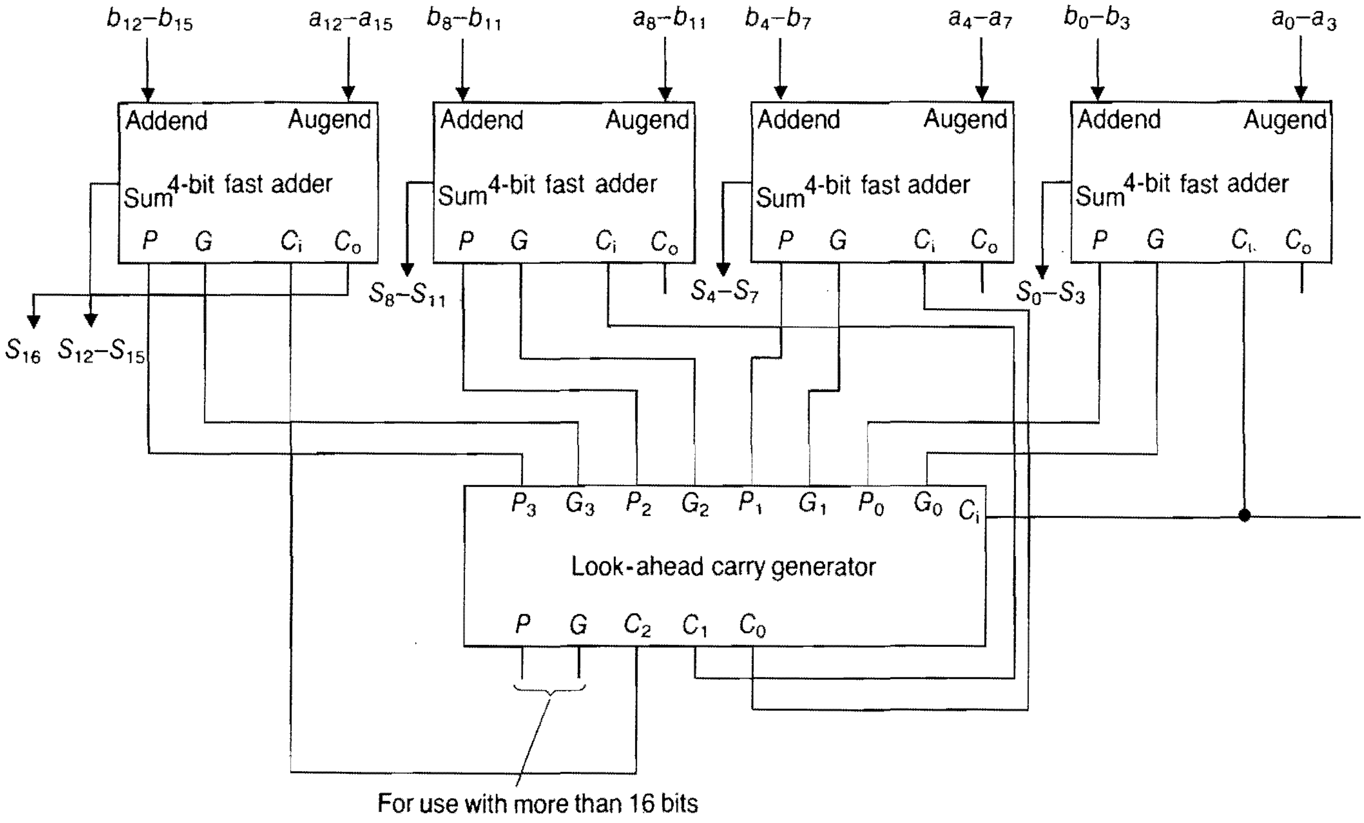
FIGURE 5.16 [*a*] Single Propagation-Generation Unit, [*b*] Single Sum Unit, and [*c*] Carry Units of a Four-Bit CLA Adder.



bers of bits. Figure 5.17 shows the connections for performing the addition of two 16-bit numbers with CLA.

Carry-propagate and carry-generate functions for more than four bits can be derived continuing the same process used for deriving C_3 and S_3. Circuit complexity makes such implementation impractical except for special-purpose, high-speed requirements.

FIGURE 5.17 **16-Bit Addition with CLA Modules.**



EXAMPLE 5.4

Design an eight-bit fast adder where the fan-in and fan-out problems are avoided by allowing the carries to ripple through after the addition of every four bits.

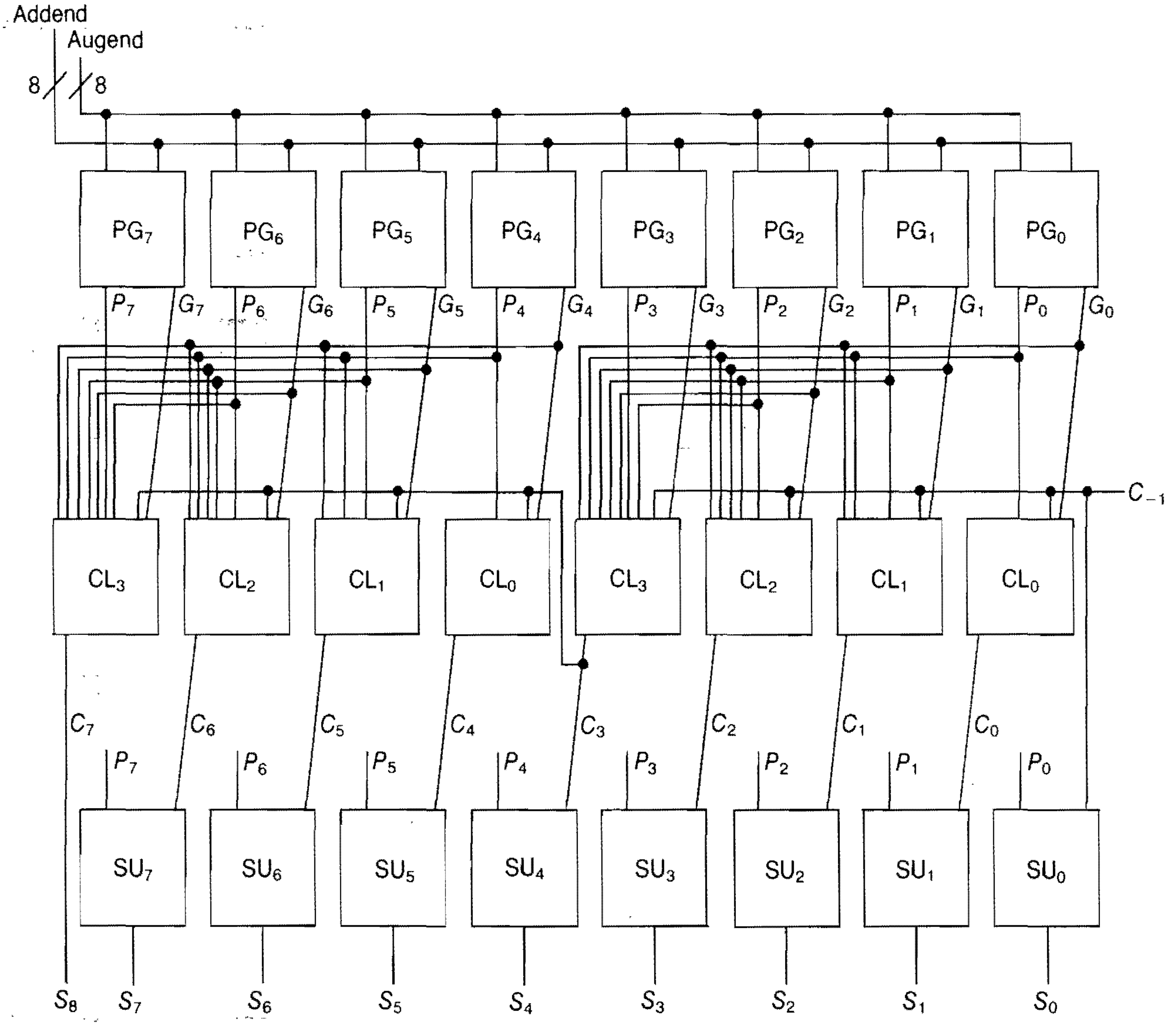
SOLUTION

The modified carry equations for an eight-bit fast adder are derived as follows:

$$C_0 = G_0 + C_{-1}P_0$$
$$C_1 = G_1 + G_0P_1 + C_{-1}P_0P_1$$
$$C_2 = G_2 + G_1P_2 + G_0P_1P_2 + C_{-1}P_0P_1P_2$$
$$C_3 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_{-1}P_0P_1P_2P_3$$
$$C_4 = G_4 + C_3P_4$$
$$C_5 = G_5 + G_4P_5 + C_3P_4P_5$$
$$C_6 = G_6 + G_5P_6 + G_4P_5P_6 + C_3P_4P_5P_6$$
$$C_7 = G_7 + G_6P_7 + G_5P_6P_7 + G_4P_5P_6P_7 + C_3P_4P_5P_6P_7$$

The first four equations are similar to Equations [5.15–18]. The last four equations have a similar form, but C_3 is treated as the carry-in to the fifth bit. The resultant circuit may now be obtained, as shown in Figure 5.18, by having two separate units for the carry section.

FIGURE 5.18



5.5 Code Converters

It was pointed out in Chapter 1 that many different binary codes exist that are used in various digital subsystems. Sometimes it is necessary to transfer data from one subsystem to another. *Code converter circuits* are required to convert one form of binary code to another. Many of these converters use combinational logic, and there are many that use sequential logic as well.

The following examples will illustrate the combinational techniques in the design of various code converters. We shall consider several conversion schemes: Gray-to-binary, binary-to-Gray, binary-to-BCD, and BCD-to-binary.

EXAMPLE 5.5

Design a circuit for converting a five-bit Gray code into its binary equivalent.

SOLUTION

The Gray code is a reflected binary code such that the Gray code for one number differs from the next number in only one bit position. In binary space the codes are one unit distance apart. The truth table of Figure 5.19 shows the corresponding Gray and binary numbers.

FIGURE 5.19

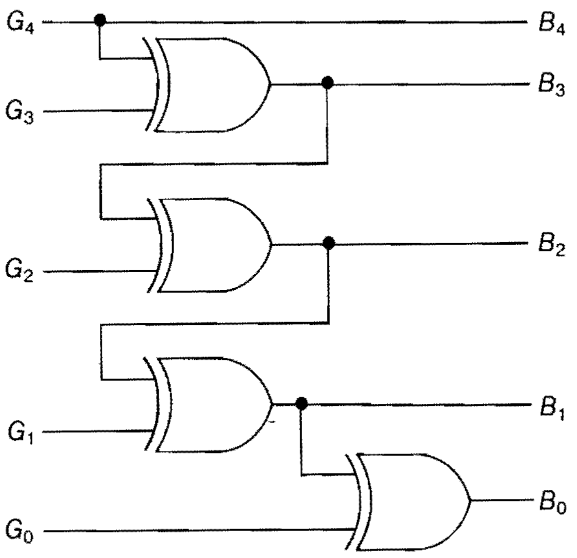
Gray					Binary				
G_4	G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0
0	0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	0	1	0	0
0	0	1	1	1	0	0	1	0	1
0	0	1	0	1	0	0	1	1	0
0	0	1	0	0	0	0	1	1	1
0	1	1	0	0	0	1	0	0	0
0	1	1	0	1	0	1	0	0	1
0	1	1	1	1	0	1	0	1	0
0	1	1	1	0	0	1	0	1	1
0	1	0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1	0	1
0	1	0	0	1	0	1	1	1	0
0	1	0	0	0	0	1	1	1	1
1	1	0	0	0	1	0	0	0	0
1	1	0	0	1	1	0	0	0	1
1	1	0	1	1	1	0	0	1	0
1	1	0	1	0	1	0	0	1	1
1	1	1	1	0	1	0	1	0	0
1	1	1	1	1	1	0	1	0	1
1	1	1	0	1	1	0	1	1	0
1	1	1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0	0	0
1	0	1	0	1	1	1	0	0	1
1	0	1	1	1	1	1	0	1	0
1	0	1	1	0	1	1	0	1	1
1	0	0	1	0	1	1	1	0	0
1	0	0	1	1	1	1	1	0	1
1	0	0	0	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1

The next step in the design process normally would be to produce five five-variable K-maps for determining B_4, B_3, B_2, B_1, and B_0 as functions of G_4, G_3, G_2, G_1, and G_0. Note, however, that G_4 and B_4 are exactly alike. In addition, B_3 is seen to be a 1 when only one of G_4 and G_3 is a 1; B_2 is a 1 when only an odd number of G_4, G_3, and G_2 are 1; B_1 is a 1 when an odd number of G_4, G_3, G_2, and G_1 are 1; and, finally, B_0 is a 1 when an odd number of the five Gray code inputs are 1. These observations eliminate the need for the K-map minimization scheme, although as a general rule, K-maps should be used and will always give a correct, if not minimum, solution. As in this particular case, however, careful observations may at times reduce complex problems to simpler ones. With our prior experience the application of X-ORs to this problem should become obvious. The binary outputs are obtained as follows:

$$B_4 = G_4$$
$$B_3 = G_3 \oplus G_4$$
$$B_2 = G_2 \oplus G_3 \oplus G_4$$
$$B_1 = G_1 \oplus G_2 \oplus G_3 \oplus G_4$$
$$B_0 = G_0 \oplus G_1 \oplus G_2 \oplus G_3 \oplus G_4$$

Four two-input X-OR gates may be cascaded, as shown in Figure 5.20, to realize the desired Gray-to-binary conversion circuit. This design could be extended to an n-bit converter by cascading an additional (n − 5) two-input X-OR gates.

FIGURE 5.20



EXAMPLE 5.6

Design an n-bit code converter for converting binary to the equivalent Gray code.

SOLUTION

The truth table of Example 5.5 could be used again to solve for G_0 through G_4 in terms of B_0 through B_4. Checking for patterns in the bit relationships to avoid going through a lengthy minimization process, we note that G_0 is a 1 only when either B_0 or B_1 is a 1. Likewise, G_1 is a 1 when either B_1 or B_2 is a 1; G_2 is a 1 when either B_2 or B_3 is a 1; and, similarly, G_{n-1} is a 1 when B_{n-1} is a 1. Using our knowledge of X-ORs we obtain

$$G_{n-1} = B_{n-1}$$

$$\cdots\cdots\cdots\cdots\cdots$$

$$G_3 = B_3 \oplus B_4$$

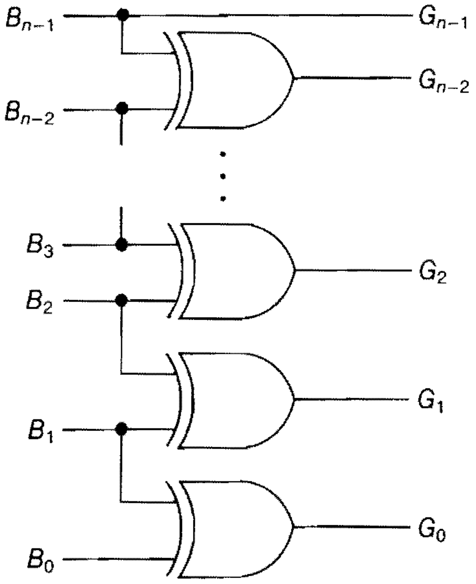
$$G_2 = B_2 \oplus B_3$$

$$G_1 = B_1 \oplus B_2$$

$$G_0 = B_0 \oplus B_1$$

The logic circuit of an n-bit binary-to-Gray converter is obtained as shown in Figure 5.21.

FIGURE 5.21



EXAMPLE 5.7

Design a four-bit module that could be used to obtain an n-bit binary-to-BCD conversion circuit.

SOLUTION

The partially complete truth table for the binary-to-BCD conversion is obtained as shown in Figure 5.22. It can be seen that the LSB of both the binary and the BCD numbers are the same. We could, therefore, design a circuit that will convert the remaining binary bits to the corresponding BCD bit: $D_4D_3D_2D_1$. A close examination of the truth table reveals that

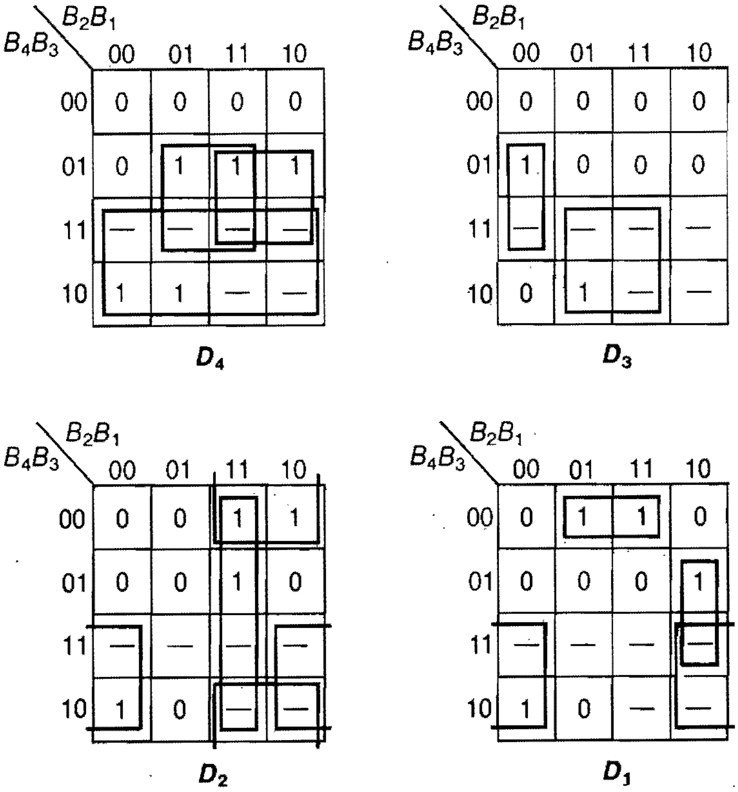
$$D_4D_3D_2D_1 = \begin{cases} (B_4B_3B_2B_1) & \text{when } 0 < (B_4B_3B_2B_1) \leq 0100 \\ (B_4B_3B_2B_1 + 0011) & \text{otherwise} \end{cases}$$

The regular design of the conversion module would consist of obtaining the minimized Boolean expressions for D_4, D_3, D_2, and D_1 from the corresponding K-maps as shown in Figure 5.23. In the K-map, however, the output

FIGURE 5.22

Binary					BCD							
B_4	B_3	B_2	B_1	B_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	1	0	1	0	0	0	0	0	1	0	1
0	0	1	1	0	0	0	0	0	0	1	1	0
0	0	1	1	1	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	1	0	0	0	0	1	0	0	1
0	1	0	1	0	0	0	0	1	0	0	0	0
0	1	0	1	1	0	0	0	1	0	0	0	1
0	1	1	0	0	0	0	0	1	0	0	1	0
0	1	1	0	1	0	0	0	1	0	0	1	1
0	1	1	1	0	0	0	0	1	0	1	0	0
0	1	1	1	1	0	0	0	1	0	1	0	1
1	0	0	0	0	0	0	0	1	0	1	1	0
1	0	0	0	1	0	0	0	1	0	1	1	1
1	0	0	1	0	0	0	0	1	1	0	0	0
1	0	0	1	1	0	0	0	1	1	0	0	1

FIGURE 5.23



B_4B_3 \ B_2B_1	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	—	—	—	—
10	1	1	—	—

D_4

B_4B_3 \ B_2B_1	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	—	—	—	—
10	0	1	—	—

D_3

B_4B_3 \ B_2B_1	00	01	11	10
00	0	0	1	1
01	0	0	1	0
11	—	—	—	—
10	1	0	—	—

D_2

B_4B_3 \ B_2B_1	00	01	11	10
00	0	1	1	0
01	0	0	0	1
11	—	—	—	—
10	1	0	—	—

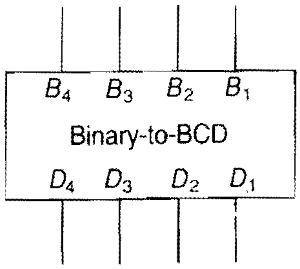
D_1

corresponding to the inputs 10–15 may be considered as don't-cares. The equations for the BCD digits are found to be

$$D_4 = B_3B_1 + B_3B_2 + B_4$$
$$D_3 = B_4B_1 + B_3\overline{B}_2\overline{B}_1$$
$$D_2 = B_2B_1 + B_4\overline{B}_1 + \overline{B}_3B_2$$
$$D_1 = \overline{B}_4\overline{B}_3B_1 + B_3B_2\overline{B}_1 + B_4\overline{B}_1$$

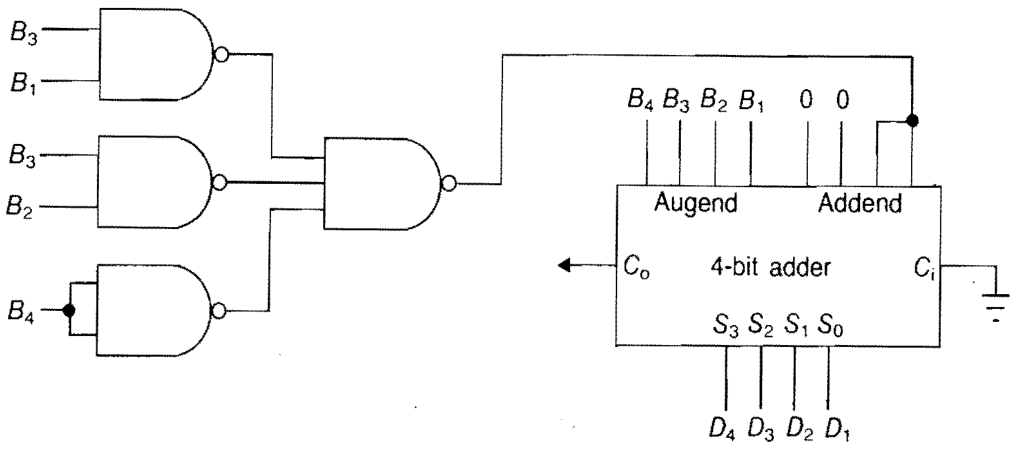
The implementation of the conversion module, as shown in Figure 5.24, is now reasonably straightforward using the preceding equations.

FIGURE 5.24



The equations found for D_4, D_3, D_2, and D_1 are the equations that would be implemented in a commercial binary-to-BCD IC. Another solution using an adder would involve designing a circuit that outputs a 1 whenever $B_4B_3B_2B_1 > 0100$. The circuit output is tied to the least significant two addend inputs of a four-bit adder while B_4, B_3, B_2, and B_1 are tied to the corresending augend inputs. The sum outputs of the four-bit adder unit, as shown in Figure 5.25, would yield the desired BCD output.

FIGURE 5.25



Before the module designed in Example 5.7 is used for conversions of more than five bits, a thorough understanding of the interrelationship between a binary number and its BCD equivalent number is necessary. The decimal value of an n-bit binary number was given by Equation [1.4] as

$$(NI)_{10} = ((\ldots(((a_{n-1})2 + a_{n-2})2 + a_{n-3})2 + \cdots + a_2)2 \\ + a_1)2 + a_0$$

where each of the coefficients, a_j, is either 0 or 1. This nested multiplication by two can be carried out by shifting the binary number $(0.a_{n-1}a_{n-2}a_{n-3} \ldots a_3a_2a_1a_0)$ to the left n times. As these bits are shifted left, each group of four consecutive bits, beginning with the binary point, represents a *decade*. Within each decade each left-shift represents a multiplication by two. However, if a bit passes from the MSB of one decade to the LSB of the next higher decade, its value increases from 8 to 10 only (instead of 16). Figure 5.26 shows the nine possible BCD digits, their values after being shifted left without correction, and their values after being corrected and then shifted.

Note that for values less than or equal to 0100, the shifted value gives the correct BCD digit. If the BCD digit is greater than 0100, the shifted number can be corrected by adding a binary 0110 to the shifted bits. An equivalent correction can also be made by adding 0011 to the BCD digit prior to the shifting. The device already designed in Example 5.7 adds 0011 to the four-bit input if the four bits represent a value greater than binary four.

FIGURE 5.26 **Shifted BCD Digits.**

BCD Digits	BCD Digits Shifted Left One Bit	BCD Digits Corrected and Then Shifted Left One Bit
0000	0 0000	0 0000
0001	0 0010	0 0010
0010	0 0100	0 0100
0011	0 0110	0 0110
0100	0 1000	0 1000
0101	0 1010	1 0000
0110	0 1100	1 0010
0111	0 1110	1 0100
1000	1 0000	1 0110
1001	1 0010	1 1000

In order to provide an understanding of the use of multiple binary-to-BCD converters in converting binary digits of more than five bits, an example involving seven bits will be solved with explanations of each step. The steps will lead us to an algorithm that can be used for the conversion of any n-bit number. The converter module we designed is used to correct a BCD digit prior to the necessary shifting mechanism. The process of shifting bits is achieved by means of hard-wiring between separate stages of converter modules. A separate stage of converter module would be necessary for each of the shift operations.

EXAMPLE 5.8

Convert $127_{10} = 1111111_2$ to BCD.

SOLUTION

Step 1. *Correct-Shift*: Note that for the first two left-shifts of 0.1111111_2 no converter module would be necessary. The third shift, however, results in an integer larger than four. Therefore, the integer needs to be corrected before further shifting of bits:

$0111\,.\,1111 \times 2^4$	uncorrected
$1010\,.\,1111 \times 2^4$	corrected BCD prior to shift
$1\ 0101\,.\,111 \times 2^3$	hybrid number after shift

After completing Step 1 we have a hybrid number; the integer portion is in BCD while the fractional portion is in binary. The value of this hybird number is

$$[15 + (7/8)] \times 2^3 = 127_{10}$$

Step 2. *Correct-Shift*: The four bits on the immediate left of the binary point could be larger than 0100, and in the present case they are. A new correct-shift operation is required, therefore:

$1\ 0101\,.\,111 \times 2^3$	from previous operation
$1\ 1000\,.\,111 \times 2^3$	corrected BCD prior to shift
$11\ 0001\,.\,11 \times 2^2$	hybrid number after shift

Also,

$$[31 + (3/4)] \times 2^2 = 127_{10}$$

Keep in mind that the shift operation just performed brought in a bit from the right of the binary point to be included in the BCD portion. This move is valid since our four-bit input binary-to-BCD converter actually converts five bits, the four connected and the one immediately to the right of those connected.

Prior to shifting, we again add 0011 to those BCD digits that could be greater than 0100. The left-most BCD digit can at most be 0011, so no correction prior to shifting is necessary.

Step 3. *Correct-Shift*:

$11\ 0001\,.\,11 \times 2^2$	from previous step
$11\ 0001\,.\,11 \times 2^2$	corrected BCD prior to shift
$110\ 0011\,.\,1 \times 2^1$	hybrid number after shift

The hybrid number gives $[63 + 1/2] \times 2^1 = 127_{10}$.

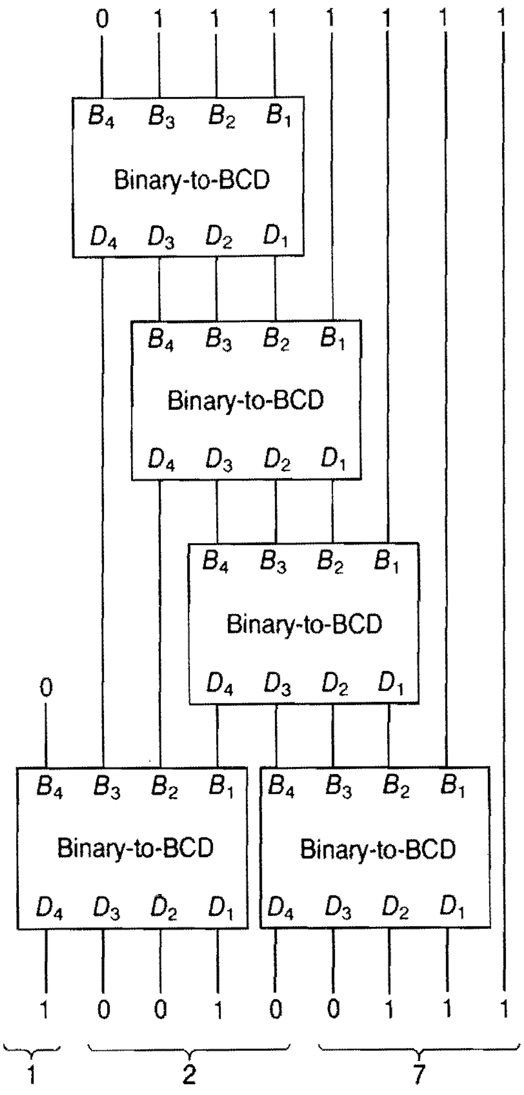
Step 4. The next correction prior to shifting requires two devices since there are sufficient bits in both BCD digits to have values greater than 0100. Again, the left-most binary-to-BCD has one input connected to 0 to allow for the possibility of a four-bit output with only three bits in:

$0110\ 0011\ .\ 1 \times 2^1$	from previous step
$1001\ 0011\ .\ 1 \times 2^1$	corrected BCD prior to shift
$1\ 0010\ 0111\ .\ 0$	hybrid number after shift

The process is now complete.

You would discover after reviewing the preceding steps that the bit positions can remain constant provided the converter modules are moved right to effect a left-shift. The process gives the circuit of Figure 5.27.

FIGURE 5.27



Consider the pattern in the circuit of Figure 5.27. The conversion of an n-bit binary number to BCD can be performed using the converter designed in Example 5.7 as follows:

1. Add a 0 to the left of the MSB position of the binary number. Connect to the inputs of a four-bit converter module a 0 and the three MSBs of the binary number.

2. Take the three least significant processed bits and the most significant unprocessed bit as inputs to a converter module. The *most significant processed bit* (*MSPB*), which is the MSB of the left-most converter module, is considered in the next step.
3. If there are three MSPBs, connect a 0 to the MSB position of a converter module and the three MSPBs to the remaining three inputs; otherwise carry the MSPBs from above operations to the next level.
4. Once a converter module is added to a level, the number of converter modules remains the same until three MSPBs accumulate, at which time another is added.
5. This process is continued until the LSB is the only unprocessed bit.

EXAMPLE 5.9

Use several of the modules designed in Example 5.7 for converting the binary number 11010110100_2 to its BCD equivalent.

SOLUTION

The logic circuit necessary for obtaining the conversion of this 11-bit binary number follows directly from the previous discussion of the *n*-bit binary-to-BCD conversion algorithm. The resulting circuit configuration is shown in Figure 5.28. After every three levels, the number of modules per level increases by one. The output of the network is 1716_{BCD} as expected.

EXAMPLE 5.10

Design a four-bit module suitable for the BCD-to-binary conversion of five bits.

SOLUTION

Since the LSBs of the binary and BCD numbers are equal, the design will consider the four MSBs of the five-bit BCD value. The design involves undoing the conversion of binary-to-BCD numbers considered in the preceding few examples. Intuitively we might assert that this will involve shifting right and then correcting the resultant values by subtracting 0011. Accordingly, the combinational module that could be used for this algorithm has the following characteristics:

$$B_4B_3B_2B_1 = \begin{cases} (D_4D_3D_2D_1) & \text{when } (D_4D_3D_2D_1) < 1000 \\ (D_4D_3D_2D_1 - 0011) & \text{otherwise} \end{cases}$$

FIGURE 5.28

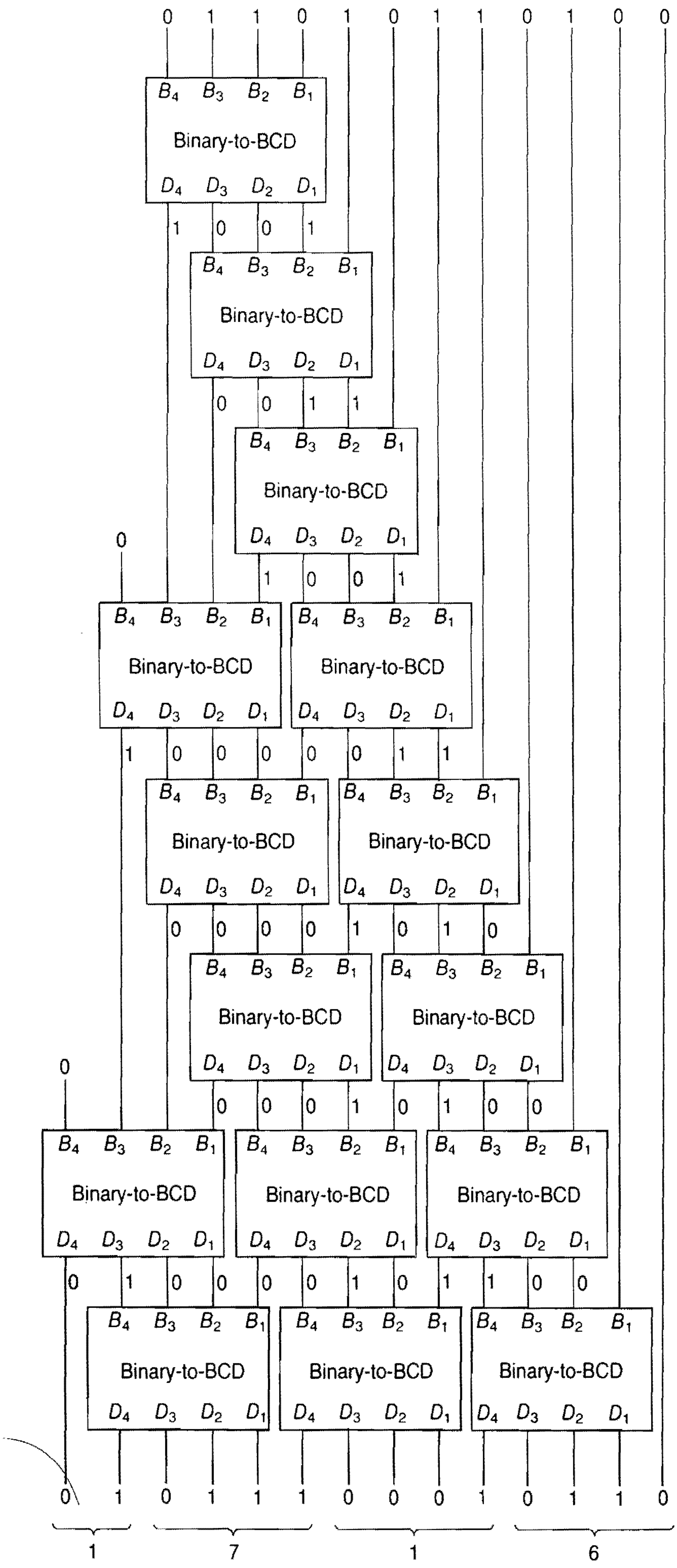

0 1 1 0 1 0 1 1 0 1 0 0
B_4 B_3 B_2 B_1
Binary-to-BCD
D_4 D_3 D_2 D_1
1 0 0 1
0 0 1 1
1 0 0 1
0
1 0 0 0 0 0 1 1
0 0 0 0 1 0 1 0
0 0 0 1 0 1 0 0
0
0 1 0 0 0 0 1 0 1 1 0 0
0 1 0 1 1 1 0 0 0 1 0 1 1 0
1
7
1
6

FIGURE 5.29

D_4	D_3	D_2	D_1	B_4	B_3	B_2	B_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	—	—	—	—
0	1	1	0	—	—	—	—
0	1	1	1	—	—	—	—
1	0	0	0	0	1	0	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	1
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	1
1	1	0	1	—	—	—	—
1	1	1	0	—	—	—	—
1	1	1	1	—	—	—	—

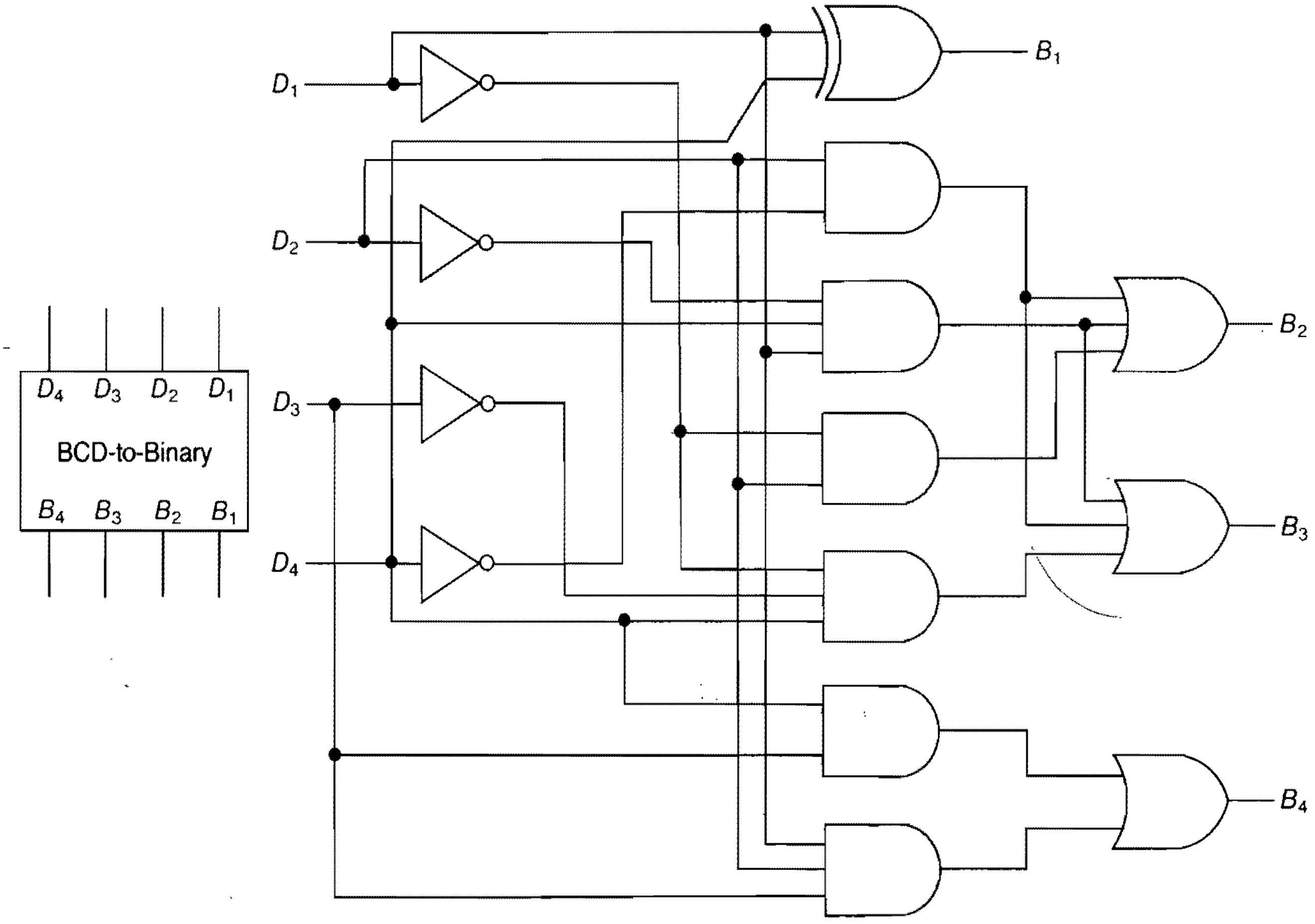
The truth table for such a module is obtained as shown in Figure 5.29.

The don't-cares in the binary table occur for six different input values that will never appear as uncorrected BCD digits. For example, a 0101 will never occur since that would imply the presence of an unlikely BCD number, either 1010 or 1011, prior to the shift-right operation. Examination of the truth table indicates that BCD digits equal to or less than 0100 require no modification. However, for BCD digits 8 through 12 a value of 3 must be subtracted for correction. Constructing and minimizing the corresponding K-maps for B_4, B_3, B_2, and B_1 gives the following Boolean equations:

$$B_1 = D_1\bar{D}_4 + \bar{D}_1D_4 = D_1 \oplus D_4$$
$$B_2 = D_2\bar{D}_4 + D_1\bar{D}_2D_4 + \bar{D}_1D_2$$
$$B_3 = D_2\bar{D}_4 + D_1\bar{D}_2D_4 + \bar{D}_1\bar{D}_3D_4$$
$$B_4 = D_3D_4 + D_1D_2D_3$$

The circuit for the BCD-to-binary module may now be obtained as shown in Figure 5.30.

FIGURE 5.30



EXAMPLE 5.11

Use the module designed in Example 5.10 to perform the conversion of 127_{10} and consequently develop an algorithm for use in an n-bit BCD-to-binary conversion. Note that

$127_{10} = 000100100111_{BCD}$

SOLUTION

As mentioned previously, this process should be the reverse of the binary-to-BCD conversion scheme developed earlier.

Step 1. Shift the BCD quantity to the right by one bit, which will replace one bit to the right of the decimal. We are considering a hybrid number again. The digits to the right of the radix point will be binary and those to the left BCD. The process of shifting results in uncorrected BCD values. The BCD-to-binary converter is used to correct these uncorrected results:

1001 0011 . 1 × 2^1	uncorrected BCD
0110 0011 . 1 × 2^1	corrected BCD

The hybrid number is $[63 + (1/2)] \times 2 = 127_{10}$.

Step 2. Note that the left-most 0011 needs no correction:

0011 0001 . 11 × 2^2	uncorrected BCD
0011 0001 . 11 × 2^2	corrected BCD

The hybrid number still gives $[31 + (3/4)] \times 4 = 127_{10}$.

Step 3. *Correct-Shift:*

0001 1000 . 111 × 2^3	uncorrected BCD
0001 0101 . 111 × 2^3	corrected BCD

And, $[15 + (7/8)] \times 2^3 = 127_{10}$.

Step 4. At this point we have five bits of BCD left to convert to binary:

1010 . 1111 × 2^4	uncorrected BCD
0111 . 1111 × 2^4	corrected BCD

And, $[7 + (15/16)] \times 2^4 = 127_{10}$.

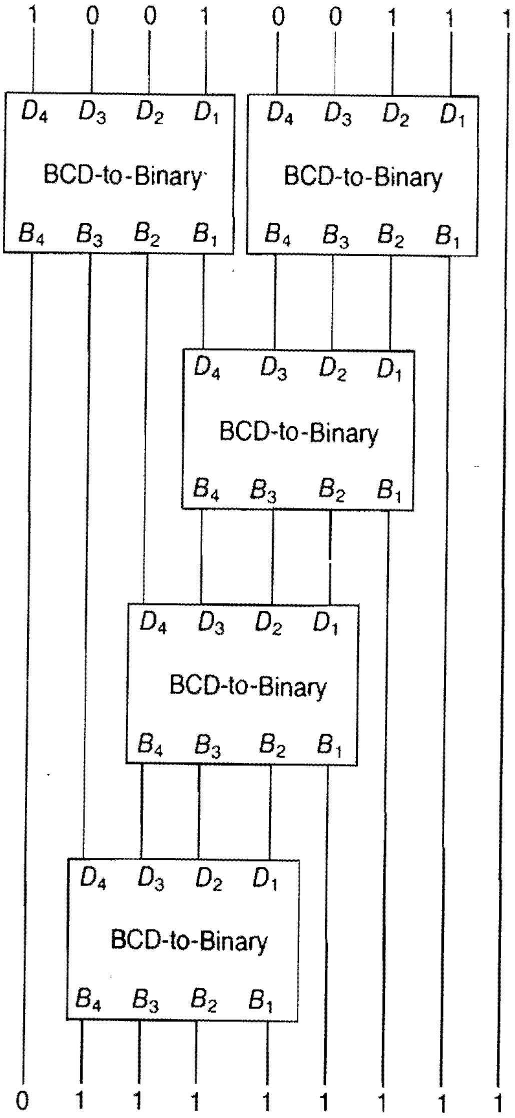
The resultant value 1111111_2 is indeed equal to 000100100111_{BCD}. The resulting circuit using the converter is obtained as shown in Figure 5.31.

An algorithm now may be developed for extending the process just examined to the conversion of n-bit BCD numbers:

Step 1. Starting at the right end, skip the LSB and connect four-input, BCD-to-binary converter modules to all remaining bits. If two or less bits would be connected to the left-most converter, leave it off and extend the bits to the next level. If all bits at a level are connected to a converter module, the MSB of the output will be zero and should not be carried to the next level.

Step 2. Skip the least significant processed bit at each level and reassign bits as directed in Step 1. Continue until the MSB (processed or unprocessed) from an upper level is included as the most significant input to a converter. Remember not to bring down bits that would be zeros from upper MSB positions (see Figure 5.31).

FIGURE 5.31



5.6 BCD Arithmetic Circuits

Even though most computers use regular binary numbers for their arithmetic operations, some special-purpose computers and calculators operate in the decimal number system using BCD. BCD-based systems require more memory to store information because of less efficient coding and complex arithmetic circuitry. However, the final results in these systems do not have to be decoded prior to display as decimal digits. BCD arithmetic is usually complicated by the fact that some of the sums or differences are invalid. When two BCD numbers are added on a binary adder, it is possible to obtain 16 different sums, of which six are undefined. In addition, when there is a carry-out, four additional conditions turn out to be undefined as well. Figure 5.32 shows the to-be-corrected and the corresponding corrected BCD sums and carry. The largest uncorrected sum could be 10011, corresponding to the sum of two BCD nines and a carry-in of one. Note that the sums that are greater than 1001 are all undefined and are in need of correction.

FIGURE 5.32 **Table for the Uncorrected and Corrected BCD Sums.**

Uncorrected					Corrected					
C_3'	S_3'	S_2'	S_1'	S_0'	C_3	S_3	S_2	S_1	S_0	
0	0	0	0	0	0	0	0	0	0	
	• • •		• • •			• • •		• • •		No correction needed
0	1	0	0	1	0	1	0	0	1	
0	1	0	1	0	1	0	0	0	0	
0	1	0	1	1	1	0	0	0	1	
0	1	1	0	0	1	0	0	1	0	
0	1	1	0	1	1	0	0	1	1	
0	1	1	1	0	1	0	1	0	0	
0	1	1	1	1	1	0	1	0	1	
1	0	0	0	0	1	0	1	1	0	
1	0	0	0	1	1	0	1	1	1	
1	0	0	1	0	1	1	0	0	0	
1	0	0	1	1	1	1	0	0	1	

A BCD adder could be designed using the design techniques we have already studied. Such an adder circuit would have a total of nine inputs and five outputs. Of the nine inputs, four inputs would be for the augend, another four for the addend, and the ninth input would be the carry-in. Of the five outputs, one would be for the carry-out and the remaining four would be for the sums. Therefore, the truth table would consist of $2^9 = 512$ different input combinations, many of which would lead to don't-care conditions. The two obvious design choices for such a circuit would be

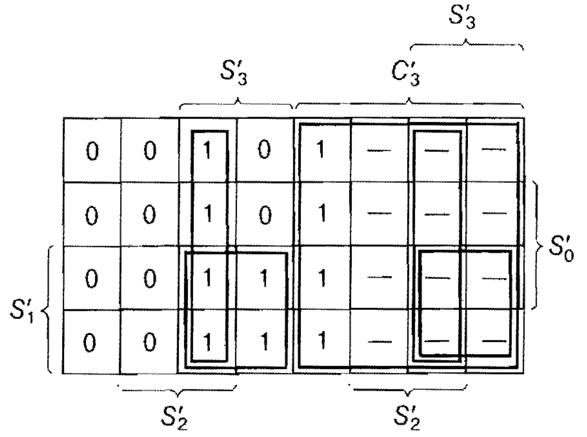
1. Make use of a ROM or PAL or PLA.
2. Minimize the function using the Q-M technique and then generate the circuit using combinational gates.

In either case, the design would be too involved to pursue further.

The easiest route to a BCD adder design is to base its operation on a four-bit FA module. If the sum of the corresponding nine inputs exceeds nine, a carry is generated and the sum is corrected by means of a correction circuit. It should be noted from Figure 5.32 that the corrected sum is obtainable from the uncorrected sum simply by adding a $(0110)_2$ to it. Furthermore, it can be seen that C_3 becomes 1 only when correction becomes necessary. The condition for the generation of a carry-out can be found, therefore, from the minimization of the K-map for C_3 shown in Figure 5.33. The Boolean expression for the corrected carry-out becomes

$$C_3 = C_3' + S_1'S_3' + S_2'S_3' \qquad [5.23]$$

FIGURE 5.33 **K-Map for the Corrected Carry Bit.**



0	0	1	0	1	—	—	—
0	0	1	0	1	—	—	—
0	0	1	1	1	—	—	—
0	0	1	1	1	—	—	—

An analysis of our previous discussion would reveal that the single-decade BCD adder would involve the following subunits:

a four-bit FA for addition,

a carry decoder circuit,

a second four-bit FA circuit for correction.

An adder made in this way is shown in Figure 5.34. The sum bits of the first adder unit are introduced at the augend inputs of the second adder unit. When the corrected carry-out, C_3, is a 0, the uncorrected sum remains unchanged. When it is a 1, the uncorrected sum is added to 0110 to give the corrected sum at the output of the second four-bit FA.

At this time the designer may be concerned with the amount of worst-case propagation delays. The propagation delays basically arise from the two FA subunits used in this circuit. A close examination of the circuit, however, reveals that the correction does not need all four of the single-bit FAs. S_0' never needs any correction. An improved version of the BCD adder may be obtained by having only two HAs and one FA in the correction unit. This improved circuit is shown in Figure 5.35. Another approach for designing BCD adders would be first to convert the BCD numbers into binary, add the resulting binary numbers, and then transform the binary sum into the correct BCD sum. This design requires both postcorrection and preconversion circuitries and, therefore, is more expensive than the one already designed.

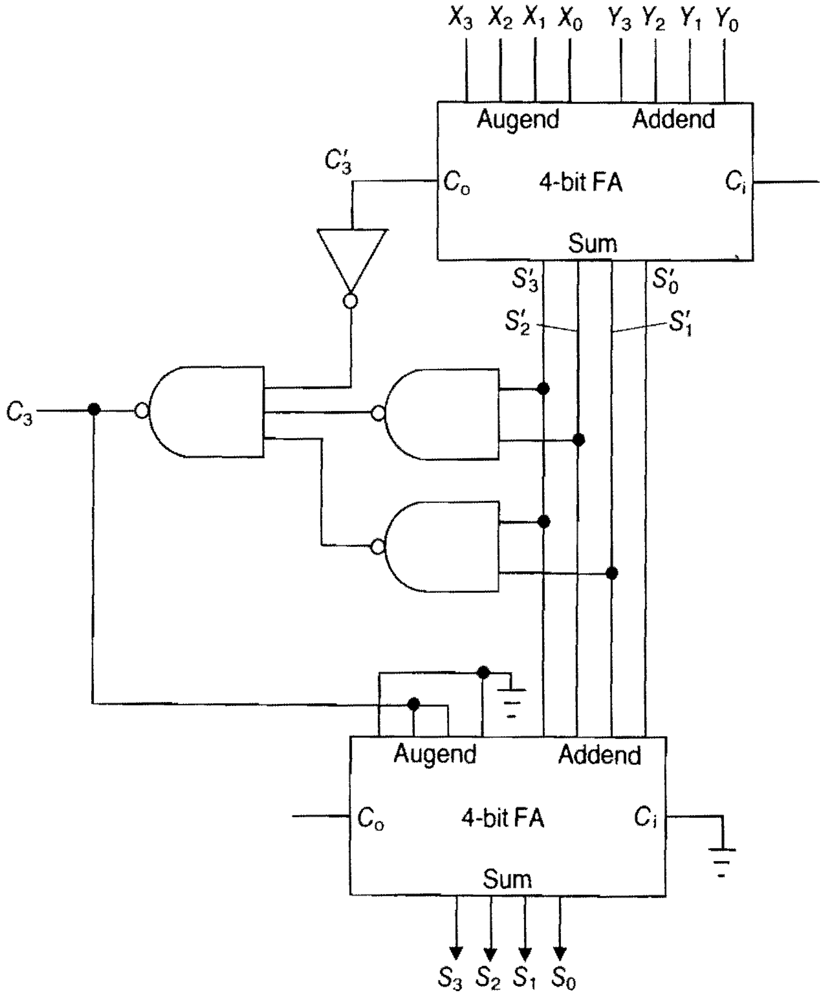


FIGURE 5.34 BCD Adder Circuit.

Next consider the design of a BCD subtracter. Subtraction is more complex than BCD addition, since the possibility of having a negative difference exists. Recall from earlier observations made in Sections 1.4, 1.5, and 5.3 that there are at least two ways to handle negative BCD numbers: either by 10's complement or by 9's complement. The technique using 10's complement would result in a circuit similar to that of Figure 5.12. The scheme using 9's complement, however, would require an end-round-carry into the system. The BCD subtracter using 9's complement could be designed using a BCD adder and a 9's complementer circuit. In order to obtain the 9's complementer circuit, consider the truth table in Figure 5.36.

Using the table of Figure 5.36, we would obtain

$$O_3 = \bar{I}_1\bar{I}_2\bar{I}_3 = \overline{I_1 + I_2 + I_3}$$

$$O_2 = I_1 \oplus I_2$$

$$O_1 = I_1$$

$$O_0 = \bar{I}_0$$

FIGURE 5.35 **Improved BCD Adder Circuit.**

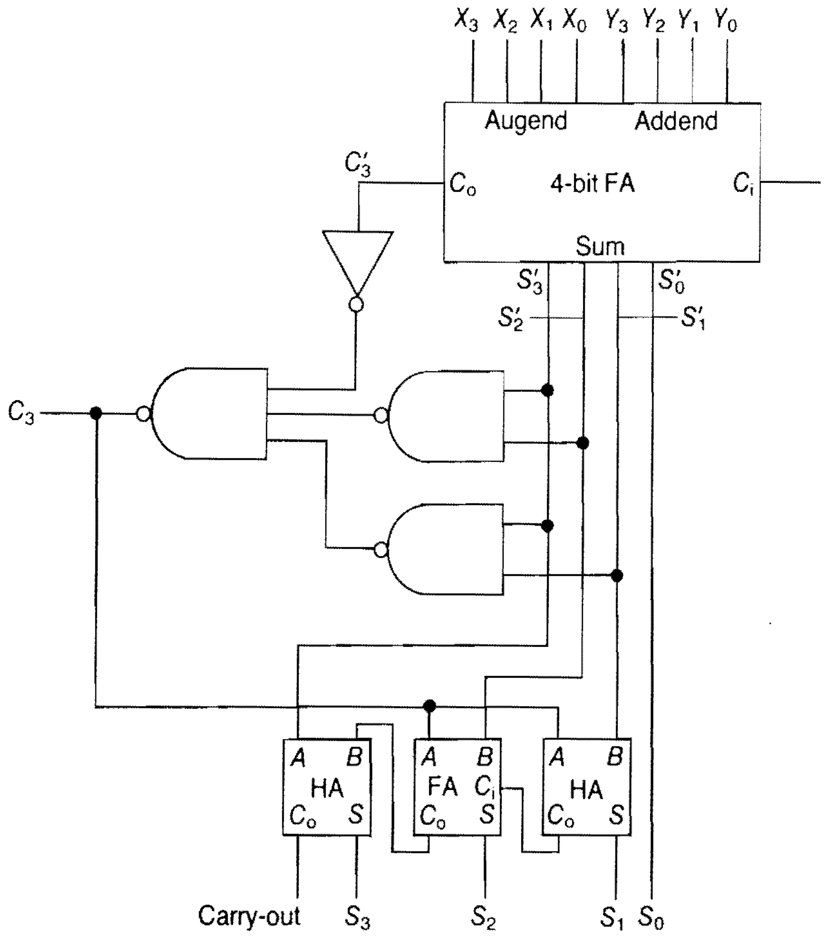
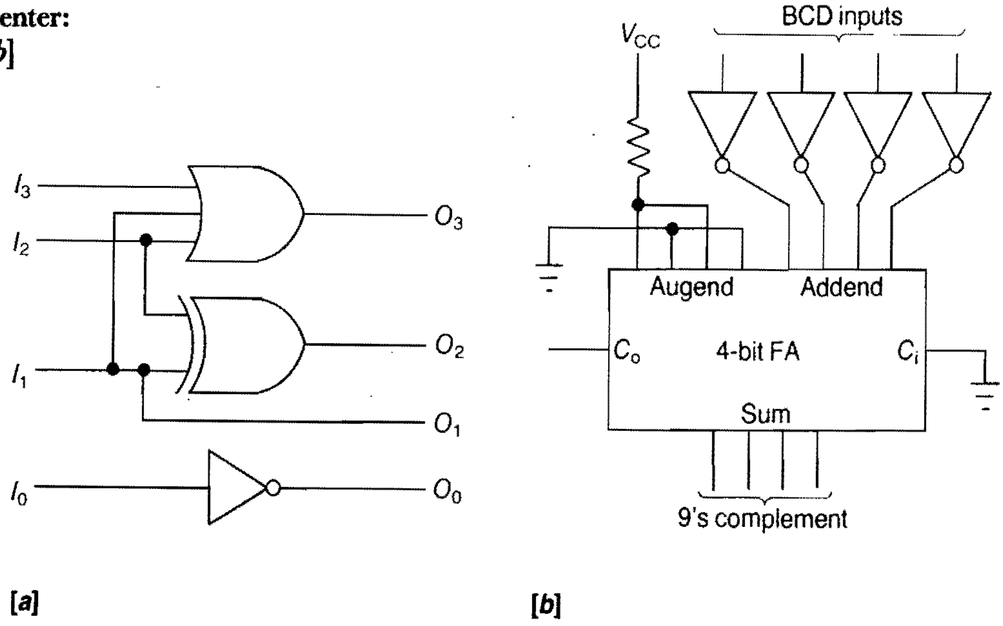


FIGURE 5.36 **Truth Table for the BCD 9's Complement.**

BCD Input				9's Complement			
I_3	I_2	I_1	I_0	O_3	O_2	O_1	O_0
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0

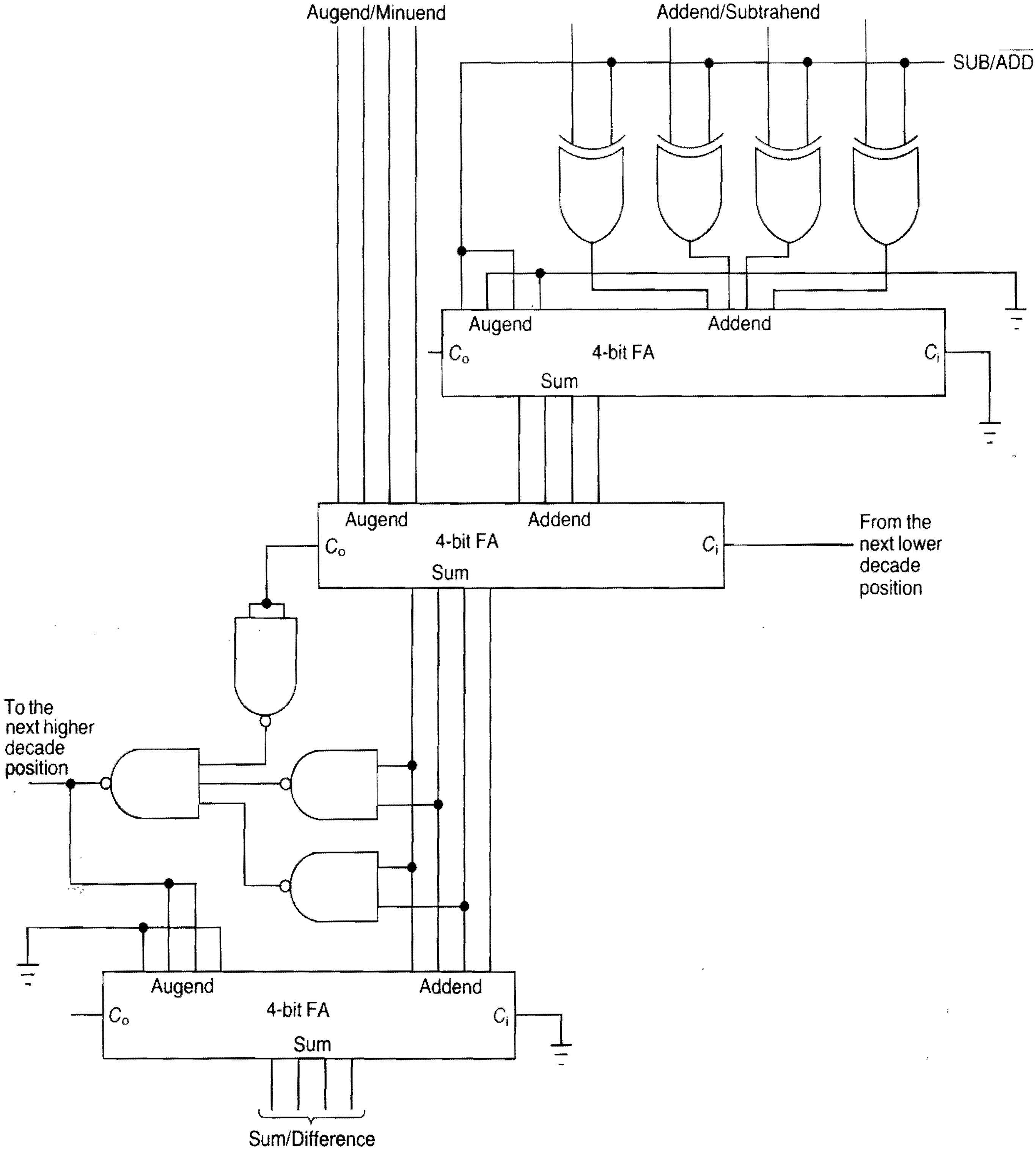
A BCD 9's complementer circuit using the preceding equations is shown in Figure 5.37[*a*]. Figure 5.37[*b*] shows how a four-bit adder may be used to achieve the same goal. The latter circuit performs the 1's complement of the BCD digits and then adds to it the 2's complement of six (1010).

FIGURE 5.37 **9's Complementer: [*a*] Using Simple Gates and [*b*] Using a Four-Bit FA.**



Once the 9's complementer is designed, the rest of the steps are obvious. If X-OR gates are used in place of the inverters of Figure 5.37[*b*], a combined BCD adder/subtracter unit may result. The BCD adder/subtracter unit so designed is shown in Figure 5.38. Each decade unit consists of three four-bit FAs. The first one works

FIGURE 5.38 **A Single-Decade BCD Adder/Subtracter Unit.**



as the 9's complementer unit, the second as the adder unit, and the third works as part of the correction circuit. When the SUB/$\overline{\text{ADD}}$ input is a 0, the unit performs simple addition as the complementer unit leaves the addend unchanged. When SUB/$\overline{\text{ADD}}$ is a 1, the subtrahend is complemented and is then added to the minuend. Consequently, the sum appears at the final output when the SUB/$\overline{\text{ADD}}$ input is a 0 and the difference is obtained otherwise. When several such units are cascaded to make a multi-decade BCD adder/subtracter, the carry-out from the most significant decade unit is fed into the carry-in of the least significant decade unit. This arrangement guarantees the inclusion of an end-around-carry to satisfy the need of the 10's complement system.

EXAMPLE 5.12

Use the XS3 code for designing a BCD adder/subtracter unit.

SOLUTION

As you can see from the truth table in Figure 5.39, the 9's complement of an XS3 number may be obtained simply by taking its 1's complement. Consequently, it is easier to obtain the 9's complement of an XS3 than that of its BCD equivalent.

It would be worthwhile to review Example 1.17 at this time. Notice that correction would be necessary for obtaining the true XS3 sum. Corresponding to the expected decimal numbers 3 and 7, the sum yielded 1001 and 1101, respectively, and no carry-out. However, 1000 and 0000 were respectively obtained along with a carry-out in place of decimal numbers 8 and 0. It can now be concluded that there are two simple rules that must be followed while adding numbers in the XS3 code:

a. If a carry is produced, 0011 is added.

b. If a carry is not produced, 0011 is subtracted. This subtraction is usually done by adding the 2's complement of 0011 (1101) and ignoring any carry if so produced.

These rules should be better understood by examining the following cases.

Case I.

$$\begin{array}{rrl}
47_{10} & 0111\ 1010_2 & \\
+\ 34_{10} & +\ 0110\ 0111_2 & \\
\hline
81_{10} & 1110\ 0001_2 & \\
 & +\ 1101\ 0011_2 & \text{correction} \\
\cline{2-2}
 & 1\ 1011\ 0100 & \text{XS3} \\
 & (8\quad 1)_{10} &
\end{array}$$

Case II.

$$\begin{array}{rrl}
47_{10} & 0111\ 1010_2 & \\
-\ 34_{10} & +\ 1001\ 1000_2 & \text{9's complement} \\
\hline
13_{10} & 1\ 0001\ 0010_2 & \\
 & \xrightarrow{\quad} 1_2 & \\
 & 0011\ 0011_2 & \text{correction} \\
\cline{2-2}
 & 0100\ 0110 & \\
 & (1\quad 3)_{10} &
\end{array}$$

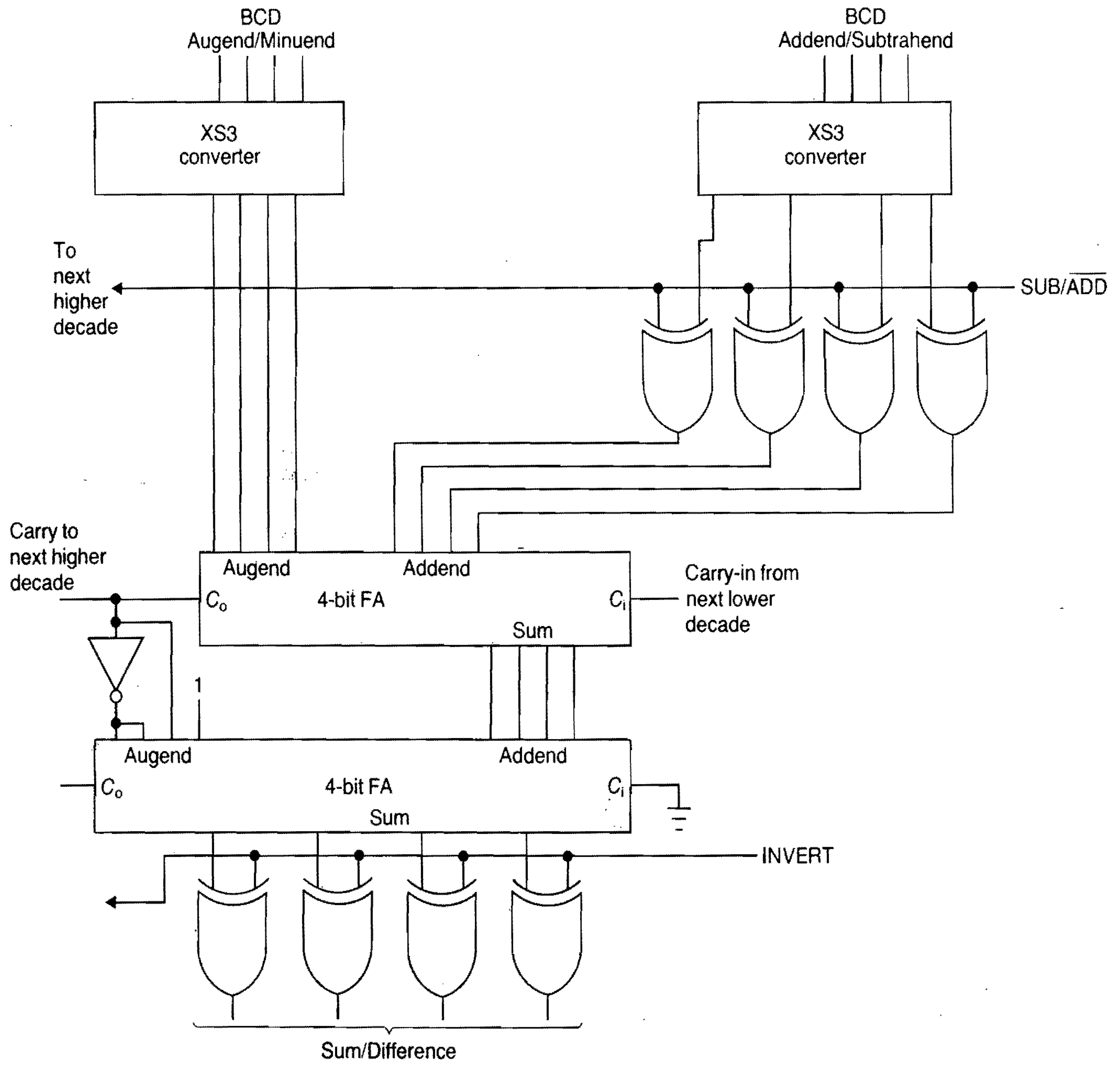
FIGURE 5.39

Decimal	XS3	9's Complement of XS3
0	0011	1100
1	0100	1011
2	0101	1010
3	0110	1001
4	0111	1000
5	1000	0111
6	1001	0110
7	1010	0101
8	1011	0100
9	1100	0011

$$\begin{array}{lrrl}
\textit{Case III.} & 34_{10} & 0110\;0111_2 & \\
 & -\;47_{10} & +\;1000\;0101_2 & \text{9's complement} \\
\cline{2-3}
 & -\;13_{10} & 1110\;1100_2 & \\
 & & 1101\;1101_2 & \text{correction} \\
\cline{3-3}
 & & 1011\;1001_2 & \\
 & & \rightarrow 0100\;0110_2 & = \;-(13)_{10}
\end{array}$$

A positive result is indicated when adding the 9's complement results in a carry-out of the MSB. This carry-out is used as the end-around-carry. Figure 5.40 shows a BCD adder/subtracter circuit that has incorporated the rules of XS3 addition and subtraction.

FIGURE 5.40



When the SUB/$\overline{\text{ADD}}$ input is a 0, the addend arrives at the first adder along with the augend. Otherwise, the complement of the subtrahend is added to the minuend. The output of the first adder is added to either 0011 or 1101, depending on whether or not a carry is generated. The INVERT input is activated when there is no end-around-carry, and in that event the output is complemented to yield the final result.

5.7 Arithmetic Logic Unit (ALU)

The examples presented in previous sections have shown some of the multi-bit arithmetic functions that are used in digital systems. Many times it may as well be necessary to perform bit-by-bit logic operations between two multi-bit operands. A multi-function circuit that can operate on groups of bits, therefore, proves to be extremely advantageous in many complex digital systems. Combinational design techniques generally are used to design such *multi-function circuits*, otherwise known as *arithmetic logic units* (*ALUs*). The various operations are usually selected by means of several control or select lines.

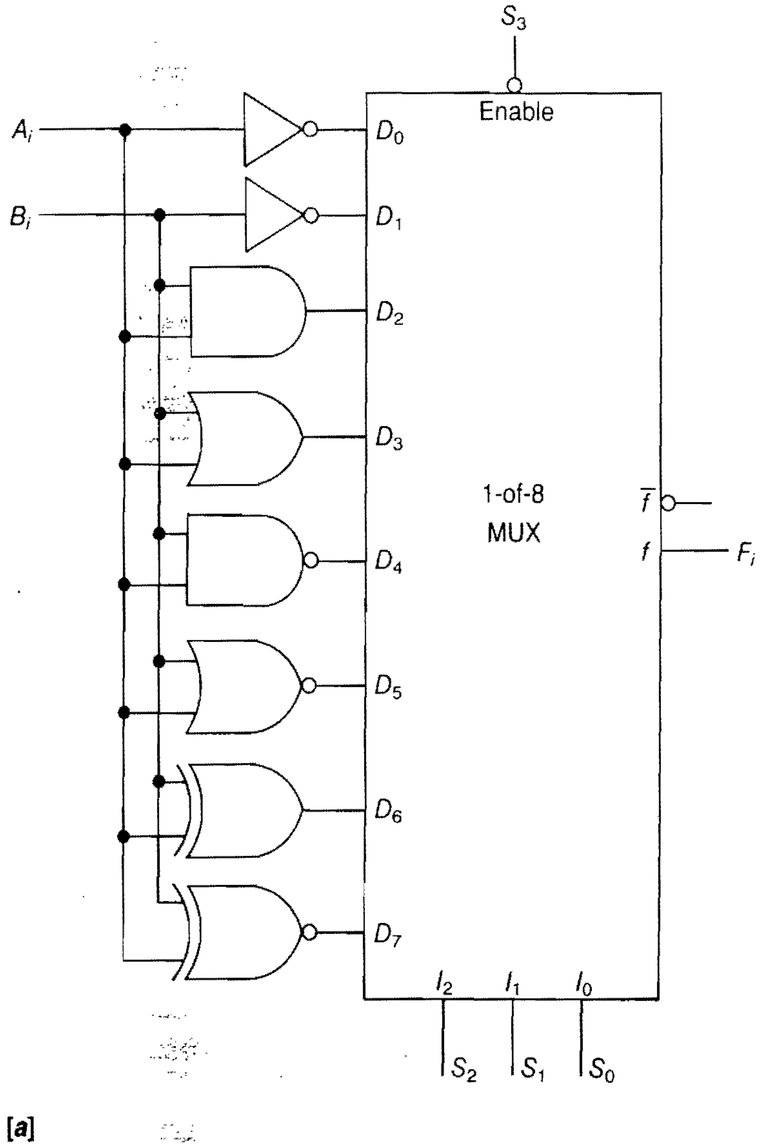
The various logic operations are realized by routing the inputs to circuits that perform various logic functions and using a MUX to select any one of the possible logic operations. There are commercially available ALUs that operate on two four-bit values with 32 arithmetic and 16 logic operations selectable by the combination of four control inputs, a mode selector, and a carry-in. In this section the design of a relatively less complex ALU will be considered to demonstrate the process and some typical functions.

Consider an ALU with two four-bit inputs, $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. Let us first consider bit-by-bit logic operations of several types. Between any two inputs, X and Y, there could be a total of 16 different types of logic outputs: 0, 1, X, Y, $\overline{X}$, $\overline{Y}$, $X + Y$, $\overline{X} + Y$, $X + \overline{Y}$, XY, $\overline{X}Y$, $X\overline{Y}$, $\overline{X + Y}$, $\overline{XY}$, $X \oplus Y$, and $\overline{X \oplus Y}$. All of these outputs are achievable by means of logic gates.

Figure 5.41[*a*] shows a logic circuit consisting of a 1-of-8 MUX and only eight logic gates. This circuit is able to perform up to eight different logic operations between its inputs, A_i and B_i. A specific operation is selected by means of the three control inputs: S_2, S_1, and S_0. The enable input, S_3, is set to a 0 to enable the MUX in all of these eight cases. The truth table in Figure 5.41[*b*] lists the possible logic operations and the corresponding control conditions. For example, when control input is 0100, the D_4 input of the 1-of-8 MUX is activated, and consequently $\overline{A_iB_i}$ becomes available at the MUX output. This 1-of-8 MUX could have been replaced with a 1-of-16 MUX for realizing 16 different logic operations. In that case, however, an additional control input would be necessary.

The arithmetic section may be designed around a four-bit ripple or CLA adder circuit. If the carry-in is utilized, it is possible to obtain a larger number of arithmetic operations for the same

FIGURE 5.41 **Logic Section of an ALU: [*a*] Circuit and [*b*] Function Table.**



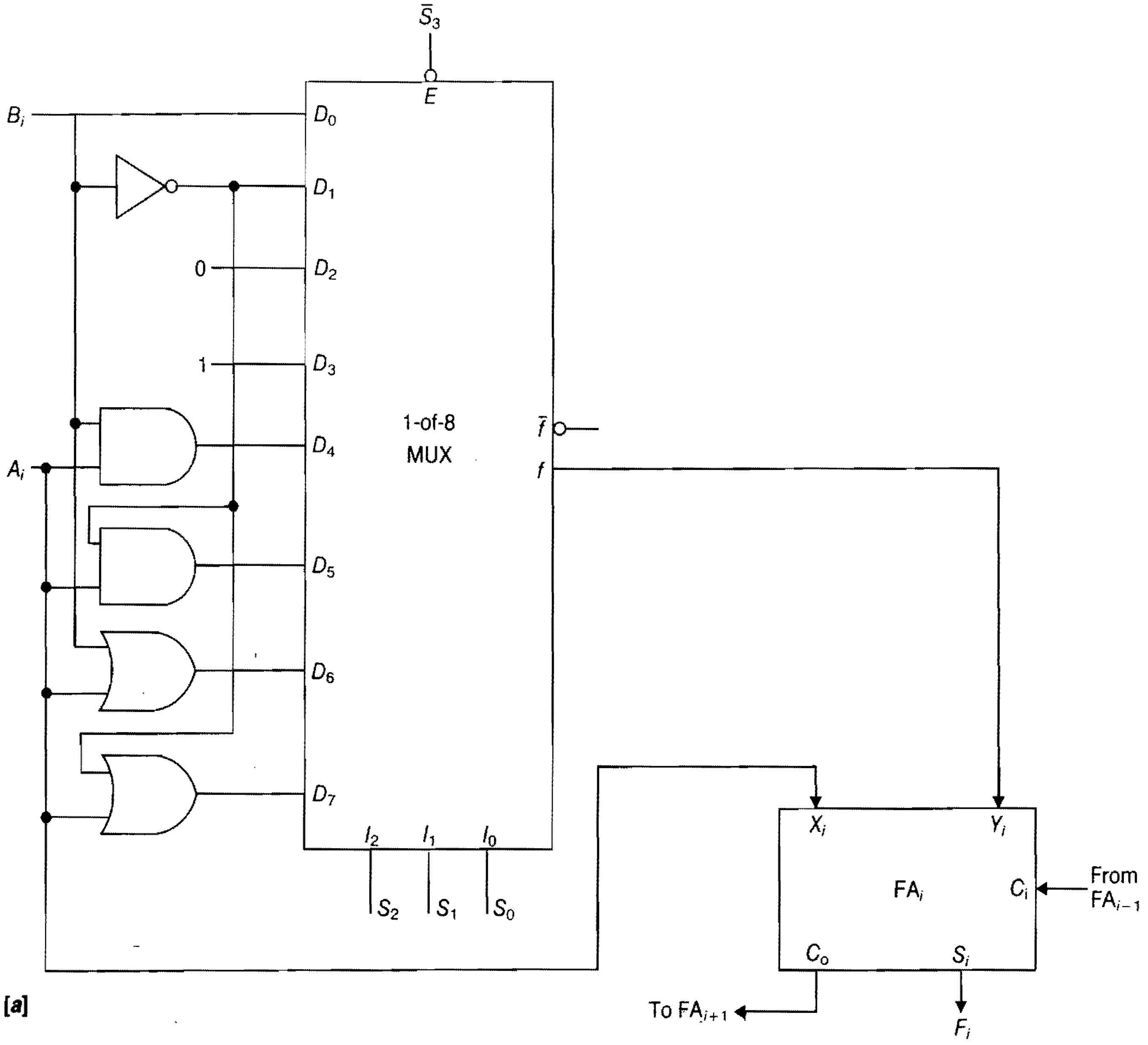
Control Inputs				Output
S_3	S_2	S_1	S_0	F_i
0	0	0	0	$\overline{A}_i$
0	0	0	1	$\overline{B}_i$
0	0	1	0	A_iB_i
0	0	1	1	$A_i + B_i$
0	1	0	0	$\overline{A_iB_i}$
0	1	0	1	$\overline{A_i + B_i}$
0	1	1	0	$A_i \oplus B_i$
0	1	1	1	$\overline{A_i \oplus B_i}$

[*b*]

number of control inputs. For every control condition the arithmetic output when $C_i = 1$ is always one greater than the corresponding output when $C_i = 0$. Figure 5.42[*a*] shows an arithmetic circuit corresponding to a single-bit input. It may be used to obtain a total of 16 different arithmetic operations. This circuit consists of a 1-of-8 MUX, a full adder, and five logic gates. In order to differentiate between the arithmetic and the logic operations, $\overline{S_3}$ is used to select arithmetic operations. The output of the MUX unit is fed into one of the adder inputs while another of the FA inputs is tied to A_i directly.

Figure 5.42[*b*] lists all of the arithmetic operations and the corresponding control inputs needed for operating this arithmetic unit. The MUX output carries into the adder either a function of both A_i

FIGURE 5.42 Arithmetic Unit of an ALU: [*a*] Circuit and [*b*] Function Table.



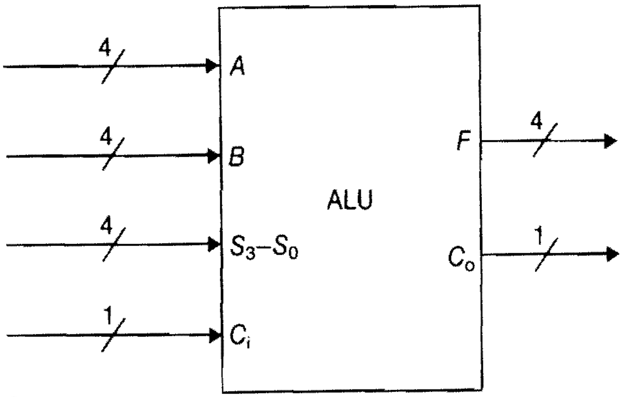
Control				F_i	
S_3	S_2	S_1	S_0	$C_i = 0$	$C_i = 1$
1	0	0	0	$A_i + B_i$	$A_i + B_i + 1$
1	0	0	1	$A_i + \bar{B}_i$	$A_i + \bar{B}_i + 1$
1	0	1	0	A_i	$A_i + 1$
1	0	1	1	$A_i - 1$	A_i
1	1	0	0	$A_i + A_iB_i$	$A_i + A_iB_i + 1$
1	1	0	1	$A_i + A_i\bar{B}_i$	$A_i + A_i\bar{B}_i + 1$
1	1	1	0	$A_i + (A_i + B_i)$	$A_i + (A_i + B_i) + 1$
1	1	1	1	$A_i + (A_i + \bar{B}_i)$	$A_i + (A_i + \bar{B}_i) + 1$

[*b*]

and B_i, or a 0, or a 1. For example, when the control input is 1100, the D_4 input of the MUX is activated, which introduces A_i, A_iB_i, and the carry-in as inputs to the FA. Thus the sum becomes A_i plus A_iB_i in the absence of a carry-in, and A_i plus A_iB_i plus 1 otherwise. Consequently, four 1-of-8 MUXs, twenty (5 × 4) discrete gates, and a four-bit FA may be connected to form an arithmetic unit for processing two four-bit binary inputs. Note when the control input is 1011 and $C_i = 0$, the four bits of A are added to a string of four 1s. This operation is equivalent to adding A to the 2's complement of 0001.

Both of these units, arithmetic and logic, may now be combined together to make an ALU, as shown by the block diagram in Figure 5.43. The two basic units could be internally ORed together to produce the ALU output. The ALU processes four bits of A and four bits of B simultaneously. If $S_3 = 0$, one of the eight logic operations would be realized, and when $S_3 = 1$, one of the 16 arithmetic operations would be available. Consequently the ALU consists of four single-bit arithmetic units and four single-bit logic units, and it could perform a total of 24 different operations. If the word size exceeds four bits, several four-bit ALUs may be cascaded by tying the carry-out of one ALU to the carry-in of the next ALU.

FIGURE 5.43 **Block Diagram of a Four-Bit ALU.**



5.8 Decoders and Encoders

Very often in digital systems it is necessary to convert one code to another. The process that determines what character, digit, or number a code represents is called *decoding*. A *decoder* is an integral part of this process. It is a specially organized combinational circuit that translates a code to a more useful or meaningful form. In this section we shall look at just a few of the many decoding functions.

One of the frequently used decoders is a BCD–to–seven-segment decoder. This particular type of decoder accepts as inputs BCD and provides outputs to drive a seven-segment LED display device, as shown in Figure 5.44, in order to decode bits into readable digits. The decimal inputs to a circuit are first changed to equivalent binary form by means of BCD-to-binary converter modules for the desired binary operation. The resultant binary output is finally reconverted back to equivalent BCD output and is usually displayed by means of seven-segment display devices. These devices are used often in calculators.

The display device consists of seven light-emitting segments that represent each of the ten decimal numbers when activated in suitable combination. As an example, segments *a*, *f*, *g*, *c*, and *d* have to be illuminated to represent a 5. There are two choices for representing a 1: either *f* and *e* or *b* and *c*. The normal decimal code for these indicators is shown in Figure 5.45. A Boolean expression corresponding to each segment of the display may now be found. The Boolean equations for the illumination of display segments may be obtained as follows:

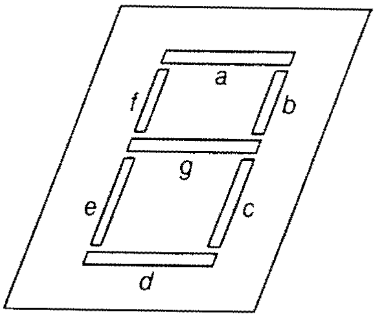


FIGURE 5.44 **Seven-Segment Display Device.**

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

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

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

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

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

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

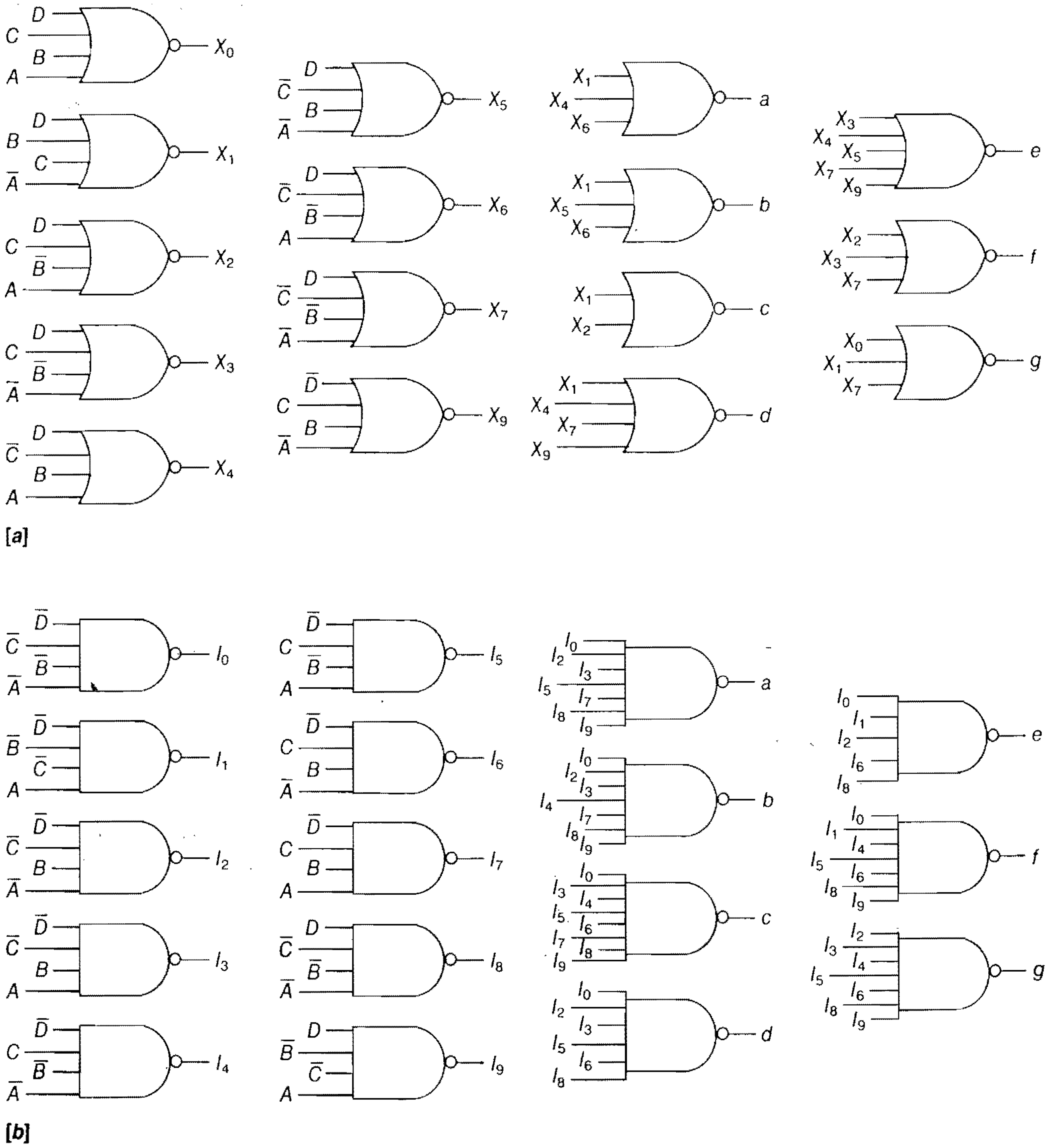
$$g(D,C,B,A) = \overline{\overline{(D + C + B + A)} + \overline{(D + C + B + \overline{A})}}$$
$$\overline{+ \overline{(D + \overline{C} + \overline{B} + \overline{A})}}$$

FIGURE 5.45 **Truth Table for Seven-Segment Decoding Function.**

Number	Segments						
	a	*b*	*c*	*d*	*e*	*f*	*g*
0 $\bar{D}\bar{C}\bar{B}\bar{A}$	1	1	1	1	1	1	0
1 $\bar{D}\bar{C}\bar{B}A$	0	1	1	0	0	0	0
2 $\bar{D}\bar{C}B\bar{A}$	1	1	0	1	1	0	1
3 $\bar{D}\bar{C}BA$	1	1	1	1	0	0	1
4 $\bar{D}C\bar{B}\bar{A}$	0	1	1	0	0	1	1
5 $\bar{D}C\bar{B}A$	1	0	1	1	0	1	1
6 $\bar{D}CB\bar{A}$	0	0	1	1	1	1	1
7 $\bar{D}CBA$	1	1	1	0	0	0	0
8 $D\bar{C}\bar{B}\bar{A}$	1	1	1	1	1	1	1
9 $D\bar{C}\bar{B}A$	1	1	1	0	0	1	1

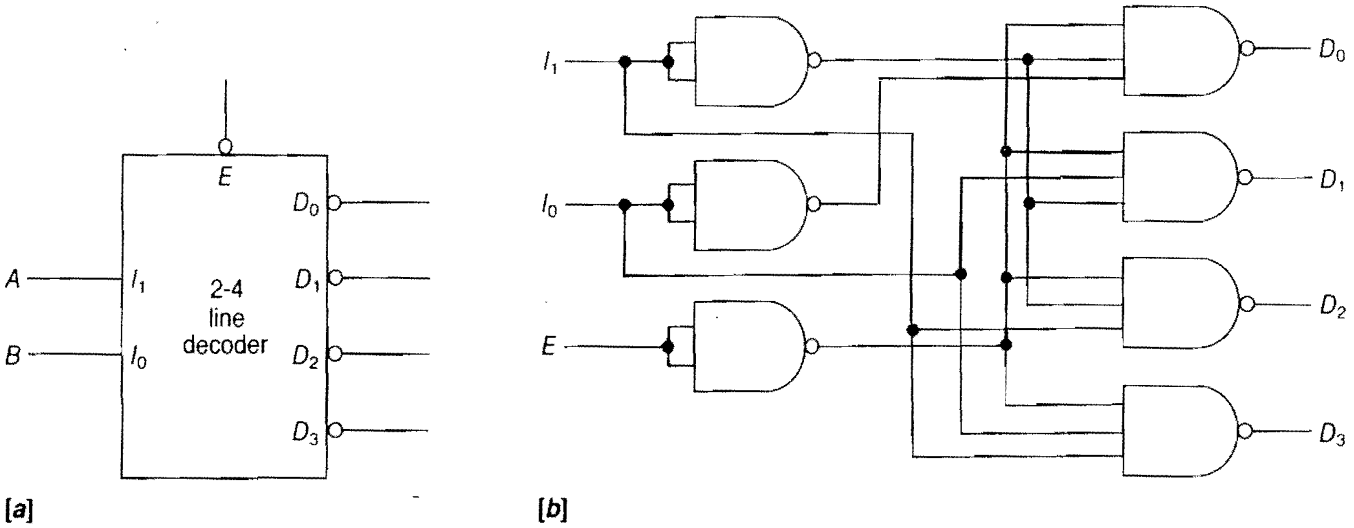
These Boolean expressions for the segments lead to the implementation of the desired decoder circuit as shown in Figure 5.46. The NOR circuit requires 16 gates and a total of 59 gate inputs while the NAND circuit requires 17 gates and a total of 87 gate inputs.

FIGURE 5.46 **BCD–to–Seven-Segment Decoder Circuit: [*a*] Using NOR Gates and [*b*] Using NAND Gates.**


D C B A X_0
D B C $\bar{A}$ X_1
D C $\bar{B}$ A X_2
D C $\bar{B}$ $\bar{A}$ X_3
D $\bar{C}$ B A X_4
D $\bar{C}$ B $\bar{A}$ X_5
D $\bar{C}$ $\bar{B}$ A X_6
D $\bar{C}$ $\bar{B}$ $\bar{A}$ X_7
$\bar{D}$ C B $\bar{A}$ X_9
X_1 X_4 X_6 a
X_1 X_5 X_6 b
X_1 X_2 c
X_1 X_4 X_7 X_9 d
X_3 X_4 X_5 X_7 X_9 e
X_2 X_3 X_7 f
X_0 X_1 X_7 g
[a]
$\bar{D}$ $\bar{C}$ $\bar{B}$ $\bar{A}$ I_0
$\bar{D}$ $\bar{B}$ $\bar{C}$ A I_1
$\bar{D}$ $\bar{C}$ B $\bar{A}$ I_2
$\bar{D}$ $\bar{C}$ B A I_3
$\bar{D}$ C $\bar{B}$ $\bar{A}$ I_4
$\bar{D}$ C $\bar{B}$ A I_5
$\bar{D}$ C B $\bar{A}$ I_6
$\bar{D}$ C B A I_7
D $\bar{C}$ $\bar{B}$ $\bar{A}$ I_8
D $\bar{B}$ $\bar{C}$ A I_9
I_0 I_2 I_3 I_5 I_7 I_8 I_9 a
I_0 I_2 I_3 I_4 I_7 I_8 I_9 b
I_0 I_3 I_4 I_5 I_6 I_7 I_8 I_9 c
I_0 I_2 I_3 I_5 I_6 I_8 d
I_0 I_1 I_2 I_6 I_8 e
I_0 I_1 I_4 I_5 I_6 I_8 I_9 f
I_2 I_3 I_4 I_5 I_6 I_8 I_9 g
[b]

We shall consider next a slightly different decoder scheme that accepts n bits of binary information and converts them to up to 2^n unique outputs. For example, two bits of binary information would result in four unique output values. One such decoder scheme is shown by the circuit of Figure 5.47. This decoder is commonly known as a 2-4 line decoder. If the enable input E is 0, the decoder is enabled, and when E is 1 the decoder is disabled. When E is 0, all outputs are 1 except for the line corresponding to the decimal value of the input bits. For example, when $AB = 00$, only D_0 is 0. Such a scheme would allow us to activate four different circuits by means of the four decoder outputs. In that case the operations of these four peripheral circuits could be controlled by means of two select inputs, A and B.

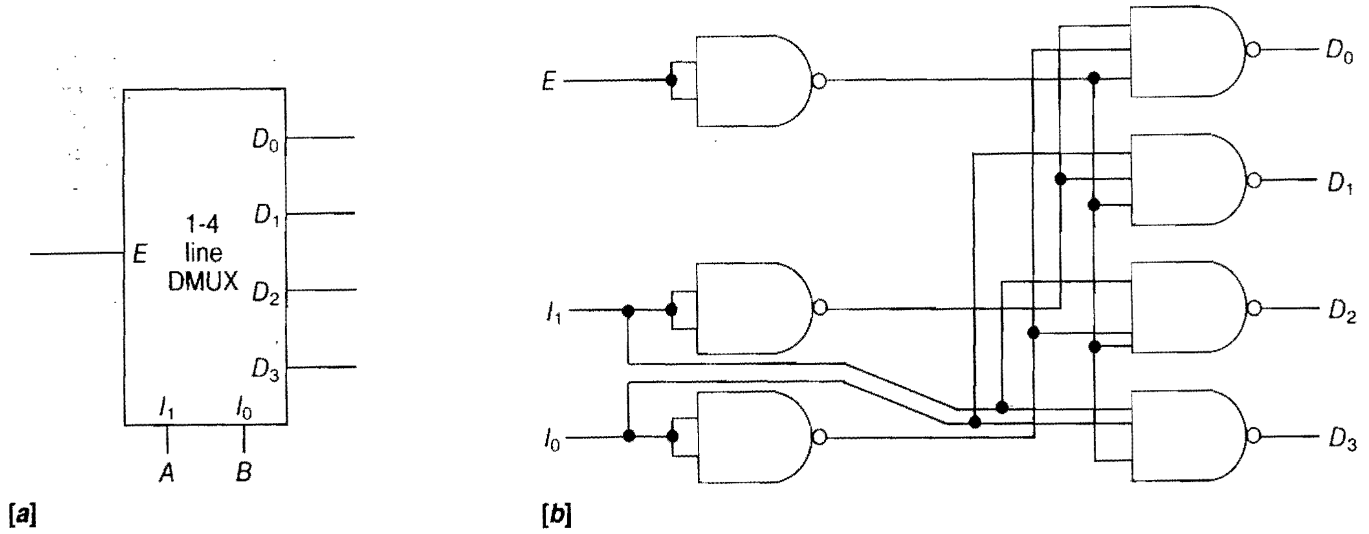
FIGURE 5.47 **2-4 Line Decoder: [*a*] Block Diagram and [*b*] Circuit.**



With proper connections a demultiplexer circuit can be made from a decoder. A *demultiplexer* (*DMUX*) receives data on a single entry line and outputs this data on one of its many output lines. Figure 5.48 shows a 1-4 line DMUX circuit where the decoder inputs are treated as the DMUX selects. The enable input E is connected to the input data, which appear at the output specified by the values of A and B.

There are times when several decoder/DMUX circuits may be cascaded together to form a larger decoder/DMUX. Figure 5.49 shows how two 2-4 line decoders are combined by means of their

FIGURE 5.48 **1-4 Line Demultiplexer: [*a*] Block Diagram and [*b*] Circuit.**

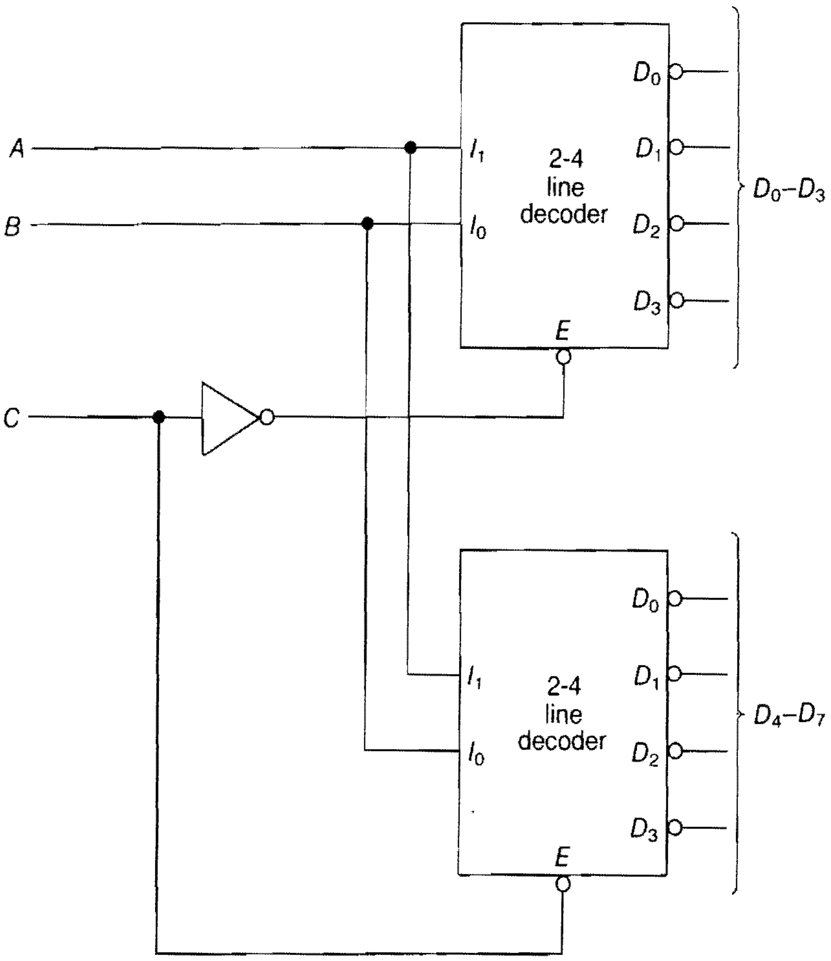


enable inputs. When $C = 1$, only the top decoder is enabled and the lower one is disabled. When $C = 0$, the top decoder is disabled and the bottom decoder is enabled. The combined circuit functions as a 3-8 line decoder.

The decoders that have been discussed thus far are often constructed using transistors in AND formation. The number of transistors used in each of the gates is approximately equal to the number of inputs to each gate, and the number of gates present is on the order of the number of decoder outputs. Therefore, the number of transistors required in a decoder circuit increases as 2^n with increasing inputs. Consequently designers would like to see a decoder scheme whose sum of gates and gate inputs is reasonably small. Figure 5.50 shows a particular configuration for a 4-16 line decoder that uses a reduced number of transistors. Such a configuration is commonly known as the *tree-type* decoding network. An examination would show that 64 transistors are needed to fabricate such a circuit. Comparatively, a regular 4-16 line decoder designed similarly to that shown in Figure 5.47 would require a total of 72 transistors.

The decoder network also may use an innovative scheme called the *balanced* decoding scheme, illustrated in Figure 5.51. It requires only 56 transistors. The significance of this improvement becomes more important as larger decoders are considered. The regular decoder network of Figure 5.47 is still the fastest because it involves only two stages of NAND gates. The inclusion of an enable input in a 4-16 line decoder, however, would involve an additional 18 transistors.

FIGURE 5.49 3-8 Line Decoder Using Two 2-4 Line Decoder Units.



An *encoder* is a combinational circuit that accepts a digit on its inputs and converts it to a coded output. In fact, an encoder reverses the function of a decoder. As an example, let us consider the design of a decimal-to-BCD encoder. The device has a total of 10 inputs—one for each decimal digit—and four outputs to represent the corresponding BCD numbers. These decimal inputs could possibly be the keys on a hand-held calculator.

The truth table for the encoder is listed in Figure 5.52[*a*] from which the following expressions may readily be obtained:

$$A = 1 + 3 + 5 + 7 + 9$$

$$B = 2 + 3 + 6 + 7$$

$$C = 4 + 5 + 6 + 7$$

$$D = 8 + 9$$

The resulting circuit is very straightforward, as shown in Figure 5.52[*b*].

FIGURE 5.50 **4-16 Line Tree-Type Decoder: [*a*] Block Diagram and [*b*] Circuit.**

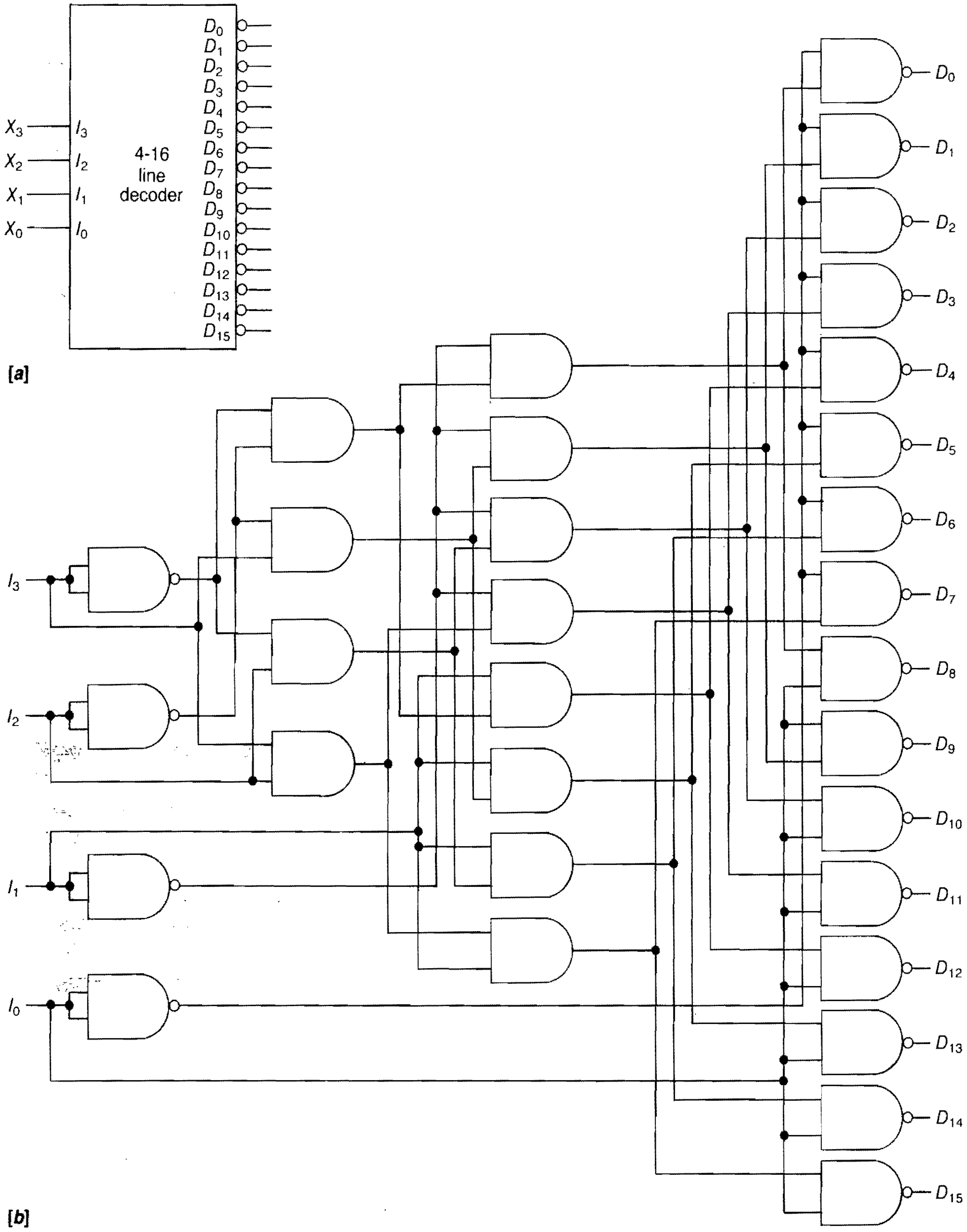


FIGURE 5.51 4-16 Line Balanced Decoder Circuit.

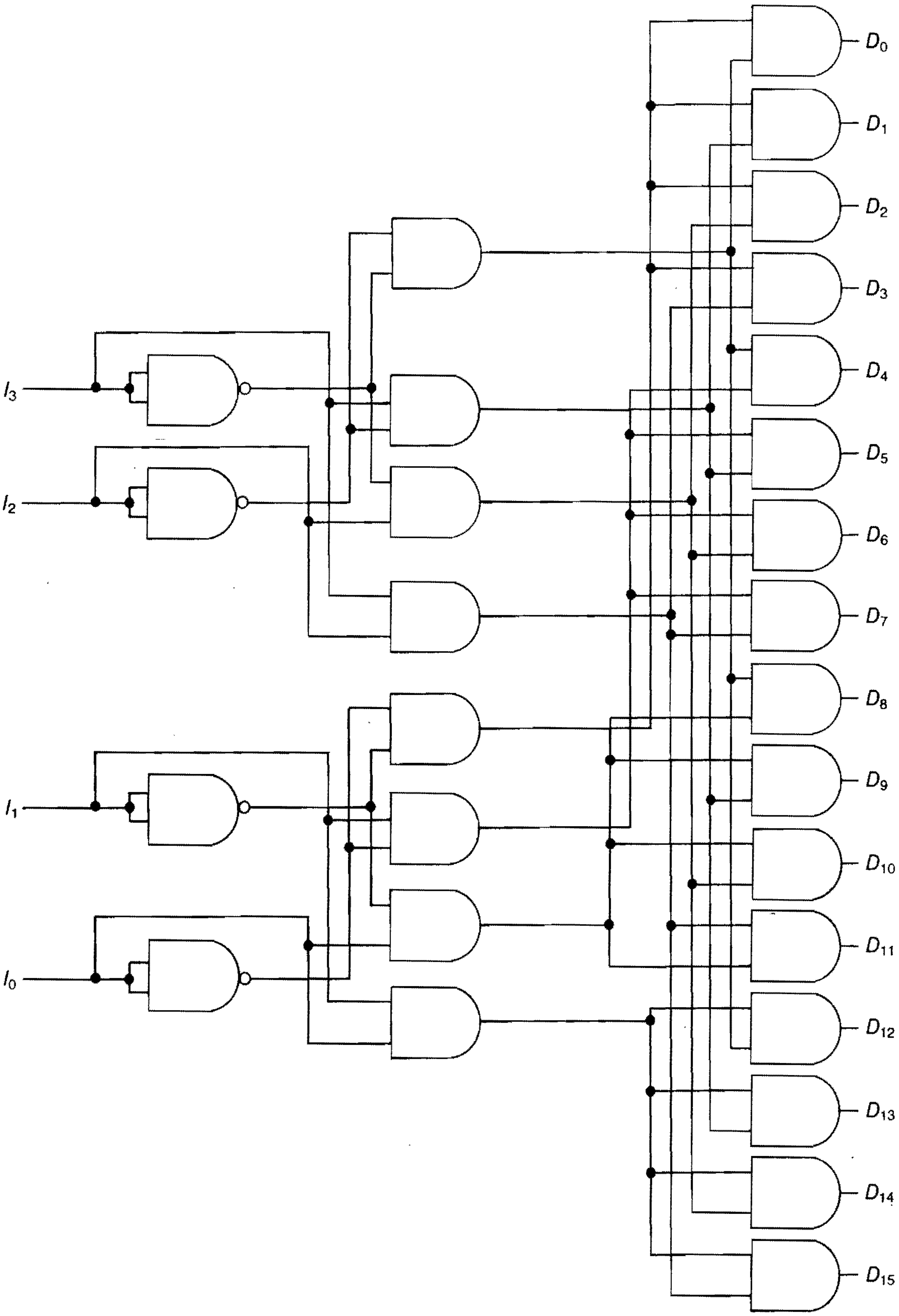
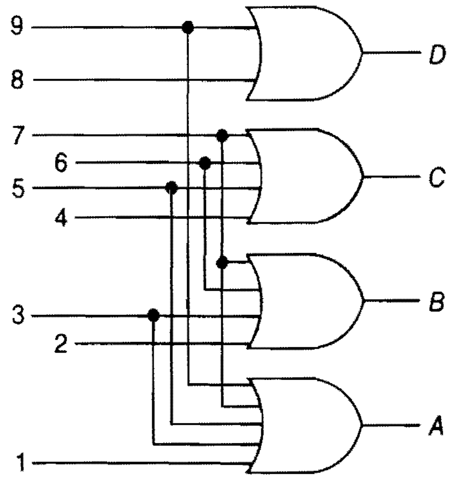


FIGURE 5.52 **Decimal-to-BCD Encoder: [*a*] Truth Table and [*b*] Circuit.**

	BCD
Decimal Input	*DCBA*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

[*a*]



[*b*]

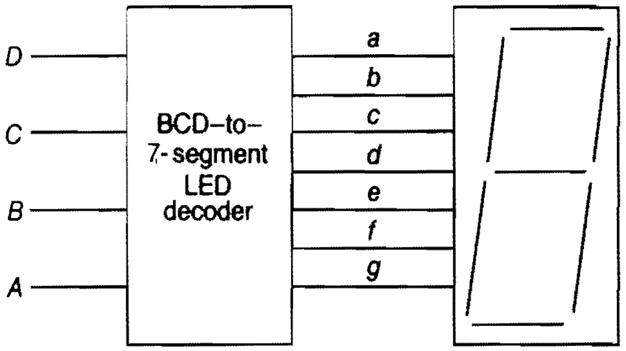
EXAMPLE 5.13

Consider the seven-segment display device of Figure 5.44. Design a scheme to store up to five BCD integers such that the left-most BCD zeros are not displayed. For example, 00932 should be displayed as only 932.

SOLUTION

This design may be accomplished by modifying the already-designed BCD-to-LED display of Figure 5.46. Each unit accepts four BCD inputs and outputs seven LED outputs, as shown by the block diagram of Figure 5.53. The circuit of Figure 5.53 requires an added feature that suppresses the display of leading zeros. First it must be determined if the MSB is a 0. If it is, the next significant bit is tested, and so on. The first nonzero bit stops further testing. Accordingly, one may design a circuit for the *i*th bit if it is a zero or not, as shown by the block diagram of Figure 5.54.

FIGURE 5.53



A high *TDS input* to the decoder implies that *This Decoder needs to be Searched.* A high *NDS output* similarly implies that the *Next Decoder needs to be Searched* as well. The NDS output of one input may be introduced to the next unit on the right as the TDS input, and so on. This feature may be accomplished by adding a circuit such as shown in Figure 5.55 to the already available circuit.

FIGURE 5.54

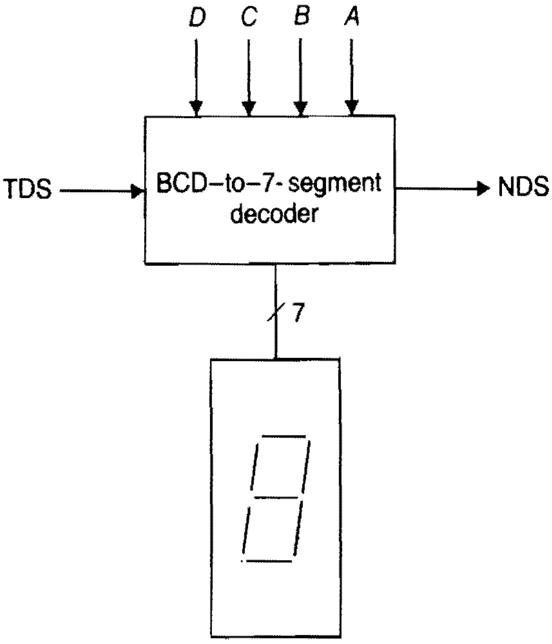
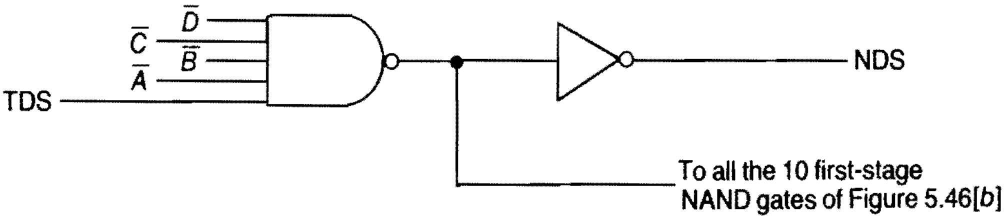
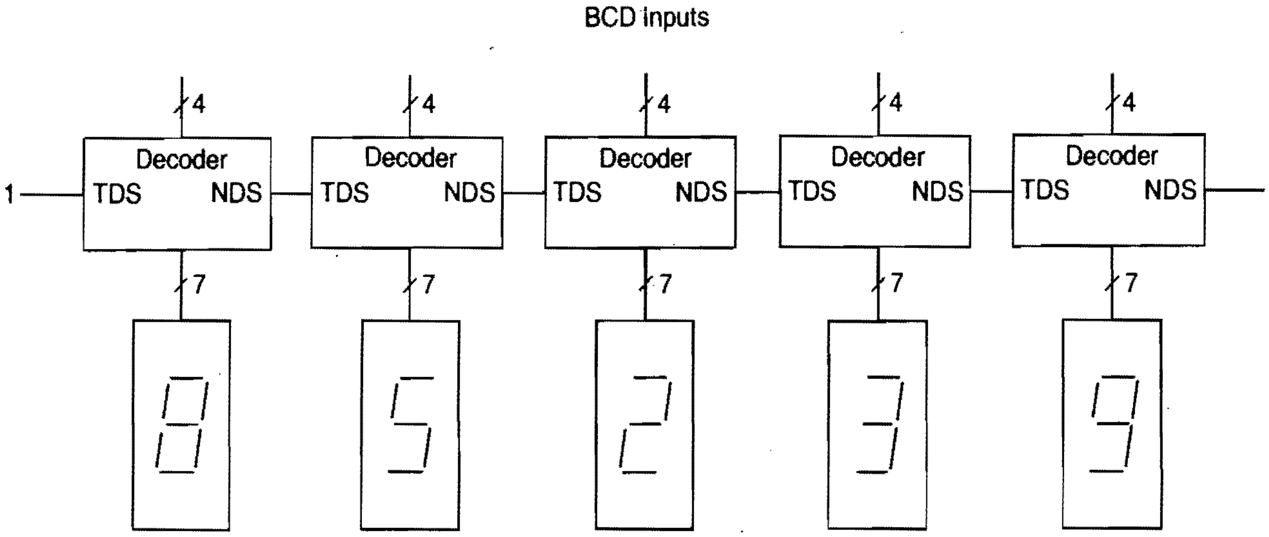


FIGURE 5.55



When the BCD digit corresponds to a 0 and the TDS input is high, the above circuit disables the decode circuit and suppresses the display. At the same time the resulting NDS becomes a 1, resulting in further search for zeros if there are any. If the BCD digit is not a 0, the decoder is not disabled and further search is abandoned. The overall five-bit circuit is obtained as shown in Figure 5.56.

FIGURE 5.56



EXAMPLE 5.14

Design a four-input priority encoder such that when two inputs, D_i and D_j, are high simultaneously, D_i has priority over D_j when $i > j$. The encoder produces a binary output code corresponding to the input that has the highest priority.

SOLUTION

The block diagram for such a device may be as shown in Figure 5.57. As a beginning step, the corresponding truth table needs to be known. The truth table for such a device is easily obtained as shown in Figure 5.58. The don't-cares are introduced under input columns whenever appropriate. The Boolean equations for the outputs are obtained directly as follows:

$$f_0 = D_3 + D_1\bar{D}_2$$

$$f_1 = D_2 + D_3$$

The four-input priority encoder circuit is obtained accordingly, as shown in Figure 5.59. The request indicator, M, shows whether or not any of the four inputs are active.

FIGURE 5.57

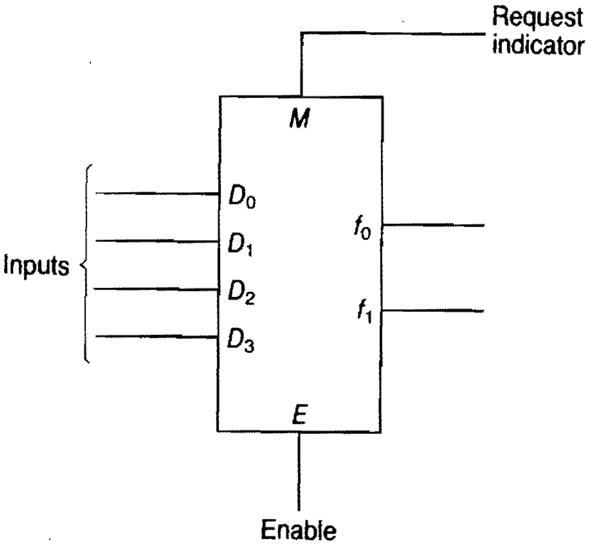
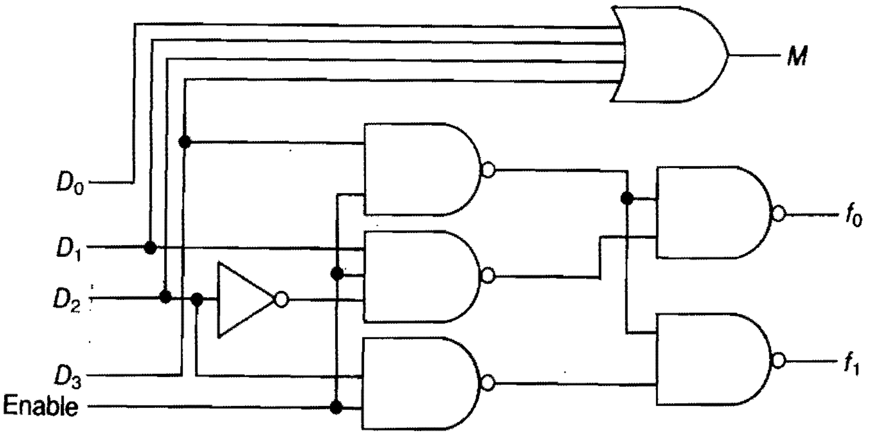


FIGURE 5.58

D_0	D_1	D_2	D_3	f_1	f_0
1	0	0	0	0	0
—	1	0	0	0	1
—	—	1	0	1	0
—	—	—	1	1	1

FIGURE 5.59



5.9 Error-Control Circuits

Errors may occur as digital codes are transmitted from one system or subsystem to another. There is always a possibility, albeit small, that a random-noise pulse will change a zero to a one or a one to a zero. It is possible, however, to code the data so that the occurrence of an error can be detected after the data have been received. The simplest approach is to add an extra bit, called a *parity bit*, to each of the number codes. If the coded data including the parity bit have an even number of ones, the code is said to have *even parity*. If the coded word including the parity bit has an odd number of ones, the code is said to have *odd parity*. Prior to sending a code, the number of ones are counted and the parity bit is set to make the number of ones odd or even as determined by the parity scheme chosen. At the receiving end, a check is made to see how many ones are present in the coded word. If odd parity is used and an even number of ones are received, it implies that an error has occurred. However, the proposed parity bit scheme cannot detect the occurrence of an even number of errors. For situations where the probability of multiple errors is high, a more sophisticated coding scheme must be used. In computers or communications equipment the possibility of random noise causing changes in more than one bit is low.

In order to check for or generate the proper parity bit in a given code, it is necessary to determine whether an odd or even number of ones are present. An X-OR gate functions in such a way that the output of an even number of ones is always a 0, and the output of an odd number of ones is always a 1. As an example, the circuit of Figure 5.60 makes use of these gates to generate an even parity bit for normal BCD input and to check for a possible error that may have been caused during transmission. The parity generator circuit at the source end examines the contents of the four data lines and accordingly generates a parity bit so that the encoded message (five bits in all) has even parity. At the receiver end the parity-checking circuit determines if an error has occurred or not. A high output at the parity-checking circuit indicates the occurrence of an error during transmission. This circuit could be made suitable for odd parity by simply replacing the final X-OR gates of both generator and checker circuits with X-NOR gates.

Several other schemes are also available, especially for coding decimal digits. The most common of these are the 2-out-of-5 code and 2-out-of-7 code. They are listed along with parity-coded BCD in the table of Figure 5.61. The 2-out-of-7 code is also known as the *biquinary code*. The zeroth through the sixth bit have positional weights of 0, 1, 2, 3, 4, 0, and 5 respectively. In both of these m-out-of-n codes, there are m ones and $(n - m)$ zeros. The advantages of these schemes are understood by comparing the different permissible codes. Two codes are said to be at a distance p if the codes differ from each other in p locations. Clearly, each code in an m-out-of-n

FIGURE 5.60 **Even Parity Generator-Checker Circuit.**

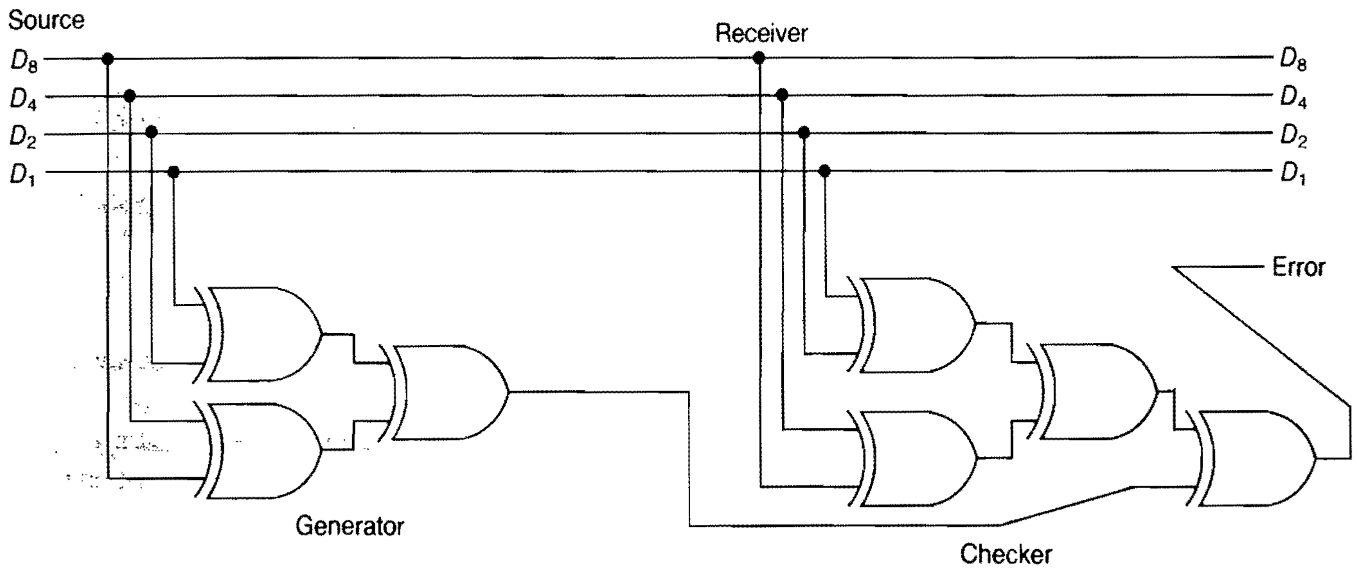


FIGURE 5.61 **Some Codes with Error Control.**

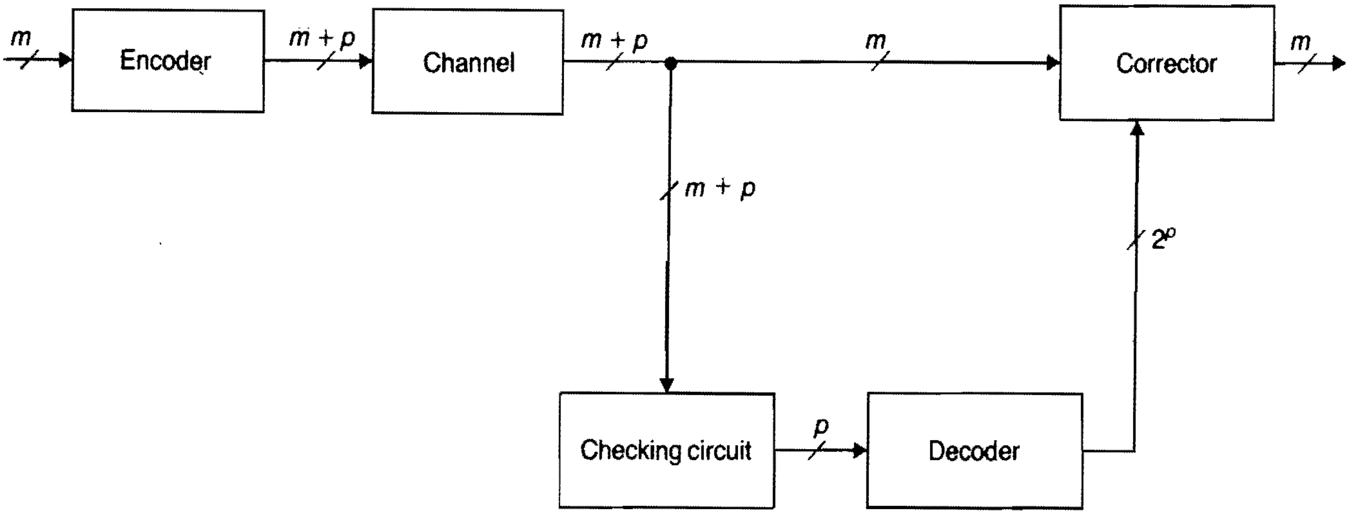
Decimal	BCD with Even Parity	BCD with Odd Parity	2-out-of-5	2-out-of-7
0	00000	00001	00011	0100001
1	00011	00010	00101	0100010
2	00101	00100	00110	0100100
3	00110	00111	01001	0101000
4	01001	01000	01010	0110000
5	01010	01011	01100	1000001
6	01100	01101	10001	1000010
7	01111	01110	10010	1000100
8	10001	10000	10100	1001000
9	10010	10011	11000	1010000

scheme is at least distance two away from the next code. Consequently, these codes can be used to detect single errors.

To correct k errors the minimum distance between two code words must not be smaller than $2k + 1$. A total of k errors would produce an error word k distance away from the correct code word. To be able to correct this error, no other k errors should be able to produce this same error word. The error word, therefore, should be at a distance at least $k + 1$ from any other code word. Accordingly, the minimum distance between two code words should be $2k + 1$. A minimum distance of two provides single-error detectability; any single error moves the code closer to where it was than to any other possible code. A circuit could be designed to move it back to its correct position. Of course, this minimum distance code could also be used instead for double-error detection. A minimum distance of four will provide both single-error correction plus double-error detection. A minimum distance of five would allow double-error correction. Next we will examine a code for which both error detection and error correction are straightforward.

One of the most useful codes is the *Hamming code*. This scheme not only provides for the detection of an error, but also locates the bit position in error so that it may be readily corrected. A block diagram of the implementation of this scheme is shown in Figure 5.62. The m bits of data are encoded with p parity bits before transmission. The received word is tested by a checking circuit to see if any error has occurred. The decoder then locates the exact position of error and, accordingly, the data are corrected by a corrector circuit.

FIGURE 5.62 **Block Diagram of the Hamming Code.**



The Hamming code uses multiple parity bits placed at specific locations of the coded word. The general rules for the generation of parity bits are summarized as follows:

1. If m is the number of information bits, then the number of parity bits, p, is equal to the smallest integer value of p that satisfies $2^p \geq m + p + 1$.
2. Parity bits are placed at the locations 1, 2, 4, 8, 16, and so on, of the coded word. Information bits are placed in order at locations 3, 5, 6, 7, 9, 10, and so on.
3. Each parity bit individually takes care of only a few bits of the coded word. To determine the bits of the coded word that are checked by a parity bit, every bit position is expressed in binary. A parity bit would check those bit positions, including itself, that have a 1 in the same loca-

tion of their binary representation as in the binary representation of the parity bit.

Example 5.15 illustrates the coding mechanisms involved.

EXAMPLE 5.15

Determine the Hamming-coded word for the message 10101 using even parity.

SOLUTION

The number of bits is $m = 5$; therefore, $p = 4$. This coding would result in a nine-bit coded word. The corresponding message bits (10101) are positioned respectively in locations 3, 5, 6, 7, and 9. These locations are specified in the table of Figure 5.63 as M_1, M_2, M_3, M_4, and M_5 respectively. To determine the exact value of each parity bit, each of the position designations is expressed at first in binary. The parity bits are generated from the following observations:

a. P_1 checks bit positions 1, 3, 5, 7, and 9 since all of these locations have a 1 in the LSB of their binary representations. The message bits present at four of these locations are, respectively, 1, 0, 0, and 1. The parity bit at location 0001, therefore, should be a 0 to maintain an even parity.

b. P_2 checks bit positions 2, 3, 6, and 7 because they all have a 1 at the same location in their binary representations. Bits 3, 6, and 7 house, respectively, a 1, 1, and 0. Therefore, P_2 must be a 0.

c. P_3 checks bit positions 4, 5, 6, and 7. Bits 5, 6, and 7 house, respectively, 0, 1, and 0, which requires that P_3 be a 1.

d. P_4 checks bit positions 8 and 9 and should be a 1 to maintain an even parity.

Therefore, the coded word is 001101011.

FIGURE 5.63

Bit Designation	P_1	P_2	M_1	P_3	M_2	M_3	M_4	P_4	M_5
Bit Position	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001
Message Bits			1		0	1	0		1
Parity Bits	0	0		1				1	

As mentioned earlier, the Hamming code also provides a means to detect and correct a single error. The *general detection algorithm* consists of the following steps:

Step 1. Check parity on each parity bit P_n and the bits for which it provides parity.

Step 2. If the test indicates the preservation of assumed parity, a 0 is assigned to the test result. A failed test is indicated by a 1.

Step 3. The binary number formed by the score of parity tests indicates the location of the bit in error.

Example 5.16 illustrates the detection mechanisms.

EXAMPLE 5.16

Determine if any bit is in error in the coded word 001101111. The message was coded with even assumption.

SOLUTION

The bit position table, as shown in Figure 5.64, is first prepared and then the coded bits are placed in their proper places. The following observations can be made regarding the coded word:

P_1 checks bits 1, 3, 5, 7, and 9. Consequently the first test fails since there are three 1s → 1 (LSB)

P_2 checks bits 2, 3, 6, and 7. This test also fails since there are three 1s → 1

P_3 checks bits 4, 5, 6, and 7. This test also fails since there are three 1s → 1

P_4 checks bits 8 and 9. This is a good check since there are two 1s → 0 (MSB)

The test score is 0111. The bit in error is the seventh bit (0111); therefore, the seventh bit is changed to 0. Therefore, the correct coded word should be 001101011. This result agrees with the earlier findings of Example 5.15.

FIGURE 5.64

Bit Designation	P_1	P_2	M_1	P_3	M_2	M_3	M_4	P_4	M_5
Bit Position	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001
Received Message	0	0	1	1	0	1	1	1	1

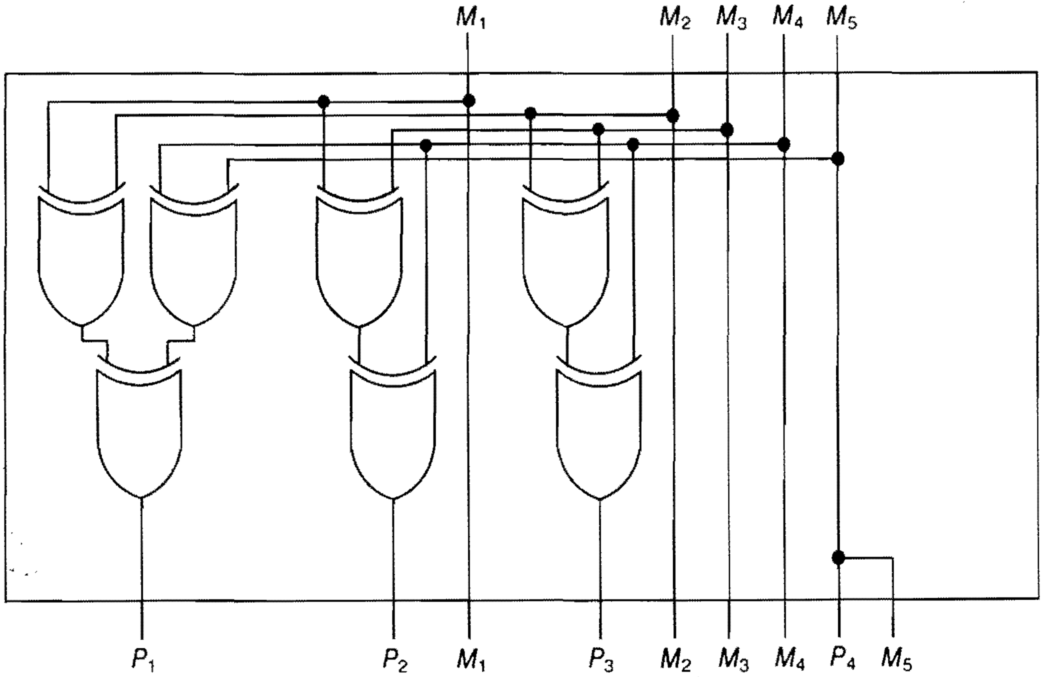
The hardware implementations of the Hamming code scheme are relatively easy to achieve. Consider as an example a logic circuit for processing five bits of data, just as in the last two examples. The parity bit generator circuit is to generate four additional bits: P_1, P_2,

P_3, and P_4. The rules for the generation of parity bits, as stated earlier, may be used to obtain

$$P_1 = M_1 \oplus M_2 \oplus M_4 \oplus M_5$$
$$P_2 = M_1 \oplus M_3 \oplus M_4$$
$$P_3 = M_2 \oplus M_3 \oplus M_4$$
$$P_4 = M_5$$

The generator circuit may be implemented with X-OR gates. The resultant circuit is shown in Figure 5.65. Correspondingly, for an odd parity assumption the final X-OR gates of all stages need to be replaced with X-NOR gates and M_5 inverted to generate P_4. It is important to note that the minimum distance between two code words is three because one change in the data bits produces at least two changes in the parity bits.

FIGURE 5.65 **Hamming-Coded Message Generator.**

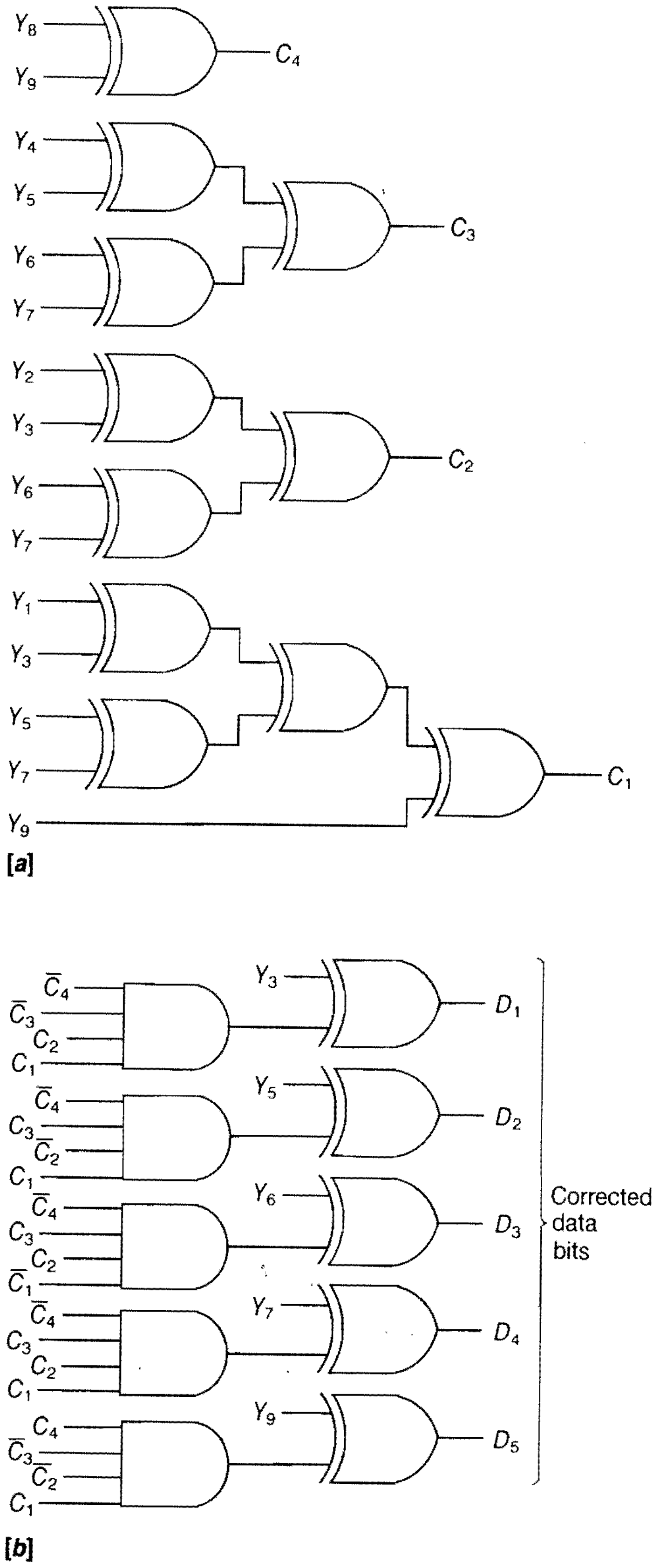


Using similar reasoning the parity-checking circuit may also be designed. The parity of each parity bit and its corresponding data bits are tested. The Boolean equations for the nine-input parity-testing circuit are readily obtained as follows:

$$C_1 = Y_1 \oplus Y_3 \oplus Y_5 \oplus Y_7 \oplus Y_9$$
$$C_2 = Y_2 \oplus Y_3 \oplus Y_6 \oplus Y_7$$
$$C_3 = Y_4 \oplus Y_5 \oplus Y_6 \oplus Y_7$$
$$C_4 = Y_8 \oplus Y_9$$

where $Y_1, Y_2, \ldots, Y_8$, and Y_9 correspond, respectively, to P_1, P_2, M_1, P_3, M_2, M_3, M_4, P_4, and M_5 of the generator circuit. The parity-checking circuit is again realized using X-OR gates and is shown in Figure 5.66[*a*]. The value $C_4C_3C_2C_1$ points to the bit in error and

FIGURE 5.66 [*a*] **Parity-Checking Circuit and [*b*] Correction Circuit.**



may be used to correct the bit. Our interest is in recovering the corrected data bits. Making use of the X-OR programmable inverter function, the equations for the corrected bits are obtained as follows:

$$D_1 = (\overline{C}_4 \cdot \overline{C}_3 \cdot C_2 \cdot C_1) \oplus Y_3$$
$$D_2 = (\overline{C}_4 \cdot C_3 \cdot \overline{C}_2 \cdot C_1) \oplus Y_5$$
$$D_3 = (\overline{C}_4 \cdot C_3 \cdot C_2 \cdot \overline{C}_1) \oplus Y_6$$
$$D_4 = (\overline{C}_4 \cdot C_3 \cdot C_2 \cdot C_1) \oplus Y_7$$
$$D_5 = (C_4 \cdot \overline{C}_3 \cdot \overline{C}_2 \cdot C_1) \oplus Y_9$$

For example, if the parity test results are 0111, then the fourth data bit (i.e., the seventh bit of the coded word) is in error. The bit in error is corrected by complementing Y_7. All of the other bits remain unchanged. The corresponding error-correcting circuit is shown in Figure 5.66[*b*].

5.10 Summary

In this chapter different aspects of combinational circuit design were presented. Primary emphasis was placed on designing smaller and manageable modules, several of which were then cascaded together to realize a robust system. In particular, the design and working principles of binary and nonbinary adders/subtracters, code converters, decoders, encoders, and various error-correcting circuits were explored. Many of these devices will be used time and again throughout the rest of this text for developing more advanced concepts.

Problems

1. Design an FA circuit using logic gates suitable for adding two bits of addend, two bits of augend, and carry-in input.
2. Obtain a single-bit FA using only MUXs.
3. Design a single-bit FA using only NOR gates.
4. Use the FAs designed in Problem 1 to perform addition of six-bit numbers. Show the configuration of the setup for adding $(110110)_2$ and $(000010)_2$.
5. Design a four-bit FA using combinational logic.
6. Design a four-bit FA using ROM technology.
7. Use bridging to implement a standard full subtracter circuit (three inputs and two outputs) using X-OR gates.
8. Verify Equations [5.4] and [5.6].
9. Design a circuit for dividing a four-bit number by a four-bit number.
10. The following message needs to be transmitted using the Hamming code under even parity assumption. Determine the

parity bits and the order in which the coded message will be sent. The to-be-coded message is 1010111001011. Show the corresponding circuit.

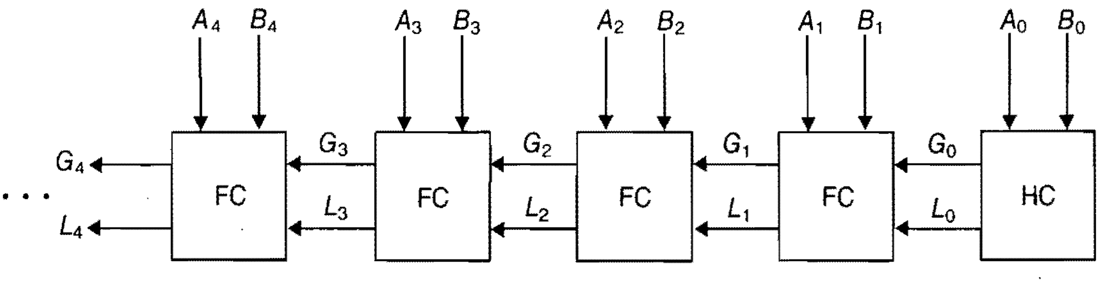
11. Design a half subtracter circuit using (a) only NOR gates and (b) only MUXs.
12. The Hamming-coded message received under odd parity assumption is 1010111001011. Determine if the message has any error and write out the correct message bits only. Obtain the corresponding correction circuit.
13. Design a full subtracter using half subtracter modules.
14. Using only a four-bit binary adder, design decimal code converters for the following conversions:
    a. 8-4-2-1 to XS3
    b. XS3 to BCD
    c. XS6 to XS3
    d. BCD to XS3
15. Remove the combinational FAs from the circuit of Example 5.2 and replace these with equivalent ROMs. Show the ROM logic for one of these units and determine the total ROM size needed for the complete circuit.
16. Design a combinational circuit capable of comparing two eight-bit binary integers (without sign bits) X and Y. The output Z should be a 1 whenever $X \geq Y$.
17. Design a controllable, dual-purpose, four-bit converter that converts binary to Gray and also Gray to binary.
18. Use the module of Example 5.7 for obtaining the following conversions:
    a. 15-bit
    b. 20-bit
    c. 25-bit

    Justify your designs using exemplary nontrivial binary inputs.
19. Use the module of Example 5.10 for obtaining the following conversions:
    a. 15-bit
    b. 20-bit
    c. 25-bit

    Justify your designs using exemplary nontrivial BCD inputs.
20. Design an adder/subtracter using cascaded ALUs. Show how it works when adding and when subtracting if $A = 84$ and $B = 32$. Repeat the problem using base-16 equivalents of the numbers.
21. Design a 12-bit FA in which carries are allowed to ripple after the first six bits of addition.

22. Show how the ALU can be used to (a) subtract one from and (b) add one to a number. Show the setup if the number is 76_{10}.
23. Show how a 3-8 line decoder could be used to generate $f(A,B,C) = \Sigma m(0,1,3,5)$.
24. Design a logic circuit that multiplies an input decimal digit (in BCD) by five. The output is also in BCD form. Show that the outputs can be obtained from the input lines without using any logic gates.
25. Implement the FA circuit of Problem 1 using MUXs.
26. Obtain the most minimal circuit that squares a three-bit binary number.
27. Design a special-purpose unit using FAs (a) for adding 12 single-bit binary numbers and (b) for adding 17 single-bit binary numbers.
28. Design a four-bit CLA circuit where the propagate function is defined as $P_i = A_i + B_i$ instead of $P_i = A_i \oplus B_i$. How does the current design differ from the one discussed in Section 5.4?
29. Use the techniques considered in Section 5.4 to obtain a four-bit fast subtracter.
30. Obtain the CLA carry equations when $n > 13$ and show that the maximum fan-out is dependent on variable $P_{(n-2)/2}$ and is equal to $\{[(n + 1)^2/4] + 2\}$ for odd n. Also show that for even n, the maximum fan-out is dependent on both $P_{(n/2)-1}$ and $P_{(n/2)}$ and is equal to $\{[n(n + 2)]/4 + 2\}$.
31. Design an n-bit binary comparator circuit to test if an n-bit number A is equal to, larger than, or smaller than a second n-bit number B. The problem could be broken into one unit of a half comparator module and $n - 1$ units of full comparator modules, as shown in Figure 5.P1. Each of the modules gives out two outputs: G_n and L_n, such that
    a. $G_n = 1$ and $L_n = 0$ if $A_n > B_n$
    b. $G_n = 0$ and $L_n = 1$ if $A_n < B_n$
    c. When $A_n = B_n$, then $G_n = G_{n-1}$ and $L_n = L_{n-1}$ for a full comparator and $G_n = L_n = 0$ for a half comparator.

FIGURE 5.P1



Describe the working principles of the design and show how you could improve upon this design.

32. Design an n-bit comparator module different from that of Problem 31 such that the comparison process begins from the MSB and moves toward the LSB until the final decision is made.
33. Obtain the circuit for a 16-input, 4-output priority encoder.
34. Design a serial-to-parallel converter circuit that routes a long sequence of binary digits into four different output lines as specified by external control signals.
35. Design a BCD adder circuit that first converts the BCD numbers into binary, then adds the resulting binary numbers, and finally converts the binary sum to the correct BCD sum.

Suggested Readings

Agrawal, D. P. "Negabinary carry look ahead adder and fast multiplier." *Elect. Lett.* vol. 10 (1974): 312.

Arazi, B. "An electrooptical adder." *Proc. IEEE.* vol. 73 (1985): 162.

Bartee, T. C. *Digital Computer Fundamentals.* New York: McGraw-Hill, 1985.

Benedek, M. "Developing large binary to BCD conversion structures." *IEEE Trans. Comp.* vol. C-26 (1977): 688.

Bin Nun, M. A., and Woodward, M. E. "Halfadders modulo-2 using read-only memories." *Elect. Lett.* vol. 10 (1974): 213.

Bywater, R. E. H. *Hardware/Software Design of Digital Systems.* Englewood Cliffs, N.J.: Prentice-Hall International, 1981.

Daws, D. C., and Jones, E. V. "Hardware-efficient bit sequential adders and multipliers using mode-controlled logic." *Elect. Lett.* vol. 16 (1980): 434.

Feldstein, A., and Goodman, R. "Loss of significance in floating point subtraction and addition." *IEEE Trans. Comp.* vol. C-31 (1982): 328.

Floyd, T. L. *Digital Fundamentals.* 2d ed. Columbus, Ohio: Charles E. Merrill, 1982.

Gaitanis, N. "Single error correcting and multiple unidirectional error detecting cyclic AN arithmetic codes." *Elect. Lett.* vol. 20 (1984): 638.

Greer, C. R., and Thompson, R. A. "Combinational logic design with decoders." *IEEE Trans. Comp.* vol. C-27 (1978): 869.

Hamming, R. W. *Coding and Information Theory.* Englewood Cliffs, N.J.: Prentice-Hall, 1980.

Lai, H. C., and Muroga, S. "Minimum parallel binary adders with NOR(NAND) gates." *IEEE Trans. Comp.* vol. C-28 (1979): 648.

Langdon, G. G., Jr., and Tang, C. K. "Concurrent error detection for group look-ahead binary adders." *IBM J. Res. & Dev.* vol. 14 (1970): 563.

Ling, H. "High-speed binary adder." *IBM J. Res. & Dev.* vol. 25 (1981): 156.

Liu, T. K.; Hohulin, K. R.; Shiau, L. E.; and Muroga, S. "Optimal one-bit full adders with different types of gates." *IEEE Trans. Comp.* vol. C-23 (1974): 69.

Majerski, S. "On determination of optimal distributions of carry skips in adders." *IEEE Trans. Elect. Comp.* vol. EC-16 (1967): 45.

Peterson, W. W. "Error correcting codes." *Sci. Am.* vol. 215 (1962): 96.

Schmookler, M. S. "Design of large ALUs using multiple PLA Macros." *IBM J. Res. & Dev.* vol. 24 (1980): 2.

Wakerley, J. *Error Detecting Codes, Self-Checking Circuits and Applications.* Amsterdam: North-Holland, 1978.

Sequential Devices

CHAPTER SIX

6.1 Introduction

A circuit is known as combinational as long as its steady-state outputs depend only on its current inputs. If, on the other hand, the present value of the outputs are dependent on both the present values of the inputs and the past values of the inputs, the circuit is considered to be a *sequential circuit.* One of the important applications of digital techniques is where digital signals are received and interpreted by the system, and control outputs are generated in accordance with the sequence in which the input signals are received. Therefore, such systems require circuits that respond to the past history of the inputs. In general, sequential circuits have the capability of storing information. Consequently, sequential circuits find wide application in digital systems as counters, registers, control logic, memories, and other complex functions.

The most common sequential circuit is the flip-flop. A *flip-flop (FF)* is an electronic device that has two stable states. One state is assigned the logic 1 value and the other the logic 0 value. The output of the FF can assume either of the stable states based on input events, and the output can be checked to determine what event occurred in the pair. There are a number of FFs in common usage in digital circuits, and they differ from one another in the number of inputs they have and in the manner in which the binary state is affected by the inputs. The possible changes in the FF outputs generally have a direct correspondence to the frequency with which the input is changing value. However, there is a type of sequential circuit memory device, known as a *monostable multivibrator*, that produces circuit output independent of the input frequency. This chapter introduces the logical behavior and control of various types of FFs. After studying this chapter, you should be able to:

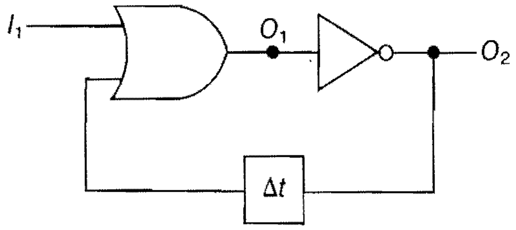
- Understand the design and working principles of latches;
- Understand the design and working principles of FFs;

- ○ Understand the design and working principles of the monostable multivibrator;
- ○ Understand the importance and significance of sequential circuits in general.

6.2 Latches

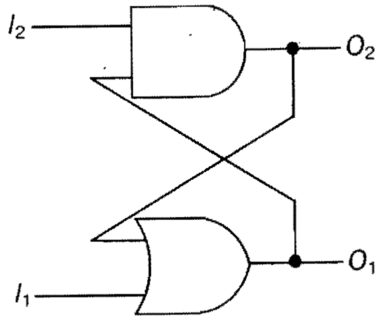
A *latch* is a bistable circuit that is the fundamental building block of a flip-flop. The latch is basically a combinational circuit that has one of its outputs fed back as an input. It can be realized from an OR gate with its complemented output fed back as one of its inputs, as shown in Figure 6.1. We have considered Δt to be the *lumped* gate delay (total of all propagation delays) of the gates that are used. If the input I_1 is held at logic 1, the OR output results in a 1 and, therefore, the complemented output, O_2, is a 0. This O_2 output is fed back to the OR gate after a time equal to the lumped delay. As a result the output of the OR gate is held at logic 1. Once the OR output is set to this condition, the gate output will remain in this same state. This phenomenon is commonly known as the *latching effect.* Consequently, this circuit could be used for the storage of logic 1. This latching condition will prevail until the feedback path is broken.

FIGURE 6.1 **Latch for Storing Logic 1.**



The NOT gate of Figure 6.1 may now be replaced by an AND gate to provide for the storage of logic 0. Such provision is known as the *unlatching* of the gate, which is illustrated in Figure 6.2. This circuit, however, is able to store both logic 1 and 0. Each of the two latch outputs, O_1 and O_2, is a logical complement of the other. If one holds I_2 input to logic 0, the feedback route between the input and the output of the OR gate will be logically broken and the OR gate

FIGURE 6.2 **AND-OR Latch.**



will return to its initial condition. When the input I_2 is a 0, the output O_2 is maintained at logic 0, and when I_1 is held at logic 1, the output O_1 remains at logic 1.

The latching concept developed in this section will be used next to come up with a standardized latch suitable for subsequent development of FFs. An FF has the capability of storing a single binary bit of information. When the values stored in the FFs change, we say that the sequential circuit *changes state*. Generally, however, an FF should have two outputs, called Q and $\overline{Q}$, that are complements of each other. The characteristic table of Figure 6.3 details the pertinent working principles of one such basic latch unit, where t is used to denote the time variable and Δt is the short time duration between a change in the input and a possible change in the output. The interval Δt is equivalent to the lumped delay of the circuit. The two inputs S (set) and R (reset) are used to control the output based on the current state of the output. If $R = 0$ and $S = 1$, the output is turned on if not already on. If $R = 1$ and $S = 0$, the output is turned off if not already off. When $S = R = 0$, no output change occurs. The to-be-designed latch circuit, however, manifests an undesirable condition when both inputs go to 1 simultaneously. When $S = R = 1$, the two outputs, Q and $\overline{Q}$, would no longer be complements of each other. In addition, the behavior of the latch would become unpredictable once the inputs returned to 0. Consequently, the simultaneous existence of $S = R = 1$ is forbidden.

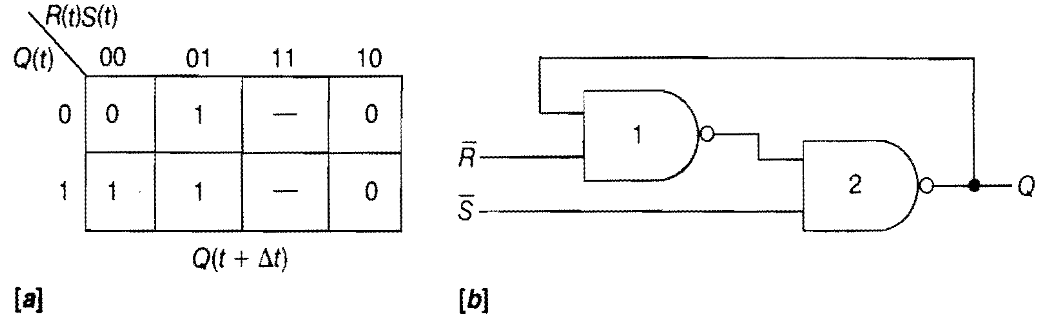
FIGURE 6.3 **Characteristic Table for a Basic Latch.**

Inputs			Output
$R(t)$	$S(t)$	$Q(t)$	$Q(t+\Delta t)$
0	0	0	0
		1	1
0	1	0	1
		1	1
1	0	0	0
		1	0
1	1	0	—
		1	—

The circuit may be determined accordingly, using the K-map of Figure 6.4[*a*]. The equation for $Q(t + \Delta t)$ is obtained as follows:

$$\begin{aligned} Q(t + \Delta t) &= S(t) + \overline{R(t)}Q(t) \\ &= \overline{\overline{S(t)} \cdot \overline{\overline{R(t)}Q(t)}} \qquad [6.1] \end{aligned}$$

FIGURE 6.4 *RS* Latch: [*a*] K-map and [b] Circuit.



$Q(t)$ \ $R(t)S(t)$	00	01	11	10
0	0	1	—	0
1	1	1	—	0

This equation, known also as the *next-state equation*, states that after a short time, Δt, the new value of Q is determined by the values of Q, R, and S at time t. The corresponding circuit is shown in Figure 6.4[*b*]. If R and S values change at time t, a new value of Q will result Δt time later. Δt time in this circuit is the total gate delay of the two NAND gates. The output of NAND gate 1 is traditionally known as the $\bar{Q}$ output since the outputs of two NAND gates are complements of each other.

FIGURE 6.5 Revised *RS* Latch Characteristic Table.

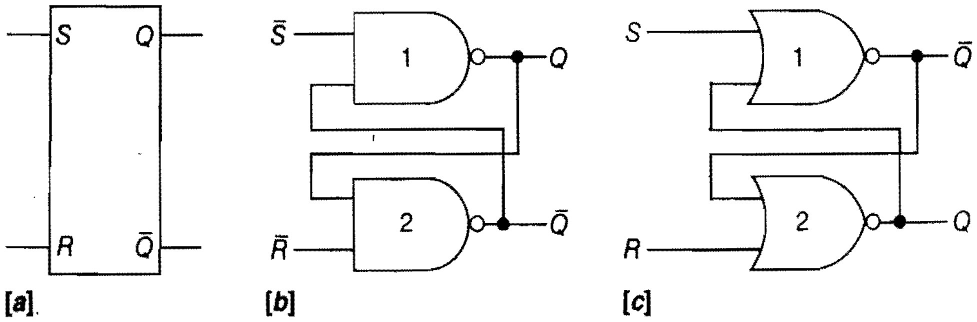
$Q(t)$	$R(t)$	$Q(t + \Delta t)$
0	0	$S(t)$
0	1	0
1	0	1
1	1	0

The revised characteristic table of Figure 6.5 shows the corresponding characteristics of the *RS latch* (also called reset-set latch) where the don't-care of the forbidden state is assumed to be equal to a 0. The equation for $Q(t + \Delta t)$ may also be obtained as follows:

$$\begin{aligned} Q(t + \Delta t) &= \bar{Q}(t)\bar{R}(t)S(t) + Q(t)\bar{R}(t) \\ &= \bar{R}(t)[Q(t) + \bar{Q}(t)S(t)] \\ &= \bar{R}(t)[Q(t) + S(t)] \\ &= \overline{R(t) + [\overline{Q(t) + S(t)}]} \qquad [6.2] \end{aligned}$$

This NOR form of latch could also be derived by grouping the zeros of the K-map of Figure 6.4[*a*]. The two circuits, NAND and NOR latches, are respectively known as *RS* and *SR* flip-flops or more commonly as latches. The corresponding latch circuits are obtained as shown in Figure 6.6.

FIGURE 6.6 *SR* Latch: [*a*] Block Diagram, [*b*] NAND Circuit, and [*c*] NOR Circuit.



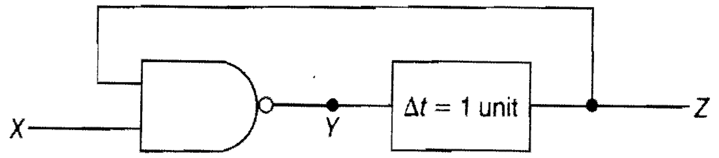
Note that the latch outputs are always complements of each other: When Q is a 1, $\bar{Q}$ is 0, and when Q is a 0, $\bar{Q}$ is 1. The forbidden state occurs when $S = R = 1$ at the same time. As long as S and R are both set at 1, Q and $\bar{Q}$ are forced to be at the same logic value simultaneously, thus violating the basic complementary nature of the outputs.

EXAMPLE 6.1

Obtain the response of a periodic square wave of period 8 units, when it is fed into the sequential circuit of Figure 6.7. Assume that the NAND gate has a total delay of 1 unit.

SOLUTION

FIGURE 6.7

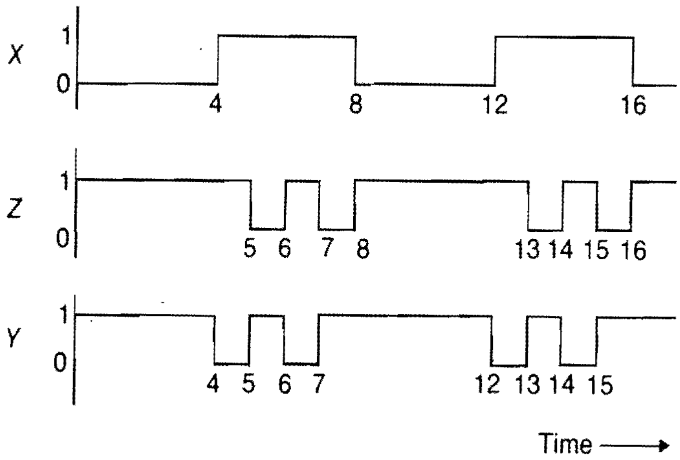


The output, Z, is given by

$$Z(t + \Delta t) = \overline{X(t)Z(t)} = \bar{X}(t) + \bar{Z}(t)$$

Consequently, the timing diagram is obtained as shown in Figure 6.8. Whenever X changes from a 0 to a 1, the output Z starts oscillating with a period of $2\Delta t$ (2 units in this case). However, when X changes from a 1 to a 0, the output Z becomes 1 after a time delay of Δt.

FIGURE 6.8



The SR latch described earlier has its time delay lumped together. However, there is another form of the latch model known commonly as the *distributed gate delay model.* Consider, for example, the NAND latch of Figure 6.6[*b*] where the gates numbered 1 and 2 are assumed to have gate delays t_1 and t_2, respectively. Accordingly,

$$Q(t + t_1) = \overline{\bar{S}(t) \cdot \bar{Q}(t)} = S(t) + Q(t) \qquad [6.3]$$

$$\bar{Q}(t + t_2) = \overline{\bar{R}(t) \cdot Q(t)} = R(t) + \bar{Q}(t) \qquad [6.4]$$

These two equations now may be combined to yield

$$\overline{Q}(t + t_1 + t_2) = \overline{\overline{R}(t + t_1) \cdot Q(t + t_1)}$$
$$= \overline{\overline{R}(t + t_1)\,[S(t) + Q(t)]} \qquad [6.5]$$

Therefore,

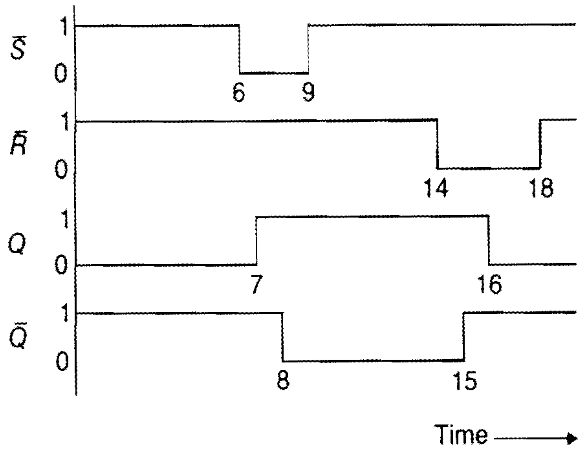
$$Q(t + t_1 + t_2) = \overline{R}(t + t_1)[S(t) + Q(t)] \qquad [6.6]$$

Assuming equal gate delays for a total of Δt delay, the resulting equation becomes

$$Q(t + \Delta t) = \overline{R}\left(t + \frac{\Delta t}{2}\right)[S(t) + Q(t)] \qquad [6.7]$$

The timing diagrams of Figure 6.9 show the behavior pattern of a NAND latch corresponding to the distributed gate delay model. For simplicity it has been assumed that both of the gates have 1 unit length of gate delay and also that Q and $\overline{Q}$ at time $t = 0$ are, respectively, 0 and 1. This timing diagram shows that the latch works as intended. However, under various input conditions the latch may have problems. Example 6.2 will illustrate one of these input conditions and its consequences.

FIGURE 6.9 **Timing Diagram of a NAND Latch.**

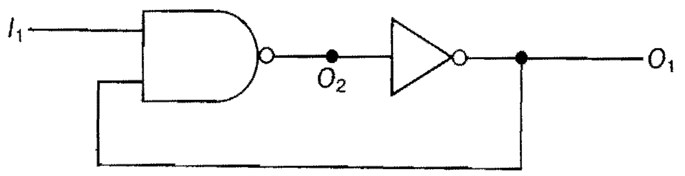


EXAMPLE 6.2

Obtain the timing diagram for the latch of Figure 6.10 when the input I_1 changes from 1 to 0 for a duration much shorter than the total gate delay.

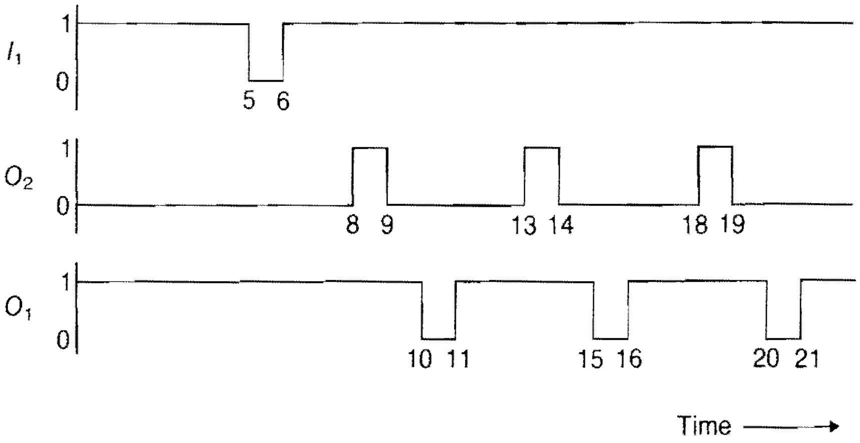
SOLUTION

FIGURE 6.10



For this example the gate delays for the NAND and NOT gates are chosen to be 3 units and 2 units, respectively. Accordingly, the timing diagrams of Figure 6.11 are obtained. The output is oscillatory in nature and is not latched to any fixed value.

FIGURE 6.11



6.3 Clocked *SR* Flip-Flop

In the last section we introduced circuits for the *SR* FF. For dependable operation of such devices one must attempt to prevent transient pulses from appearing on either of the inputs. It is advantageous to control the times when the *SR* FF output is allowed to change by means of an additional input. This additional signal is commonly called a *clock*. The clock pulses (*CK*) can be periodic or a set of random pulses. Almost always, however, they are periodic.

The purpose of the clock input is to force the FF to remain in its rest (or hold) state while changes occur on the set and reset inputs. *CK* is set to logic 1 once the inputs have settled. The NAND and NOR latches with clock input are shown in Figure 6.12. In order to operate these devices effectively, the following conditions must be met:

1. The FF inputs should be allowed to change only when $CK = 0$.
2. The clock input should be long enough so that the outputs will be able to reach steady states.
3. The condition $S = R = 1$ must not be allowed to occur when *CK* is equal to logic 1. For proper operation, therefore, $S(t)R(t)$ should always equal zero.

It can be seen that the circuit action can occur only when the *CK* signal is high. When $CK = 0$, the FF outputs do not change. The *S* and *R* inputs may, however, be simultaneously high when the clock is absent since the FF will be inhibited. The overall functioning of the gated *SR* FF is illustrated by the characteristic table of Figure 6.13. Note in the timing diagram shown in Figure 6.14 that it is

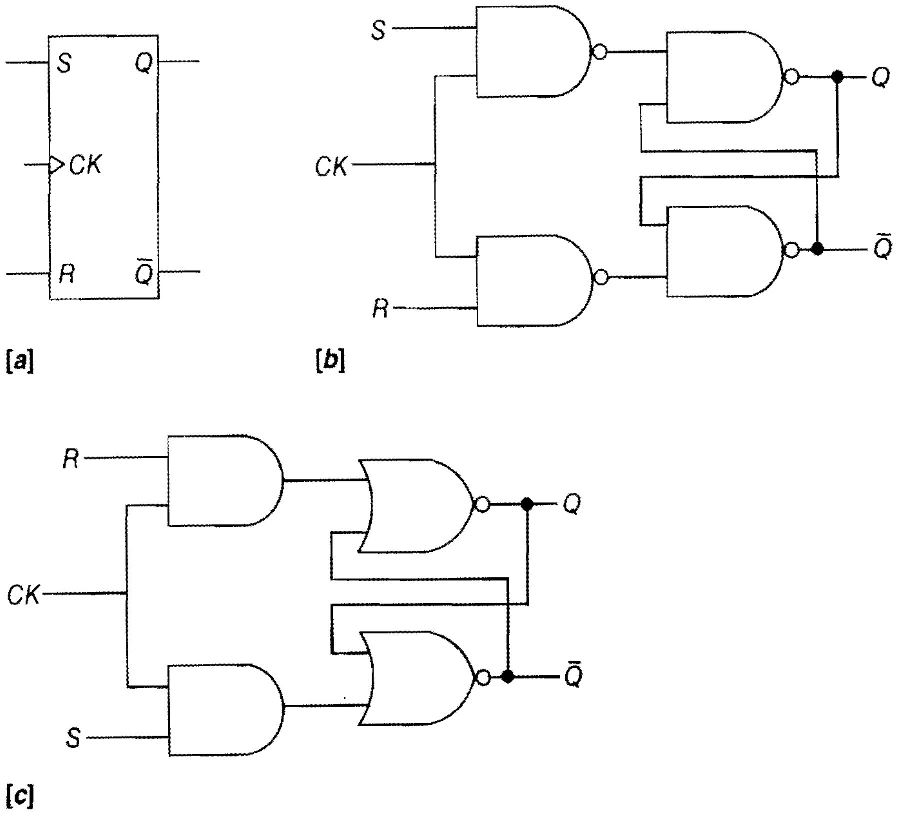
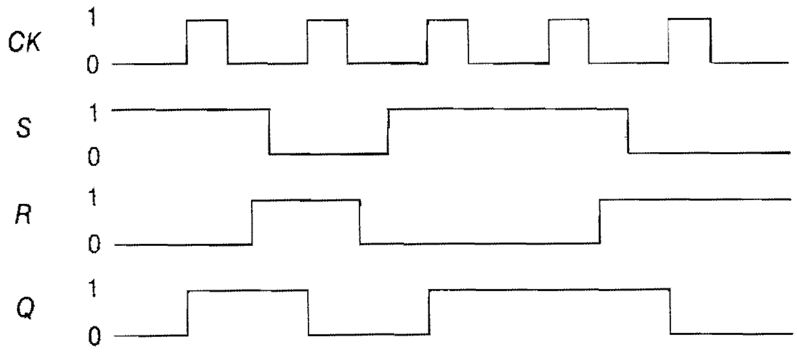


FIGURE 6.12 Clocked *SR* FF: [*a*] Logic Symbol, [*b*] NAND Gate Circuit, and [*c*] NOR Gate Circuit.

FIGURE 6.13 Characteristic Table for the Clocked *SR* FF.

Inputs				Outputs	
$CK(t)$	$S(t)$	$R(t)$	Mode	$Q(t + \Delta t)$	$\bar{Q}(t + \Delta t)$
0	—	—	No action	$Q(t)$	$\bar{Q}(t)$
⎍	0	0	Hold	$Q(t)$	$\bar{Q}(t)$
⎍	0	1	Reset	0	1
⎍	1	0	Set	1	0
⎍	1	1	Invalid	—	—

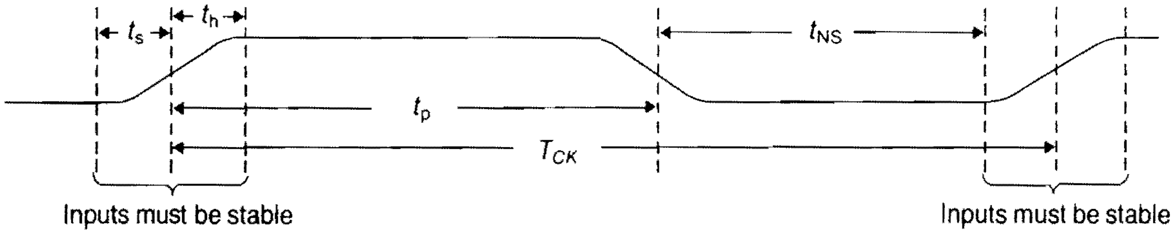
FIGURE 6.14 Timing Diagram of a Clocked *SR* FF.



necessary to consider the circuit only at the time *CK* changes from low to high to see if the output changes.

It is now appropriate to introduce several operational characteristics that are commonly associated with the FF usages. Figure 6.15 shows some of these specifications, of which setup and hold time are the most important ones. The *setup time*, t_s, is the time necessary for

FIGURE 6.15 Timing Characteristics under Worst-Case Condition for Maximum Clock Frequency Determination.



the input data to stabilize before the triggering edge of the clock. Its value is extremely critical since it manifests itself either by ignoring actions or by resulting in partial transient outputs, commonly referred to as *partial set* and *partial reset* outputs. Consequently, it is possible to begin a set or reset mode, causing the output to start to change, but to withdraw back to its initial state. In some cases the output might even end up in a metastable state in which the FF is neither set nor reset. Again, the *hold time*, t_h, is the time necessary for the data to remain stabilized beyond the triggering edge of the clock. This is also a critical parameter in determining the correct behavior of a FF.

The maximum allowable clock frequency for an FF is usually determined from a knowledge of setup time; hold time; FF propagation delay, t_p; and propagation delay of the next-state decoder, t_{NS}. The maximum clock frequency, f_{CK}, under worst-case condition is obtained from

$$f_{CK} = \frac{1}{T_{CK}} \leq \frac{1}{t_s + t_p + t_{NS}} \qquad [6.8]$$

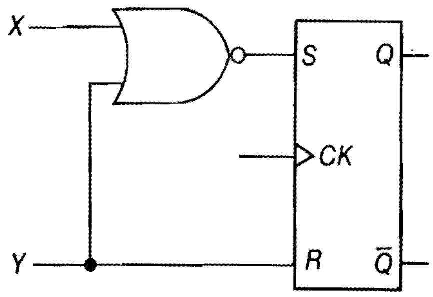
The constraint of Equation [6.8] must be met when using any FF, integrated or not.

EXAMPLE 6.3

Obtain $Q(t + \Delta t)$ as a function of the inputs and $Q(t)$ in the circuit of Figure 6.16.

SOLUTION

FIGURE 6.16



From Equation [6.1] the next-state equation follows as

$$Q(t + \Delta t) = S(t) + \bar{R}(t)Q(t)$$

In this circuit $S(t) = \overline{X(t) + Y(t)} = \bar{X}(t) \cdot \bar{Y}(t)$ and $R(t) = Y(t)$. Note also that $S(t)R(t) = \bar{X}(t) \cdot \bar{Y}(t) \cdot Y(t) = 0$. Hence, $S(t)$ and $R(t)$ are not simultaneously equal to 1 and, therefore, the circuit meets all conditions necessary for a perfect operation. Consequently, we may obtain

$$Q(t + \Delta t) = \bar{X}(t)\bar{Y}(t) + \bar{Y}(t)Q(t)$$

The table, as shown in Figure 6.17, lists all possible combinations of the inputs and the corresponding outputs.

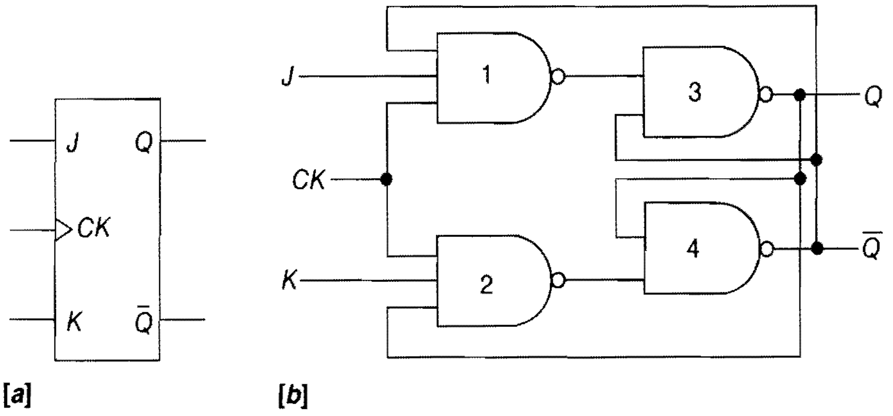
FIGURE 6.17

$X(t)$	$Y(t)$	$Q(t)$	$Q(t + \Delta t)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

6.4 *JK* Flip-Flop

We saw in the last section that the clocked *SR* FF has an indeterminate state. When using clocked *SR* FFs the designer is required to be cautious about the FF inputs. This troublesome restriction can be removed by modifying the *SR* FF; the refined FF is known as the *JK* FF. This modification involves feeding the outputs of the FF back into the inputs of the circuit shown in Figure 6.12[*a*]. The resulting circuit, its block diagram, and its functional behavior are shown in Figures 6.18[*a*–*c*].

FIGURE 6.18 ***JK* FF: [*a*] Block Diagram, [*b*] Logic Circuit, and [*c*] Characteristic Table.**



Mode	$J(t)$	$K(t)$	$Q(t)$	$Q(t + \Delta t)$
Hold	0	0	0	0
			1	1
Reset	0	1	0	0
			1	0
Set	1	0	0	1
			1	1
Toggle	1	1	0	1
			1	0

[*c*]

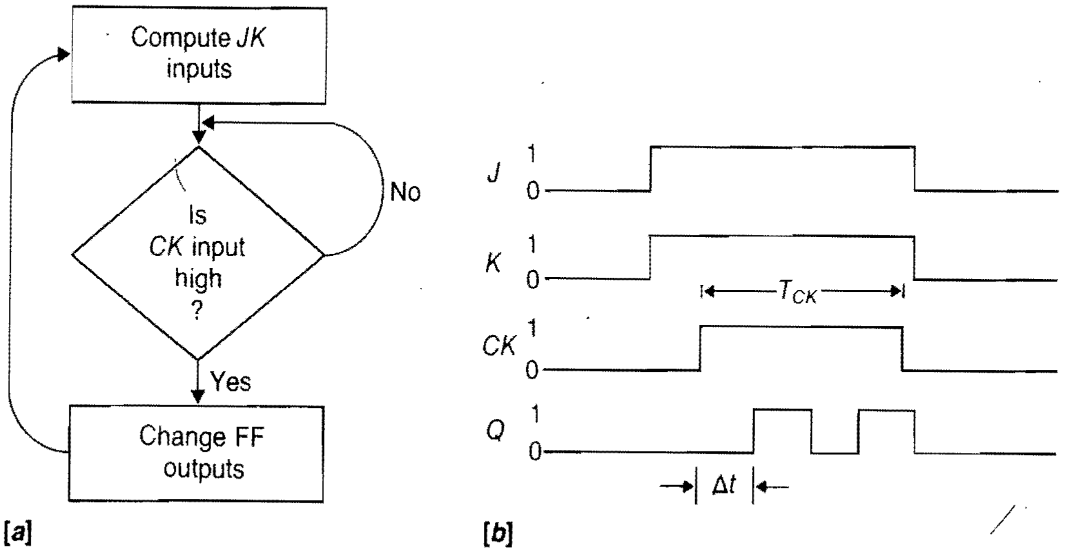
Even when the J and K inputs are both 1, the outputs of NAND gates 1 and 2 cannot simultaneously be 0. With $Q = 0$, NAND gate 2 outputs a 1, and when $Q = 1$, NAND gate 1 outputs a 1. Consequently, the input restriction of the *SR* FF is automatically eliminated. The additional feedback provides for an additional switching mode, called *toggle*, to the FF. The characteristic table of Figure 6.18[*c*] describes in detail the actions of the FF. The next-state equation may accordingly be obtained as follows:

$$Q(t + \Delta t) = J(t)\overline{Q}(t) + \overline{K}(t)Q(t) \qquad [6.9]$$

If $J = 1$ and $K = 0$, the FF is set to an output of 1 if not already set. Similarly when $J = 0$ and $K = 1$, the FF resets to 0 if not already reset. If $J = K = 1$, the FF output is complemented (toggled), and when $J = K = 0$, no change takes place.

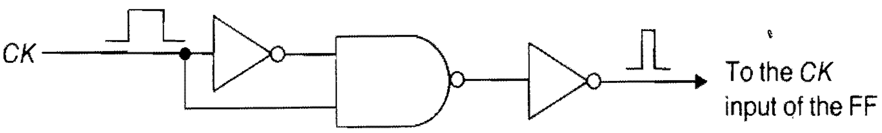
In spite of many advantages the *JK* FF still has a serious limitation, which is illustrated in Figure 6.19 and should be understood by the designer. When the clock input goes high, the FF responds according to the J and K inputs. The flowchart of Figure 6.19[*a*] shows the desired circuit operation and the consequence of having a clock pulse that is too long. If the Q output changes before the termination of the clock input, then the input conditions to NAND gates 1 and 2 change again, and this leads to subsequent change in the Q output. As a consequence, Q may be indeterminate at the termination of the clock input. Such a possibility exists as long as Δt is less than T_{CK}. It is desired that only one FF change occur during each clock input. This may be easily accomplished by maintaining the clock width much smaller than the total delay time.

FIGURE 6.19 [*a*] *JK* FF Flowchart and [*b*] Pulse Width Problem in *JK* FF.



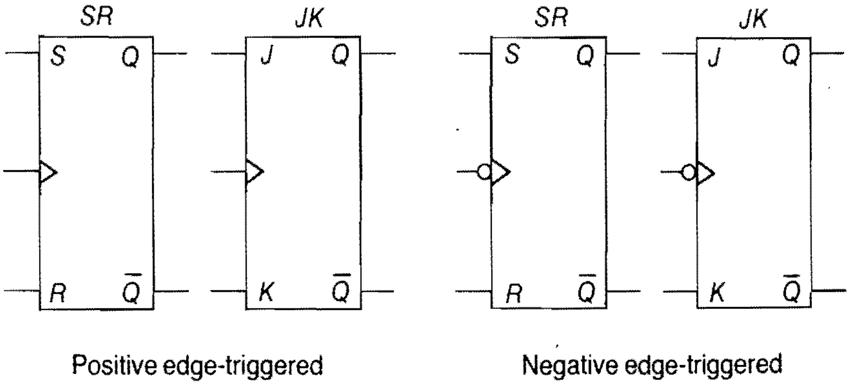
Another way to eliminate the problem caused by a clock pulse width that is too long is to design the FFs to respond to only transitions of the clock, either 1 → 0 or 0 → 1. Edge-triggered FFs are provided with a pulse-narrowing circuit, as shown in Figure 6.20. As you can see, there is a small delay on one of the NAND gate inputs so that the inverted clock pulse arrives at the gate input a couple of nanoseconds later than the true clock pulse. This results in an output spike of an extremely small time duration at the very beginning of the clock pulse. This narrow pulse is then used for the clock input of the *JK* FF, eliminating the necessity for narrow clock pulses. It is appropriate to consider the effect this narrow pulse might have on the FF actions. Depending on the parameters of the gates that are being used in the circuit of Figure 6.20, the resulting pulse could be too narrow to trigger an FF. In such an event more than one, but only an odd number of, NOT gates could be used in place of the first NOT gate. Again the number of NOT gates should not be too large, because a clock input that is too wide might result.

FIGURE 6.20 **Pulse-Narrowing Circuit.**



Edge-triggered devices are of two types. Positive edge-triggered devices respond when the clock input makes the transition 0 → 1, and the negative edge-triggered devices respond when the clock input makes the transition 1 → 0. Figure 6.21 shows the logic symbols for both positive edge-triggered and negative edge-triggered *SR* and *JK* FFs where the arrowhead input corresponds to the *CK* input. Another way to achieve edge triggering is to use a special FF known as the *master-slave FF* that appears to trigger only on the clock edge.

FIGURE 6.21 **Logic Symbols for Edge-Triggered FFs.**



EXAMPLE 6.4

Obtain the timing diagram for the sequential circuit shown in Figure 6.22 for at least six clock cycles. Assume that $Q_1(0)Q_2(0) = 00$.

SOLUTION

FIGURE 6.22

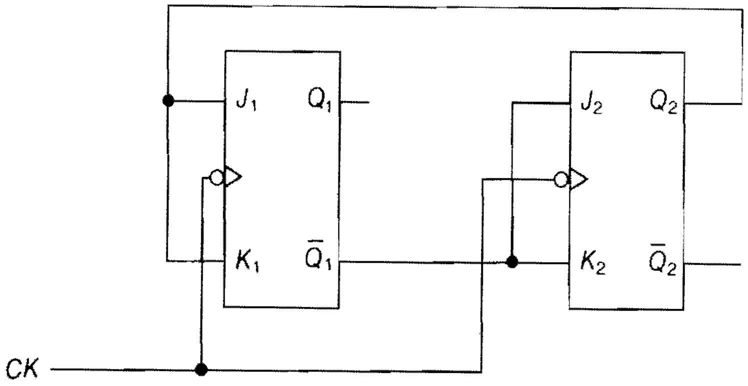
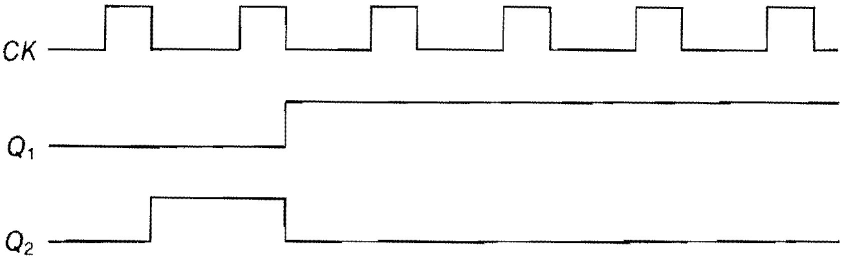


Figure 6.23 shows the timing diagram that is obtained readily by making use of the function table of a *JK* FF. It may be seen that within two cycles Q_1 is set and Q_2 is reset. The waveform will not change until Q_1 is reset externally.

FIGURE 6.23

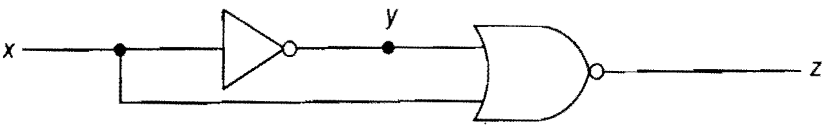


EXAMPLE 6.5

Obtain the response of the circuit of Figure 6.24, where each of the gates is assumed to have 1 unit of gate delay. The input x remains high for a duration longer than 4 units.

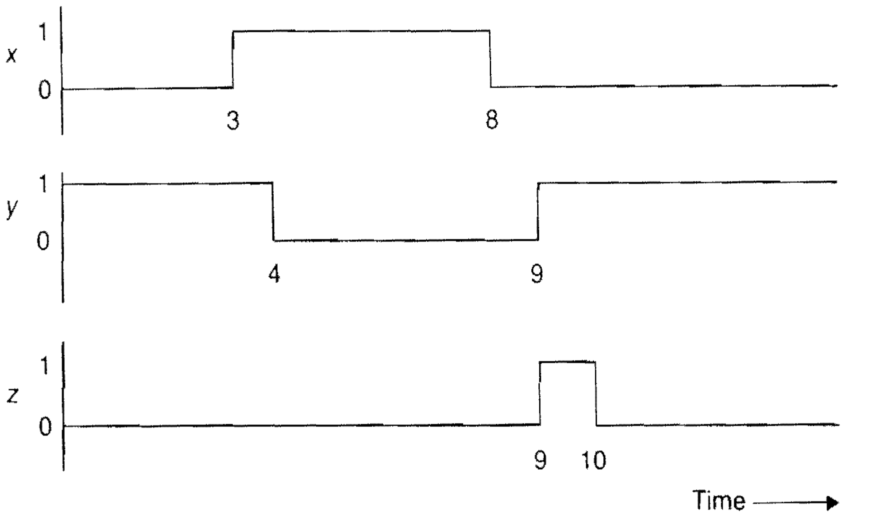
SOLUTION

FIGURE 6.24



The input, x, is assumed to be 5 units wide. The timing diagram is then obtained as shown in Figure 6.25. This circuit locates the trailing edge of the input pulse. Note that the circuit of Figure 6.20 functions likewise but locates the leading edge of an input.

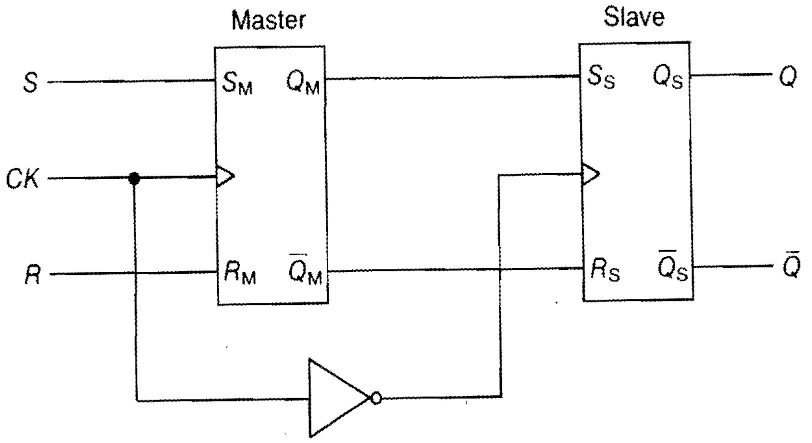
FIGURE 6.25



6.5 Master-Slave Flip-Flop

The master-slave concept is introduced into the FF circuitry to eliminate the requirement for limiting the clock width of a value determined by the circuit gate delays. This type of FF is composed of two sections: the master section and the slave section. This device is dependent not on the synchronous clocking of both units, but rather on their alternate turn-on and turn-off characteristics. The logic circuit of an *SR* master-slave FF is shown in Figure 6.26. It consists of a master FF, a slave FF, and an inverter for achieving out-of-phase clocking of the two units.

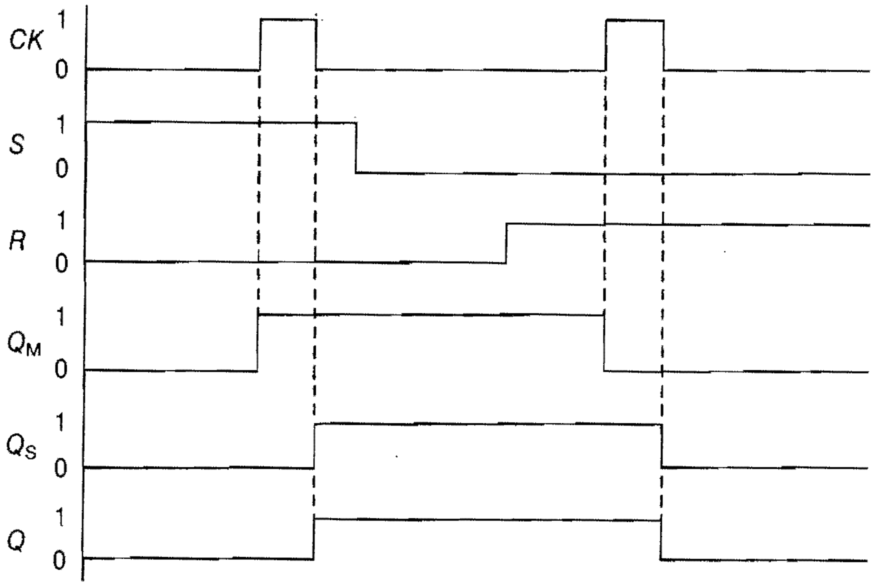
FIGURE 6.26 **Master-Slave *SR* FF.**



As a result of the presence of the inverter, the master unit is turned on and the slave unit is turned off when $CK = 1$. When $CK = 0$, the master unit is turned off and the slave unit is turned on. The circuit works as follows: for all inputs of S and R, except when $S = R$, $Q_M = S$ and $\overline{Q}_M = R$ when $CK = 1$. At this time the slave unit remains turned off. When the clock input goes to 0, $Q_S = Q_M$, $\overline{Q}_S = \overline{Q}_M$, and the master unit is turned off.

The timing diagram shown in Figure 6.27 illustrates the sequence of operations that takes place in a master-slave *SR* FF. The overall master-slave outputs appear to change at the negative edge of the clock input. However, there are many IC FFs that are the positive edge-triggered type. The master-slave cascading may be accomplished for any FF by similar introduction of an inverter between the two sections. As another example, Figure 6.28 shows the logic diagram of a master-slave *JK* FF. This is slightly different from that of a master-slave *SR* FF in that the outputs of NAND gates 7 and 8 are introduced as inputs to NAND gates 2 and 1, respectively. We have learned from Figure 6.19 that the Q and $\bar{Q}$ outputs might change several times during a wide clock pulse, leading to an unpredictable FF condition. Similar clock inputs would still cause the master outputs, Q_M and $\bar{Q}_M$, to change; but the slave outputs, Q_S and $\bar{Q}_S$, would not change because the inverted clock pulse disables the slave section. The values for J and K are still determined by the preclock values of Q_S and $\bar{Q}_S$. When the clock input to the master section goes low, the clock input to the slave section goes high, transferring Q_M and $\bar{Q}_M$ to the slave section. We have thus eliminated the problem of clock pulses that are too wide by the master-slave principle. A representative timing diagram for the master-slave *JK* FF is shown in Figure 6.29.

FIGURE 6.27 **Timing Diagram of a Master-Slave *SR* FF.**



A master-slave *JK* FF is also not without problems. The master section of the FF is vulnerable during the period when the clock is high and, therefore, may be set or reset by appropriate changes of the input. This results in "1s and 0s catching" problems. When $Q_S = 0$, $\bar{Q}_S = 1$, and the clock is high and while still high, the J input becomes high, Q_M is set, and consequently Q_S "catches" a 1 on the trailing edge of the clock input. Again when $Q_S = 1$, $\bar{Q}_S = 0$, and

FIGURE 6.28 Master-Slave *JK* FF.

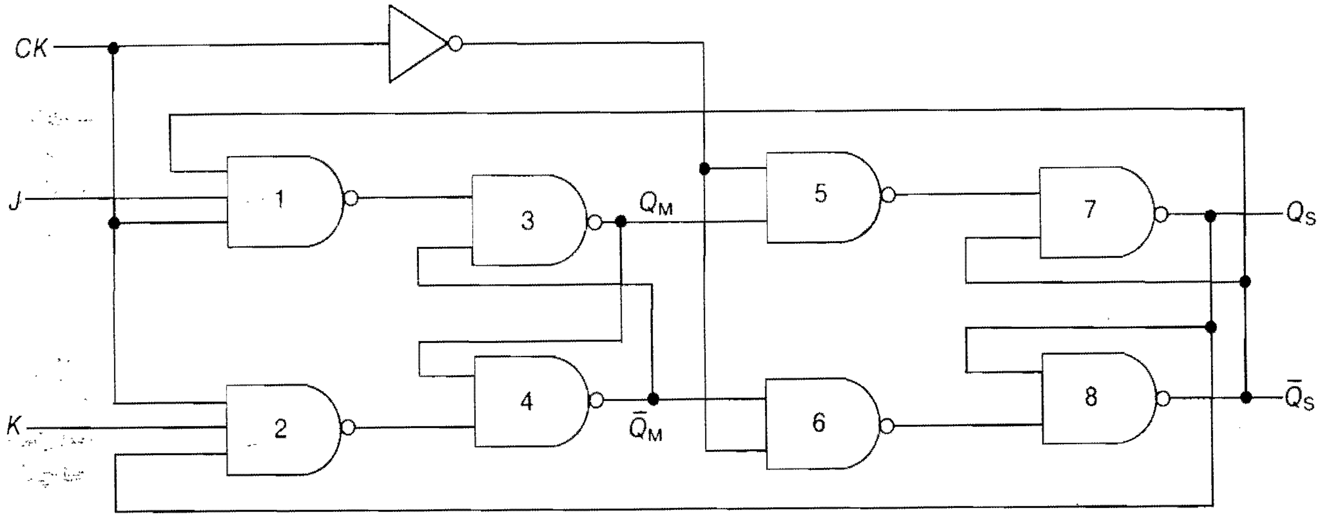
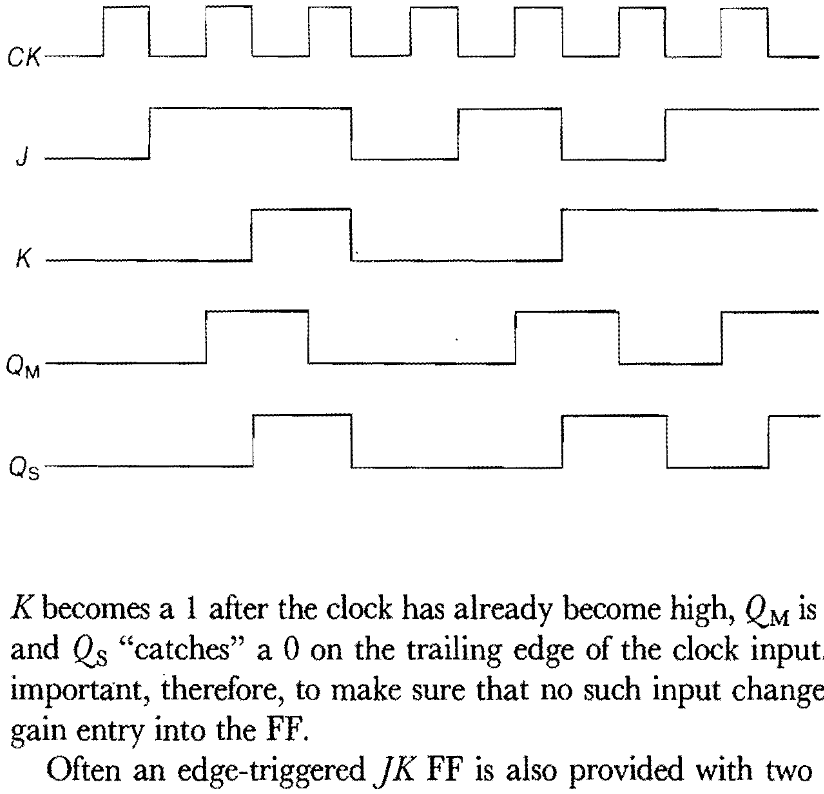


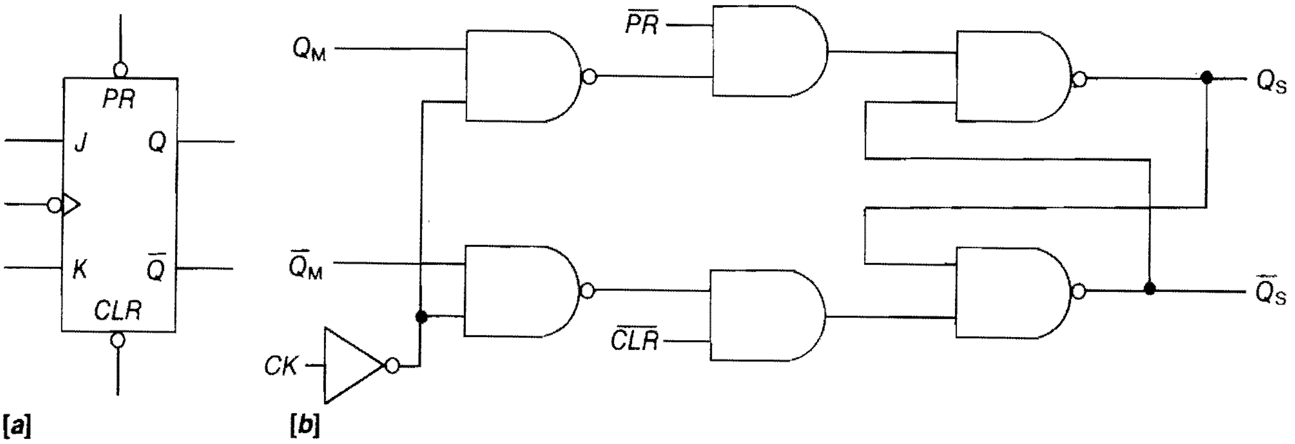
FIGURE 6.29 Master-Slave *JK* FF Timing Diagram.



K becomes a 1 after the clock has already become high, Q_M is reset, and Q_S "catches" a 0 on the trailing edge of the clock input. It is important, therefore, to make sure that no such input changes can gain entry into the FF.

Often an edge-triggered *JK* FF is also provided with two additional control inputs: preset and clear. The *preset* (*PR*) and *clear* (*CLR*) inputs allow initializing the FF to either a set ($Q = 1$) or a reset ($Q = 0$) condition. Addition of these two control inputs requires alteration of only the slave section of the FF. Figure 6.30 shows the logic diagram and the corresponding slave section of the FF that allows preset and clear inputs. Throughout this text both preset and clear inputs are considered to be active when low. Often these two FF control inputs are not labeled in the FF logic diagram. In such cases preset and clear inputs are always indicated by verti-

FIGURE 6.30 Complete Master-Slave *JK* FF: [*a*] Logic Diagram and [*b*] Slave Circuit.



cal inputs (with a bubble) respectively at the top and bottom of the corresponding FF logic diagram.

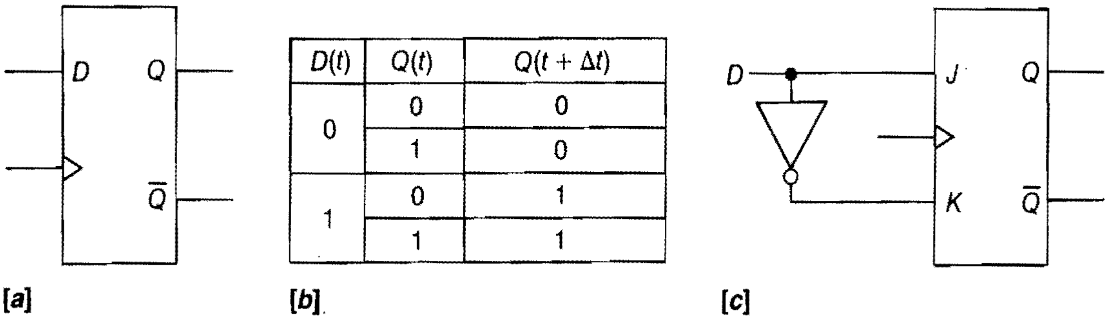
6.6 Delay and Trigger Flip-Flops

There are two other types of flip-flops that are commonly used: the *delay* (*D*) and the *trigger* (*T*) FFs. Unlike those in the previous sections, these two FFs have only one control input line besides the clock (excluding set and preset). Both of these FFs can be realized by externally manipulating the inputs of a *JK* FF.

Often it is necessary to have a sequential device that simply retains the input data value between clock pulses. The *D* FF performs this function. The FF output follows the FF input whenever a clock pulse is 1 and holds the value the input had when the clock changed to 0. The logic diagram and the characteristic table for a *D* FF are shown in Figures 6.31[*a*–*b*]. A comparison of this characteristic table with that for the *JK* FF (Figure 6.18[*c*]) reveals that a *D* FF is realizable from a *JK* FF by making $K = \bar{J}$ and using *J* as the *D* input, as illustrated in Figure 6.31[*c*]. The next-state equation of the *D* FF is given by

$$Q(t + \Delta t) = D(t) \qquad [6.10]$$

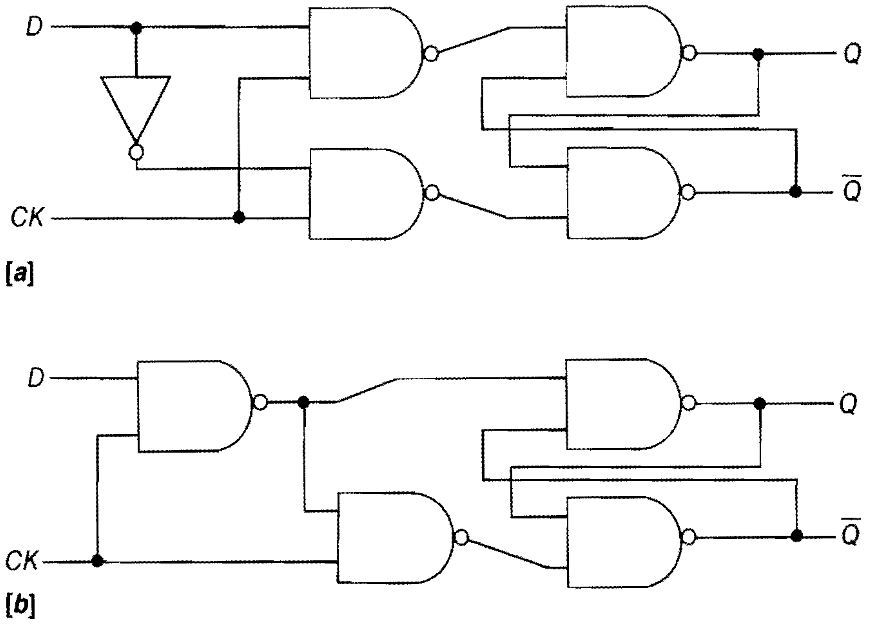
FIGURE 6.31 *D* FF: [*a*] Logic Diagram, [*b*] Characteristic Truth Table, and [*c*] Circuit Implementation.



D(t)	Q(t)	Q(t + Δt)
0	0	0
	1	0
1	0	1
	1	1

If severe restrictions placed on the clock input of an *SR* FF pose no problem, the *SR* FF can also be used to produce a *D* FF. The resulting circuit is shown in Figure 6.32[*a*]. Figure 6.32[*b*] shows a slight variation of the circuit of Figure 6.32[*a*] where advantage is taken of the special properties of NAND gates to eliminate one gate and still retain the characteristics of a *D* FF.

FIGURE 6.32 **D FF: [*a*] Using *SR* FF and an Inverter and [*b*] Using Modified NAND Latch.**



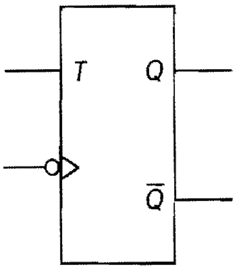
[*a*]

[*b*]

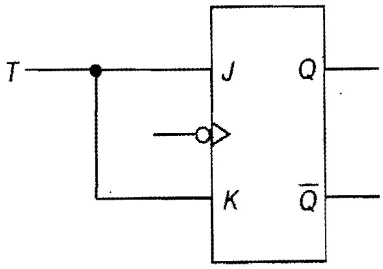
The *T* (trigger) FF, often called a toggle FF, has a single input that causes the output to change each time a pulse occurs at the input. The output remains unchanged as long as $T = 0$. The logic diagram and the characteristic table for a *T* FF are shown in Figures 6.33[*a–b*]. It should be noted that the *JK* FF has this mode available. The *JK* FF can be reorganized for realizing a *T* FF, as shown in Figure 6.33[*c*]. As long as both the *T* input and the *CK* input are high, the FF output will change. Its next-state equation, therefore, is obtained as follows:

$$Q(t + \Delta t) = Q(t) \oplus T(t) \qquad [6.11]$$

FIGURE 6.33 **T FF: [*a*] Logic Diagram, [*b*] Characteristic Table, and [*c*] Circuit Implementation.**



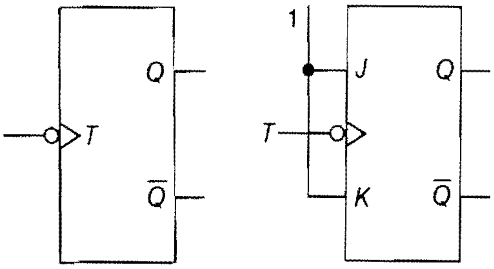
$T(t)$	$Q(t)$	$Q(t + \Delta t)$
0	0	0
	1	1
1	0	1
	1	0



A different version of the *T* FF involves a one-input device. Both *J* and *K* inputs of the *JK* FF are tied to a 1 to realize this unclocked

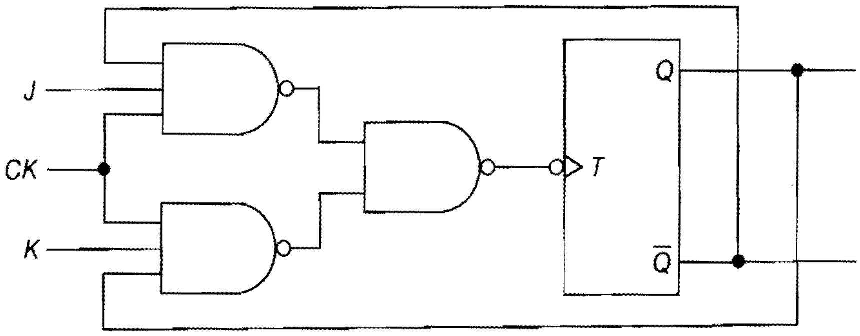
T FF. The input data are then introduced at the original clock input. The corresponding circuit for the unclocked *T* FF is shown in Figure 6.34.

FIGURE 6.34 **Unclocked *T* FF: [*a*] Block Diagram and [*b*] Logic Circuit.**



The unclocked *T* FFs are very important but are not made commercially. The logic usually is obtained using a *JK* FF as shown in Figure 6.34. One may even obtain this function from a *D* FF. In fact it is also easy to transform an unclocked *T* FF back to a *JK* FF. Such a conversion circuit is shown in Figure 6.35.

FIGURE 6.35 **Conversion of an Unclocked *T* FF to a Regular *JK* FF.**

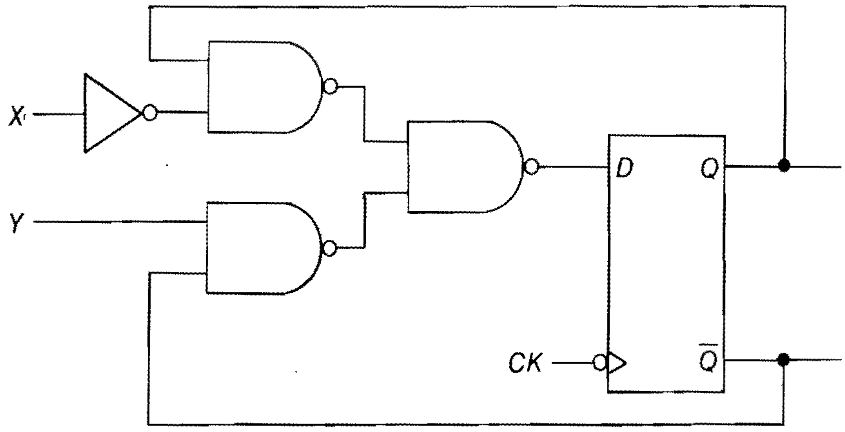


EXAMPLE 6.6

Analyze the circuit of Figure 6.36.

SOLUTION

FIGURE 6.36



The next-state equation of this circuit is as follows:

$$\begin{aligned} Q(t + \Delta t) &= D(t) \\ &= Y(t)\bar{Q}(t) + \bar{X}(t)Q(t) \end{aligned}$$

A close examination of this equation reveals that it is very similar to Equation [6.9] in that Y acts like the J input and X acts like the K input. In other words, this circuit functions exactly like a JK FF.

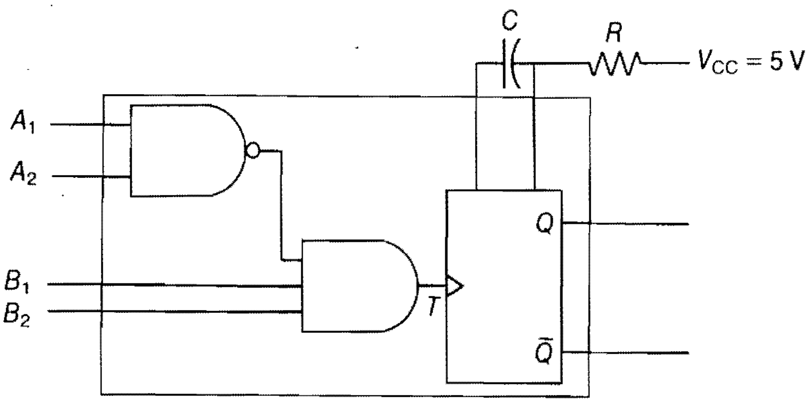
6.7 Monostable Flip-Flop

The *monostable* FF, also known as a *one-shot*, is an edge-triggered device used for producing output pulses of a duration independent of the input frequency. It produces an output pulse of specified width that is initiated by an input trigger signal. After a specified period of time the output returns to its quiescent state. The pulse duration is determined by the parameters of the resistor-capacitor network located external to the one-shot.

One-shots are of two types: nonretriggerable and retriggerable. In the nonretriggerable one-shot, if the device receives two successive trigger pulses of separation ζ less than the width δt of the output pulse generated by a single trigger, the second trigger input is ignored by the device. The retriggerable one-shot would be activated by the second trigger pulse, resulting in an output pulse of width approximately $\zeta + \delta t$. By applying a succession of trigger pulses separated by $\zeta < \delta t$, the output of a retriggerable one-shot could be maintained high (logic 1) as long as desired.

A one-shot may be designed using basic logic gates and a resistor-capacitor network. However, it is more convenient to use an IC one-shot because they are widely available and relatively inexpensive. Figure 6.37 shows the logic diagram and the function table of a standard retriggerable one-shot. The four inputs, A_1, A_2, B_1, and B_2, are available to provide flexibility of operation. The capacitor, C,

FIGURE 6.37 **Retriggerable One-Shot FF: [*a*] Logic Diagram and [*b*] Trigger Conditions.**



[*a*]

A_1	A_2	B_1	B_2	T
1 → 0	1	1	1	0 → 1
1	1 → 0	1	1	0 → 1
—	0	0 → 1	1	0 → 1
0	—	0 → 1	1	0 → 1
—	0	1	0 → 1	0 → 1
0	—	1	0 → 1	0 → 1

[*b*]

and the resistor, R, are external to the IC one-shot and are used to control the duration of the output pulse. Note that the trigger pulse, T, is given by $\overline{A_1 \cdot A_2} \cdot B_1 \cdot B_2$. The triggering conditions, as shown in Figure 6.37[*b*], cause T to change from a 0 to a 1.

The duration of the output pulse, δt, is determined by the resistor-capacitor network. Adjustable resistors and/or capacitors may be used to trim the output pulse to the desired width. In general, δt is given by

$$\delta t = f(R,C) \qquad [6.12]$$

where $f(R,C)$ is a function of the resistor and capacitor. The manufacturer provides the exact numerical relationship or curves, giving the output pulse width as a function of the timing resistors and capacitors. The minimum output pulse usually is realized using no external capacitor. Note, however, that there will be some stray capacitance existing between the terminals even in the absence of the external capacitor.

The retriggerable one-shot may be transformed into a nonretriggerable one-shot by feeding the Q output as one of the NAND gate inputs, say, A_2, while the other input A_1 is treated as the only triggering input. The remaining two inputs, B_1 and B_2, should be tied to a 1.

It is advisable to use monostable FFs only when no other solution can be found. Circuits with a number of monostables are very difficult to troubleshoot. Monostables can be falsely triggered by noise in the power supply voltage, causing serious circuit malfunctions.

6.8 Sequential Circuits

The general form of a sequential circuit is shown in Figure 6.38. The circuit in consideration has p inputs, q outputs, and r FFs used as memory. The combinational part of the circuit monitors the

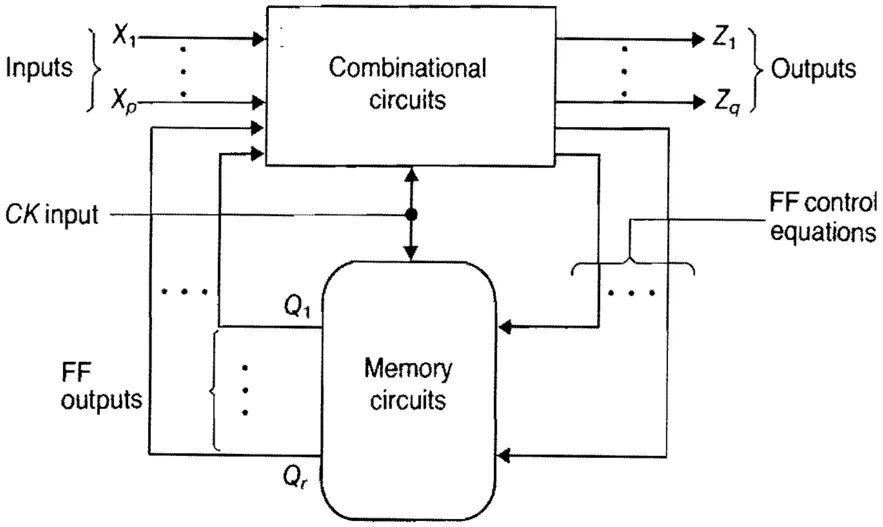


FIGURE 6.38 **Block Diagram of a General Sequential Circuit.**

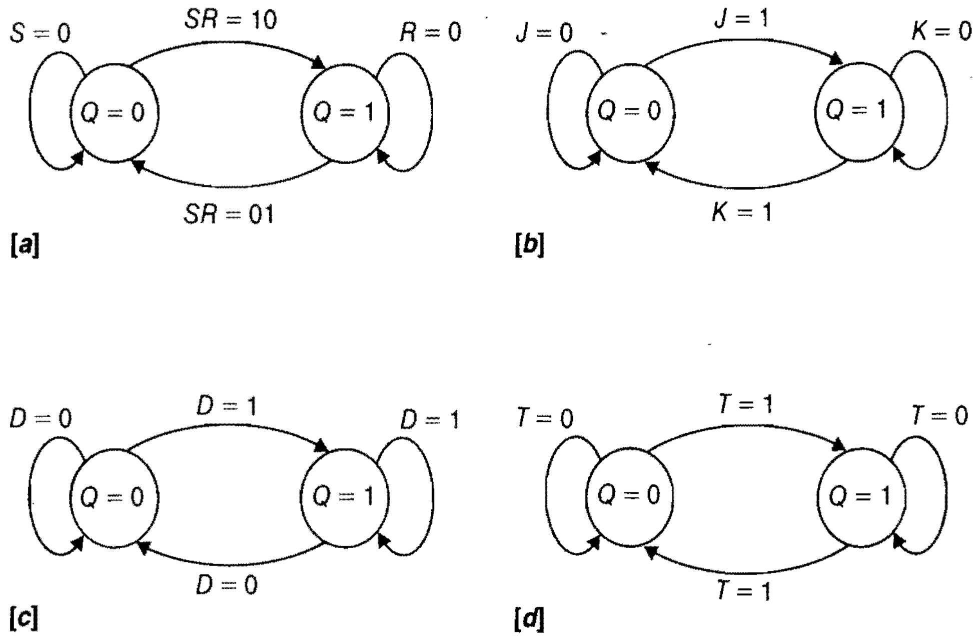
input values, X_j, checks the FF states, Q_k, and computes the FF control variables to assure that the next initiating action causes the correct changes to be made in the FF values. In addition the combinational part also computes the correct outputs, Z_l, for the circuit. Thus the current inputs and the previous-state information stored in the circuit's memory (FFs) are used to generate the current outputs and to determine the next state in the sequential circuits. The clock input is used only in clocked sequential circuits (the predominant type of sequential circuit).

The memory part of the circuit may be provided by using bistable devices such as FFs, relays, magnetic devices, switches, and so on. The most commonly used bistable device, however, is the FF. The control characteristics of various FFs are summarized in Figure 6.39, which provides the FF excitation inputs necessary to cause change in the FF output, Q. For example, the output of a *JK* FF can be changed from 1 to 0 by setting $K = 1$ while the J input could be tied either to a 1 or to a 0. The corresponding state transitions between $Q = 0$ and $Q = 1$ for each of the four FFs are shown in Figure 6.40 where the conditions for transitions are indicated next

FIGURE 6.39 **FF Control Characteristics.**

$Q(t)$	$Q(t + \Delta t)$	S	R	J	K	D	T
0	0	0	—	0	—	0	0
0	1	1	0	1	—	1	1
1	0	0	1	—	1	0	1
1	1	—	0	—	0	1	0

FIGURE 6.40 **Transition Diagrams: [*a*] *SR* FF, [*b*] *JK* FF, [*c*] *D* FF, and [*d*] *T* FF.**



to the transition lines. These characteristics will play an extremely important role in the design of complex sequential circuits.

One of the advantages of sequential systems over purely combinational systems is that circuit savings may be possible through the repetitive use of the same logic circuit. Some examples are multi-bit adder/subtracter circuits, multi-bit comparator circuits, and multi-bit code-converter circuits. It was shown in Chapter 5 that n different combinational circuit units are needed to accomplish an n-bit parallel operation. However, the price we will pay for using the same logic circuit repetitively is in the circuit operation speed.

Consider the four-bit ripple adder circuit shown in Figure 6.41. This addition operation is called parallel since two four-bit numbers are fed as inputs simultaneously to the adder circuit, and after the gate delays, the resultant bits become available simultaneously. However, addition can also be implemented serially. A *sequential circuit* that can perform such an operation is illustrated in Figure 6.42. The process can be started by introducing A_0 and B_0 to the FA. The resultant sum bit is stored in a multi-bit storage device, made up of several FFs and called a *register*, and the carry-out is stored in an FF and fed back into the FA. The process is repeated until all of the bits have been considered. The final sum would consist of the last carry-out and the sum bits stored in the register (see Problem 1 at the end of the chapter). The characteristics of the registers and how the input bits are fed sequentially to the adder will be considered in a later chapter. For the time being it will suffice to say that the register is a storage area where the bits can be moved slowly to the right as the storage process continues.

However, in spite of all their merits sequential systems are not devoid of deficiencies. Some of these disadvantages are very critical to the functioning of the system; therefore, they must be considered very carefully. Consider a very simple circuit such as the one shown in Figure 6.43. When the input I is a 0, the output O becomes a 1. This output is then fed back again to the NAND gate. As a consequence, the values of the output disagree with that of the input,

FIGURE 6.41 **Four-Bit Parallel Adder Circuit.**

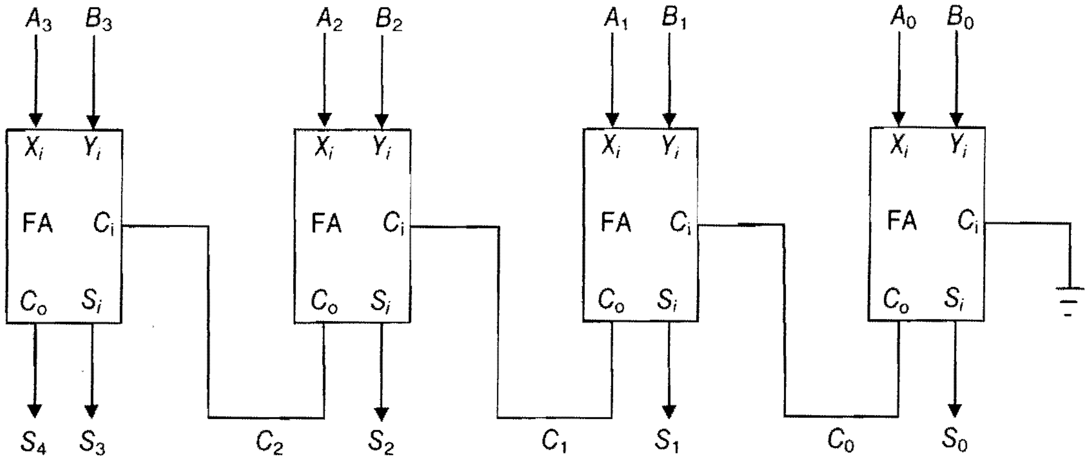


FIGURE 6.42 **Primitive Block Diagram for the Parallel Adder.**

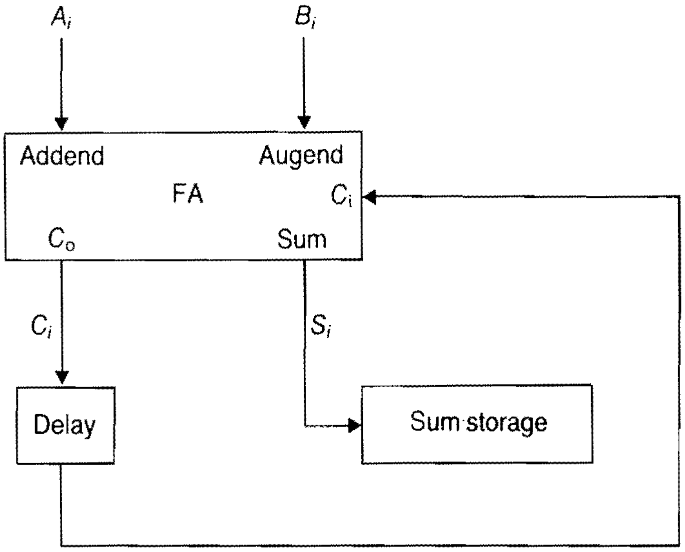
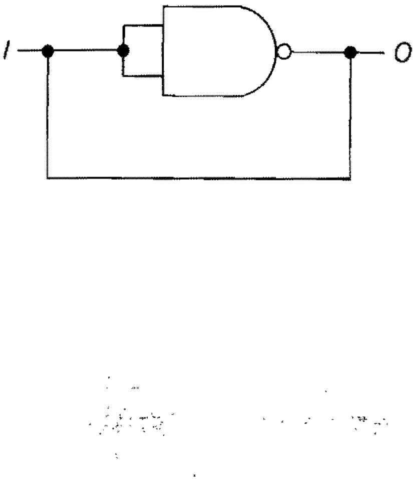


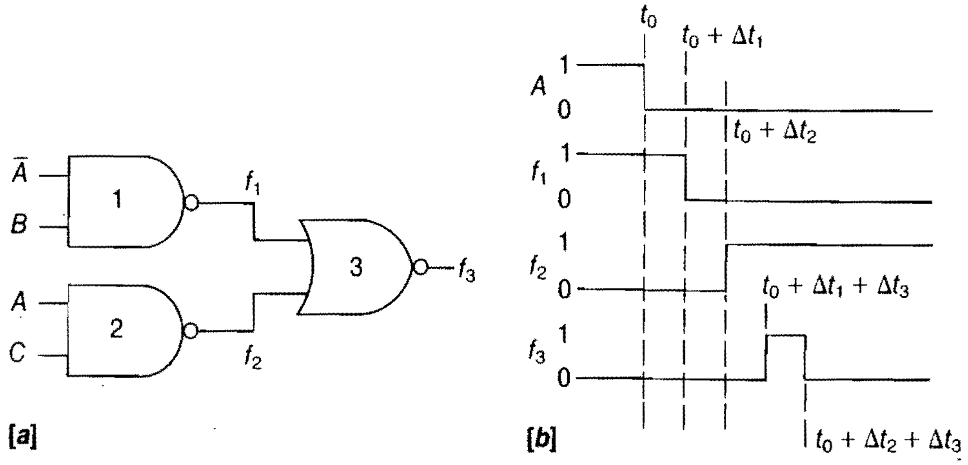
FIGURE 6.43 **Indeterminate Feedback Circuit.**



resulting in an indeterminate feedback system. Such problems, however, may be avoided by eliminating the conditions of continuing oscillations.

Next consider the combinational circuit of Figure 6.44[*a*] where the gates numbered 1, 2, and 3, respectively, have gate delays of Δt_1, Δt_2, and Δt_3. Assume that at time t_0 the inputs are $A = B = C = 1$, and at time $t > t_0$ the input A changes from 1 to 0. Assume further that $\Delta t_1 < \Delta t_2$. The resulting timing diagram is illustrated in Figure 6.44[*b*]. It is apparent that the combinational circuit output results in a transient error pulse for a duration of $\Delta t_2 - \Delta t_1$. This error pulse is small but not negligible. If the output of this circuit is introduced as the clock, preset, or clear inputs to an FF, we can expect to see additional problems in the sequential circuit. However, errors will not occur until the pulse width exceeds the time required to trigger the FF. These and other problems are tackled by taking proper precautions either during the design or during the operation of a sequential system.

FIGURE 6.44 **[*a*] Combinational Circuit and [*b*] Its Corresponding Timing Diagram When $B = C = 1$.**



There are three different types of sequential circuits. They are classified according to the characteristics of the inputs and memory types.

Synchronous sequential circuits: Synchronous sequential circuits involve FF action that occurs in synchronization with the clock input. The values of the external variables that control the FF states may change while the clock is not present. All transients due to the previous clock must have disappeared prior to the next clock for correct circuit action.

Pulse-mode circuits: The input variables of pulse-mode circuits can have only mutually exclusive pulses. In addition, the FFs that are used are not clocked since no clock is present in this mode.

Fundamental-mode circuits: Fundamental-mode circuits involve level inputs and asynchronous memory devices. The FFs change state whenever an input variable logic level changes.

While the first type listed, also known as a *clocked* sequential circuit, is synchronous, the other two are asynchronous in character. However, synchronous sequential circuits account for the overwhelming majority of sequential circuits.

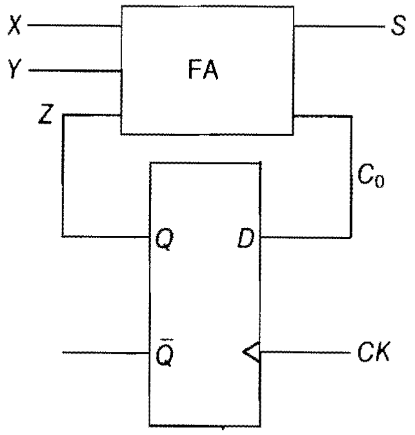
6.9 Summary

In this chapter the concept of a sequential circuit was introduced. The design and working principles of latches, various FFs, and the monostable multivibrator were discussed. Particular emphasis was placed on the various practical limitations that these devices have. Finally, the possibility of having different classes of sequential circuits was explored. The design and the characteristics of these sequential systems will be presented respectively in the next three chapters.

Problems

1. The FA receives two external inputs X and Y; the third input Z comes from the output of a D FF as shown in Figure 6.P1. The carry-out is transferred to the FF at every clock pulse.

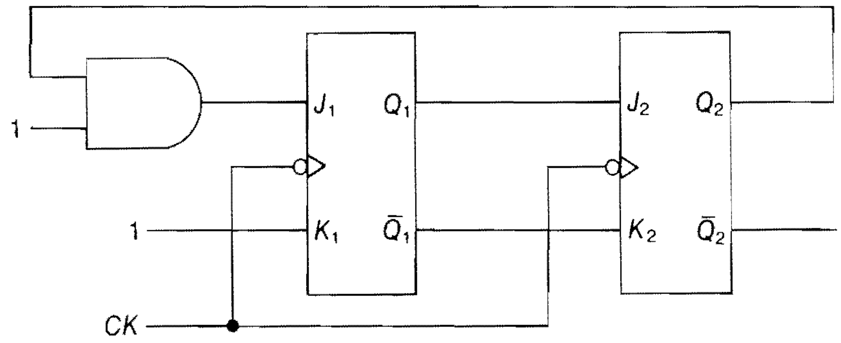
FIGURE 6.P1



The S output represents the sum. Obtain the state equations for S and C_o.

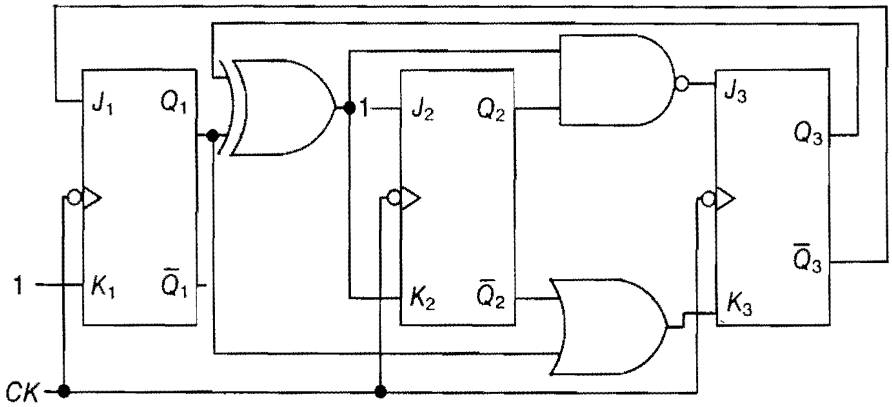
2. Draw the timing diagram for the given input signal and circuit of Figure 6.P2. Assume the starting value of $Q_2Q_1 = 00$.

FIGURE 6.P2



3. What sequence should repeat for the sequential circuit of Figure 6.P3 for the following initial inputs:
a. $Q_3Q_2Q_1 = 001$ b. $Q_3Q_2Q_1 = 100$

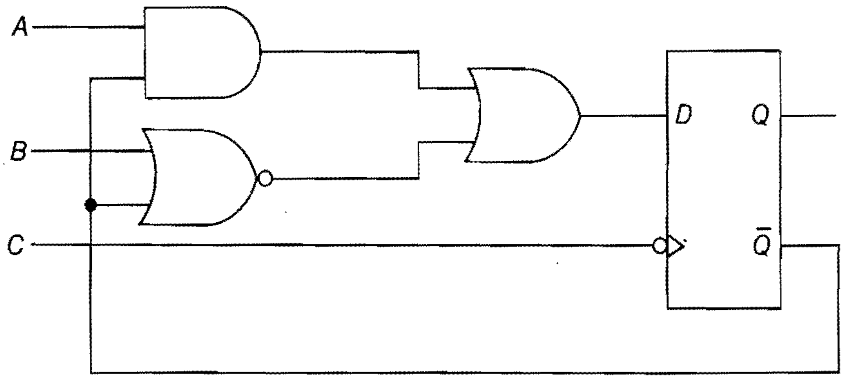
FIGURE 6.P3



4. Obtain a T FF from a D FF.

5. Explain the behavior of the circuit of Figure 6.P4.

FIGURE 6.P4



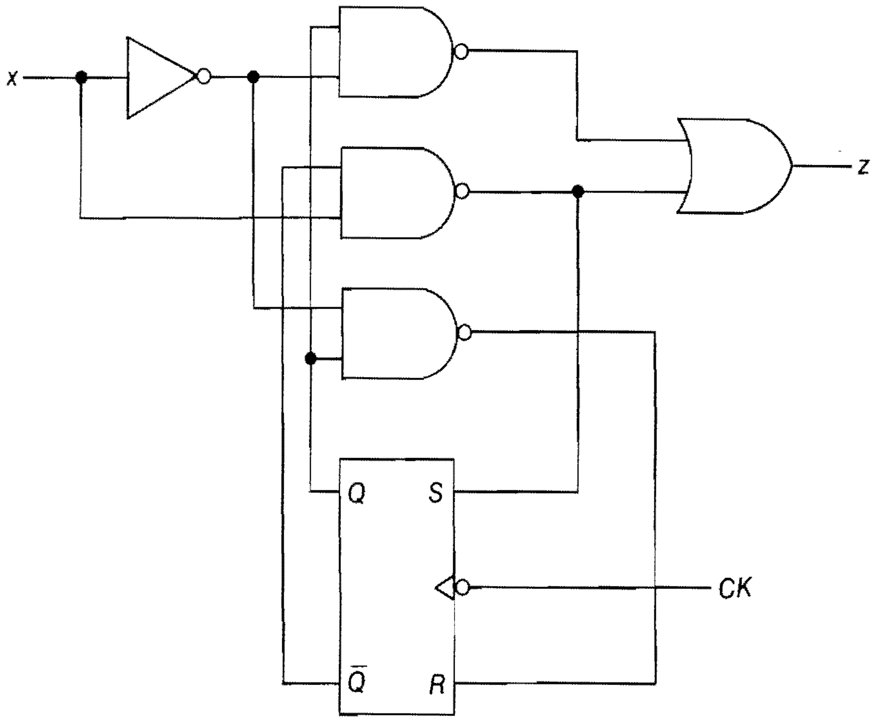
6. A sequential circuit has two inputs, X and Y, and one output, Z, such that

$$J_1 = XQ_2 + \overline{Y}\overline{Q}_2$$
$$J_2 = X\overline{Q}_1$$
$$K_1 = X\overline{Y}\overline{Q}_2$$
$$K_2 = X\overline{Y} + Q_1$$
$$Z = XYQ_1 + \overline{X}\overline{Y}Q_2$$

Obtain the logic diagram and state equations.

7. Find the output and state sequences for the circuit of Figure 6.P5 if the initial state is $Q = 0$ and the input sequence is
a. $x = 101101100$ b. $x = 111011101$

FIGURE 6.P5



8. Show the detailed working of the circuit of Figure 6.35.
9. Repeat Example 6.2 when the NAND gate has a delay of 4 units and the NOT gate has a delay of 3 units.
10. Repeat Example 6.2 when the gate delays are lumped together.
11. Repeat Example 6.2 when the NAND gate has a delay of 4 units and the NOT gate has a delay of 3 units, but assume a lumped model for the circuit.
12. Comment on the behaviors of the two circuits of Figure 6.P6.
13. Analyze the circuit of Figure 6.P7.
14. Obtain a sequential system for performing a multi-bit, BCD-to-binary conversion. Describe the general working principles of your circuit.

FIGURE 6.P6

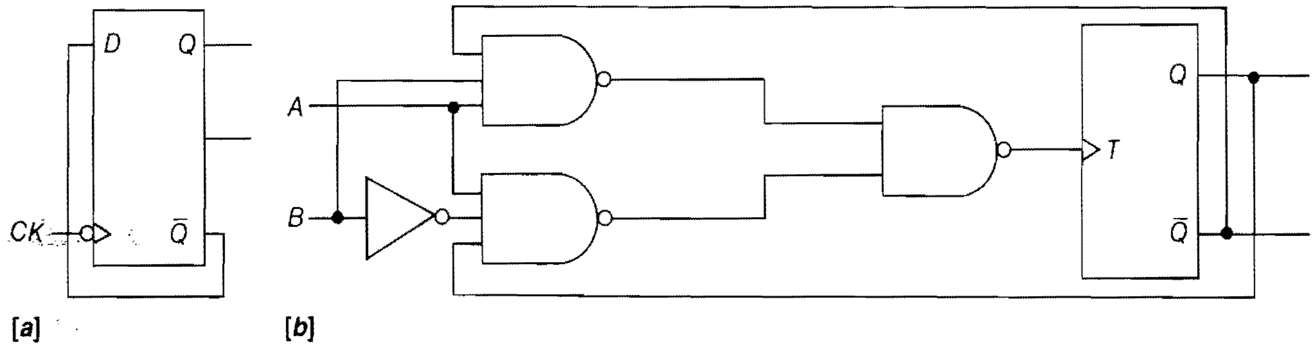
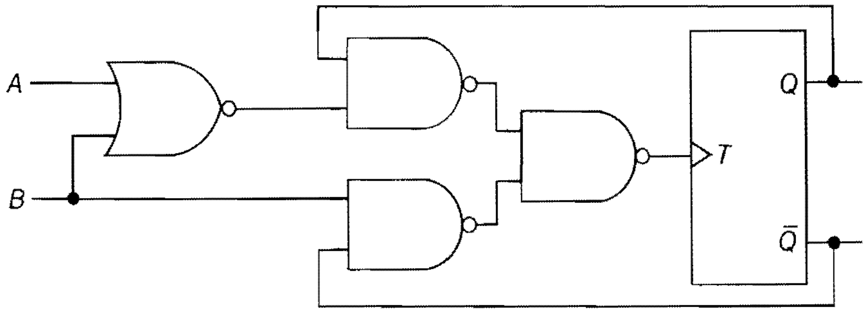


FIGURE 6.P7



15. Obtain a sequential system for performing a multi-bit, binary-to-BCD conversion. Describe the general working principles of your circuit.
16. Obtain a sequential system for performing a multi-bit, Gray-to-binary conversion. Describe the general working principles of your circuit.
17. Obtain a sequential system for performing a multi-bit, binary-to-Gray conversion. Describe the general working principles of your circuit.
18. Obtain a sequential system for performing a multi-bit comparison between two numbers. Describe the general working principles of your circuit.

Suggested Readings

Almaini, A. E. A. "Sequential machine implementations using universal logic modules." *IEEE Trans. Comp.* vol. C-27 (1978): 951.

Beraru, J. "A one-step process for obtaining flip-flop input logic equations." *Comp. Des.* vol. 6 (1967): 58.

Chen X., and Hurst, S. L. "A comparison of universal-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks." *IEEE Trans. Comp.* vol. C-31 (1982): 140.

Fleischhammer, W., and Dortok, O. "The anomalous behavior of flip-flops in synchronizer circuits." *IEEE Trans. Comp.* vol. C-28 (1979): 273.

Fletcher, W. I. *An Engineering Approach to Digital Design.* Englewood Cliffs, N.J.: Prentice-Hall, 1980.

Floyd, T. L. *Digital Fundamentals.* 2d ed. Columbus, Ohio: Charles E. Merrill, 1982.

Gaitanis, N., and Halatsis, C. "Negative failsafe sequential circuits." *Elect. Lett.* vol. 16 (1980): 615.

Huffman, D. A. "The synthesis of sequential switching circuits." *J. Frank. Inst.* vol. 257 (1954): 275.

Joseph, J. "On easily diagnosable sequential machines." *IEEE Trans. Comp.* vol. C-27 (1978): 159.

Kline, R. *Structured Digital Design Including MSI/LSI Components and Microprocessors.* Englewood Cliffs, N.J.: Prentice-Hall, 1983.

Manning, F. B., and Fenichel, R. R. "Synchronous counters constructed entirely of *J-K* flip-flops." *IEEE Trans. Comp.* vol. C-25 (1976): 300.

Mealy, G. H. "A method for synthesizing sequential circuits." *Bell Syst. Tech. J.* vol. 34 (1955): 1045.

Nanda, N. K., and Bennetts, R. G. "Reconvergence phenomenon in synchronous sequential circuits." *Elect. Lett.* vol. 16 (1980): 303.

Noe, P. S., and Rhyne, V. T. "Optimum state assignment for *D* flip-flop." *IEEE Trans. Comp.* vol. C-25 (1976): 306.

Taub, D. M. "Hardware method of synchronizing processes without using a clock." *Elect. Lett.* vol. 19 (1983): 772.

Unger, S. H. "Self-synchronizing circuits and nonfundamental mode operation." *IEEE Trans. Comp.* vol. C-26 (1977): 278.

Voith, R. P. "Minimum universal logic module sequential circiuts with decoders." *IEEE Trans. Comp.* vol. C-26 (1977): 1032.

Wilkens, E. J. "Realization of sequential machines using random access memory." *IEEE Trans. Comp.* vol. C-27 (1978): 429.

Witte, H.-H., and Moustakas, S. "Simple clock extraction circuit using a self-sustaining monostable multivibrator output signal." *Elect. Lett.* vol. 19 (1983): 897.

FIGURE 7.1 **State Diagram.**

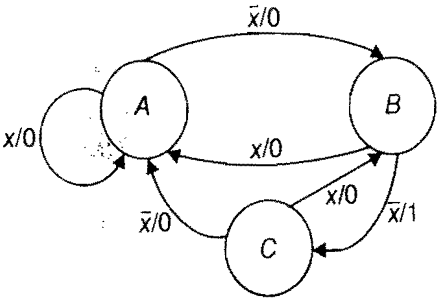


FIGURE 7.2 **State Table Corresponding to the Diagram of Figure 7.1.**

Present State (PS)	Next State (NS) Output (Z)	
	x = 0	x = 1
A	B,0	A,0
B	C,1	A,0
C	A,0	B,0

already encountered some examples of two-state state diagrams in Figure 6.40.

The state diagram in Figure 7.1 represents a synchronous circuit with three states, A, B, and C, and an input variable, x. In each state it is necessary for the circuit to be able to determine which state it is in and what the current value of x is, and then to set up the FF inputs such that the correct state is entered when the clock input occurs. The arrows connecting the states represent the occurrence of a clock input, and the variables a! ngside the arrows show the input condition that causes that pain to be followed.

If the circuit is currently in state A and $x = 1$, the circuit will remain in state A when the clock occurs ($x/0$). If $x = 0$ the circuit enters into state B when the clock occurs ($\bar{x}/0$). In both of these cases the circuit yields an output of 0. The output value and the input condition are both indicated next to the corresponding transition paths. If the circuit is in state B and $x = 1$ when the clock occurs, the circuit returns to state A ($x/0$). However, if $x = 0$ when the clock occurs, the state C is entered, resulting in an output of 1 ($\bar{x}/1$). And finally the circuit moves coincident with the clock from state C to states A and B, respectively, when $x = 0$ and $x = 1$. In either case the output remains 0. The state diagram must show each of the states of the circuit and all conditions necessary for entering or exiting the states. In this case two transition lines leave each of the states.

The implementation of a sequential circuit with n states will require m FFs where $2^m \geq n$. The outputs of these FFs are called the *state variables* and are used to identify which state the circuit is in. An additional design tool that contains the same information as the state diagram in tabular form is the *state table*. The state table of the system shown in the state diagram of Figure 7.1 is given in Figure 7.2. It can be seen that the outputs are associated with the transition paths only and are not functions of any transition states. Circuits such as this are generally known as Mealy-type machines. The *Mealy outputs*, in most cases, are pulses coincident with the input pulse causing the state transition. An alternate output type, called the *Moore output*, is associated with the present state only. The general forms of the Mealy and Moore circuits are shown in Figure 7.3. These circuits take their names, respectively, from G. H. Mealy and E. F. Moore, two of the most famous pioneers in sequential design. The outputs from Moore-type circuits are independent of the inputs. The Moore outputs change their values only when the states change because of a change of the inputs.

There are many systems that possess both Mealy and Moore outputs; in other words, some outputs are conditional on both the inputs and the state of the circuit, while others are dependent only on the state of the circuit. Note, however, that the Mealy output is

Design of Synchronous Sequential Circuits

CHAPTER SEVEN

7.1 Introduction

In this chapter we will examine clocked sequential circuits. These circuits will employ combinational circuits and flip-flops. All circuit action will take place under the control of a periodic sequence of pulses called a clock. Each clock pulse will permit the circuit to either remain in its present state (present set of FF values) or move to another state (a new set of FF values). The advantage of clocked sequential circuits is that glitches that occur due to the imperfect nature of the logic devices will have no effect. This is possible only if we choose the clock period such that all glitches due to multiple delay paths end before the FFs encounter future changes.

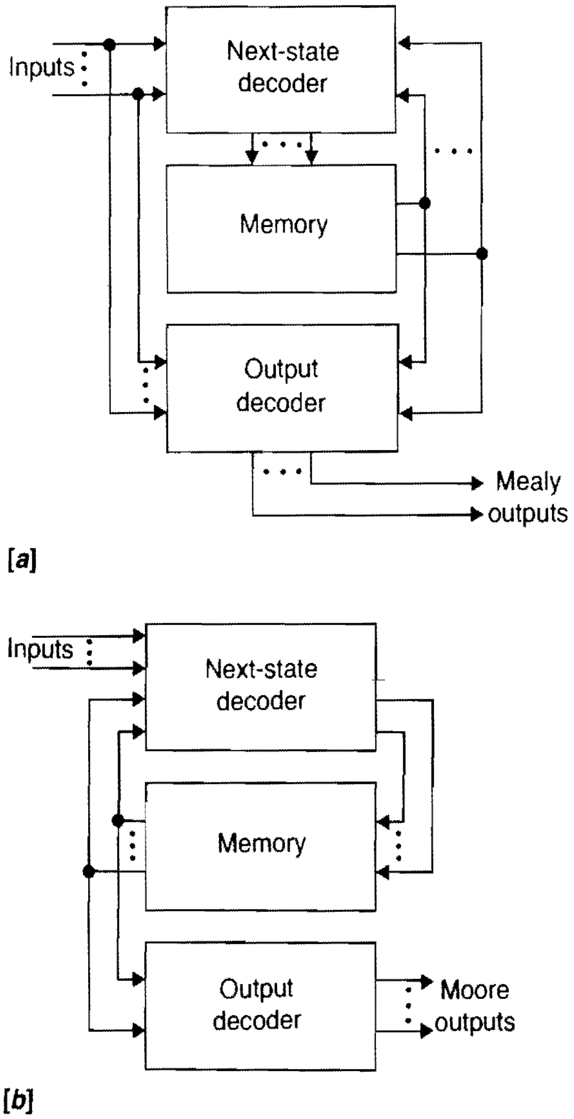
The synthesis of sequential circuits consists of obtaining a table or diagram for the time sequence of inputs, outputs, and internal states. Boolean expressions are then derived by incorporating the behavior patterns of FF memory elements. In the following sections we will introduce these design sequences along with several synchronous sequential circuit examples. After studying this chapter, you should be able to:

- ○ Obtain a state diagram for a synchronous sequential machine;
- ○ Eliminate redundant states;
- ○ Realize a sequential circuit from the state table;
- ○ Differentiate between Mealy and Moore circuits.

7.2 State Diagrams and State Tables

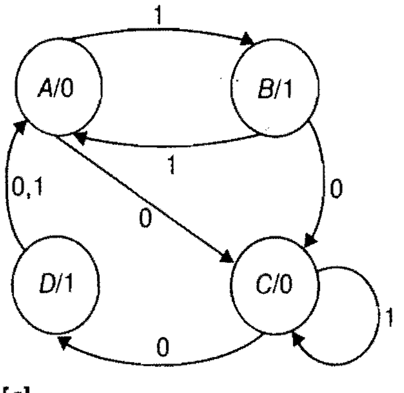
The functional interrelationship that exists among the input, the output, the present state, and the next state is best illustrated by the state diagram or the state table. The *state diagram* is a graphical representation of a sequential circuit in which the states are represented by circles and transitions between states shown by arrows. We have

FIGURE 7.3 General Model of Sequential Machines: [*a*] Mealy and [*b*] Moore.



easily convertible to equivalent Moore outputs and vice versa. (More about this conversion will be said in one of the worked-out examples.) Figure 7.4[*a*] shows the format for a state diagram where the Moore-type outputs are circled along with the corresponding present states. Figure 7.4[*b*] shows the state table corresponding to

FIGURE 7.4 Moore Model for a Sequential Circuit.



[*a*]

PS	NS		*Z*
	x = 0	*x* = 1	
A	*C*	*B*	0
B	*C*	*A*	1
C	*D*	*C*	0
D	*A*	*A*	1

[*b*]

the state diagram of part [*a*]. It is important to point out that both Mealy- and Moore-type circuits are equally applicable to both synchronous and asynchronous circuits; and the minimum number of external inputs to any one of these circuits is one. For a synchronous circuit, that one input must be the system clock.

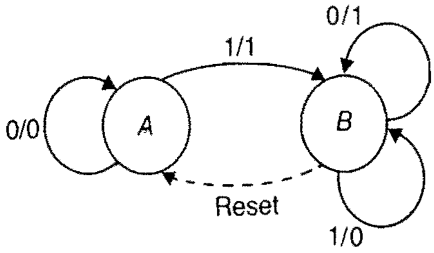
EXAMPLE 7.1

Obtain the state diagram of a controller for a serial machine that performs the 2's complement operation (see Example 5.3 for the equivalent parallel scheme).

SOLUTION

The realization of the state diagram for the controller is very straightforward, as shown in Figure 7.5. This follows from Rule 2(b) of Section 1.4. The 2's complement of a number is obtained by complementing all bits to the left of the least significant 1 in that number. State *A* takes care of the situation when none of the serial inputs are changed, whereas state *B* corresponds to the changing (1's complement) of inputs. The controller remains in state *A* as long as the low-order 0s of the input are encountered. The first input of 1 moves the machine to state *B* so that all subsequent inputs are complemented. To begin a new conversion, the machine needs to be reset back to state *A* (indicated by the broken line).

FIGURE 7.5



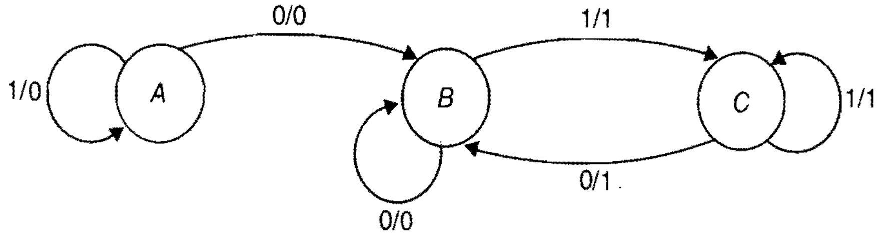
EXAMPLE 7.2

Obtain the state table for a synchronous sequential machine that detects a 01 sequence. The detection of sequence sets the output, $Z = 1$, which is reset only by a 00 input sequence.

SOLUTION

The state diagram for this machine is obtained as shown in Figure 7.6. The machine resides in state *A* as long as the sequence does not begin. This situation would include two distinct cases: either (a) the machine is yet to see a single bit, or (b) the machine has so far examined a 1 or a string of 1s. However, once the first bit, 0, of either sequence, 01 or 00, has been detected, the machine moves to state *B*. Finally, state *C* is reached if either (a) the complete sequence, 01, has been located, or (b) the resetting

FIGURE 7.6



sequence, 00, is yet to begin. The state table of the machine readily follows from the state diagram. It is shown in Figure 7.7.

FIGURE 7.7

PS	NS, *Z*	
	x = 0	*x* = 1
A	*B*,0	*A*,0
B	*B*,0	*C*,1
C	*B*,1	*C*,1

EXAMPLE 7.3

Obtain the Moore equivalent state table for the Mealy machine of Figure 7.8.

SOLUTION

FIGURE 7.8

PS	NS, *Z*	
	x = 0	*x* = 1
A	*C*,0	*A*,0
B	*B*,0	*A*,0
C	*D*,1	*C*,1
D	*D*,0	*B*,0
E	*C*,1	*A*,0

The state table of Figure 7.8 is a Mealy type since the outputs are not associated only with the states. It is the desired goal of this problem to associate each of these outputs with a particular state. It may be seen that the two next states, *C* and *D*, are associated with two different outputs, 0 and 1. The next states, *A* and *B*, are associated always with an output of 0 only. Accordingly, four new states, *C*′, *C*″, *D*′, and *D*″, may be introduced to replace the states *C* and *D*. The states *C*′ and *D*′ correspond, respectively, to states *C* and *D* when the output is a 0. Similarly, the states *C*″ and *D*″ correspond, respectively, to states *C* and *D* when the output is a 1. The state table obtained after the introduction of only *C*′ and *C*″ is shown in Figure 7.9. Next, *D*′ and *D*″ may be included to obtain the state table as shown in Figure 7.10.

FIGURE 7.9

PS	NS, *Z*	
	x = 0	*x* = 1
A	*C*′,0	*A*,0
B	*B*,0	*A*,0
C′	*D*,1	*C*″,1
C″	*D*,1	*C*″,1
D	*D*,0	*B*,0
E	*C*″,1	*A*,0

FIGURE 7.10

PS	NS, *Z*	
	x = 0	*x* = 1
A	*C*′,0	*A*,0
B	*B*,0	*A*,0
C′	*D*″,1	*C*″,1
C″	*D*″,1	*C*″,1
D′	*D*′,0	*B*,0
D″	*D*′,0	*B*,0
E	*C*″,1	*A*,0

At this time each of the states has only one output associated with itself. Accordingly one could obtain the equivalent Moore machine as shown in Figure 7.11. This table has been constructed in a way so that it resembles the format of Figure 7.4[*b*].

FIGURE 7.11

PS	NS		Z
	$x = 0$	$x = 1$	
A	C'	A	0
B	B	A	0
C'	D''	C''	0
C''	D''	C''	1
D'	D'	B	0
D''	D'	B	1
E	C''	A	—

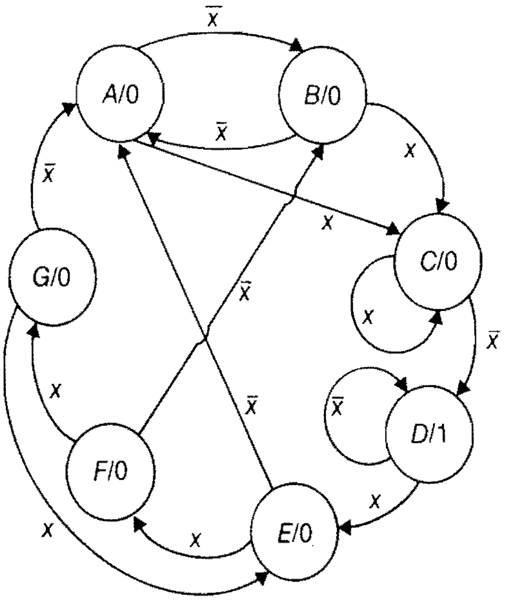
Note in Example 7.3 that the Moore machine of Figure 7.11 is equivalent to the original Mealy machine of Figure 7.8 only in the sense that its output appears as pulses. The outputs that occur in C'' and D'' are pulses that are high (1) for a full clock period. In the Mealy circuit the pulses are high (1) while the clock is high. Moreover, two additional states were necessary to make this conversion complete. In fact, the Moore equivalent machine generally consists of more states than the corresponding Mealy machine.

7.3 Equivalent States

When constructing a state diagram from the word statement of a design problem, a state that is identical to another state may inadvertently be included. The redundant state or states will increase the number of total states and may require the addition of another FF, making the circuit more expensive. Redundant states also decrease the number of don't-cares or unused states and thus increase the overall complexity of the circuit equations. In addition, fault diagnostic techniques used for failure analysis are based on the assumption that no redundant states exist. One of the design goals, therefore, is to eliminate all redundant states from the state diagram and/or table.

Sometimes redundant states are obvious. In Figure 7.12 a state diagram and a state table are given that include several redundant states. Two states are defined as *equivalent* if they have identical outputs and make transitions to the same states for a given control variable value. In Figure 7.12 no state can be equivalent to state D because it is the only state with an output of 1. We can make the statement that state E is equivalent to state F only if state A is equivalent to B and state F is equivalent to state G. Again, states F and G are equivalent only if state G is equivalent to state E and also

FIGURE 7.12 Sequential Circuit with Redundant States: [*a*] State Diagram and [*b*] State Table.



[*a*]

PS	NS		Z
	x = 0	x = 1	
A	B	C	0
B	A	C	0
C	D	C	0
D	D	E	1
E	A	F	0
F	B	G	0
G	A	E	0

[*b*]

if state *A* is equivalent to *B*. This line of reasoning becomes extremely confusing for large systems.

A systematic way of looking for redundant states usually is accomplished by means of a new tool called an *implication table*. The numbers of rows and columns in this table are both equal to $n - 1$ where n is the number of states in the to-be-reduced state table. An implication table that is used to locate the possible redundant states of Figure 7.12[*b*] is illustrated in Figure 7.13[*a*]. This table provides a bookkeeping technique that allows a systematic way to find states that are equivalent. The algorithm employed for the equivalency search in the implication table is as follows:

1. All states except the very first one are used as row labels of the table, and all states except the very last one are used as column labels.
2. The entries at each of the cells are made by comparing the next-state columns of the two states that are used to identify the cell in question. No entries are made in a cell if it corresponds to states that have the same output and the same next states for all control variables. If the equivalency of the two states is dependent on whether or not the states *P* and *Q* are equivalent, then the pair (*PQ*) is entered in the corresponding cell. A cross "X" is placed over those cells for which outputs of the two states are different for any one input condition.

FIGURE 7.13 State Reduction by Implication: [*a–b*] Implication Tables and [*c*] Equivalence Partition Table.

	A	B	C	D	E	F
B	AB					
C	~~BD~~	~~AD~~				
D	~~BD~~ ~~CE~~	~~AD~~ ~~CE~~	~~DE~~			
E	AB CF	CF	~~AD~~ ~~CF~~	~~AD~~ ~~EF~~		
F	CG	AB CG	~~BD~~ ~~CG~~	~~BD~~ ~~EG~~	AB FG	
G	AB CE	CE	~~AD~~ ~~CE~~	~~AD~~	EF	AB EG

?

[*a*]

	A	B	C	D	E	F
B	AB					
C	~~BD~~	~~AD~~				
D	~~BD~~ ~~CE~~	~~AD~~ ~~CE~~	~~DE~~			
E	~~AB~~ ~~CF~~	~~CF~~	~~AD~~ ~~CF~~	~~AD~~ ~~EF~~		
F	~~CG~~	~~AB~~ ~~CG~~	~~BD~~ ~~CG~~	~~BD~~ ~~EG~~	AB FG	
G	~~AB~~ ~~CE~~	~~CE~~	~~AD~~ ~~CE~~	~~AD~~	EF	AB EG

[*b*]

F	(FG)
E	(EFG)
D	(EFG)
C	(EFG)
B	(EFG)
A	(AB)(EFG)
	(AB)(C)(D)(EFG)

[*c*]

3. The next step involves elimination of as many cells as possible based on the respective cell entries. If the entries in any cell include at least one pair of states that are nonequivalent (i.e., the cell corresponding to that pair has already been crossed out), then that cell is eliminated by marking a X on it.
4. Successive passes are then made through the entire table to determine if any more states should be crossed out to indicate nonequivalency. The cell having an entry *PQ* should be crossed out only if the particular cells corresponding to the labels *P* and *Q* have already been eliminated. This pro-

cess of elimination is continued until no other cells can be eliminated.

5. Redundancy is then determined by examining the surviving cells of the implication table. Each surviving cell corresponds to an equivalency condition between the two states that are used to label that cell.

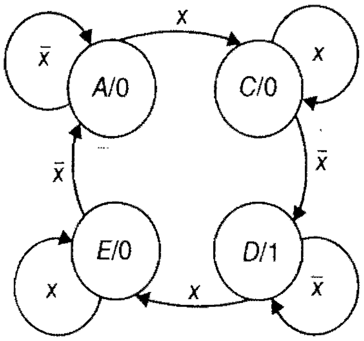
The role of the preceding algorithm will become meaningful when we use it to investigate the state table of Figure 7.12[*b*].

The entries in the table of Figure 7.13[*a*] are made by comparing each of the states with the others. For example, *CG* is entered in the cell corresponding to *A* and *F* since the equivalency of these two states is dependent on the equivalency of states *C* and *G*. Likewise, both *CG* and *AB* are entered in the cell corresponding to *B* and *F* since these two states would be equivalent only if both (a) *C* and *G* and (b) *A* and *B* are equivalent. Similar reasonings are made in completing the remaining cells.

Referring to Figure 7.13[*a*] and the state table of Figure 7.12[*b*], *D* is the only state that has an output of 1, so it cannot be equivalent to any other state. Therefore, all cells that have either a row or a column designated by *D* are crossed out. Next all the cells that have an entry composed of *D* are crossed out. As a consequence we find that all cells corresponding to the row and column designated by *C* also have been eliminated. This indicates that state *C*, like *D*, is different from all other states as well. We next cross out all cells that have at least one pair of entries involving *C*. Continuing this process, we come up with the table of Figure 7.13[*b*]. Only four surviving cells are left in the implication table.

An *equivalence partition table*, as shown in Figure 7.13[*c*], is next obtained by listing all horizontal labels (in the reverse order) as its row labels. A check is now made of each column of the final implication table (Figure 7.13[*b*]), from right to left, making note of the cells that have not been eliminated. The row and column labels of the surviving cell form an equivalent pair that is noted in the equivalent column identifier in the partition table. Equivalent pairs such as (*PQ*) and (*QR*) imply the presence of a larger group of equivalent states (*PQR*). For each column of the implication table an entry is made in the equivalence partition table, provided there is at least one surviving cell in that column. The entries from the previous rows of the partition table are entered as long as there is no repetition of the entry. In the example in question, the cell corresponding to *G* and *F* is not crossed out, and therefore the equivalent pair (*FG*) is listed next to *F* in the equivalence partition table. There are two cells corresponding to rows *F* and *G* and column *E* that result in (*FE*) and (*EG*) or the equivalent group (*EFG*). Therefore, we write (*EFG*) next to *E*, and since (*EFG*) already contains (*FG*), nothing from the previous row is repeated here. Since columns *B*, *C*, and *D* have no surviving cells, the entry (*EFG*) is repeated three times. No new equivalencies result until we come to consider *A*, for which the entry (*AB*) is added to the list. The last line of the partition table reveals that there are two sets of equivalent states: (*AB*) and (*EFG*). This implies that state *A* is equivalent to state *B* and states *E*, *F*, and *G* are equivalent to each other. Consequently, the redundant states are now eliminated by considering only one state from each equivalent group. The resultant state diagram and the state table are shown in Figure 7.14 where *B*, *F*, and *G* have been removed. The state *B* has been replaced by *A* and states *F* and *G* have been replaced with *E*.

FIGURE 7.14 **[*a*] State Diagram with No Redundancy and [*b*] State Table with No Redundancy.**



[*a*]

PS	NS		Z
	x = 0	x = 1	
A	A	C	0
C	D	C	0
D	D	E	1
E	A	E	0

[*b*]

7.4 State Assignments

State assignment is the process of adopting a binary coding scheme for the symbolic states of the state table so that it is possible for the circuit to remember which state it is in. Each bit in the code represents the output of an FF and is called a *state variable.* For *n* number of states, a total of *m* FFs will be necessary such that *m* is the smallest integer satisfying the relationship $2^m \geq n$. Any unique assignment is valid; however, it is always better if an attempt is made to assign codes in such a way that the number of cases where more than one bit in a code changes when states change is kept to a minimum. When the symbolic states are replaced with the binary coding scheme, a binary state table, commonly known as the *transition table*, results.

The state diagram of Figure 7.1 has three unique states, and, therefore, at least two FFs are necessary for designing the corresponding logic circuit. This implies that of the four different codes—00, 01, 10 and 11—only three can be used. For this example, if one chooses to assign $A = 00$, $B = 01$, and $C = 11$, the resulting transition table of Figure 7.15 is obtained.

FIGURE 7.15 **Transition Table for the Circuit of Figure 7.1.**

PS	NS	
Q_1Q_2	x = 0	x = 1
00	01,0	00,0
01	11,1	00,0
11	00,0	01,0
10	--,-	--,-

We may see that the present state $Q_1Q_2 = 01$, upon receiving the input $x = 0$, moves to the next state 11 with the resultant output of 1. During this transition the Q_1 value changes from 0 to 1, as shown highlighted in the table. In the same present state when $x =$

1, Q_2 changes from 1 to a 0. In either case the binary state changes only one of its bits. However, the present state $Q_1Q_2 = 11$ is different from the others, because when $x = 0$ both of the FF bits need to change from 1 to 0. There are situations where such a condition could cause problems, as we shall discover later.

EXAMPLE 7.4

Obtain a transition table for the sequential machine of Figure 7.14[*a*].

SOLUTION

There are four states, and, therefore, only two FFs need to be considered. The transition table results when $A = 00$, $C = 01$, $D = 11$, and $E = 10$, as shown in Figure 7.16. For the state assignments made, the transition table shows that there are no transitions for which both state variables change. Consequently, this assignment of states is considered to be very good.

FIGURE 7.16

PS	NS		
Q_1Q_2	$x = 0$	$x = 1$	Z
00	00	01	0
01	11	01	0
11	11	10	1
10	00	10	0

7.5 Excitation Maps

Up to this point when considering FFs we have been concerned with how they respond to various inputs. We will now encounter the design problem of determining their inputs such that the proper values are present to cause the next state to result when the clock input occurs. This input control is accomplished by deriving the respective excitation equations. The output equations and the state variable excitation equations are derived separately, as shown in Figures 7.17[*a*–*b*]. The FF input maps are usually called *excitation maps*.

present state

FIGURE 7.17 **[*a*] Output Table and [*b*] Excitation Maps.**

$x \backslash Q_1Q_2$	00	01	11	10
0	0	1	0	--
1	0	0	0	--

Z

[*a*]

$x \backslash Q_1Q_2$	00	01	11	10
0	0–	1–	–1	--
1	0–	0–	–1	--

J_1K_1

$x \backslash Q_1Q_2$	00	01	11	10
0	1–	–0	–1	--
1	0–	–1	–0	--

J_2K_2

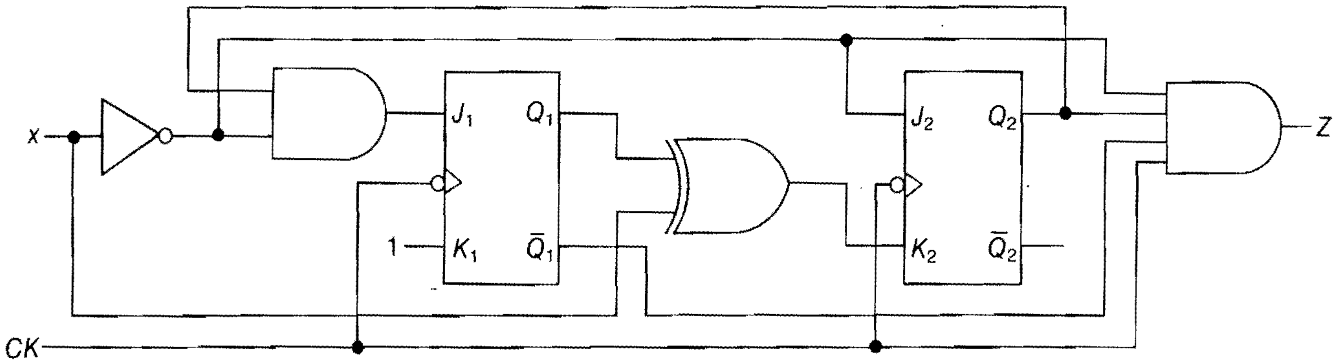
[*b*]

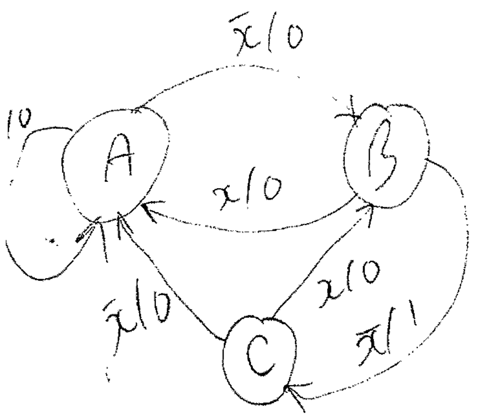
The combinational circuit corresponding to the transition table shown in Figure 7.15 has three variables—x, Q_1, and Q_2—that generate the FF inputs. Q_1 and Q_2 represent the current state of the circuit, and x is the external control input that in combination with the present state determines which action is to occur. The excitation table entries are the values of FF inputs that would cause the transition to next-state variables when the clock input occurs. We have chosen JK FFs, as an example only, for generating the excitations. One particular transition is emphasized for illustration both in the transition table and in the corresponding excitation map of Figure 7.17. The transition of Q_1, as shown in Figure 7.15, from 0 to 1 requires the FF input condition $J_1 = 1$ and $K_1 = -$. This information is entered in the corresponding cell, $x = 0$ and $Q_1Q_2 = 01$, of the excitation map. Continuing this procedure, the excitation and output equations may be obtained from the K-maps of Figure 7.17:

$$J_1 = \bar{x}Q_2$$
$$K_1 = 1$$
$$J_2 = \bar{x}$$
$$K_2 = \bar{Q}_1x + Q_1\bar{x} = Q_1 \oplus x$$
$$Z = \bar{Q}_1Q_2\bar{x} \cdot CK$$

Note that the clock input, CK, is ANDed with $\bar{Q}_1Q_2\bar{x}$ to produce the desired output of a synchronous machine. The resultant sequential circuit is obtained as shown in Figure 7.18.

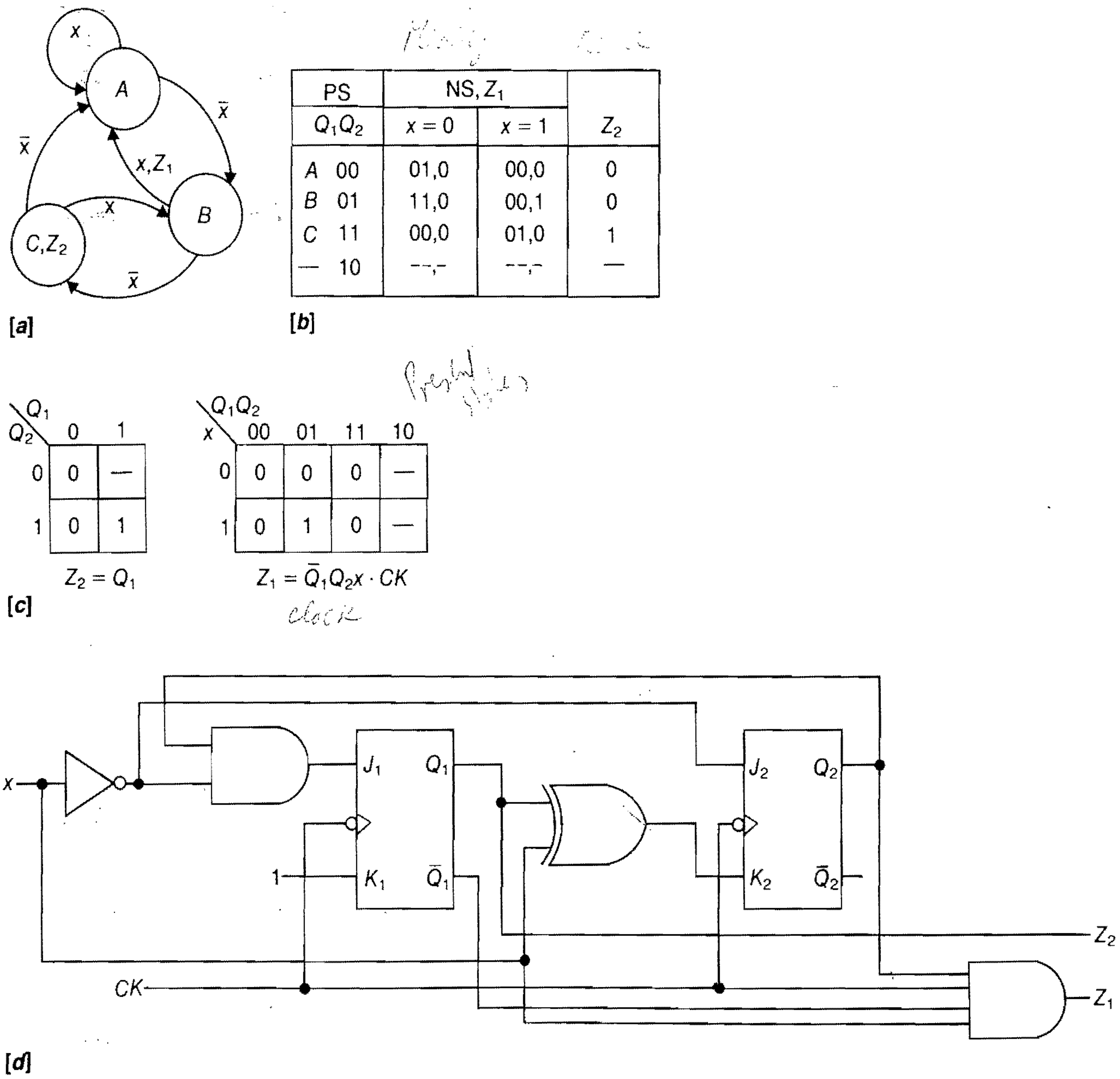
FIGURE 7.18 **Circuit Implementation of the Example.**





Consider a circuit similar to that of Figure 7.1 but having a Mealy output, Z_1, and a Moore output, Z_2, as shown in Figure 7.19[*a*]. There is to be an output coincident with the clock input when the circuit moves from state B to state A, and another output whenever the circuit is in state C. For the Z_1 output, assuming negative edge-triggered FFs, the circuit must be in state 01 and both x and clock must occur. The Mealy output is given by $Z_1 = \bar{Q}_1Q_2x \cdot CK$. If the FFs being used for the state variables were positive edge-triggered, the Z_1 output would occur in the state following 01 since the positive edge of the clock would move the circuit into the next state. The next state's state variables, 00, and the inputs would be ANDed to form Z_1 in this case. However, the Moore output is a 1

FIGURE 7.19 **Circuit with Mealy and Moore Outputs: [*a*] State Diagram, [*b*] Transition Table, [*c*] Output Tables, and [*d*] Circuit.**



[*a*]

PS	NS, Z_1		
Q_1Q_2	$x = 0$	$x = 1$	Z_2
A 00	01,0	00,0	0
B 01	11,0	00,1	0
C 11	00,0	01,0	1
— 10	--,-	--,-	—

[*b*]

$Q_2 \backslash Q_1$	0	1
0	0	—
1	0	1

$Z_2 = Q_1$

$x \backslash Q_1Q_2$	00	01	11	10
0	0	0	0	—
1	0	1	0	—

$Z_1 = \bar{Q}_1 Q_2 x \cdot CK$

clock

[*c*]

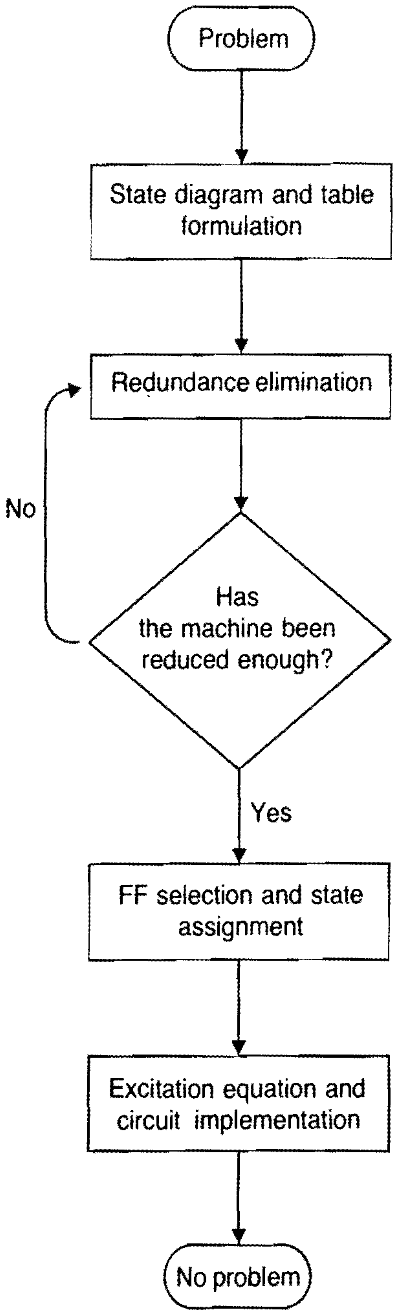
[*d*]

only when the circuit is in state *C*. Figures 7.19[*b*-*c*] show the steps involved in obtaining the final circuit of Figure 7.19[*d*].

7.6 Design Algorithm

We have examined the steps of the design of a synchronous sequential machine in the last few sections. Figure 7.20 gives a comprehensive flowchart of an algorithm for the design of sequential machines.

FIGURE 7.20 **Sequential Circuit Design Flowchart.**


Problem
State diagram and table formulation
Redundance elimination
No
Has the machine been reduced enough?
Yes
FF selection and state assignment
Excitation equation and circuit implementation
No problem

The algorithm can be summarized by the following steps:

Step 1. Obtain the state diagram from the word statement of the problem.

Step 2. Obtain the state table from the state diagram.

Step 3. Eliminate the redundant states.

Step 4. Make state assignments.

Step 5. Determine the type of FFs to use and obtain the corresponding excitation maps.

Step 6. Determine the output and FF equations.

Step 7. Construct the logic circuit.

These design steps often lead to a rather lengthy process that varies from problem to problem. The following examples illustrate the implementation of the sequential design algorithm.

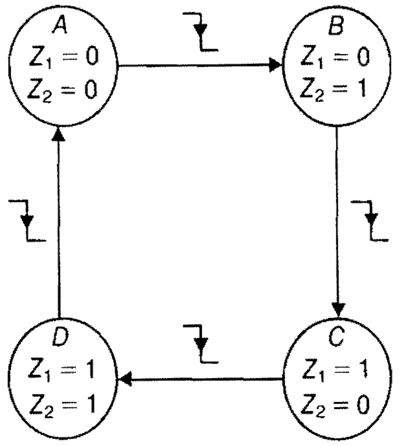
EXAMPLE 7.5

Design a two-bit clocked sequential counter circuit that counts clock pulses.

SOLUTION

Steps 1–2. We can visualize such a device as the one that receives clock pulses as input. Each time a clock pulse is received, the counter should count up. No control variable is needed; only the occurrence of the clock pulse is necessary for a state change. We shall assume further that the states change at the trailing edge of the clock input. The counter could be designed such that the outputs go through the sequence 00 → 01 → 10 → 11 and repeat. Since the counter is only a two-bit device, it would have to reset at the fourth clock. The state diagram and the corresponding state table for the counter are obtained as shown in Figure 7.21. We might choose to have the outputs directly from the FF outputs, in which case such outputs would be classed as Moore type.

FIGURE 7.21



[*a*]

PS	NS	Z_1Z_2
A	*B*	00
B	*C*	01
C	*D*	10
D	*A*	11

[*b*]

Step 3. The four states—*A*, *B*, *C*, and *D*—are all different since they stand for completely different events. Consequently, we may conclude without any doubt that none of these are redundant states.

Step 4. Making state assignments is an important step for at least one reason. The assignments of the states are crucial in determining the simplicity, or for that matter complexity, of the resultant circuit. The output circuits can be eliminated totally if the state assignments for the states *A*, *B*, *C*, and *D* are made the same as the corresponding Moore outputs of each state. Accordingly, the chosen assignments are *A* = 00, *B* = 01, *C* = 10, and *D* = 11.

Step 5. The number of FFs are indeed two. This fact was also given in the initial word statement of the problem. We might choose *JK* FFs, for example. Consequently both the transition table and excitation map are obtained as shown in Figure 7.22.

FIGURE 7.22

PS Q_1Q_2	NS	Z_1Z_2
00	01	00
01	10	01
10	11	10
11	00	11

[a]

$Q_2 \backslash Q_1$	0	1
0	0—	—0
1	1—	—1

J_1K_1

$Q_2 \backslash Q_1$	0	1
0	1—	1—
	—1	—1

J_2K_2

[b]

Step 6. The excitation maps are minimized to give

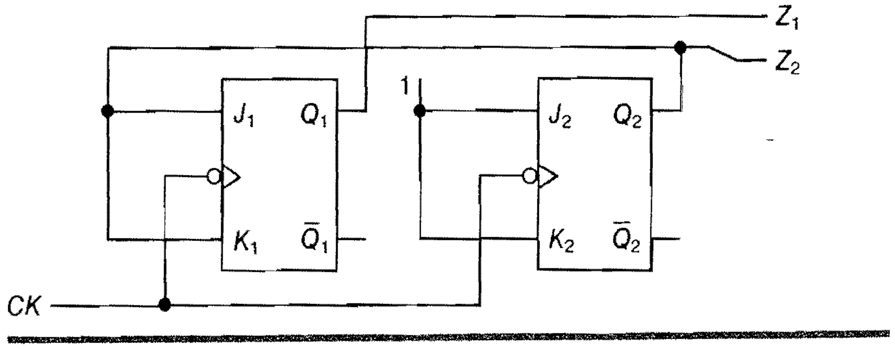
$$J_1 = Q_2$$
$$K_1 = Q_2$$
$$J_2 = 1$$
$$K_2 = 1$$

The outputs are easily realizable directly from Figure 7.22[*a*]. They are as follows:

$$Z_1 = Q_1\bar{Q}_2 + Q_1Q_2 = Q_1(\bar{Q}_2 + Q_2) = Q_1$$
$$Z_2 = \bar{Q}_1Q_2 + Q_1Q_2 = Q_2(\bar{Q}_1 + Q_1) = Q_2$$

Step 7. The resulting circuit obtained from these excitation and output equations is shown in Figure 7.23.

FIGURE 7.23



EXAMPLE 7.6

Repeat the design of Example 7.5 by assigning A = 00, B = 01, C = 11, and D = 10. Construct the corresponding timing diagram as well.

SOLUTION

Since the FF state assignments are different from the corresponding Moore outputs, Z_1 and Z_2, the circuit will require additional gates. The corresponding transition table, the *JK* excitation maps, and the output map are shown in Figure 7.24. The corresponding Boolean equations are obtained as follows:

$$J_1 = Q_2$$
$$K_1 = \bar{Q}_2$$
$$J_2 = \bar{Q}_1$$
$$K_2 = Q_1$$
$$Z_1 = Q_1$$
$$Z_2 = Q_1\bar{Q}_2 + \bar{Q}_1 Q_2 = Q_1 \oplus Q_2$$

FIGURE 7.24

PS		
Q_1Q_2	NS	Z_1Z_2
00	01	00
01	11	01
11	10	10
10	00	11

[a]

$Q_2 \backslash Q_1$	0	1
0	0—	—1
1	1—	—0

J_1K_1

$Q_2 \backslash Q_1$	0	1
0	1—	0—
1	—0	—1

J_2K_2

$Q_2 \backslash Q_1$	0	1
0	00	11
1	01	10

Z_1Z_2

[b]

The resulting logic circuit diagram is obtained as shown in Figure 7.25. It is obvious that the circuit in Example 7.5 is simpler. The timing diagram of Figure 7.26 illustrates the operation of the circuit of Figure 7.25.

FIGURE 7.25

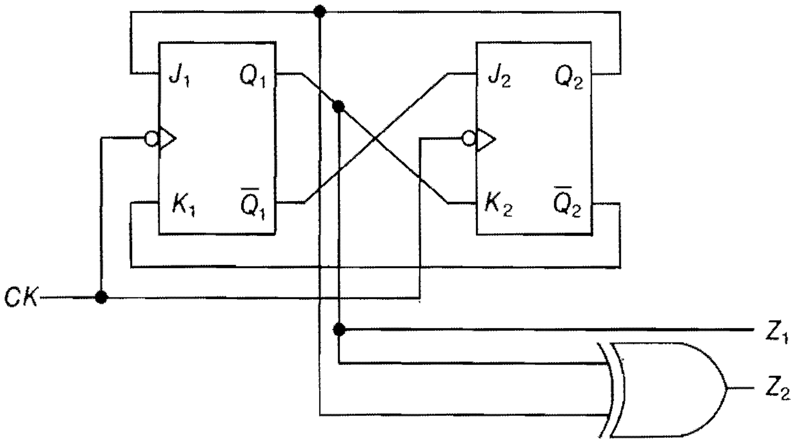
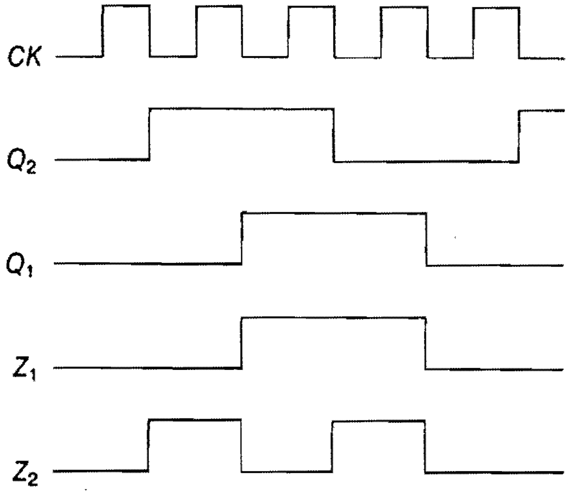


FIGURE 7.26



EXAMPLE 7.7

Complete the design of a clocked sequential circuit that recognizes the input sequence 1010, including overlapping such that for input x = 00101001010101110 the corresponding output Z is 00000100001010000.

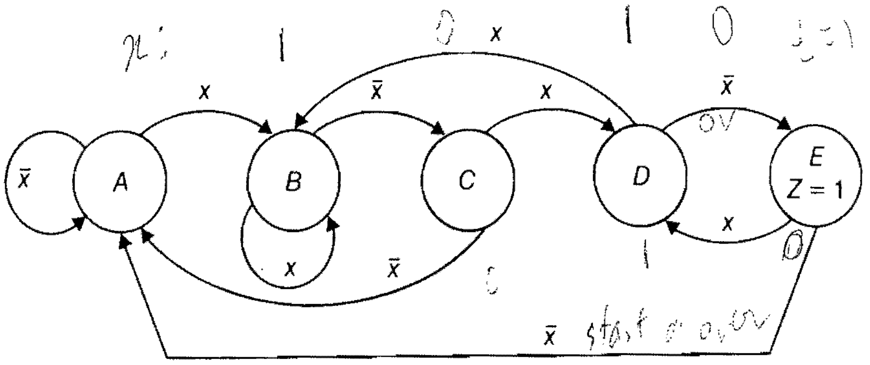
SOLUTION

The state diagram consists of five states, A–E. States B, C, D, and E represent, respectively, the occurrence of the first, second, third, and fourth bits of the sequence 1010. State E has a Moore output $Z = 1$ indicating the completion of a sequence. A subsequent input of 0 would move the circuit to state A, which indicates the input is out of sequence. An input of 1 while in E moves the circuit from state E to state D since sequence overlapping is allowed. The corresponding state diagram showing the transitions for each value of x is provided in Figure 7.27.

FIGURE 7.27

x = 0010 1001 01 01 01110
z = 0000 0100 001 01 0000

1010 0 1 0 1 0 1 0 1 1 1 0
1 2 3



Upon power-up, the circuit begins from state A. As long as the string of input is devoid of 1 (i.e., the first bit of a 1010 sequence), the circuit remains at this beginning state. Once a 1 has been located the circuit moves to state B, indicating that the first bit has already been detected. Subsequent detection of 0, 1, and 0, in that order, would amount to moving the circuit to states C, D, and E, respectively. Once the state E is reached, the circuit gives an output indicating the completion of a 1010 sequence. However, while at state B if the circuit detects a 1, the circuit reenters state B. This is due to the possibility that the most recently observed 1 might be the beginning of a 1010 sequence. For similar reasons, the detection of a 1 at state D causes the circuit to move to state B also. Again at state C, a detection of 0 eliminates the possibility of having the desired sequence, 1010. So, the circuit resets back to state A. Likewise, the circuit resets from state E to state A if it locates a 0. However, an interesting case happens when the circuit is at state E and it has just detected a 1. This time the circuit moves back to state D. This is due to the fact that detection of a 1 at state E is equivalent to detecting the third bit of a newer 1010 sequence.

The problem involves five states requiring three FFs. Three of the eight possible states will remain unused. The state table and transition table corresponding to the arbitrary assignments of A = 000, B = 001, C = 011, D = 111, and E = 101 are shown in Figure 7.28. The output and excitation maps corresponding to the use of JK FFs may now be obtained as shown in Figure 7.29. Proper grouping of the K-map cells results in the following equations:

FIGURE 7.28

PS	NS		Z
	x = 0	x = 1	
A	A	B	0
B	C	B	0
C	A	D	0
D	E	B	0
E	A	D	1

PS	NS		Z
$Q_1Q_2Q_3$	x = 0	x = 1	
000	000	001	0
001	011	001	0
011	000	111	0
111	101	001	0
101	000	111	1

$$Z = Q_1\bar{Q}_2 \cdot CK$$

$$J_1 = Q_2x$$

FIGURE 7.29

0	—	—	—
0	0	0	1

Q_1 (columns 2–3), Q_3 (row 2), Q_2 (columns 2–3)

Z

0—	--	--	--
0—	--	--	--
0—	1—	—1	—0
0—	0—	—0	—1

Q_1, Q_3, Q_2

J_1K_1

0—	--	--	--
0—	--	--	--
0—	—0	—1	1—
1—	—1	—1	0—

Q_1, Q_2

J_2K_2

0—	--	--	--
1—	--	--	--
—0	—0	—0	—0
—0	—1	—0	—1

Q_1, x, Q_2

J_3K_3

$$K_1 = Q_2x + \bar{Q}_2\bar{x} = \overline{Q_2 \oplus x}$$

$$J_2 = Q_1x + \bar{Q}_1Q_3\bar{x}$$

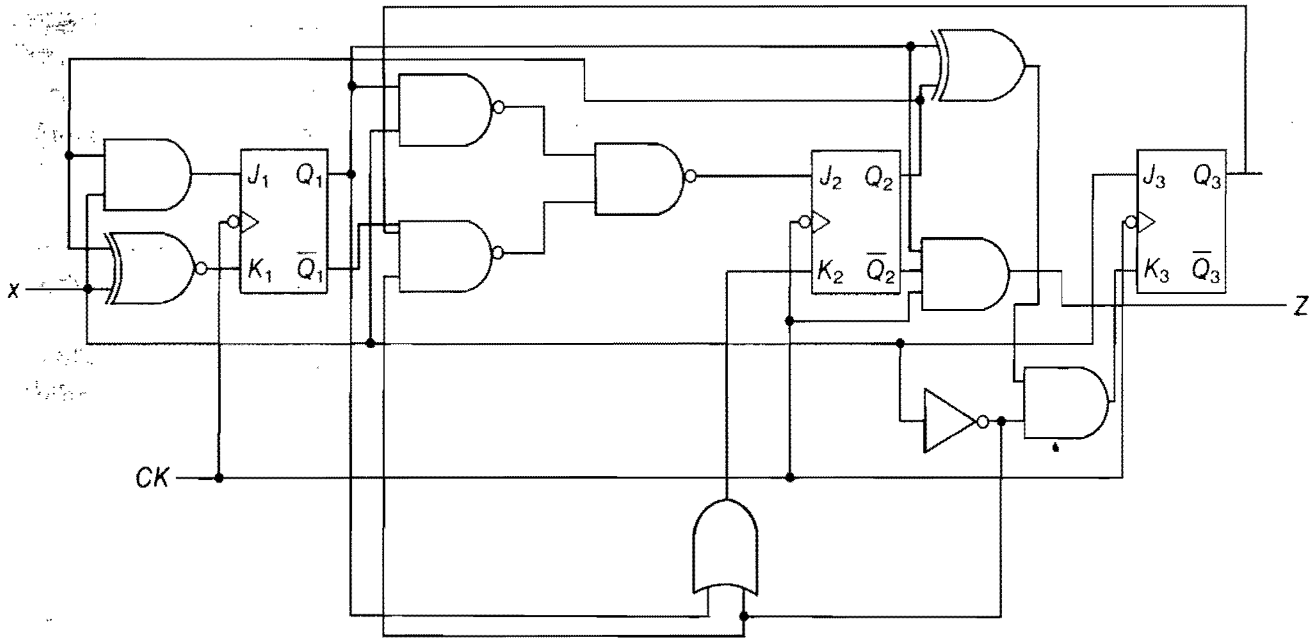
$$K_2 = \bar{x} + Q_1$$

$$J_3 = x$$

$$K_3 = \bar{Q}_1Q_2\bar{x} + Q_1\bar{Q}_2\bar{x} = (Q_1 \oplus Q_2)\bar{x}$$

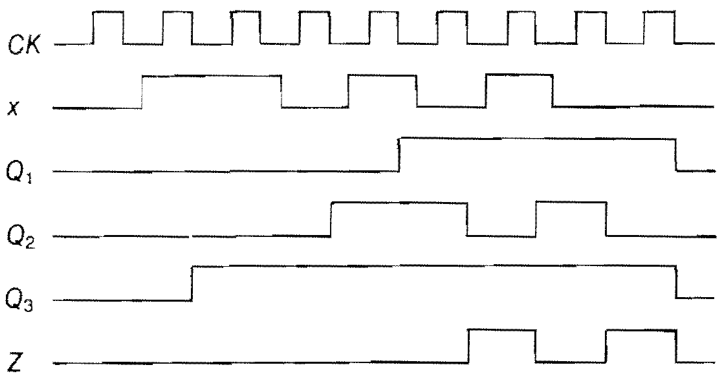
The sequential circuit of the 1010 sequence detector is obtained using the above equations and is shown in Figure 7.30[*a*]. The timing diagram of Figure 7.30[*b*] shows the relationship between the output, the clock, and the input.

FIGURE 7.30a



[*a*]

FIGURE 7.30b



[*b*]

In the examples considered thus far we have used *JK* FFs and gates. *D* FFs and other logic devices may also be used. The excitation table for all of the FFs is given in Figure 6.39. In Examples 7.8 and 7.9, *D* FFs are used.

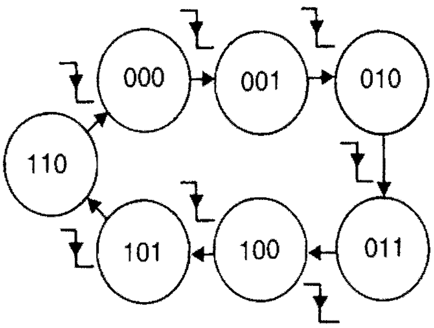
√

EXAMPLE 7.8

Obtain a scale-of-seven up-counter, as shown in the state diagram of Figure 7.31, using *D* FFs and PLA. Assume that the counter is tied to a seven-segment display device.

SOLUTION

FIGURE 7.31



The state table of the counter may be obtained as shown in Figure 7.32.

FIGURE 7.32

PS	
$Q_3Q_2Q_1$	NS
000	001
001	010
010	011
011	100
100	101
101	110
110	000

D_3 D_2 D_1

The excitation K-maps corresponding to *D* FFs are next obtained as shown in Figure 7.33. The Boolean equations, therefore, are as follows:

$$D_3 = \bar{Q}_2Q_3 + Q_1Q_2$$

$$D_2 = Q_1\bar{Q}_2 + \bar{Q}_1Q_2\bar{Q}_3$$

$$D_1 = \bar{Q}_1\bar{Q}_2 + \bar{Q}_1\bar{Q}_3$$

FIGURE 7.33

Q_1 \ Q_3Q_2	00	01	11	10
0	0	0	0	1
1	0	1	—	1

D_3

Q_1 \ Q_3Q_2	00	01	11	10
0	0	1	0	0
1	1	0	—	1

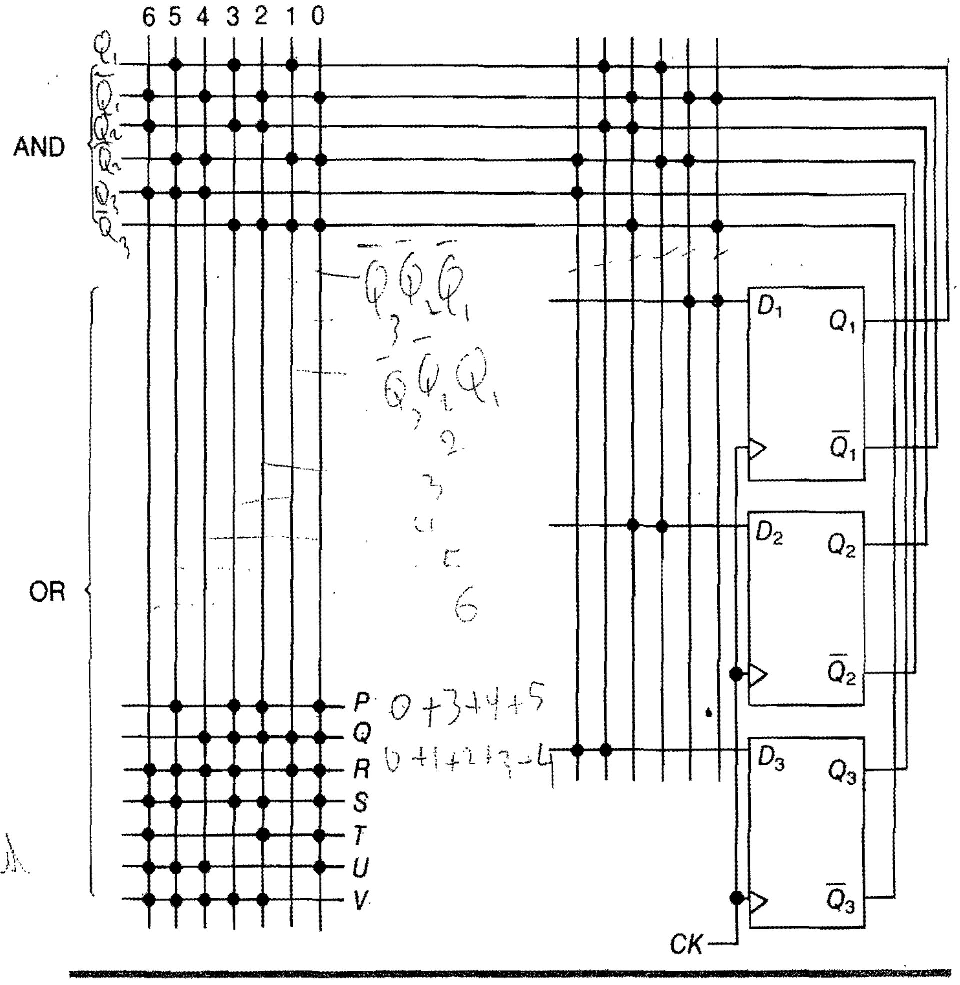
D_2

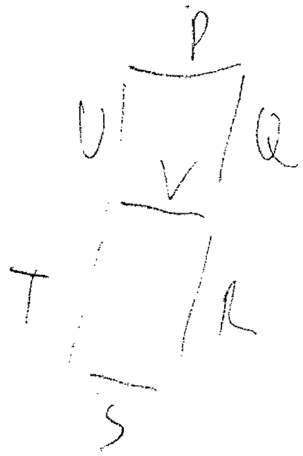
Q_1 \ Q_3Q_2	00	01	11	10
0	1	1	0	1
1	0	0	—	0

D_1

The PLA circuit configuration follows as shown in Figure 7.34. The dots in the intersection matrix correspond to either an OR or an AND operation. The segment allocation for the LED display has already been defined in Example 4.8.

FIGURE 7.34





no need

EXAMPLE 7.9

Using D FFs and assorted gates, design a sequential comparator that is to determine which of the two multi-bit numbers, X and Y, of equal length is larger.

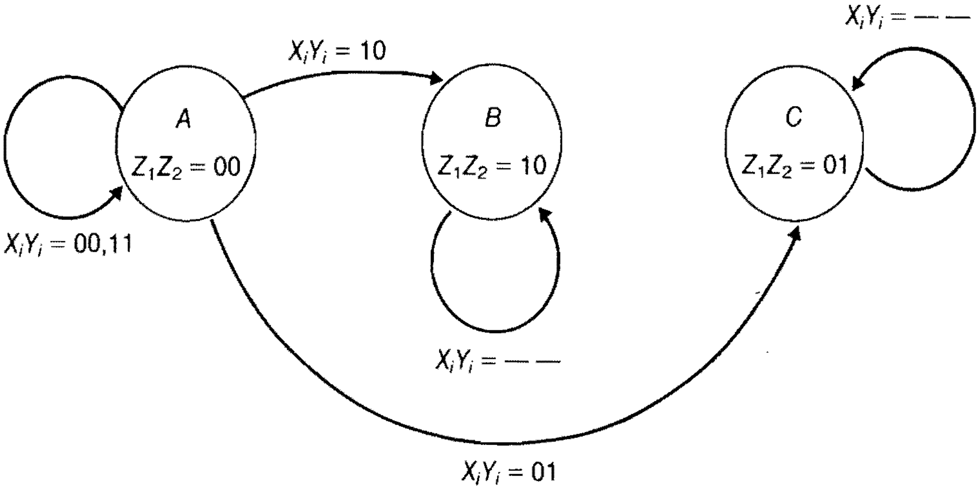
SOLUTION

The MSB of both numbers may be fed as inputs to the comparator. Two outputs, Z_1 and Z_2, may be assumed to accompany the circuit. If $X > Y$, then $Z_1 = 1$; if $X < Y$, then $Z_2 = 1$; and if $X = Y$, then $Z_1 = Z_2 = 0$.

The state diagram of the comparator is obtained as shown in Figure 7.35, where X_i and Y_i are the ith bits of X and Y, respectively. The transition table may then be obtained as shown in Figure 7.36. For simplicity, we may assign $A = 00$, $B = 10$, and $C = 01$. These assignments would allow us to derive circuit outputs directly from the FFs. That this is possible would become obvious by comparing Examples 7.5 and 7.6. Accordingly, the excitation equations are given by

$$D_1 = X_i\bar{Y}_i\bar{Q}_2 + Q_1$$
$$D_2 = \bar{X}_iY_i\bar{Q}_1 + Q_2$$

FIGURE 7.35



PS = outputs = Moore

FIGURE 7.36

Q_1Q_2

PS	NS X_iY_i = 00	01	11	10	Z_1Z_2
00 A	00 A	01 C	A	B	00
10 B	10 B	10 B	B	B	10
01 C	01 C	01 C	C	C	01

Transition table

Q_1Q_2	NS xy 00	01	11	10	Z
00	00	01	00	10	00
10	10	10	10	10	10
01	01	01	01	01	01
11	–	–	–	–	–

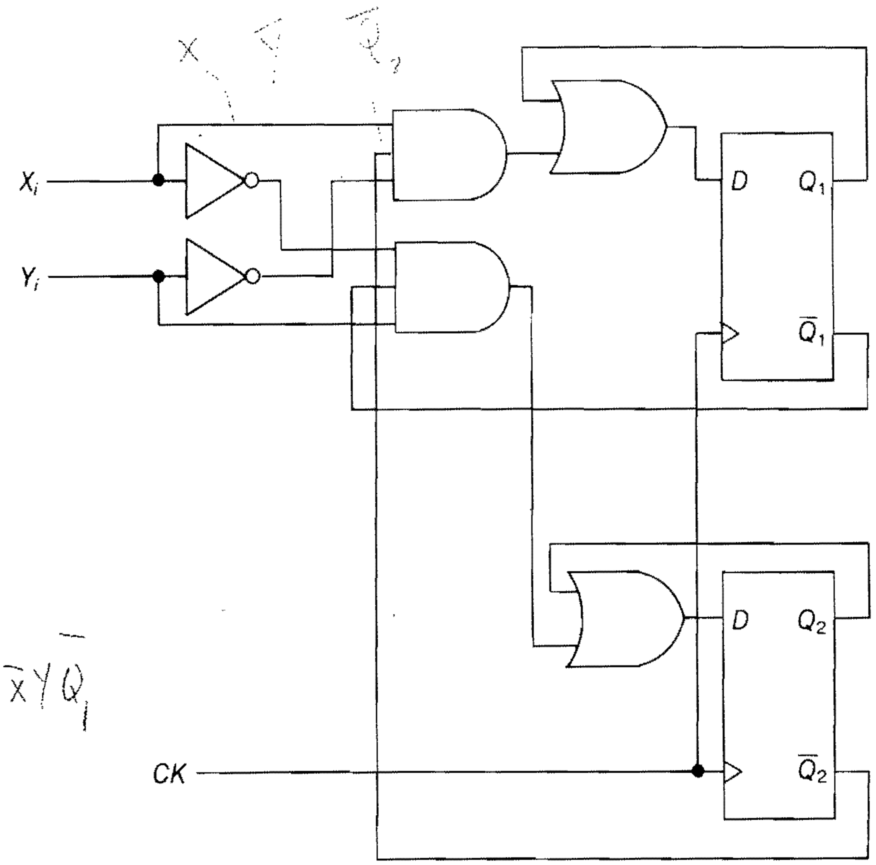
Present state = Q_1Q_2

xy \ Q_1Q_2	00	01	11	10
00	0	0	–	1
01	0	0	–	1
11	0	0	–	1
10	1	0		1

$D1 = Q_1 + x\bar{y}.\bar{Q}_2$

The resultant circuit for the sequential comparator may now be readily obtained. The circuit is illustrated in Figure 7.37.

FIGURE 7.37



XY \ Q_1Q_2	00	01	11	10
00	0	1	–	0
01	1	1	–	0
11	0	1	–	0
10	0	1	–	0

$D2 = Q_2 + \bar{X} Y Q_1$

7.7 Incompletely Specified Diagrams

All problems considered thus far in this chapter have been *completely specified*; that is, all next-state and output values were completely defined in their state diagrams and state tables. In this section we shall consider state diagrams and/or state tables that are termed *incompletely specified.* Such sequential circuits include don't-care outputs in their respective state tables and state diagrams. These circuits have an added advantage over the completely specified circuits since the presence of don't-cares may contribute to simpler Boolean expressions.

The minimization process of state tables that contain don't-cares is tedious and requires special consideration. Implication tables are used for removing state redundancies, but the steps involved are different from those for the completely specified state tables (see Section 7.3 for details). The steps for incompletely specified state tables involve the following variations:

1. The entries in the implication table are made exactly as before, but a don't-care in the output is considered to be a 1 or a 0 depending on whether this particular choice aids the formation of an equivalent group.

2. Once the successive passes and crossing out of the cells have been completed, the designer should make entries in the equivalence partition table as before. However, it must be understood that two equivalent pairs like (AB) and (AC) do not automatically imply the existence of a larger equivalent group (ABC) unless there exists an equivalent group (BC) or (BCX). This extra condition is necessary because the don't-cares may have been treated as both 0 and 1 under different conditions.
3. The maximum number of states in the reduced circuit is equal either to the number of sets of maximal compatibles or to the number of states in the original circuit, whichever is less.
4. A *closure table* is obtained by considering the maximal compatibles as states and grouping their next states under respective input columns. The reduced state table is constructed by renaming the sets of maximal compatibles. Note, however, that the resulting reduced state table might still be incompletely specified.

The application of these rules is illustrated in Example 7.10.

EXAMPLE 7.10

Obtain the reduced state table for the sequential machine shown in Figure 7.38.

SOLUTION

FIGURE 7.38

	NS, Z	
PS	$x = 0$	$x = 1$
A	A,—	B,1
B	G,—	D,0
C	B,1	B,—
D	A,1	B,—
E	C,—	A,—
F	F,—	C,—
G	G,—	G,—

The implication table is as obtained as shown in Figure 7.39. The intersection of A and B is crossed out, which indicates that A and B are unequal. This is because when $x = 1$, the next states result in different outputs. More boxes are crossed out as repeated passes are made through the table. To start, the cells with the entry AB are crossed out. At the end of such search the equivalence partition table is obtained as shown in Figure 7.40. For better understanding of the operations, consider row B. In row B there are seven groups–(BG), (BE), (BD), (BC), (CG), (EG), and (DFG)–that reduce to four possible groups of three states: (BEG), (DFG), (BCG), and

FIGURE 7.39

B	☒					
C	~~AB~~	BG BD				
D		AG BF	~~AB~~			
E	~~AB AC~~	CG AD	~~BC AB~~	~~AB AC~~		
F	BC	~~FG CD~~	~~BF BC~~	AF BC	~~CF AC~~	
G	AG BG	DG	BG	AG BG	CG AG	CG
	A	B	C	D	E	F

FIGURE 7.40

F	(FG)	
E	(EG)(FG)	
D	(DG)(DF)(EG)(FG)	
D	(EG)(DFG)	Using step 2
C	(CG)(EG)(DFG)	
B	(BG)(BE)(BD)(BC)(CG)(EG)(DFG)	
B	(BEG)(DFG)(BCG)(BDG)	Using step 2
A	(AD)(AF)(AG)(BEG)(DFG)(BCG)(BDG)	
A	(ADFG)(BEG)(BCG)(BDG)	Using step 2

FIGURE 7.41

PS	NS, Z	
	$x = 0$	$x = 1$
ADFG	AFG,1	BCG,1
BEG	CG,—	ADG,0
BCG	BG,1	BDG,0
BDG	AG,1	BDG,0

FIGURE 7.42

PS	NS, Z	
	$x = 0$	$x = 1$
P	P,1	R,1
Q	R,—	P,0
R	QRS,1	S,0
S	P,1	S,0

(*BDG*). (*BG*), (*BE*), and (*EG*) yield (*BEG*); (*BG*), (*BC*), and (*CG*) yield (*BCG*); and (*BG*), (*BD*), and (*DFG*) yield (*BDG*). Note that (*BG*) has been used in all three determinations, and such manipulations are valid. In the final form we have ~~five~~ four possible sets renamed as follows:

$P = (ADFG)$

$Q = (BEG)$

$R = (BCG)$

$S = (BDG)$

The closure table can now be constructed as shown in Figure 7.41. In the first row of the closure table, *AFG* are the next states for states *ADFG* when $x = 0$. For $x = 1$, *BCG* are the next states for states *ADFG*. Now we may construct the reduced state table by using the variables *P*, *Q*, *R*, and *S*. The table can be organized as shown in Figure 7.42. Corresponding to $x = 0$, state *R* could move to any one of the three states, *Q*, *R*, and *S*. This is because (*BG*) is present in all of those three sets of compatibles. Note that the reduced table is still incompletely specified.

7.8 Ideal State Assignments

In the previous sections state assignments were made arbitrarily with no consideration of the consequences. It will be seen that the combinational circuit complexity is different for different sets of state assignments. The number of possible state assignments for any given problem is impressive. For n present states and p flip-flops, there are $2^p!/[n!(2^p - n)!]$ ways of selecting n out of the 2^n possible combinations. For each of these ways there are $n!$ permutations of assigning the n combinations to the n states. Again, for each of these assignments there are 2^p ways of interchanging logic 0 and logic 1 and there are $p!$ ways of interchanging the FFs. Consequently, there may be a total of $[(2^p - 1)!]/[(2^p - n)!p!]$ unique assignments. For example, the number of unique assignments for a nine-state system can be calculated to be 10,810,800.

The *optimum state assignment* is one that reduces the amount of combinational logic of a sequential system when compared to other assignments. Many different approaches to this state assignment problem have been developed. The complexity and cost of the circuit will differ for different combinations of state assignments. The identification of the best state assignments has been the subject of a considerable amount of research. We can attempt to locate the best set by generating those output and excitation tables that allow the formation of large clusterings of ones. Use of the following guidelines will probably result in simpler circuits:

1. Adjacent assignments should be given to those states that have the same next state for any given input.
2. Two or more states that are the next states of the same state, under adjacent inputs, should be given adjacent assignments.
3. States that have the same output for a given input should be given adjacent assignments.

The term *adjacent assignments* means that the states appear next to each other on the mapped representation of the state table. The assignment guidelines work best with D and JK FFs. These rules usually lead to a good, but not necessarily to the optimum, solution. It may not always be possible to satisfy all of the guidelines at the same time. In case of conflicts, rule 1 is preferable. An attempt should be made to satisfy the maximum number of suggested adjacencies. However, remember that an ideal state assignment may not always reduce the cost, and it is true also that the cost of the devices is often an insignificant part of the overall cost of a digital system.

7.9 Summary

In this chapter all aspects of the design of a synchronous sequential circuit were considered. It is possible to design numerous types of digital systems using synchronous sequential design. However, there are many digital systems that are of the asynchronous type as well. We will investigate the nature of both pulse-mode and fundamental-mode circuits prior to elaborating an additional application of sequential circuits. Chapter 10 presents a variety of such applications that include sequential circuits of all three types and some involving combinations of all three. Next, in Chapter 8, we shall consider pulse-mode sequential circuits.

Problems

1. Design a three-bit counter that counts up when a control variable $E = 0$, and counts down when $E = 1$.
2. Design a four-bit binary up-counter using *JK* FFs.
3. Design a synchronous sequential circuit using *SR* FFs that results in an output of 1 whenever each the following sequences occurs:
   a. 0001
   b. 0101
   c. 1101
   d. 1011
   e. 10010
   f. 11011
   g. 10011
   h. 110110
4. Repeat Problem 3 using *JK* FFs.
5. Repeat Problem 3 using *T* FFs.
6. Assume a two-bit binary counter that counts up when $A = 1$ and $B = 0$; counts down when $A = 0$ and $B = 1$; halts when $A = 0$ and $B = 0$; and is forbidden to operate when $A = B = 1$. Obtain the state diagram and the *JK* equations.
7. Obtain the equivalent Mealy state table from the machine of Figure 7.14[*b*].
8. Obtain the equivalent Mealy circuit for the circuit of Figure 7.14[*b*].
9. Given the state tables of Figure 7.P1, find the logic equations and logic diagrams for each table using *JK* FFs.

FIGURE 7.P1

PS	NS, *Z*	
	$x = 0$	$x = 1$
A	*A*,0	*C*,0
B	*D*,1	*A*,0
C	*F*,0	*F*,0
D	*E*,1	*B*,0
E	*G*,1	*G*,0
F	*C*,0	*C*,0
G	*B*,1	*H*,0
H	*H*,0	*C*,0

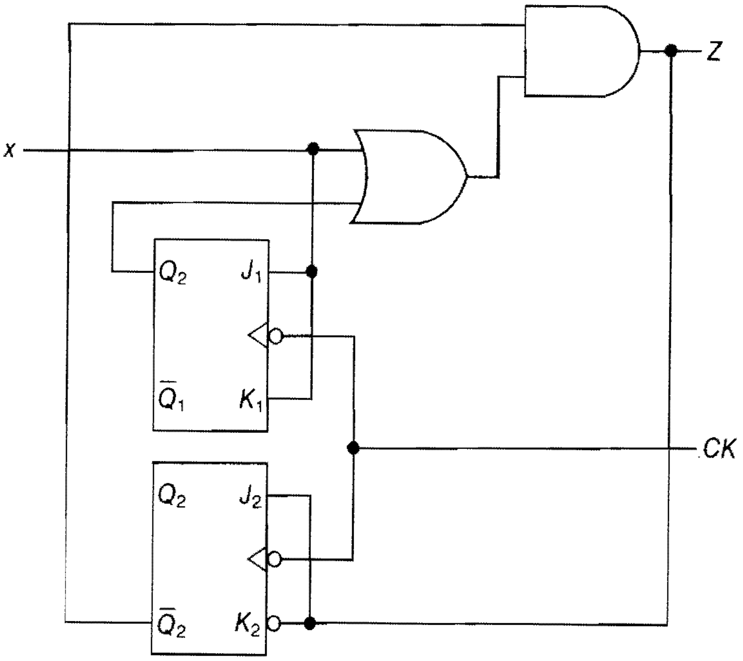
[*a*]

PS	NS, *Z*	
	$x = 0$	$x = 1$
A	*A*,0	*B*,0
B	*C*,0	*B*,0
C	*D*,0	*B*,0
D	*A*,1	*B*,0

[*b*]

10. Repeat Problem 9 using *SR* FFs.
11. Repeat Problem 9 using *D* FFs.
12. Design a synchronous sequence detector that produces an output of 1 whenever any one of the sequences 1100, 1010, and 1001 occurs. The circuit resets to its initial state after a 1 has been generated.
13. Find the Moore equivalent circuits of the two machines given in Problem 9. Use *JK* FFs.
14. Analyze the sequential circuit shown in Figure 7.P2. Obtain the state equations and the state diagram.

FIGURE 7.P2



15. Construct the state diagram for the following synchronous equations:

$$D_1 = \bar{x}Q_2 + \bar{Q}_1Q_2 + xQ_1\bar{Q}_2$$
$$D_2 = \bar{x}Q_2 + x\bar{Q}_1Q_2$$
$$z = Q_2 \cdot CK$$

16. Design a BCD counter with (a) *JK* FFs and (b) *D* FFs.
17. Design a four-bit Gray code up-counter using (a) *JK* FFs and (b) *D* FFs.
18. Design counters that follow each of the following binary sequences. For example, "0, 1, 3, 2, 5, 7, 4, and repeat" implies that the counter repeats the sequence 0, 1, 3, 2, 5, 7, 4, 0, 1, 3, 2, 5, 7, 4, 0, 1, 3, 2, 5, 7, 4, 0, 1 and so on.
    a. 0, 1, 3, 2, 5, 7, 4, and repeat. Use *SR* FFs.
    b. 0, 1, 3, 2, 6, 4, 5, and repeat. Use *T* FFs.
    c. 0, 2, 4, 6, 1, and repeat. Use *JK* FFs.

19. A synchronous sequential circuit is shown with its state diagram in Figure 7.P3. Determine the state diagram for the circuit if the primary and secondary variables are interchanged as shown in Figure 7.P4.

FIGURE 7.P3

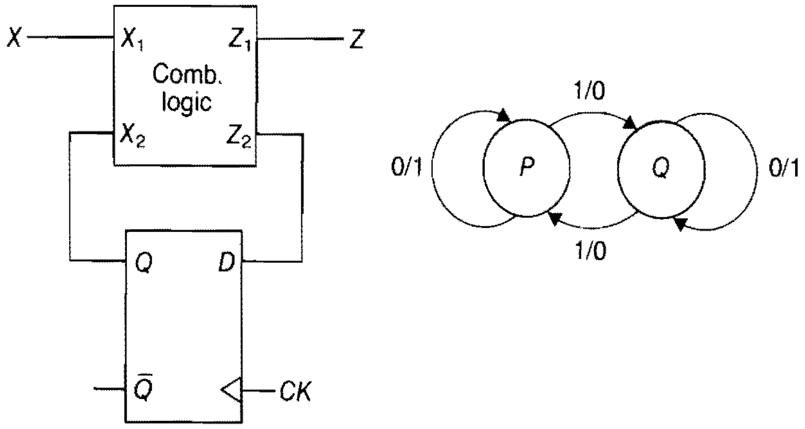
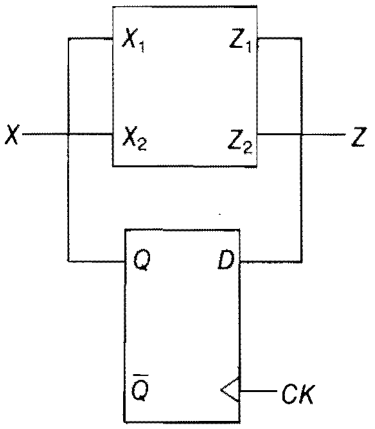


FIGURE 7.P4



20. Obtain the reduced state machine from the state tables shown in Figure 7.P5. Obtain the corresponding sequential circuits using *D* FFs.

FIGURE 7.P5

PS	NS, Z	
	$x = 0$	$x = 1$
A	*F*,0	*D*,1
B	*C*,1	*F*,1
C	*F*,1	*B*,1
D	*E*,1	*G*,1
E	*A*,1	*D*,1
F	*G*,0	*B*,1
G	*A*,0	*D*,1

[*a*]

PS	NS, Z	
	$x = 0$	$x = 1$
A	*C*,0	*B*,1
B	*C*,1	*A*,1
C	*E*,0	*B*,1
D	*F*,0	*A*,0
E	*A*,0	*G*,1
F	*D*,1	*C*,1
G	*E*,1	*C*,1

[*b*]

PS	NS, Z	
	$x = 0$	$x = 1$
A	*E*,0	*B*,1
B	*F*,0	*D*,1
C	*E*,0	*B*,1
D	*F*,0	*B*,0
E	*C*,0	*F*,1
F	*B*,0	*C*,0

[*c*]

21. Repeat Problem 20 using *JK* FFs.
22. Repeat Problem 20 using *SR* FFs.

Suggested Readings

Almaini, A. E. A. "Sequential machine implementations using universal logic modules." *IEEE Trans. Comp.* vol. C-27 (1978): 951.

Armstrong, D. B. "A programmed algorithm for assigning codes to sequential machines." *IRE Trans. Elect. Comp.* vol. EC-11 (1962): 466.

Dunworth, A., and Hartog, H. V. "An efficient state minimization algorithm for some special classes of incompletely specified sequential machines." *IEEE Trans. Comp.* vol. C-28 (1979): 531.

Fojo, J. M., and Barreiro, F. D. "A method for reducing the number of input variables to synchronous sequential circuits." *IEEE Trans. Comp.* vol. C-24 (1975): 1029.

Hartmanis, J. "On the state assignment problem for sequential machines I." *IRE Trans. Elect. Comp.* vol. EC-10 (1961): 157.

Joseph, J. "On easily diagnosable sequential machines." *IEEE Trans. Comp.* vol. C-27 (1978): 159.

McCluskey, E. J., and Unger, S. H. "A note on the number of internal variable assignments for sequential switching circuits." *IRE Trans. Elect. Comp.* vol. EC-8 (1959): 439.

Nagle, H. T., Jr.; Carroll, B. D.; and Irwin, J. D. *An Introduction to Computer Logic.* Englewood Cliffs, N.J.: Prentice-Hall, 1975.

Nanda, N. K., and Bennetts, R. G. "Reconvergence phenomenon in synchronous sequential circuits." *Elect. Lett.* vol. 16 (1980): 303.

Noe, P. S., and Rhyne, V. T. "Optimum state assignment for *D* flip-flop." *IEEE Trans. Comp.* vol. C-25 (1976): 306.

Pattavina, A., and Trigila, S. "Combined use of finite-state machines and Petri nets for modelling communicating processes." *Elect. Lett.* vol. 20 (1984): 915.

Paull, M. C., and Unger, S. H. "Minimizing the number of states in incompletely specified sequential switching functions." *IEEE Trans. Elect. Comp.* vol. EC-8 (1959): 356.

Rao, C. V. S., and Biswas, N. N. "Minimization of incompletely specified sequential machines." *IEEE Trans. Comp.* vol. C-24 (1975): 1089.

Stearns, R. E., and Hartmanis, J. "On the state assignment problems for sequential machines II." *IRE Trans. Elect. Comp.* vol. EC-10 (1961): 593.

Unger, S. H. "Self-synchronizing circuits and nonfundamental mode operation." *IEEE Trans. Comp.* vol. C-26 (1977): 278.

Waxman, J., and Rootenberg, J. "Logic circuit for cyclic detection in a state diagram." *IEEE Trans. Comp.* vol. C-26 (1977): 303.

Yamamoto, M. "A method for minimizing incompletely specified sequential machines." *IEEE Trans. Comp.* vol. C-29 (1980): 732.

Yang, C. C. "Closure partition method for minimizing incomplete sequential machines." *IEEE Trans. Comp.* vol. C-22 (1973): 1109.

Introduction to Counters, Registers, and Register Transfer Language

CHAPTER TEN

10.1 Introduction

With the study of flip-flops and sequential circuits behind us, the study of counters and registers will be a natural and straightforward extension. Counters and registers are essential to the design of advanced circuits found in digital computers. *Counters* are employed to keep track of a sequence of events, and *registers* are used to store and manipulate data that contribute to all or many of these events. In other words, most of the robust digital systems would have two major functional units: a unit where the manipulations are conducted and a unit that is used for regulating the events of the first unit. Registers and associated logic subunits help to make the first unit, and counters could be used for running the second unit. Therefore, without an understanding of flip-flops, counters, and registers, design of digital systems would be impossible.

Counters are particularly common in the control and arithmetic units of processors, where they are used to keep track of the sequence of instructions in a program, to distribute the sequence of timing signals, for frequency division for causing time delays, for counting, and a host of other similar operations. Counters may count in binary or in nonbinary fashion. They are commercially available in a large variety of medium-scale integrated devices. The basic operational characteristic of a counter is sequential; for every present state there is a well-defined next state. The design of a counter involves designing combinational logic that decodes the present state and enables entry into the next state of the counting sequence. Counters are generally classified into two groups: synchronous and asynchronous. A *synchronous counter* has all FFs change state synchronously with the clock input whether a periodic clock or an aperiodic pulse occurs. An *asynchronous* (or *ripple*) *counter* is made up of FFs that do not change simultaneously with the clock input.

Another application for FFs is for storing bits of information. When FFs are configured to store multi-bit information, they are

referred to as *registers*. Registers are classified according to the way information bits are stored and retrieved. If data are stored and removed at either end of a multi-bit register, one bit at a time, the register is referred to as a *serial* or *shift register*. However, if all bits of the word are stored or retrieved simultaneously, the register is referred to as a *parallel register*.

Another area that needs to be investigated is how a digital system, however small it may be, is built, integrated, and run. The control unit of a system can be designed using the methods that were developed in the last three chapters. However, these sequential design techniques are inadequate for the representation of subsystems that are used strictly in the manipulation of data. The tool that has been found useful in accomplishing this representation is known as the *register transfer language* (RTL). RTL helps to translate a specification mechanically into its hardware realization.

The beginning of this chapter is devoted to the development and study of various counters and registers. This discussion is then gradually expanded to include the basics of RTL. Finally, RTL is used in the design of complete digital systems. After studying this chapter, you should be able to:

○ Design and analyze both synchronous and asynchronous counters;

○ Design and analyze serial, parallel, and hybrid registers;

○ Design and analyze systems that have counters and registers;

○ Translate complex operations into equivalent RTL sequences;

○ Use RTL in the design of data and control units.

10.2 Synchronous Binary Counters

Synchronous counters are distinguished from asynchronous (or ripple) counters in that the clock pulses in synchronous counters initiate changes in the FFs used in the counter. The simplest possible counter is a single-bit counter that alternates between two states, 0 and 1. A toggle FF using a single *JK* FF, with both inputs tied to 1 ($J = K = 1$), will function as a single-bit counter alternating between the two states with the occurrence of each clock. The output of the FF has a frequency that is one-half the clock frequency.

A two-bit binary up-counter with four states was already designed in Example 7.5. Such a counter consists of two *JK* FFs whose states Q_2Q_1 could be assumed to move in sequence through 00, 01, 10, 11, 00, 01, and so on. The corresponding *J* and *K* inputs of the two FFs are given by

$$J_1 = K_1 = 1$$
$$J_2 = K_2 = Q_1$$

Note that these equations are slightly different from those given in Example 7.5. The positions of the FFs have been reversed and output equations are abandoned altogether. We can take the outputs directly from the FFs.

We shall now attempt to synthesize a three-bit binary up-counter of the nonterminal type. With every clock input the counter moves to the next higher state. Consider the FF outputs to correspond to the present state. The first step in the design sequence of a sequential circuit is to begin with a state diagram and a state table followed by the assignment of states. The state diagram, the state table, and excitation maps of a three-bit counter are shown in Figure 10.1.

The excitation maps of Figure 10.1[*c*] may be used to obtain the *JK* equations as follows:

$$J_1 = K_1 = 1$$
$$J_2 = K_2 = Q_1$$
$$J_3 = K_3 = Q_1Q_2$$

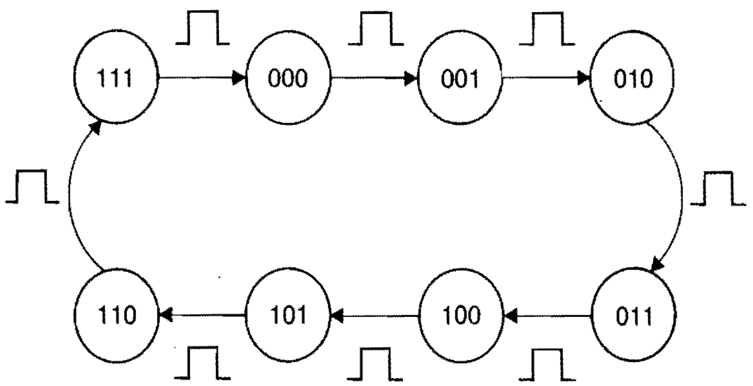
The resulting circuit diagram is shown in Figure 10.2.

We learned in Chapter 5 that many of the complex combinational designs are realized using heuristic techniques. This simplification is more true when modularity is evident in the system. Quite similarly, many sequential design problems can be accomplished without going beyond state tables. A close examination of the state table of Figure 10.1[*b*] reveals the presence of a certain degree of regularity. Note that Q_1 changes with every clock pulse and, in general, Q_i changes state if all less significant bits are 1. Similar conclusions about the regularity of counter design can be made by inspecting the respective *JK* equations of the three counters that we have considered thus far. An inspection of the *J* and *K* equations leads us to the conclusion that, based on a regular pattern, these equations can be extended to *J* and *K* equations for the *n*th bit of a multi-bit up-counter as follows:

$$J_n = Q_{n-1}Q_{n-2}\cdots Q_3Q_2Q_1 = Q_{n-1}J_{n-1}$$
$$K_n = Q_{n-1}Q_{n-2}\cdots Q_3Q_2Q_1 = Q_{n-1}K_{n-1}$$

There are two different ways of connecting the inputs to successive FFs based on the two forms of the equation for the *n*th term. Both are illustrated in Figure 10.3. Part [*a*] of the figure shows a configuration where the FF outputs are combined in parallel. The propagation delay at the input of each FF is the same for all stages. However, the fan-in to the AND gate and the fan-out of each FF increase as the number of counter stages is increased. Figure 10.3[*b*] shows an equivalent configuration using the second form of the J_n and K_n equations. The fan-in of the AND gates is always two; how-

FIGURE 10.1 Three-Bit Binary Up-Counter: [*a*] State Diagram, [*b*] State Table, and [*c*] Excitation Maps.



[*a*]

PS	
$Q_3Q_2Q_1$	NS
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

[*b*]

FF 3 2 1
J_3 J_2 J_1
K_3 K_2 K_1

J_1

Q_3 \ Q_2Q_1	00	01	11	10
0	1	—	—	1
1	1	—	—	1

J_2 (Q_2)

0	1	—	—
0	1	—	—

J_3 (Q_1Q_2)

0	0	1	0
—	—	—	—

K_1 (Q_1)

Q_3				
	—	1	1	—
1	—	1	1	—

K_2 (Q_1)

—	—	1	0
—	—	1	0

K_3 (Q_1, Q_1Q_2)

—	—	—	—
0	0	1	0

[*c*]

FIGURE 10.2 Logic Circuit of a Three-Bit Up-Counter.

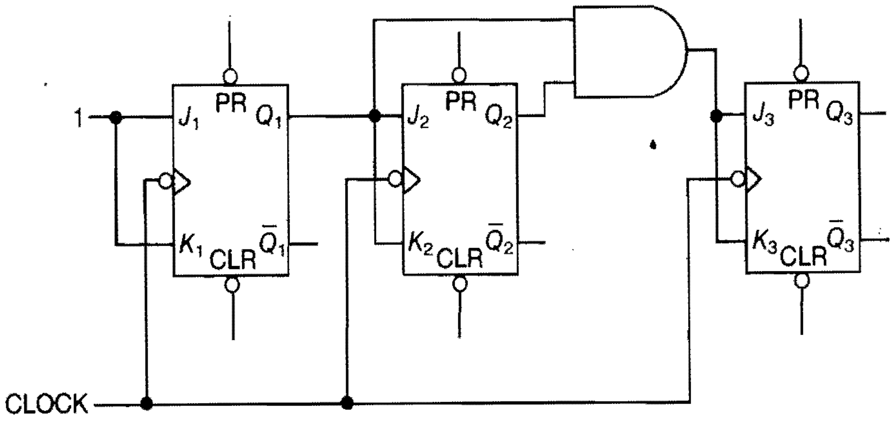
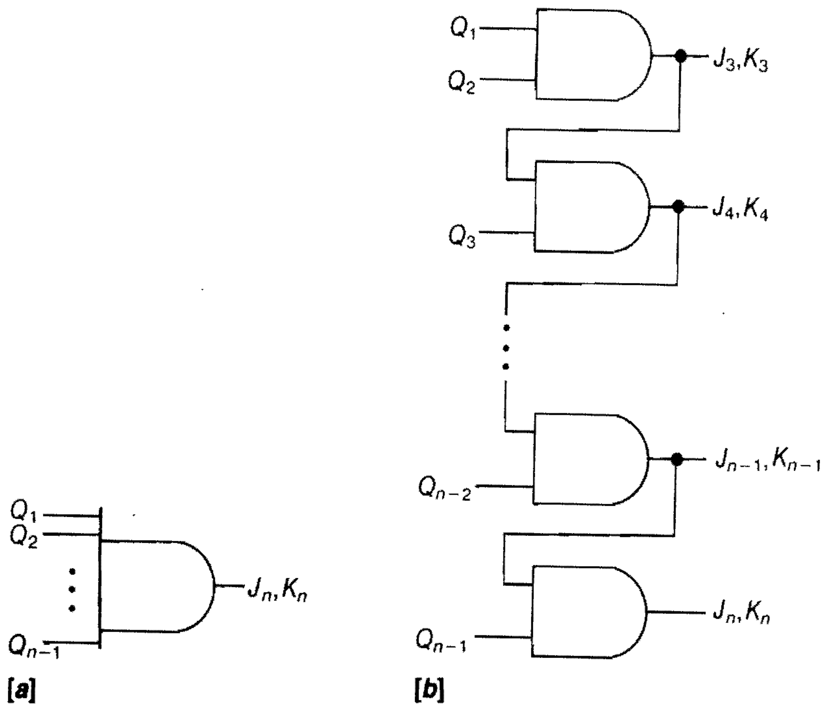


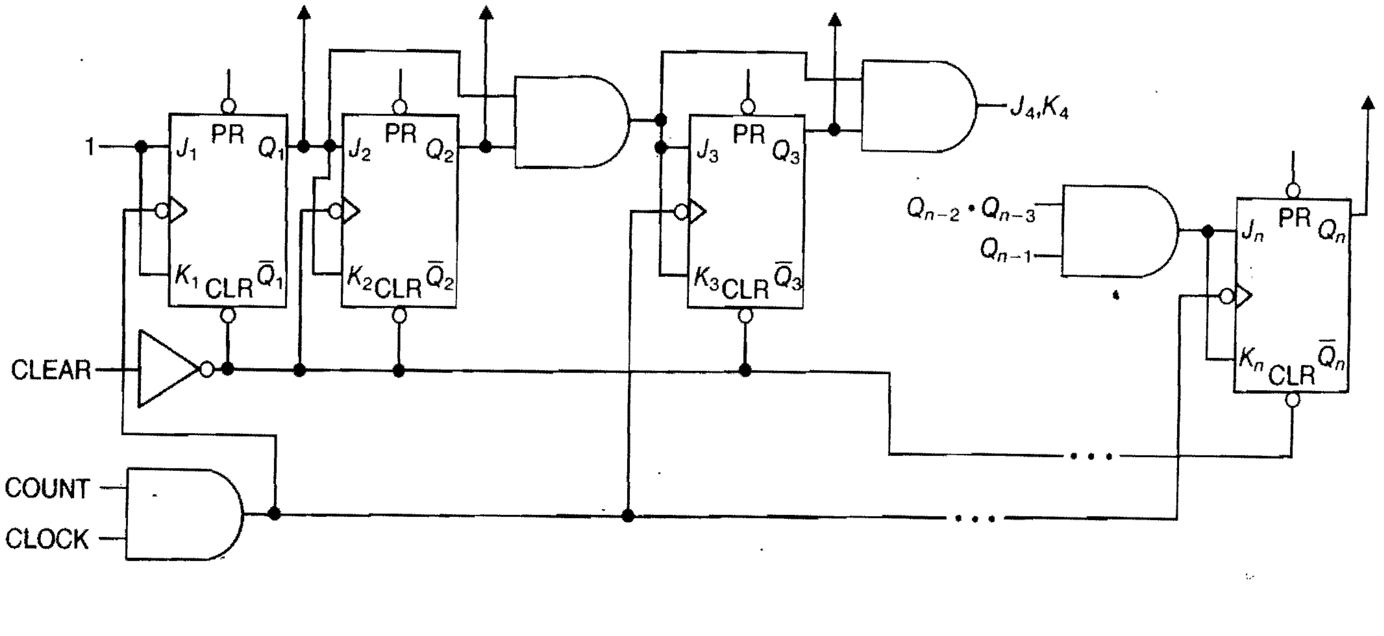
FIGURE 10.3 **Configurations for the *J* and *K* Inputs of an *n*-Bit Counter: [*a*] Parallel and [*b*] Serial.**



ever, the propagation delay to the *n*th FF increases as the number of counter stages is increased. Note that the first method allows faster clocking and counting but results in fan-in and fan-out problems for large counters.

Figure 10.4 shows a synchronous *n*-bit binary up-counter using *JK* FFs and two-input AND gates. Two control signals are added in this circuit, CLEAR and COUNT. A high on the CLEAR input resets the counter; the counter remains reset until the CLEAR signal is withdrawn. The COUNT signal is used to disable the clock pulses. The designer can use this to block the clock input and hold any nonzero count state. If the preset (PR) inputs are effectively

FIGURE 10.4 **n-Bit Synchronous Up-Counter.**



used, one may even be able to set the counter to its maximum count state.

There are many occasions in a digital system when a down-counter is required. A binary number is set into the counter that then counts toward zero as the clock pluses occur. These counters can be designed in the same way as up-counters. The equations for the down-counters are also seen to have regularities. The J and K equations for an n-bit binary down-counter are obtained as follows:

$$J_1 = K_1 = 1$$
$$J_2 = K_2 = \bar{Q}_1$$
$$J_3 = K_3 = \bar{Q}_1\bar{Q}_2 = J_2\bar{Q}_2 = K_2\bar{Q}_2$$

....

$$J_n = K_n = \bar{Q}_{n-l}\,\bar{Q}_{n-2}\ldots\bar{Q}_3\bar{Q}_2\bar{Q}_1 = J_{n-1}\bar{Q}_{n-1} = K_{n-1}\bar{Q}_{n-1}$$

By comparing these equations with those of the up-counters, we note that the resulting circuit is similar in nature. The J_n and K_n inputs are taken from the two-input AND gate, whose inputs are $\bar{Q}_{n-1}$ and K_{n-1}. Note in the case of the up-counters the corresponding AND inputs were Q_{n-1} and J_{n-1}.

For some of the applications a counter may be required to count up or down. One such application could be a counter device that keeps track of total cars inside a parking garage. As each car enters the garage, the counter counts up; as each car leaves, the counter counts down. This is a more complex design than that of all counters considered so far since it requires at least one control signal, E, to determine the direction of the count. We may assume that when $E = 1$ the circuit counts up, and when $E = 0$ the circuit counts down. One way to synthesize such a circuit would be to follow the standard steps for the design of sequential circuits (see Chapter 7, Problem 6, for example). However, we can combine the equations for up- and down-counters to derive the respective J and K equations of an n-bit, up-down counter as follows:

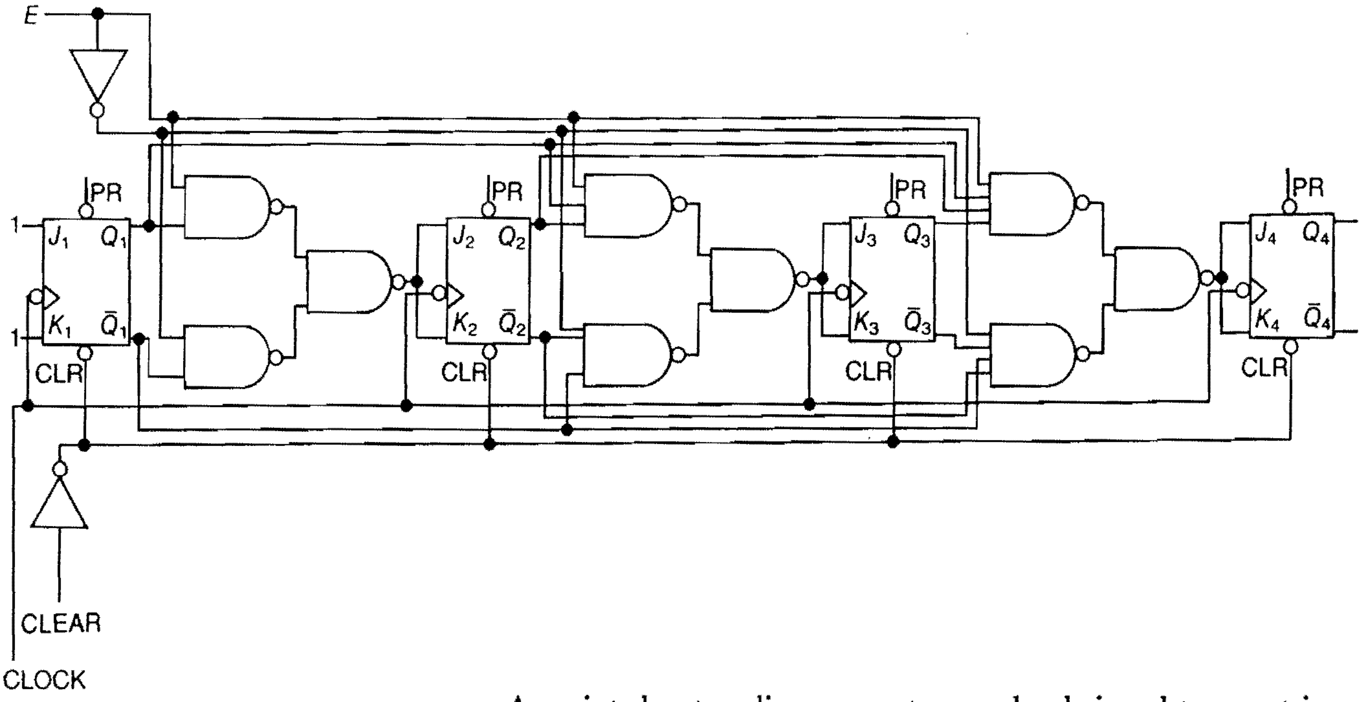
$$J_1 = K_1 = 1$$
$$J_2 = K_2 = EQ_1 + \bar{E}\bar{Q}_1$$
$$J_3 = K_3 = EQ_1Q_2 + \bar{E}\bar{Q}_1\bar{Q}_2$$

....

$$J_n = K_n = EQ_{n-1}Q_{n-2}\ldots Q_2Q_1 + \bar{E}\bar{Q}_{n-1}\bar{Q}_{n-2}\ldots\bar{Q}_2\bar{Q}_1$$

The excitation function of the up-counter is ANDed with E, and that of the down-counter is ANDed with $\bar{E}$, and, finally, the two corresponding composite functions are ORed to obtain the J and K equations. Consequently, when $E = 1$ these equations reduce to those of an up-counter, and when $E = 0$ the equations reduce to those of a down-counter. The implementation of a four-bit, up-down counter is shown in Figure 10.5.

FIGURE 10.5 **Four-Bit Binary Up-Down Counter.**



As pointed out earlier, a counter can be designed to count in a nonbinary manner as well. Two examples of nonbinary counters are a BCD decade counter and a Gray code counter. In the former, the counter counts 0000 through 1001 and then resets back to 0000. A four-bit Gray code counter, on the other hand, counts 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, and 1000 in that order and then resets to 0000 before resuming count-up operation again. Example 10.1 illustrates the design of a BCD decade counter.

EXAMPLE 10.1

Obtain the *J* and *K* equations for a BCD up-counter.

SOLUTION

The design steps, followed in the usual sequence, consist of the state diagram (Figure 10.6), the state and transition table (Figure 10.7), and excita-

FIGURE 10.6

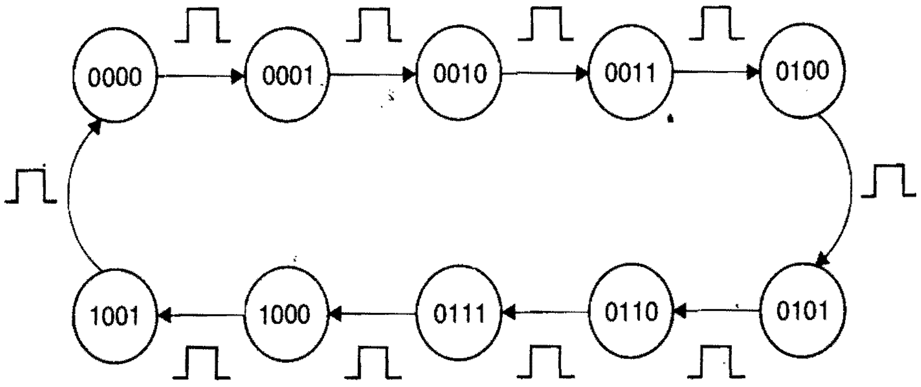


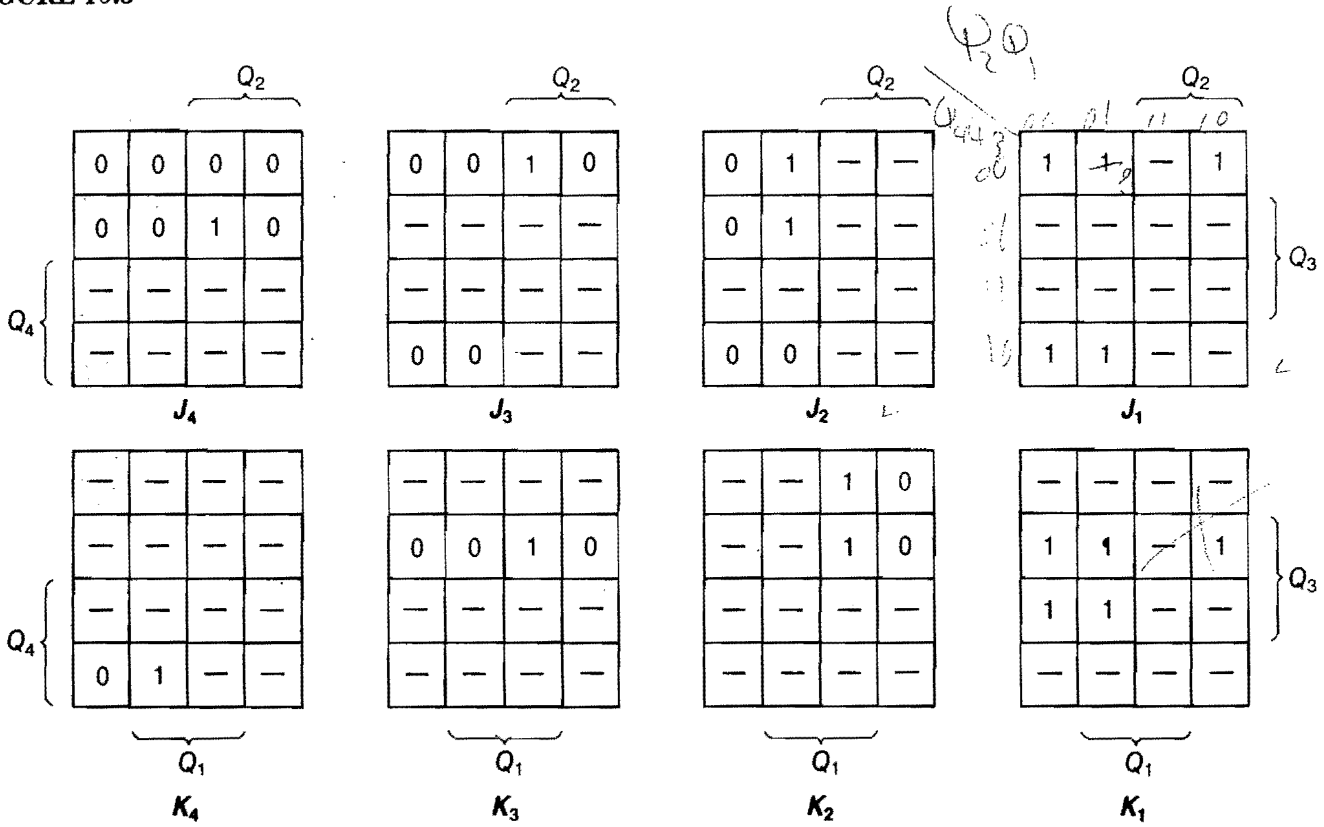
FIGURE 10.7

PS					
$Q_4Q_3Q_2Q_1$	NS	J_4K_4	J_3K_3	J_2K_2	J_1K_1
0000	0001	0—	0—	0—	1—
0001	0010	0—	0—	1—	—1
0010	0011	0—	0—	—0	1—
0011	0100	0—	1—	—1	—1
0100	0101	0—	—0	0—	1—
0101	0110	0—	—0	1—	—1
0110	0111	0—	—0	—0	1—
0111	1000	1—	—1	—1	—1
1000	1001	—0	0—	0—	1—
1001	0000	—1	0—	0—	—1

tion maps (Figure 10.8). The resulting J and K equations are obtained from Figure 10.8 as follows:

$$J_1 = K_1 = 1$$
$$J_2 = Q_1\overline{Q}_4$$
$$K_2 = Q_1$$
$$J_3 = K_3 = Q_1Q_2$$
$$J_4 = Q_1Q_2Q_3$$
$$K_4 = Q_1$$

FIGURE 10.8



J_4

0	0	0	0
0	0	1	0
—	—	—	—
—	—	—	—

J_3

0	0	1	0
—	—	—	—
—	—	—	—
0	0	—	—

J_2

0	1	—	—
0	1	—	—
—	—	—	—
0	0	—	—

J_1

1	—	—	1
—	—	—	—
—	—	—	—
1	1	—	—

K_4

—	—	—	—
—	—	—	—
—	—	—	—
0	1	—	—

K_3

—	—	—	—
0	0	1	0
—	—	—	—
—	—	—	—

K_2

—	—	1	0
—	—	1	0
—	—	—	—
—	—	—	—

K_1

—	—	—	—
1	1	—	1
1	1	—	—
—	—	—	—

Steps similar to those in Example 10.1 can be used to determine the circuit of a BCD down-counter. Such an exercise gives the J and K equations for a BCD down-counter as follows:

$$J_1 = K_1 = 1$$
$$J_2 = \bar{Q}_1(Q_3 + Q_4)$$
$$K_2 = \bar{Q}_1$$
$$J_3 = \bar{Q}_1 Q_4$$
$$K_3 = \bar{Q}_1 \bar{Q}_2$$
$$J_4 = K_4 = \bar{Q}_1 \bar{Q}_2 \bar{Q}_3$$

The design of the BCD counter considered in Example 10.1 is a classical one. It is not necessary for a designer to go through all of these cumbersome steps if an already working circuit can be modified to suit the requirement of the new design. For example, consider a four-bit binary up-counter, also known as the *modulo-16* (or divide-by-16) counter. A synchronous BCD counter works exactly as the already designed four-bit, modulo-16 counter until state 9 ($Q_4Q_3Q_2Q_1 = 1001$) is reached. The four-bit counter advances to 1010, while the BCD counter starts all over at 0000. Assuming that a four-bit counter is to be modified to a BCD counter, it is necessary to reset FFs when they are about ready to go to 1010 otherwise. The modification process requires circuit changes to satisfy the following conditions when the four-bit up-counter reaches count 1001:

Q_1, the LSB, should become a 0,

Q_4, the MSB, should become a 0,

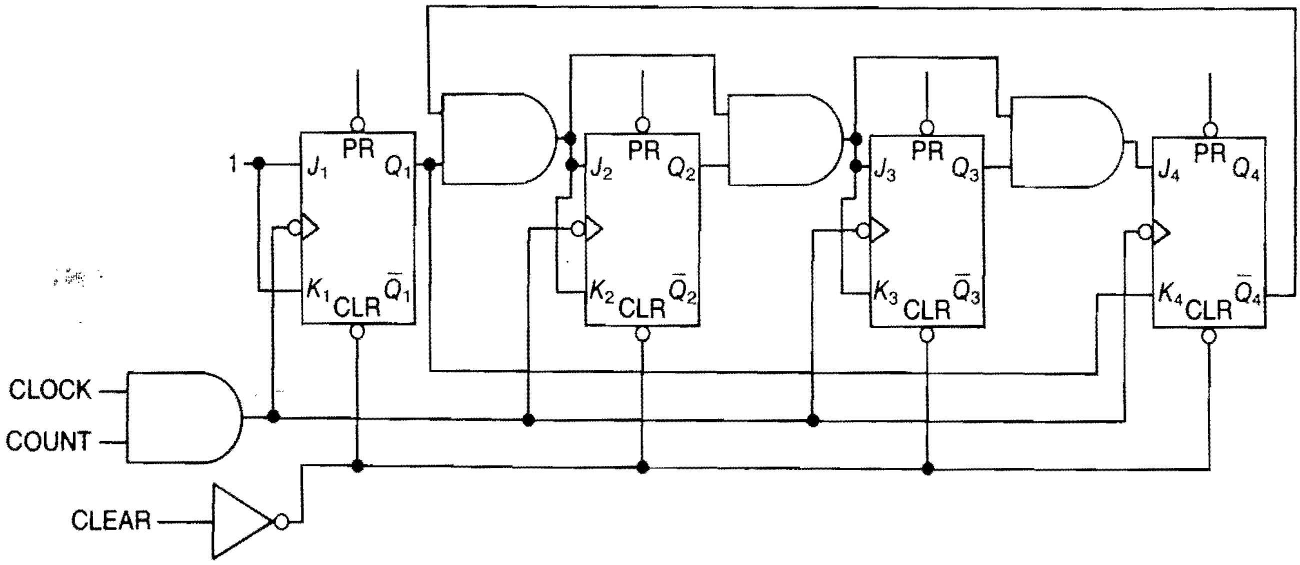
Q_2 should be prevented from becoming a 1.

An inspection of the circuit of Figure 10.4 reveals that the least significant FF is always in a toggle mode, which implies that Q_1 will change to a 0 by itself. Note also that the MSB, Q_4, of the four-bit counter changes only when the count reaches either 0111 or 1111. This change happens because the inputs J_4 and K_4 are both held to a 0 during the other counts. Consequently, it is necessary to supply K_4 with a 1 instead of a 0 when the count reaches 1001. This can be accomplished if Q_1 is fed directly to the K_4 input and K_4 is disconnected from the J_4 input. The next consideration is to prevent Q_2 from switching back to 1 once the count reaches 1001. This action is accomplished by supplying J_2 and K_2 with the ANDed output of Q_1 and $\bar{Q}_4$. This modification does not cause any problem because $\bar{Q}_4$ is a 1 until the count reaches 1000. And once the count reaches 1000, Q_2 need not be turned on in a BCD counter. In summary,

Q_1 should be fed to K_4,
K_4 and J_4 should not be connected,
$Q_1\bar{Q}_4$ should be fed to J_2 and K_2.

Consequently, the BCD counter may be obtained by modifying the modulo-16 counter circuit, as shown in Figure 10.9.

FIGURE 10.9 **Synchronous BCD Up-Counter.**



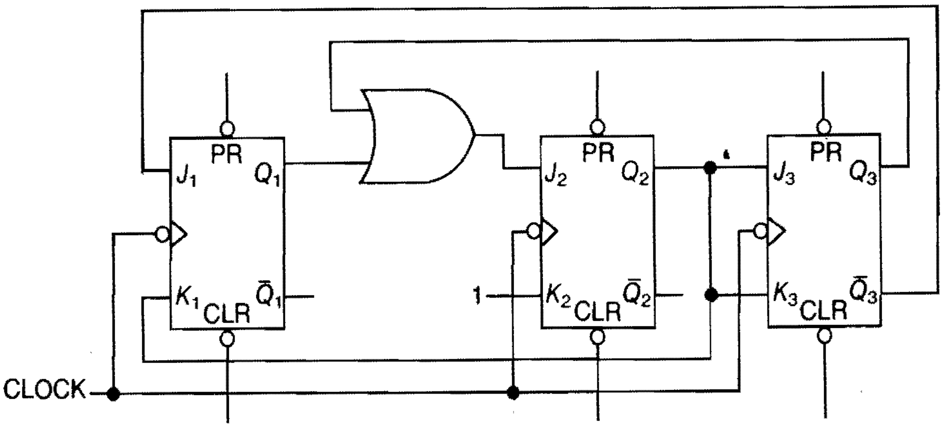
Synchronous circuits requiring irregular count sequences, such as the Gray counter mentioned earlier, are usually designed using the familiar step-by-step technique. A count sequence is said to be *irregular* if it is not magnitude ordered. In general, it is harder to come up with an irregular counter by modifying a regular counter. Examples 10.2 and 10.3 relate to counters with irregular count sequences.

EXAMPLE 10.2

Obtain the state table for the counter shown in Figure 10.10, starting from the count 000. The MSB of the count is at Q_3.

SOLUTION

FIGURE 10.10



This circuit can be analyzed by assuming a present state and using our knowledge of *JK* FF operation. Corresponding to each of the present states, the next state is found by determining the corresponding *J* and *K* values of each FF. When $JK = 00$, the corresponding Q remains unchanged; when $JK = 01$, Q is reset; when $JK = 10$, Q is set; and when $JK = 11$, Q toggles. The resulting state table is obtained as shown in Figure 10.11. The circuit has an irregular state sequence 0, 1, 3, 4, 6, and repeat. Consequently this circuit is an irregular counter.

FIGURE 10.11

PS	J_3K_3	J_2K_2	J_1K_1	NS
000	00	01	10	001
001	00	11	10	011
011	11	11	11	100
100	00	11	00	110
110	11	11	01	000

EXAMPLE 10.3

Obtain a synchronous counter that produces the count sequence 0, 2, 4, 3, 6, 7, 0, ...

SOLUTION

The state table and the *JK* excitations are obtained as shown in Figure 10.12. The easiest way to solve this problem would be to consider the present and corresponding next states. The *JK* excitations necessary for the corresponding transitions are then determined for each of the cases. Note that states 1 and 5 don't occur in the count sequence. Consequently, we assume the corresponding next states and the *JK* excitations to be don't-cares. This assumption should help in the minimization step. The *J* and *K* Boolean equations are obtained using K-maps. They are as follows:

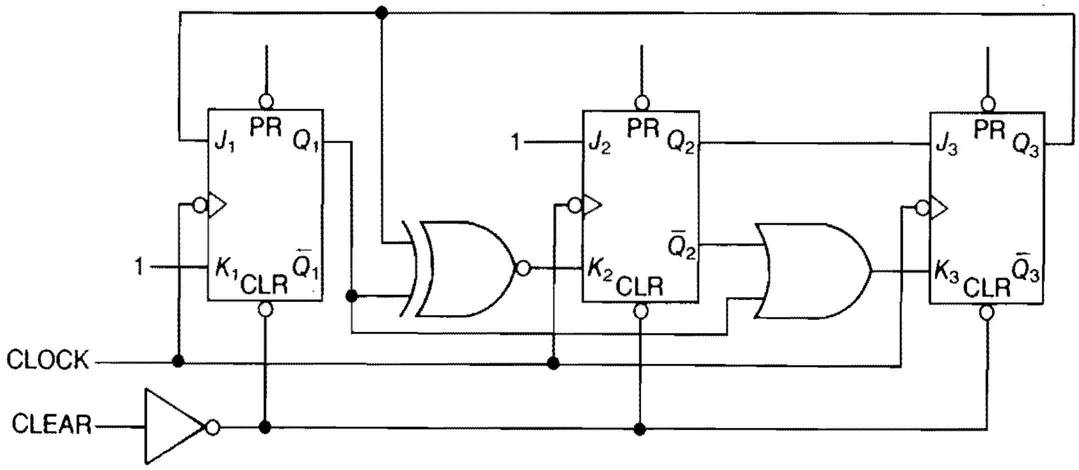
FIGURE 10.12

PS	NS			
$Q_3Q_2Q_1$	$Q_3Q_2Q_1$	J_3K_3	J_2K_2	J_1K_1
000	010	0—	1—	0—
001	——	——	——	——
010	100	1—	—1	0—
011	110	1—	—0	—1
100	011	—1	1—	1—
101	———	——	——	——
110	111	—0	—0	1—
111	000	—1	—1	—1

$$J_1 = Q_3$$
$$K_1 = 1$$
$$J_2 = 1$$
$$K_2 = \overline{Q_1 \oplus Q_3}$$
$$J_3 = Q_2$$
$$K_3 = Q_1 + \bar{Q}_2$$

The resulting counter is obtained as shown in Figure 10.13

FIGURE 10.13



It is appropriate at this point to offer a word of caution. Since there is are various cascadable multi-mode binary and nonbinary counters available, a designer might be tempted to use them in his or her design as a starting chip. Before doing so, the designer must thoroughly examine the circuit specifications and diagrams. If *JK* FFs are used, the familiar 1's-catching problem (see Section 6.5 for details) may crop up here. More specifically, if the clock input is not at the proper level, erroneous operation might result if a particular mode is changed. There may be times, therefore, when a counter may have to be designed from scratch.

10.3 Asynchronous Binary Counters

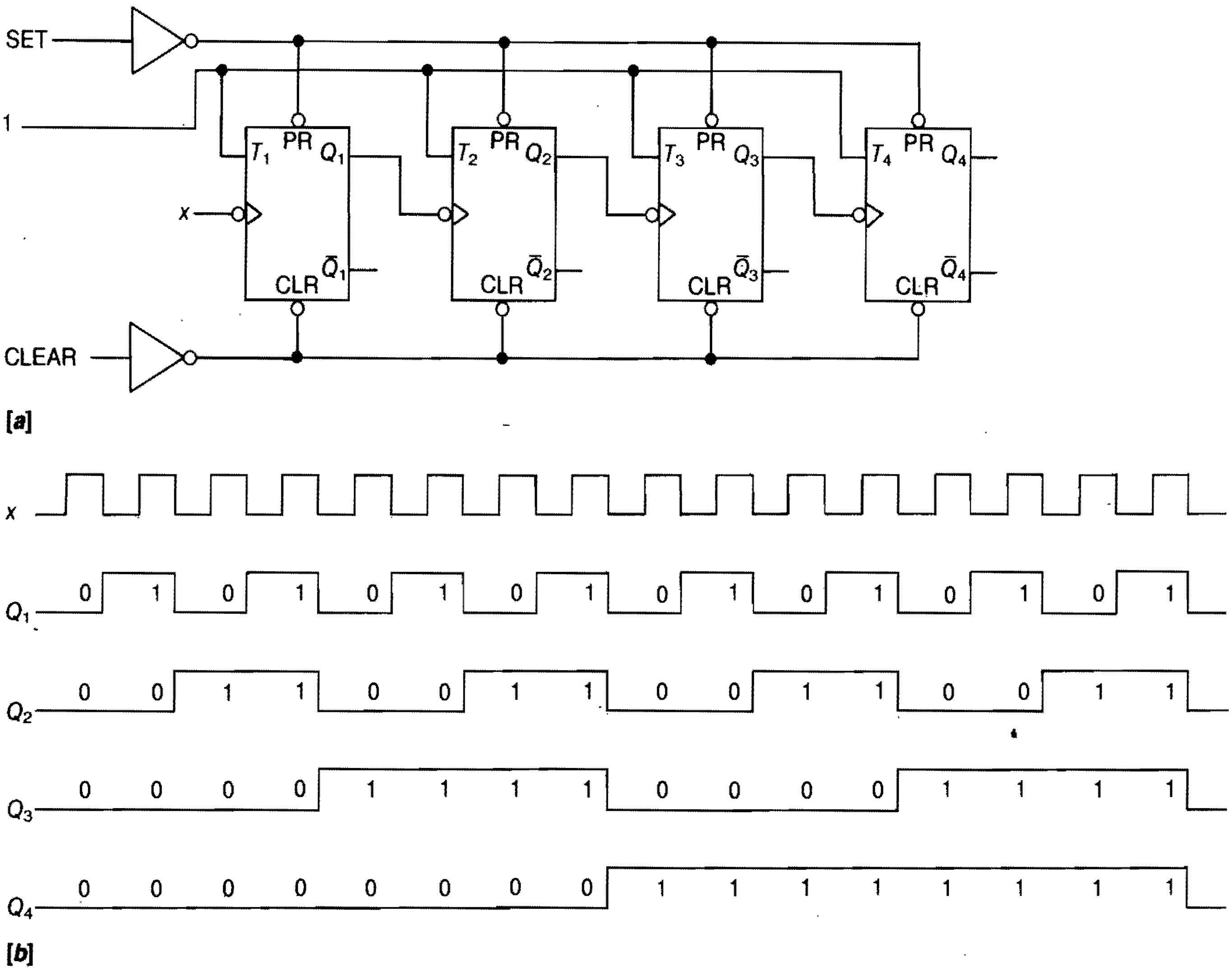
All of the counters considered thus far employed synchronous circuits, that is, the FF actions were synchronized with the clock pulse. The advantage of a synchronous counter is that all bits of a count change simultaneously except for slight differences in FF delays. If a specific count is being decoded, all bits are available at the same time, eliminating momentary errors at the decode output. We shall now introduce asynchronous counters, also known as ripple counters. The FF clock inputs in a ripple counter are not tied together.

In fact, the clock inputs are cascaded from output to input (almost like the rippling of carries in a ripple adder). They are used to reduce the amount of control logic required to construct a binary counting sequence. Asynchronous counters have limitations but also provide less expensive counter options for those cases where their limitations will not affect the circuit. The AND gates between FFs in the synchronous binary counter design may be eliminated by observing the counter state table. The LSB needs to be changed in every present-state to next-state transition. In all bit locations, Q_i changes each time Q_{i-1} makes a transition from a 1 to a 0.

A negative edge-triggered *JK* FF in a toggle mode changes state each time the signal connected to the clock input makes a 1 → 0 transition. An asynchronous counter using *T*-configured *JK* FFs has its least significant FF activated by the system clock. The 1 → 0 transitions of that FF may be used as the trigger (clock input) signal for the next most significant FF. This process of triggering is continued for as many bits as desired.

The logic circuit of a four-bit ripple counter is shown in Figure 10.14[*a*], where four *T* FFs are cascaded together. The output of

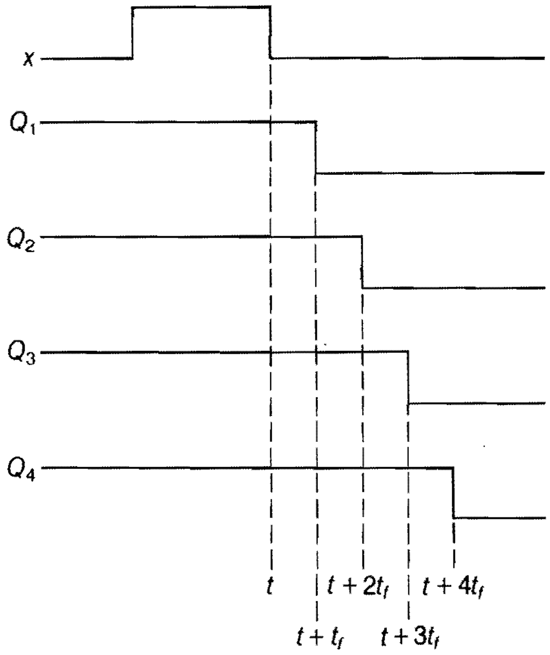
FIGURE 10.14 **Four-Bit Ripple Counter: [*a*] Circuit and [*b*] Timing Diagrams.**


SET
1
CLEAR
X
T_1 PR Q_1 CLR $\bar{Q}_1$
T_2 PR Q_2 CLR $\bar{Q}_2$
T_3 PR Q_3 CLR $\bar{Q}_3$
T_4 PR Q_4 CLR $\bar{Q}_4$
[a]
X
Q_1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Q_2 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Q_3 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
Q_4 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
[b]

each FF provides the clock signal for the next FF. The timing diagram without delays for the four-bit ripple counter is shown in Figure 10.14[*b*]. Examination of the timing diagrams shows that the frequency of the Q_4 pulse is one-sixteenth of the frequency of the input pulse x. Each stage of the counter divides the frequency of the preceding stage by two.

Counters like the one shown in Figure 10.14[*a*] are simple in concept, but have at least two disadvantages: a forced regular binary sequence and speed. The first disadvantage is not so much of a serious handicap, but the speed is. In reality, the rippling effect through the FFs causes delay between each count that is proportional to the number of FFs in the counting chain. Consider the timing diagram of Figure 10.15 that shows the situation existing in the counter when the count is 1111. Q_1 does not change to 0 coincident with the trailing edge of the sixteenth x-pulse until time t_f, the propagation delay of each FF. The same is generally true for the synchronous counter, but for the asynchronous counter Q_2, Q_3, and Q_4 change at times $2t_f$, $3t_f$, and $4t_f$, respectively, beyond the negative edge of the sixteenth x-pulse. For an n-bit ripple counter to reach a valid count before the next clock pulse, $T > nt_f$, where T is the period of the input pulse. If the final count is all that is of interest, the condition $T > t_f$ is all that must be met. In this situation, changes in the LSB are constantly rippling to higher-order bits. After the last pulse is input, it will be nt_f before the final count can be correctly read. Note that in Figure 10.15 the counter does not go through the transition 1111 → 0000. Instead the counter passes through the state transition sequence 1111 → 1110 → 1100 → 1000 → 0000. These transitions occur in rapid succession but they result in undesired transient conditions that might cause further problems

FIGURE 10.15 **Timing Consequences of a Four-Bit Asynchronous Counter During 1111 → 0000 Transition.**



in a circuit that is being driven by such a counter. This must be borne in mind when such counters are used in generating a controller of a complex system.

If the circuit of Figure 10.14[*a*] is modified so that $\bar{Q}_1$, $\bar{Q}_2$, and $\bar{Q}_3$, are used respectively as the clock inputs for the second, third, and fourth *T* FFs, then the circuit will operate as a four-bit ripple down-counter. The ripple counter of Figure 10.14[*a*] may also be modified to generate a resettable counter. Say, for example, we are interested in generating only BCD counts. Such a counter is shown in Figure 10.16. The circuit operates as a modulo-16 ripple counter, but when the state 1010 is reached the counter resets immediately. This resetting is accomplished by means of a combinational decoding scheme that is tied with the reset inputs of every FF. The counter stays at the count of 1010 for the decoding gate delay plus the reset input-to-output delay.

The ripple counter of Figure 10.16 exhibits other transient

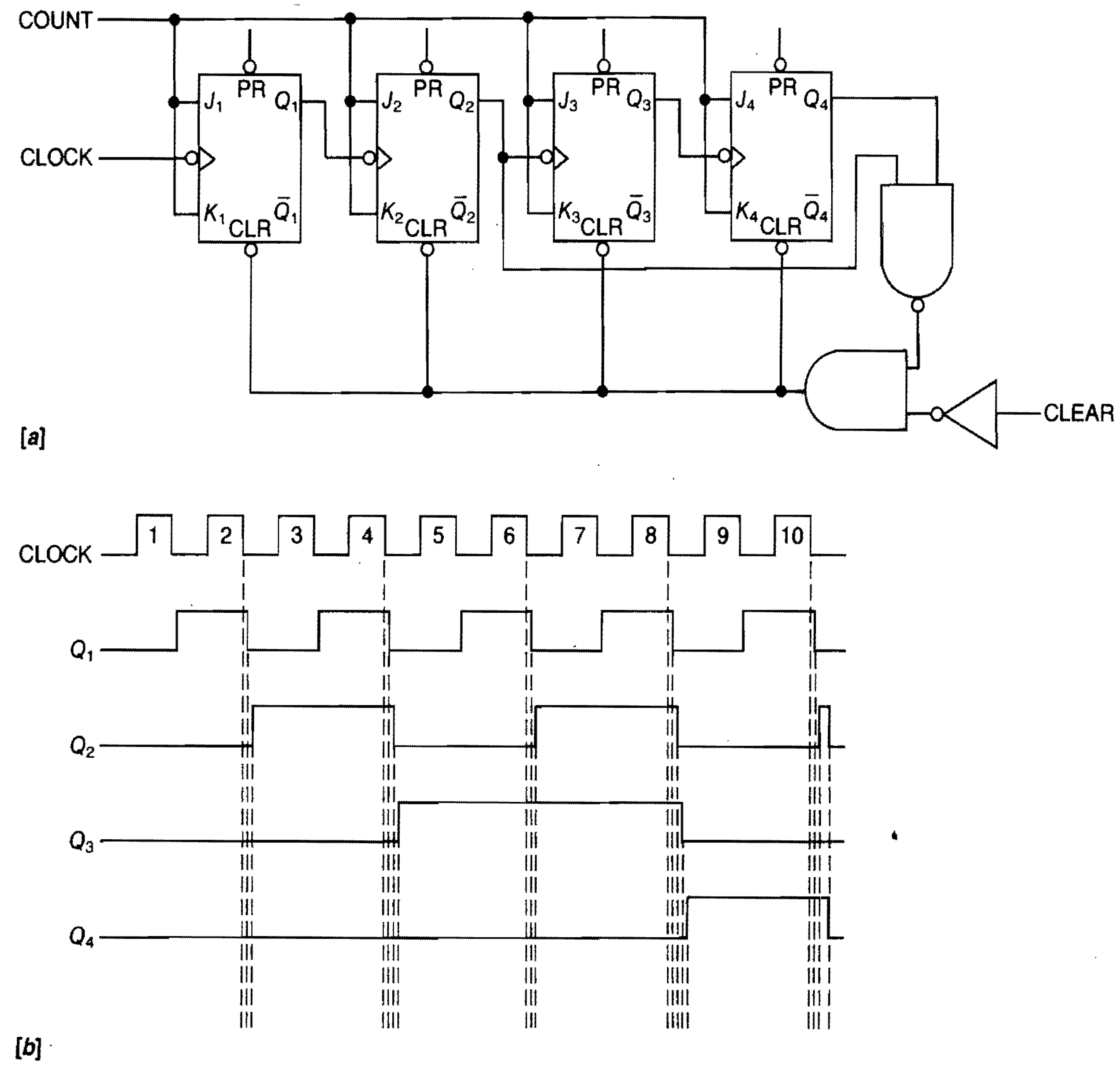


FIGURE 10.16 **Asynchronous BCD Up-Counter: [*a*] Circuit and [*b*] Timing Diagram.**

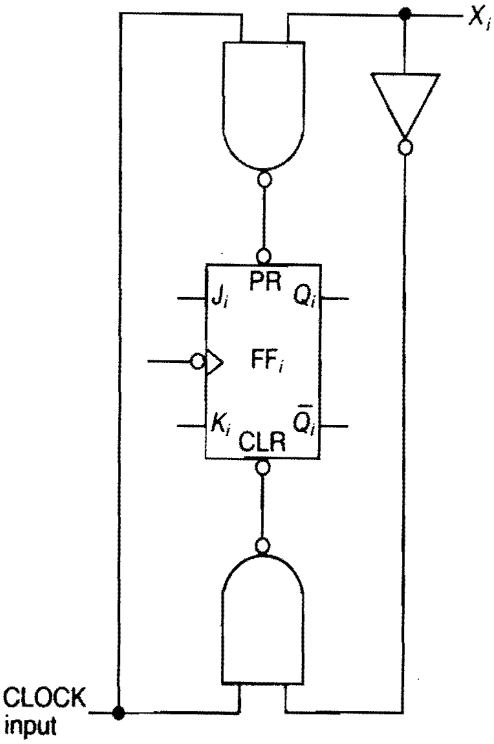
behavior that needs to be closely examined. The transient behavior of the counter is listed as follows:

0001 → 0010 (ideal), 0001 → 0000 → 0010 (actual),
0011 → 0100 (ideal), 0011 → 0010 → 0000 → 0100 (actual),
0101 → 0110 (ideal), 0101 → 0100 → 0110 (actual),
0111 → 1000 (ideal), 0111 → 0110 → 0100 → 0000 → 1000 (actual),
1001 → 0000 (ideal), 1001 → 1000 → 1010 → 0000 (actual).

The worst-case transient occurs during the transition 0111 → 1000. Since an intermediate count is to be decoded, the clock period must be longer than the 0111 → 1000 propagation delay.

Another alternative to the BCD counter design involves feeding the clock input of a modulo-5 counter with the output of a single T FF. The combination of a modulo-2 and modulo-5 counter results in a modulo-10 counter. Such cascading of one counter with another should be pursued whenever possible. Some of the counter designs that we have considered thus far have demonstrated how to reset a counter once a specific count has been reached. There are cases where a different approach may be necessary. It is always possible to preset a counter to a specific count by means of a *jam-entry* scheme, as shown in Figure 10.17. The bit that is to replace the old value at the Q_i location is fed directly to the corresponding entry point, X_i. The new bit is transferred to the FF output when the clock pulse occurs. The counter will begin counting from the preset count

FIGURE 10.17 **Jam-Entry Scheme for Presetting the *i*th FF.**



as further clock pulses arrive at the input. This is possible only in counters made up of independent FFs.

Because of the problems of asynchronous counters, they should be used with a certain degree of caution. The designer must be aware of their limitations. The next section introduces the characteristics of IC counters. In a subsequent section (Section 10.9), more counter configurations will be introduced that exhibit several advantages over the synchronous and asynchronous counters.

10.4 IC Counters

In the first three sections of this chapter we have used both classical and heuristic design techiques to design counters. Similar multi-bit counters are available in IC form. A typical four-bit binary counter module with inputs and outputs is shown in Figure 10.18. We are already familiar with most of its features. The different inputs are described as follows:

A, B, C, D: These inputs are used for presetting the counter to an initial value. It has been assumed that *A* is the LSB and *D* is the MSB.

Q_A, Q_B, Q_C, Q_D: These are the FF outputs of the counter. Q_A corresponds to the LSB and Q_D corresponds to the MSB.

CARRY-OUT (CO): This output becomes a 1 when the count equals 1111 and the ENABLE control input is a 1. CARRY-OUT is equivalent to $Q_AQ_BQ_CQ_D \cdot$ ENABLE.

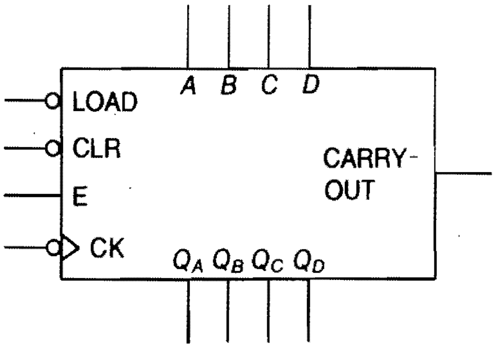
LOAD: This control input is used to load *A, B, C*, and *D* values into the counter. When the LOAD input is a low (0), the values are either loaded immediately if loading is asynchronous or loaded at the next clock pulse in the synchronously operated counter.

CLEAR (CLR): This control input when set to a 0 causes the counter to be cleared. The counter is cleared immediately if it has an asynchronous clear. In a counter with synchronous clear the output changes coincident with the next clock pulse.

ENABLE (E): This input must be high for the counter to count.

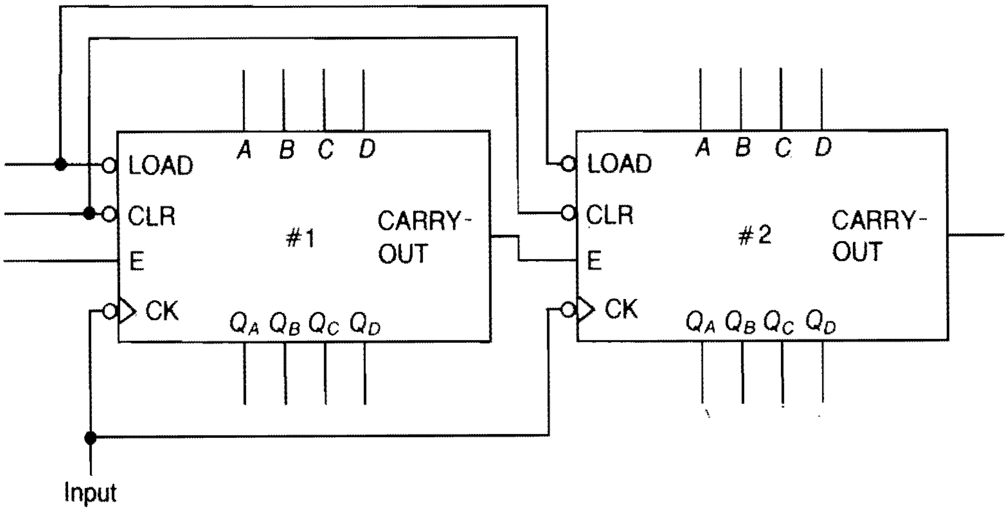
CLOCK (CK): The 1 → 0 transitions of this input are counted by the counter when the ENABLE input is a 1.

FIGURE 10.18 Four-Bit Binary Counter Module.



The four-bit IC counters may be cascaded together, as shown in Figure 10.19, to form counters of any bit length. When counter #1 has a count of 1111 and its ENABLE is a 1, the CARRY-OUT is a 1, which activates the second counter. When the next clock occurs, counter #1 resets to 0000 and counter #2 counts up to 0001 and the CARRY-OUT of counter #1 is reset to a 0. The eight-bit counter output then becomes 0001 0000. During the next 15 pulses, counter #2 would remain disabled and counter #1 would keep on counting upward. Again when counter #1 reaches count 1111, counter #2 counts up to 0010. And this process continues as more inputs appear at counter #1. Each time the setup of Figure 10.19 receives a total of 16 clock pulses, the #2 counter counts up by 1.

FIGURE 10.19 **Eight-Bit IC Counter.**



The LOAD and CLR controls can be manipulated in many different ways to obtain many count variations. Example 10.4 illustrates such manipulations of a binary IC counter mode.

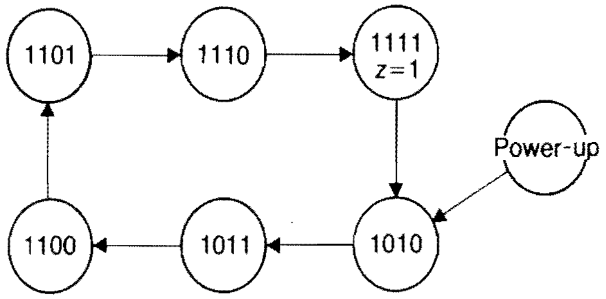
EXAMPLE 10.4

Design a counter using the module of Figure 10.18 that outputs a 1 each time six counts have been received.

SOLUTION

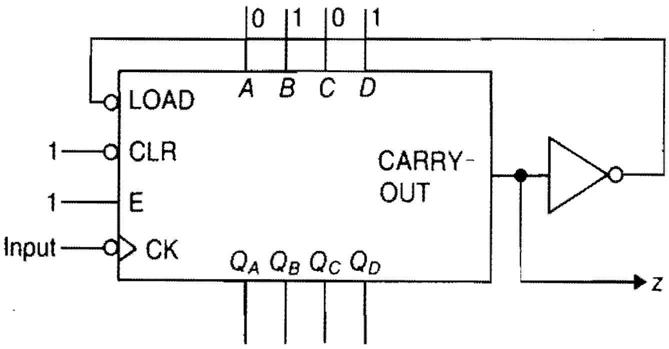
One of the ways to accomplish this design is to make use of the CARRY-OUT output, which is 1 when the count reaches 1111. We can load the counter with an initial value that will cause an output to occur five pulses later. The CARRY-OUT is then used to reload the initial value. We begin from 1010, and when the counter reaches 1111 the CARRY-OUT would give a 1. Note that the first output upon turning the power on may not even be 1010. The CARRY-OUT either may become 1 before six pulses have occurred or may require up to 15 pulses. The number of pulses is dependent on what value the counter assumes upon power-up. The state diagram of the required sequence is shown in Figure 10.20.

FIGURE 10.20



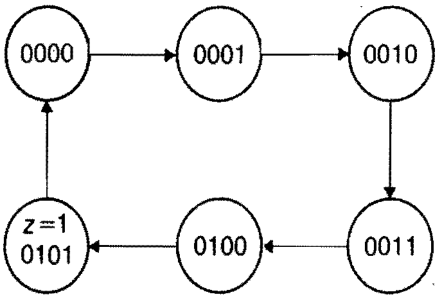
Successive outputs should occur after every six pulses. The circuit configuration using an IC counter with a synchronous load is obtained as shown in Figure 10.21. When the counter reaches 1111 the CARRY-OUT becomes a 1, causing a low at the LOAD input. This low forces the counter to begin again from the 1010 state.

FIGURE 10.21



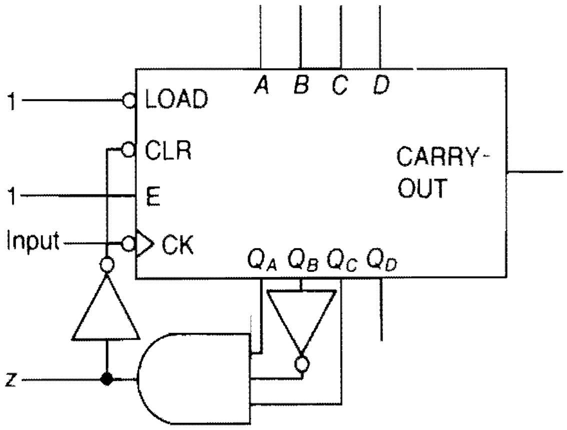
An alternate solution to this problem may be obtained by making use of the CLR input. In this case we begin from 0000, and once the count reaches 0101 an output is made to occur, and at the same time the counter is reset. A count of 0101 is decoded using external logic, and the counter is then made to reset using the synchronous CLR. The first pulse may require as many as seven pulses or none, depending on what the counter state is after power-up. In the worst case the beginning state could be 0110 and the counter would be reset once the count reaches 1101. The state diagram and the corresponding circuit configuration are obtained as shown in Figure 10.22. Note that the output z will be a 1 whenever the count is 0101 or 1101. The count will be 1101 only if the counter shows a count larger than 0101 at power-up.

FIGURE 10.22[a]



[a]

FIGURE 10.22[b]



[b]

Given a modulo-*m* counter to count *n* pulses, where $n \leq m$, and then to restart the count, it is thus advisable to follow either of the following schemes:

a. Use the CARRY-OUT and load $m - n$ into the preset inputs. The CARRY-OUT will be 1 every *n* pulses.

b. Decode a count of $n - 1$ and use the output of the decode gate to clear the counter. The decode gate will be active every *n* clock pulses.

10.5 Basic Serial Shift Registers

The shift register is one of the most extensively used functional devices in digital systems. A *shift register* consists of a group of FFs connected so that each FF transfers its bit of information to the adjacent FF coincident with with each clock pulse. In other words, shift registers store bits of information, behaving like temporary memory, and upon external command shift those information bits one position to either right or left, depending on the design of the device.

The action of a right-shift register whose shift-right serial input is tied to a 1 is illustrated in Figure 10.23. The bits shifting out of the right-most FF are lost. With each clock input the bits move one position to the right while a 1 moves in at the MSB. After 11 clock pulses, the data in the register prior to shifting are replaced by a string of 1s. A quite useful application of shift-right registers requires a connection of the right-most FF output to the input of the left-most FF. In that case the LSB is not lost but appears at the MSB. Consequently, after two clock pulses, for example, 00101100101 would be replaced by 01001011001. Such a shift-right register is known as a *circulate-right* register. In the event these same data were stored in a shift-left register and the MSB output was connected to the LSB input, the data would be 10110010100 after two clock pulses. This latter type of register is known as a *circulate-left* register.

FIGURE 10.23 **Shift-Right Register Action.**

After Clock	Bit Pattern
0	00101100101
1	10010110010
2	11001011001
3	11100101100
4	11110010110
5	11111001011
⋮	⋮
10	11111111110
11	11111111111

A typical stage, n, in a multi-bit serial shift-right register can be designed using the design procedures described in the earlier chapters. Figure 10.24 shows the state diagram of the FF concerned. The FF state reflects the current content of that position of the shift register. If the current state of the nth bit of the shift register is a 0 and that of the $(n + 1)$th bit is a 0, the nth bit remains unchanged when the clock occurs. Similarly, Q_n does not change if the present states of both Q_n and Q_{n+1} are 1. For the other cases, Q_n changes and takes the value of Q_{n+1}. Figures 10.23 and 10.24 illustrate the action of serial shift-right registers and make it obvious that the function of each of the FFs is governed by the same next-state equations. This observation reduces the design of a multiple-bit shift register to the design of a single stage. For n bits, n such stages are cascaded.

The circuit action described by the state diagram in Figure 10.24 is that of a D FF. This can be verified by comparing the state diagram of Figure 10.24 with that of Figure 6.40[*c*]. Multi-bit, shift-right registers can be built using edge-triggered D FFs or JK FFs connected as D FFs. Such shift-right registers are shown in Figure 10.25 where SRI is the entry point for *shift-right input* and SRO is the exit point for *shift-right output.* While using D FFs, the Q output of each stage is connected to the D input of the succeeding stage. If JK FFs are used, the Q and $\bar{Q}$ outputs of each stage are connected to

FIGURE 10.24 **State Diagram for the Q_n Bit of a Multi-Bit, Shift-Right Register.**

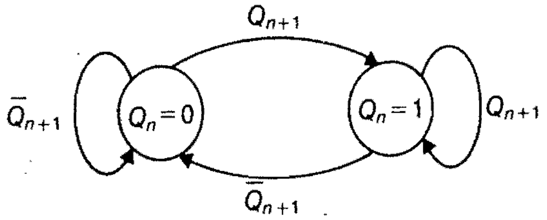
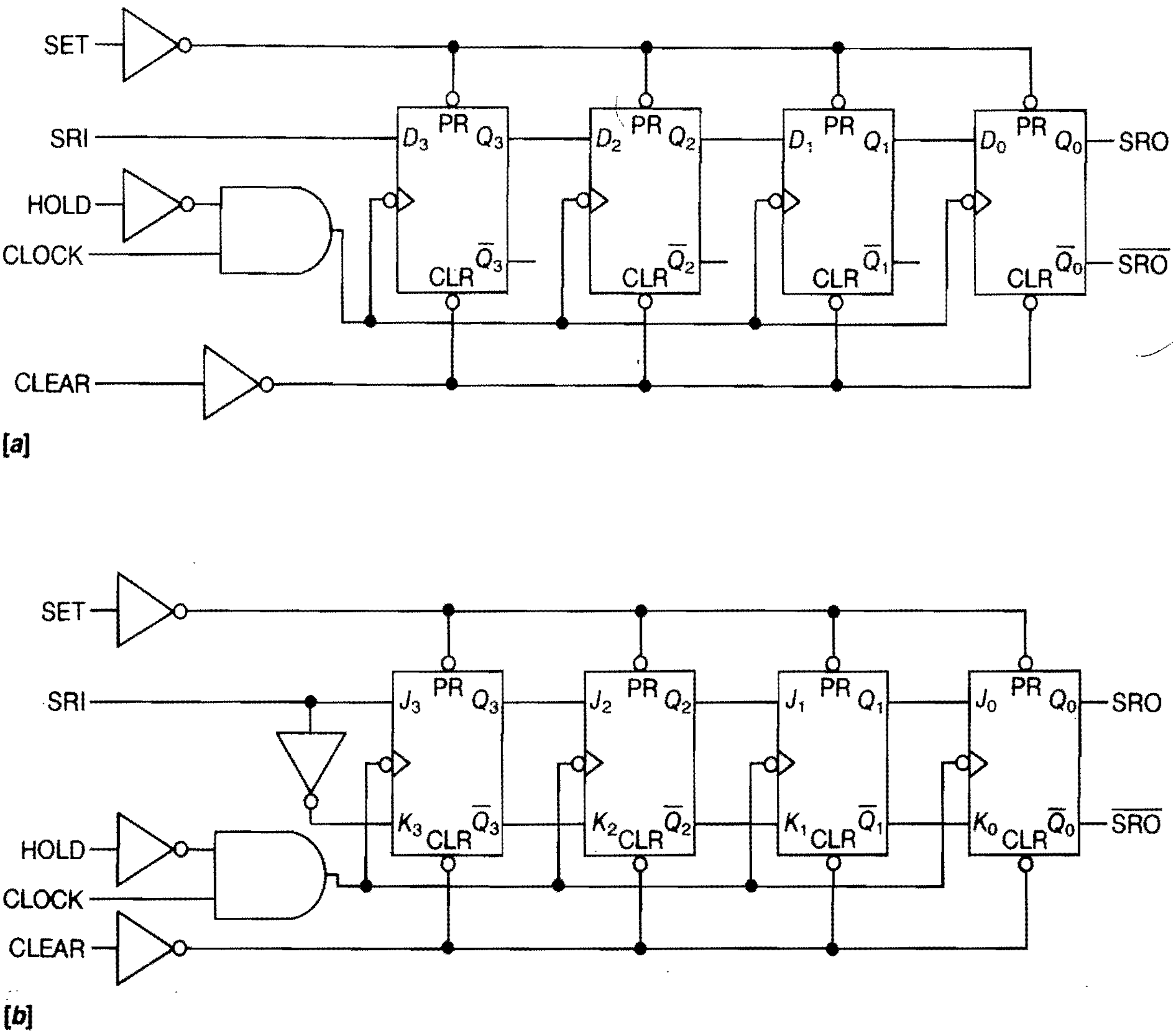


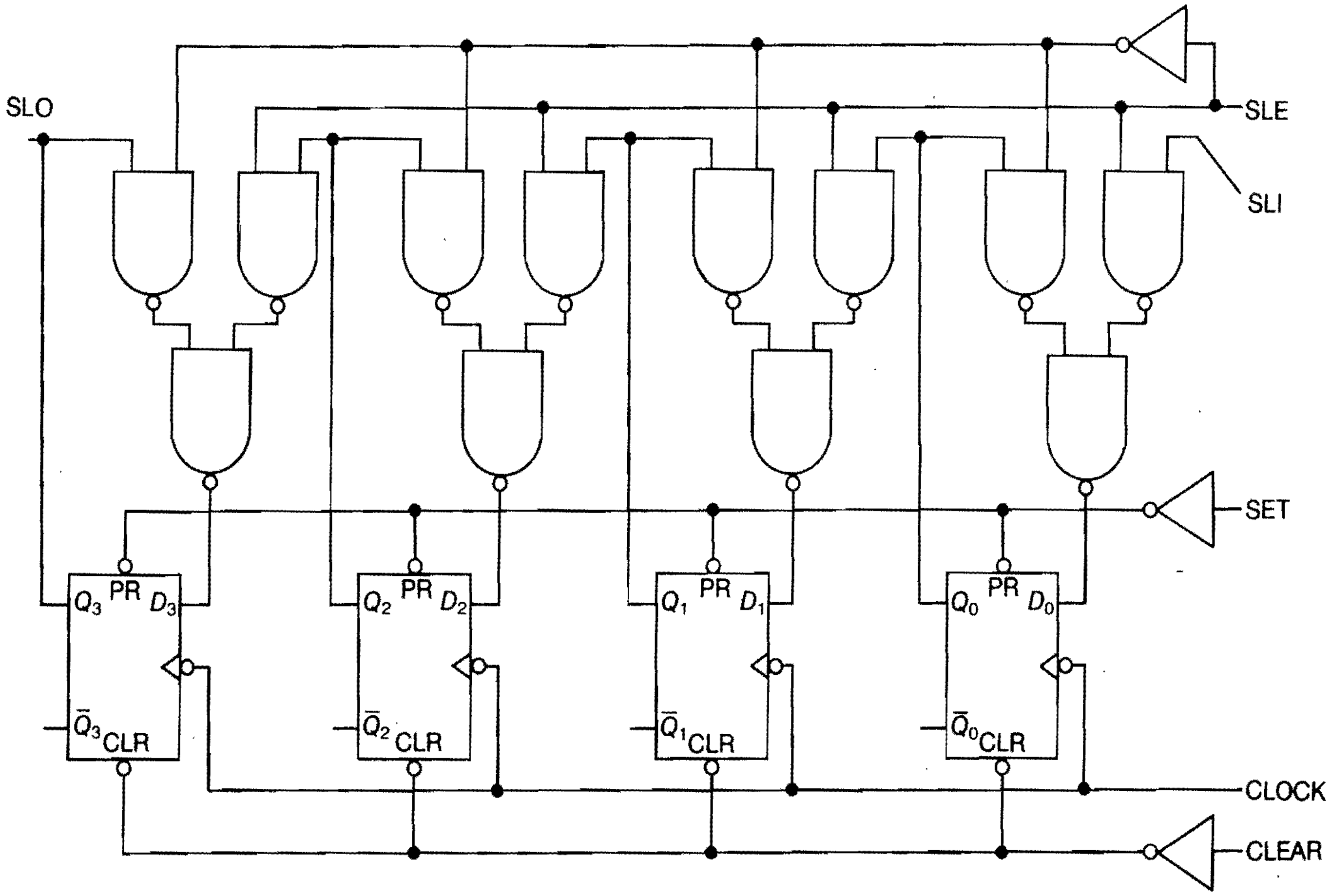
FIGURE 10.25 **Four-Bit, Shift-Right Register: [*a*] Using *D* FFs and [*b*] Using *JK* FFs.**



the J and K inputs of the next stage. The first JK FF is modified to that of a D FF by supplying the data directly to the J input and the complement of data to the corresponding K input. Note that the remaining FFs do not have an *inverter* between their J and K inputs since each of the J inputs is already a complement of the corresponding K input. A high at the CLR input would reset the register FFs, and similarly a high at the SET input would place a 1 at each of the FFs. A high at the HOLD input would disable the FFs and this could be used for storing the bits indefinitely. SET, CLR, and HOLD inputs must be fed with 0 for operating the register in serial mode.

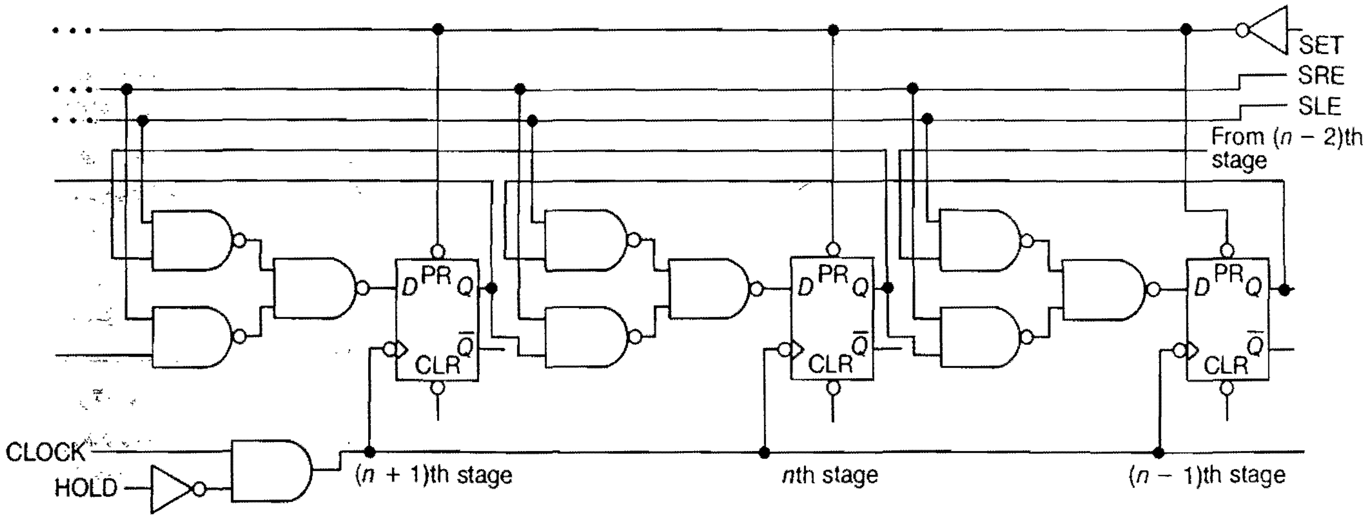
The shift registers of Figure 10.25 are classed as serial-in, serial-out, shift-right registers. Similar design techinques may be used to obtain a shift-left register. In fact, both shift-right and shift-left capabilities may be combined to obtain a bidirectional serial-in, serial-out shift register. A controlled shift-left register is shown in Figure 10.26 that has an additional control input SLE that determines what it does on the next clock pulse. The SLI is the entry point for *shift-left input* and SLO is the exit point for *shift-left output.* When the *shift-left enable*, SLE, is low, the FF output is fed back to its data input. In this way, digital information bits may be restored indefinitely. It is interesting to see how the HOLD input has been eliminated in this case. The clock inputs are still allowed to excite the FFs. However, in the previous case, as shown in Figure 10.25, the clock input was not allowed into the FFs. Again when the SLE control is high, the serial input sets up the right-most FF, Q_0 sets up the second FF, Q_1 sets up the third FF, and so on. With SLE high, the circuit functions as a shift-left register. The serial output is obtained out of the left-most FF.

FIGURE 10.26 **Controlled Shift-Left Register.**



The shift-left and shift-right registers can be combined to obtain the bidirectional shift register of Figure 10.27. This register is similar in design to that of an up-down counter where an up-counter is combined with a down-counter. The direction of shift is controlled by the inputs SLE and SRE. SLE and SRE control inputs cannot be 1 simultaneously, except when HOLD = 1, so they are mutu-

FIGURE 10.27 Bidirectional Shift Register.



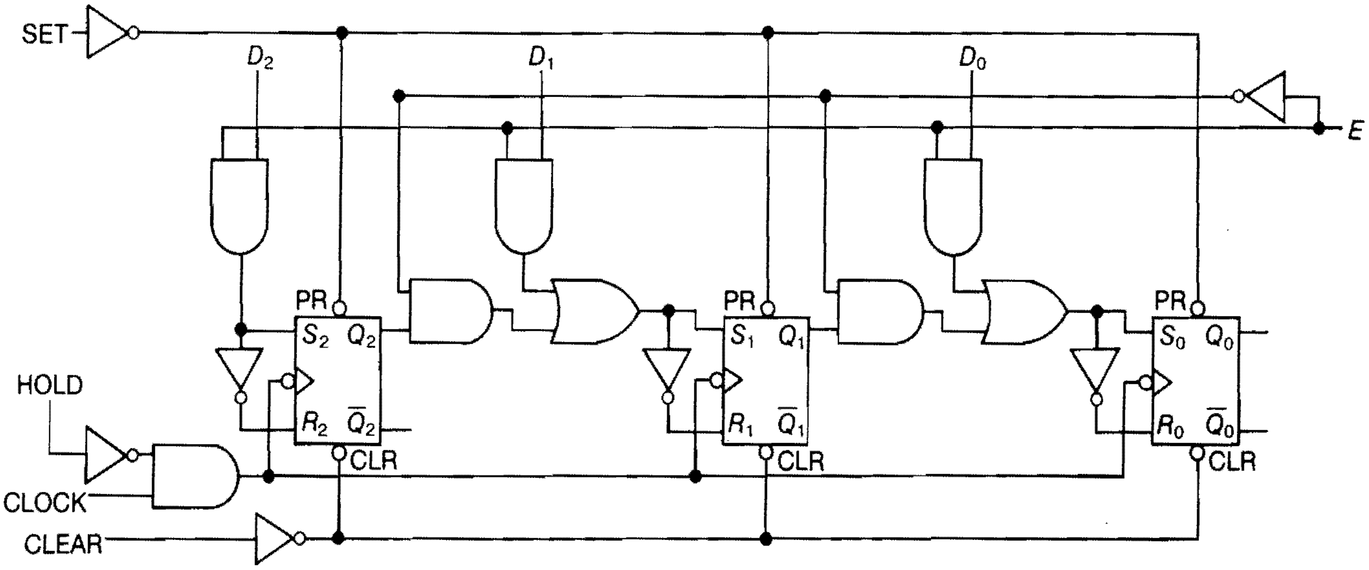
ally exclusive. When the SRE input is high, each of the FFs loads the respective Q output of the FF on its immediate left. When the SLE input is high, each of the FFs loads the Q output of the FF on its immediate right. When both SLE and SRE are 0, the FFs are all reset. Note this time how the CLR control has been eliminated. When HOLD = 0 the register functions in its serial mode, and when HOLD = 1 the old data bits are restored. We might decide to eliminate one of the two shift controls. In that event we may decide to keep SRE only by making sure that SLE has been replaced with the complement of SRE. The register would function as a shift-right type when SRE = 1 and as a shift-left type when SRE = 0. But this arrangement causes a problem if we need to reset the FFs at any time. This problem could be solved, however, by feeding the complement of a CLR control to each of the FF resets.

An important application of shift registers is their role in arithmetic operations. A binary number can be multiplied by 2 by shifting the number one bit to the left and divided by 2 by shifting the register content one bit to the right. As we will see later, the bits shifted in at one end and out at the other end are not unimportant; they are used in arithmetic operations in many instances.

10.6 Parallel-Load Shift Registers

An n-bit, serial-load shift register requires n clock pulses to load an n-bit word. A *parallel-load shift register*, in comparison, loads all information bits simultaneously. Both serial-in and parallel-load shift registers have specific applications in digital systems. A parallel-in, serial-out shift register using master-slave *SR* flip-flops is shown in Figure 10.28. The parallel data are loaded using the jam-entry

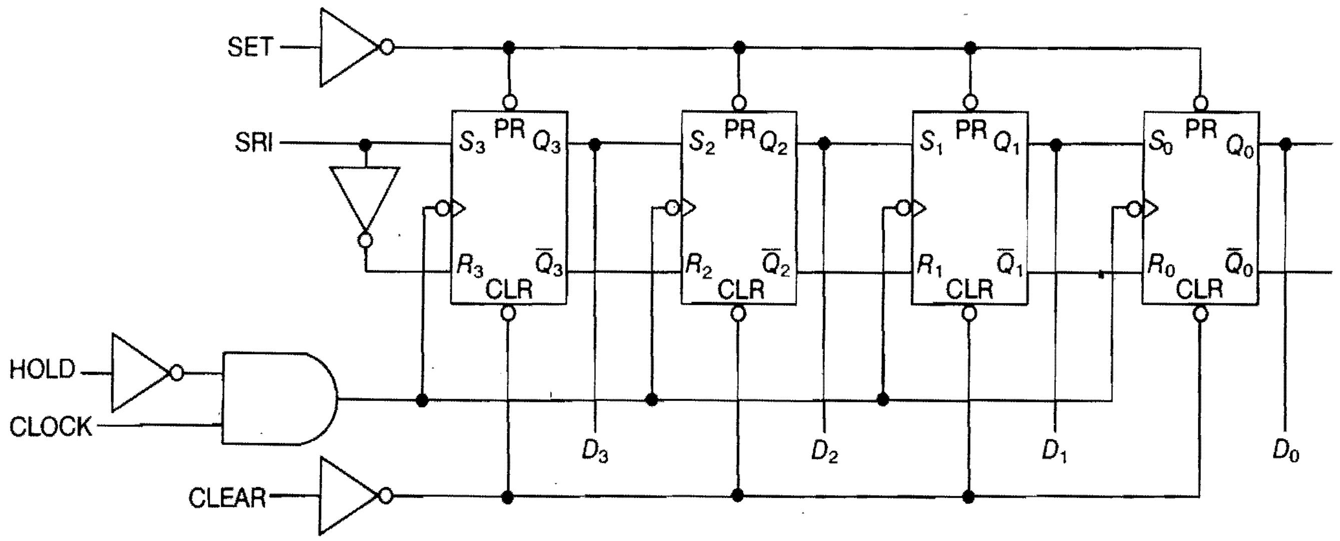
FIGURE 10.28 Three-Bit, Parallel-In, Serial-Out Shift Register.



scheme that was discussed in Section 10.3. When the enable signal E is high, the data are loaded into the register in parallel. Again, if E is low, the Q output of the FF of every stage is shifted to the right by means of the combinational gates. In either case the HOLD control must be held low. Parallel-in, serial-out shift registers allow accepting data n bits at a time on n lines and then sending them one bit after another on one line. This is a standard mode of communication between digital systems.

At the receiving end of two digital systems communicating over a single data line, it is necessary to collect n bits and then transfer them in parallel to the receiving system that is designed to handle n bits simultaneously. Figure 10.29 shows the logic circuit of such a

FIGURE 10.29 Four-Bit, Serial-In, Parallel-Out Shift Register.



serial-in, parallel-out register. The serial data are entered to the S input of the left-most FF while the data are transferred in parallel from the Q outputs. The register is organized in exactly the same way as that of Figure 10.25[b]; however, all of its Q outputs are available, which is not the case for all shift registers. Both parallel loading access features are included in the four-bit, parallel-in, parallel-out register shown in Figure 10.30. The CLR, SET, and HOLD inputs are set low for normal operation. When the LOAD is low, the shift register performs the shift-right operation. When the LOAD is high, the inputs I_3, I_2, I_1, and I_0 are loaded in parallel into the register coincident with the next clock pulse. The outputs O_3, O_2, O_1, and O_0 are available in parallel from the Q output of the FFs.

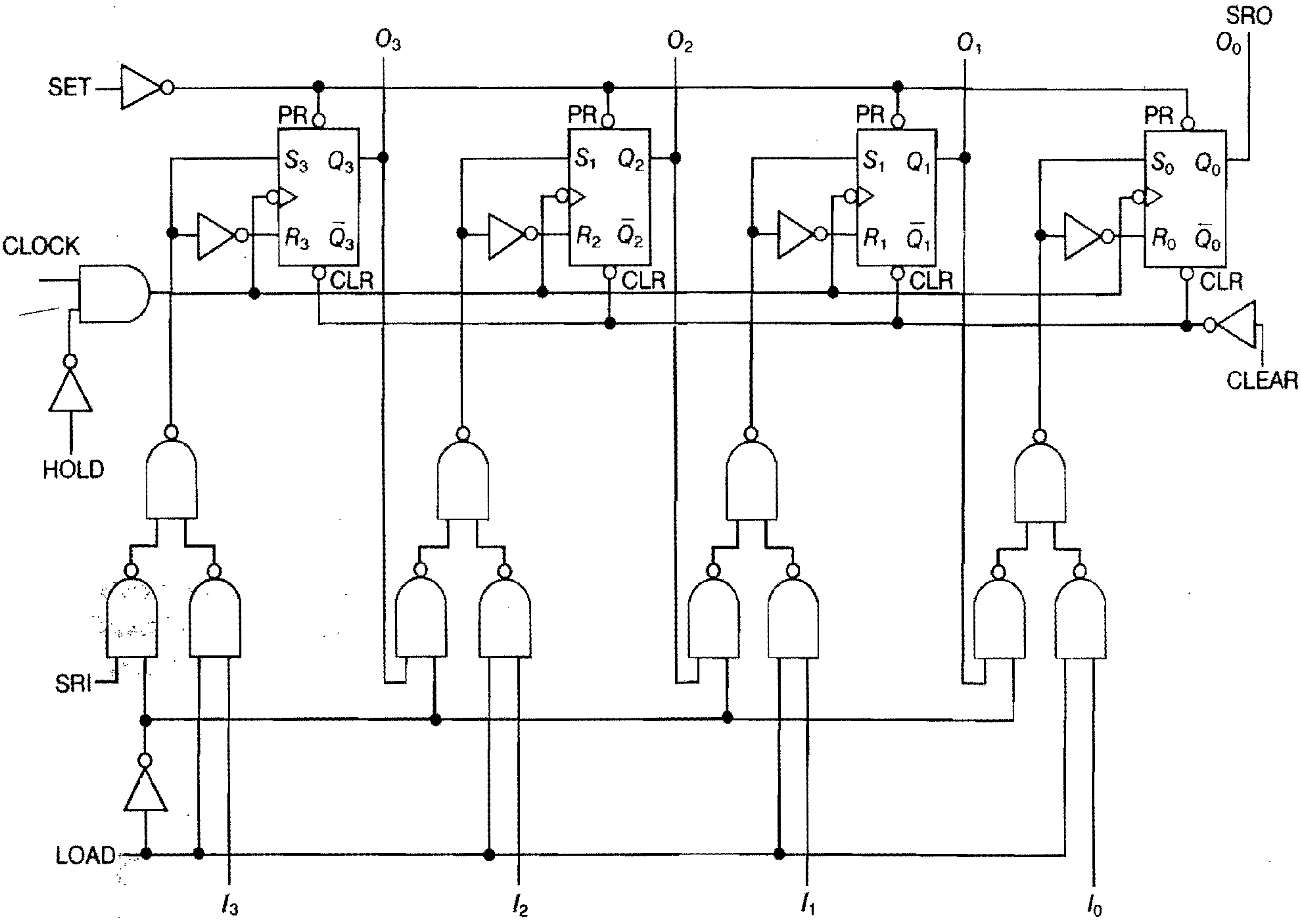


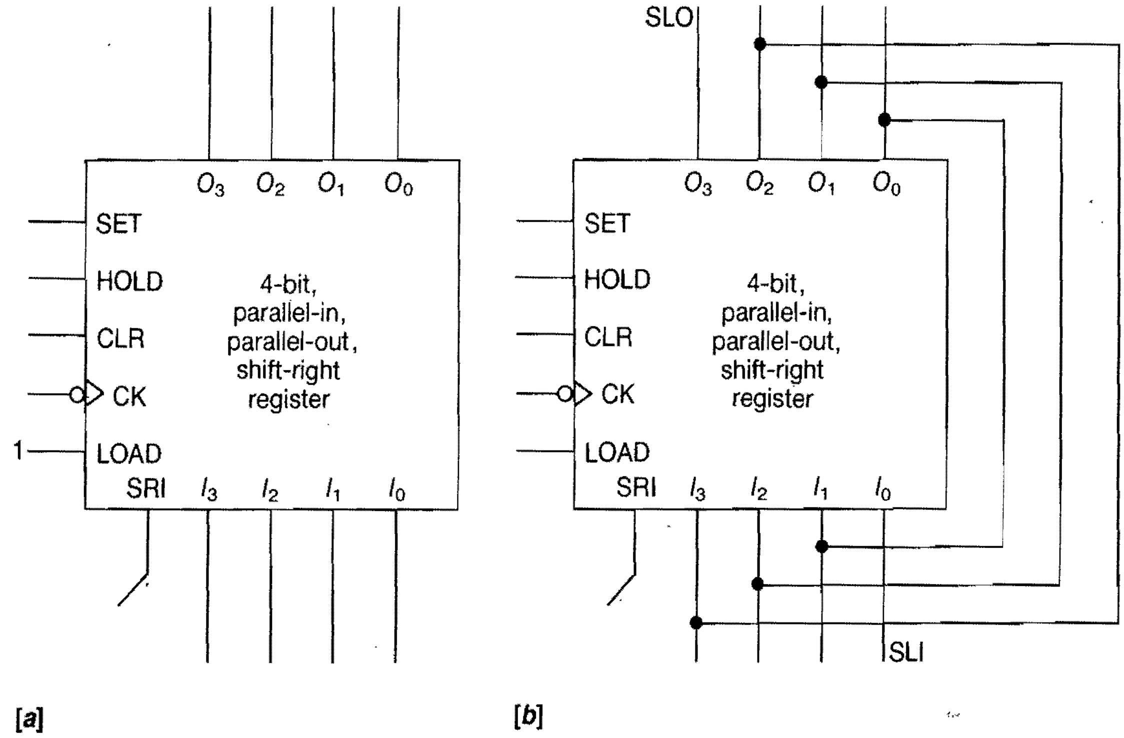
FIGURE 10.30 **Four-Bit, Parallel-In, Parallel-Out Shift Register.**

10.7 Universal-Shift Registers

A *universal-shift register* is a versatile shift register that has capabilities for parallel loading, parallel outputs, bidirectional shifting, and bidirectional serial input and output. In other words, it is capable of operating in all of the register modes described previously. There could be two different ways to realize a universal-shift register: either by modifying a parallel-in, parallel-out shift register or by building one from scratch.

The circuit of Figure 10.31[*a*] shows the logic diagram of a four-bit, parallel-in, parallel-out shift-right register. Its internal circuitry and functions are exactly like those of the circuit of Figure 10.30. It has five entry points for the data inputs–SRI, I_3, I_2, I_1, and I_0–four outputs–O_3 through O_0–and five control inputs–SET, HOLD, CLR, CK, and LOAD. The register is capable of either serial or parallel entry and serial and parallel output. Note that in the circuit of Figure 10.30 the normal serial mode is only shift-right. Such devices are also available commercially. When LOAD is low the register works as a shift-right register, and when LOAD is high it works as a parallel-in register.

FIGURE 10.31 **Block Diagram of Figure 10.30: [*a*] in Shift-Right Mode and [*b*] in Shift-Left Mode.**

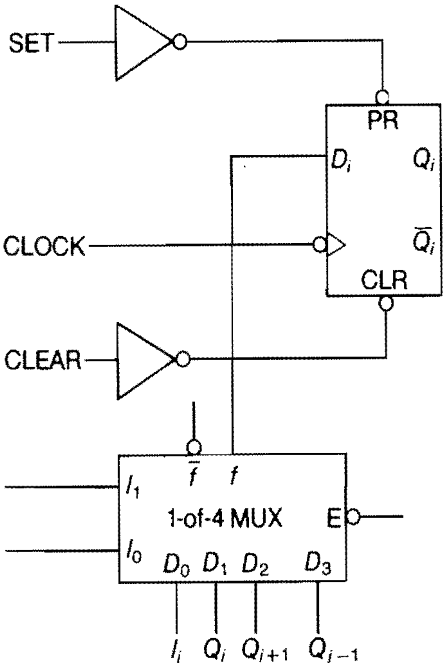


O3 O2 O1 O0
SET
HOLD
CLR
CK
LOAD
SRI
I3 I2 I1 I0
4-bit, parallel-in, parallel-out, shift-right register
1
[a]
SLO
O3 O2 O1 O0
SET
HOLD
CLR
CK
LOAD
SRI
I3 I2 I1 I0
4-bit, parallel-in, parallel-out, shift-right register
SLI
[b]

The circuit of Figure 10.30 can be externally modified to perform a left-shift operation, as shown in Figure 10.31[*b*]. This modification is accomplished by selecting the parallel-mode, LOAD = 1, and by connecting each parallel input to the output of the FF on the immediate right. This form of shifting is called a *wired shift* since the shifting action is accomplished by external connections.

An *n*-bit, universal-shift register can be designed by means of *n* *D* FFs and *n* 1-of-4 multiplexers. The *i*th bit of such a universal register is shown in Figure 10.32. Now all that remains is to cascade *n* such units. When *AB* = 00 the register performs a parallel-in operation, when *AB* = 01 the register restores the old value, when *AB* = 10 the register performs a right-shift operation, and when *AB* = 11 the register performs a left-shift operation.

FIGURE 10.32 *i*th Bit of a Universal-Shift Register.



10.8 Shift Registers as Counters

Shift register ICs are used at times to generate counts or controlled sequences. As a result registers are used extensively in multiple address coding, parity bit generators, and random bit generators. The output of each stage and its complement must be accessible for these applications. These register outputs are used to drive combinational feedback logic, as shown in Figure 10.33. The feedback logic determines the next state of Q_n. In the case of a bidirectional register, the feedback logic controls shift-left and shift-right signals and sets up a 1 or 0 to the appropriate SLI and SRI input.

The state diagram of a four-bit shift register with the J_3 input of the input FF available is shown in Figure 10.34. If the shift register is initially in the state $Q_3Q_2Q_1Q_0$ = 1001, then there are two possi-

FIGURE 10.33 Feedback Shift-Right Register Configuration.

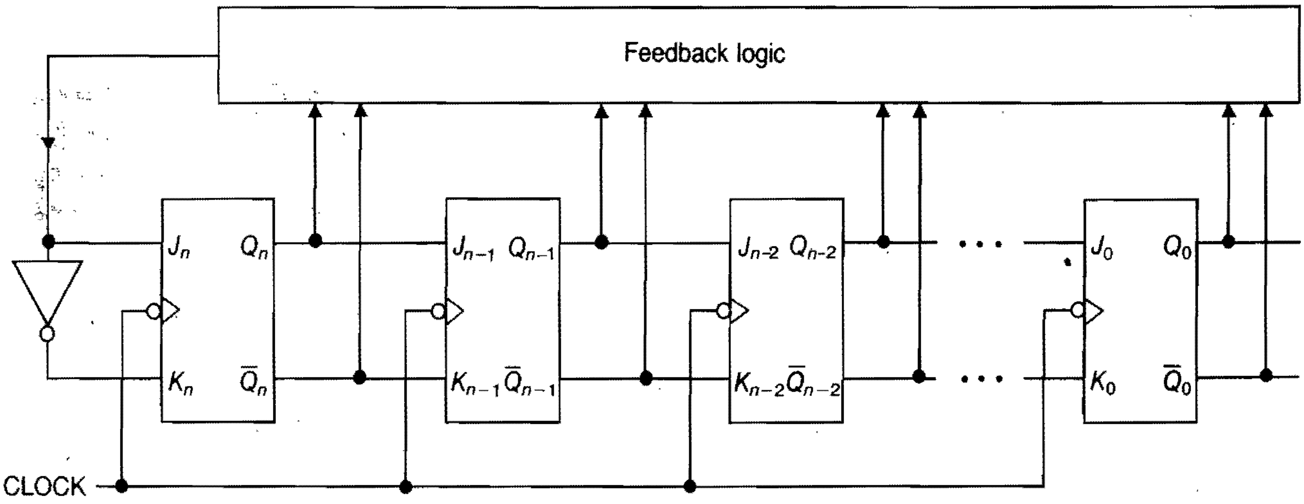
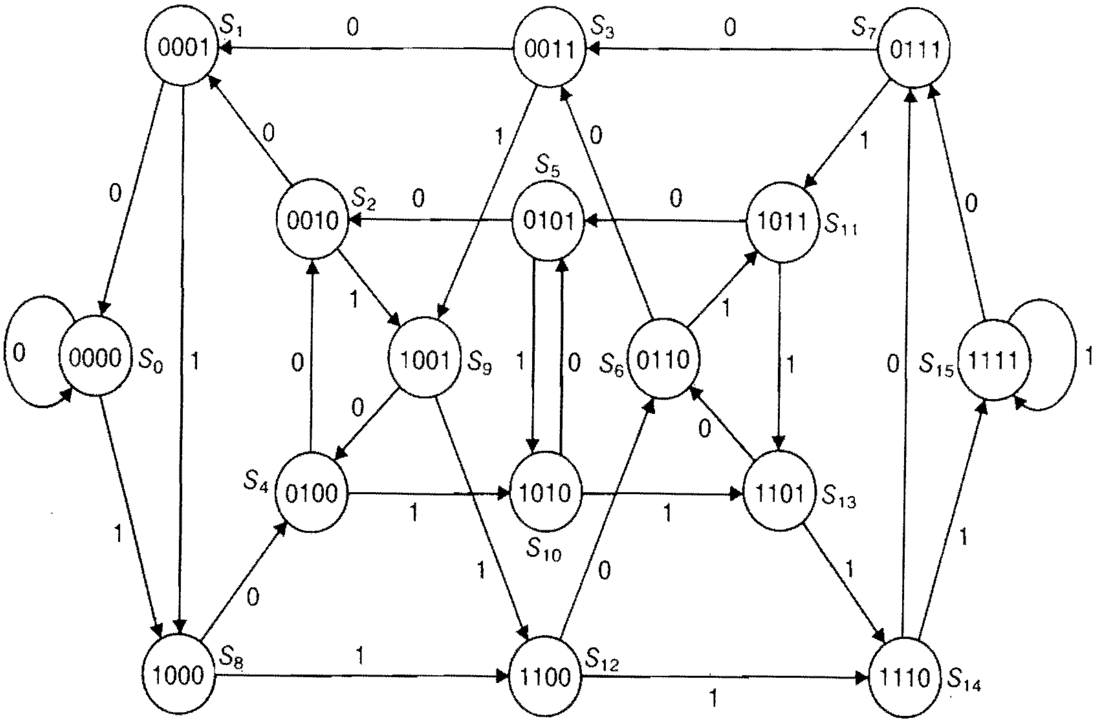


FIGURE 10.34 **Universal State Diagram for a Four-Bit Feedback Register.**



ble next states. These are 0100 if the J_3 input is a 0, or 1100 if the J_3 input is a 1. These values correspond to a shift-right operation. All possible internal states of the register and all possible transitions between the states are considered in this state diagram.

In order to design a counter or sequence generator, the designer selects the desired sequence of states on the universal state diagram. Based on the desired sequence the feedback logic is designed so that the register will cycle through the selected sequence of states. Example 10.5 illustrates the technique.

EXAMPLE 10.5

Design a decade sequence generator using a shift register that follows the sequence

$S_0 \rightarrow S_8 \rightarrow S_{12} \rightarrow S_{14} \rightarrow S_{15} \rightarrow S_7$
$\rightarrow S_{11} \rightarrow S_5 \rightarrow S_2 \rightarrow S_1 \rightarrow S_0$

SOLUTION

The state transitions and the corresponding feedback logic condition are obtained as shown in Figure 10.35. The resulting K-map for the feedback function J_3 is obtained as shown in Figure 10.36. The feedback function is

$$J_3 = \bar{Q}_1\bar{Q}_0 + Q_2\bar{Q}_0 + \bar{Q}_3Q_2Q_1$$

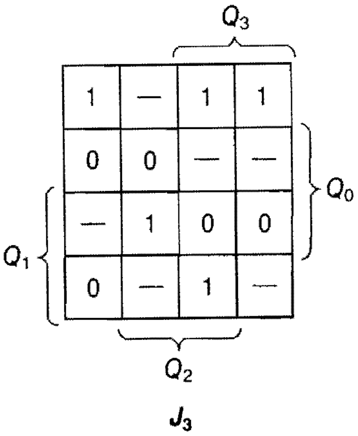
The feedback logic is realized using combinational logic. It is then fed to the J_3 input of the shift-right register, as shown in Figure 10.37, to obtain the desired sequence generator. Note that the nonsequence states lead eventually to the sequence as follows:

$S_3 \rightarrow S_1$, $S_4 \rightarrow S_{10} \rightarrow S_5$, $S_9 \rightarrow S_4$, and $S_{13} \rightarrow S_6 \rightarrow S_{11}$

FIGURE 10.35

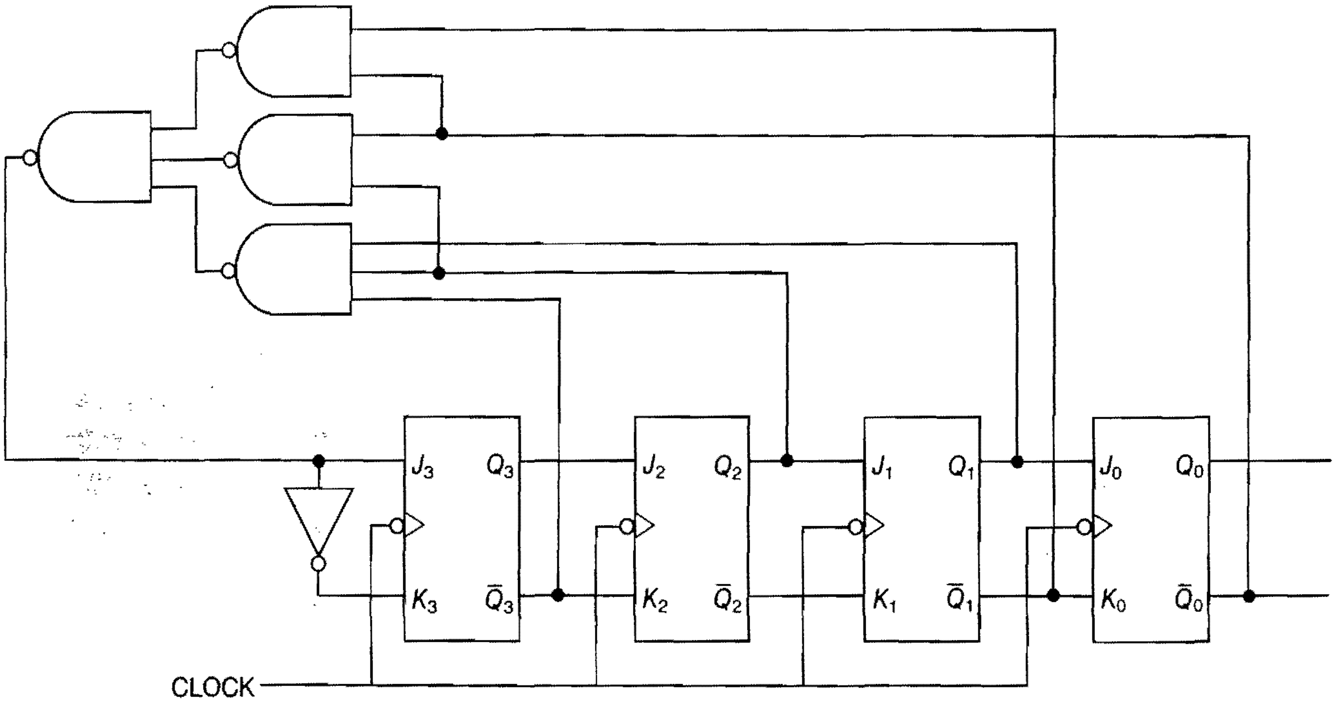
Transitions PS	 → NS	J_3
0000	1000	1
1000	1100	1
1100	1110	1
1110	1111	1
1111	0111	0
0111	1011	1
1011	0101	0
0101	0010	0
0010	0001	0
0001	0000	0

FIGURE 10.36



1	—	1	1
0	0	—	—
—	1	0	0
0	—	1	—

FIGURE 10.37



Consider, for example, that J_3 is a 0 whenever the present register state is at S_3 (= 0011). Therefore, the system would shift from state S_3 to S_1 (= 0001) as per the state diagram of Figure 10.34. If the sequence generator in this example enters an unused state due to a glitch or on power-up, it will return to the decade sequence after a maximum of two clock pulses.

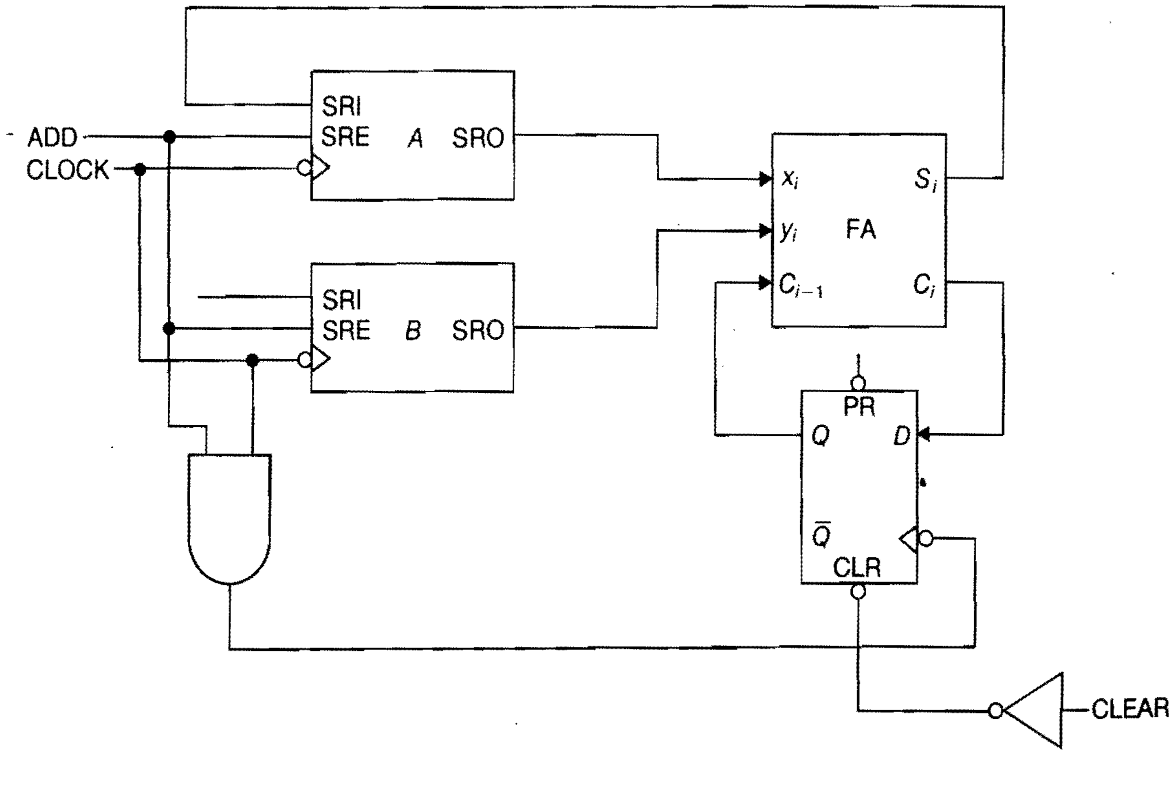
10.9 Counter and Register Applications

There are hundreds of applications of counters and shift registers. They are used extensively in computers. In general, digital computers process numbers by repeated arithmetic and logic operations. Execution of a specific instruction usually involves moving the instruction and data between registers. The data are operated on by the ALU as they are transferred between registers. These transfer sequences are in turn controlled by sequential circuits. In particular, registers provide the means for the storage of bits as they are being processed. On the other hand, counters keep track of the next memory location and count the intervals in the sequences that control these complex operations. In this section we shall consider only a few of their many important applications.

The operations in digital computers are performed in parallel in most cases since this is a faster mode of operation. In comparison, serial operations are slower but require less complicated and less expensive circuits. Consider the *add* function. In Chapter 5 the design of parallel addition circuits was examined in detail. The techniques developed were reasonably fast but they involved very complex circuitry. Frequently, the designer must make a trade-off between time and the number of components.

The add operation can also be performed by loading the addend and augend into two serial shift registers and shifting one bit from each register into a single-bit FA, as shown in the block diagram of Figure 10.38. The carry-out of the FA is stored in a *D* FF and fed back as the carry-in to the FA to be added to the next pair of signifi-

FIGURE 10.38 **Serial Adder Configuration.**

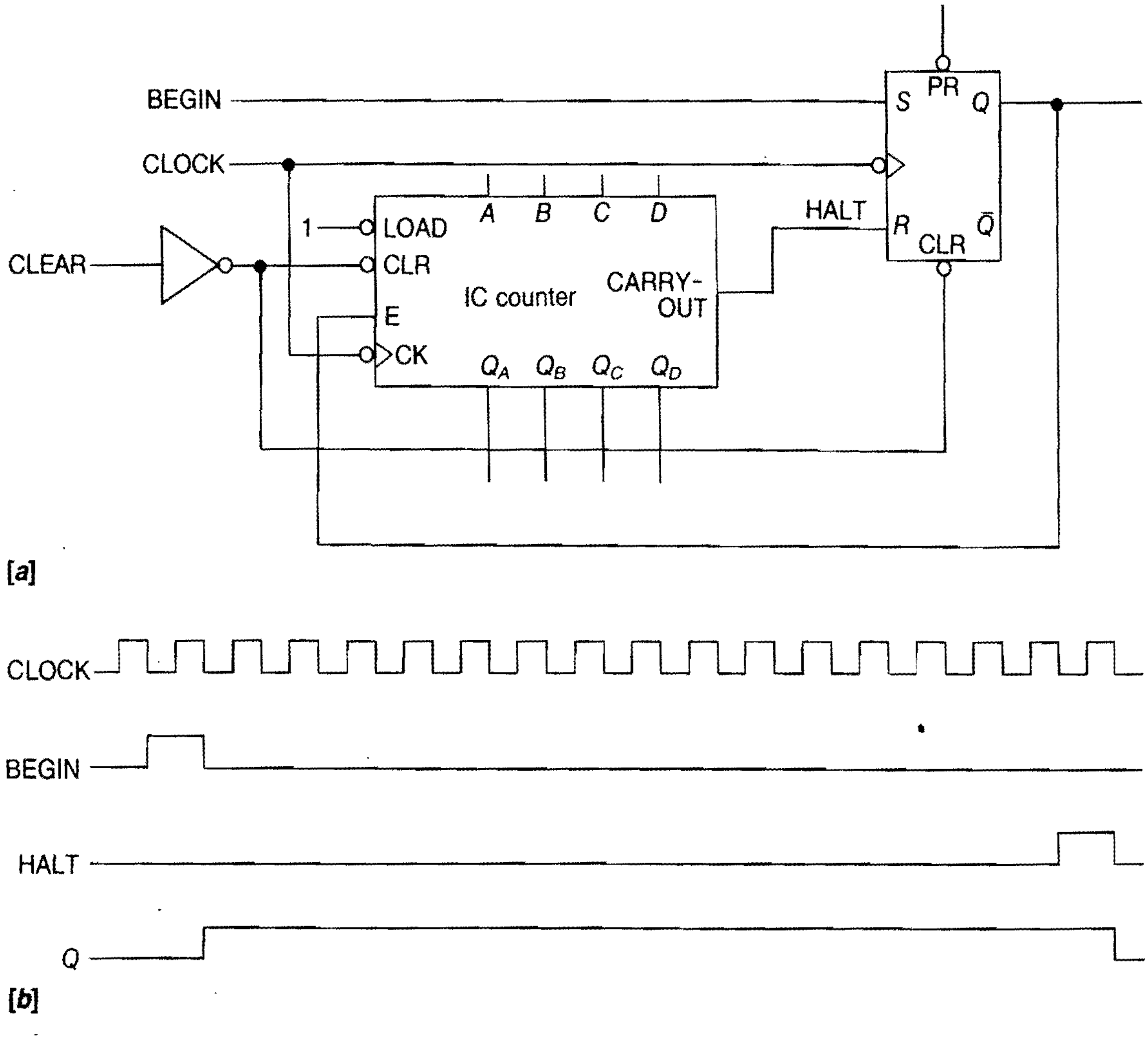


cant bits from the shift registers. The sum bit is shifted into the shift register containing the augend as the augend bits are continually being shifted out to the right.

Initially the shift registers *A* and *B* hold the augend and addend and the *D* FF is cleared. The summation is achieved by connecting each pair of bits, through shifting, together with the previous carry-out into the FA circuit and by transferring the sum bits serially, into the register *A*. The ADD command starts and stops the operation. When ADD is high the registers perform a shift-right operation at each clock, and when ADD is low the registers maintain a hold mode. In the next chapter we shall consider every aspect of how to design such a serial adder circuit. In the meantime we will develop other relevant concepts.

Operations in digital systems are controlled by a sequence of timing pulses. The control unit in a serial computer must generate a signal that remains high for a number of pulses equal to the number of bits in the shift registers. For example, the serial adder system of Figure 10.38 requires a control signal, ADD, for its operation. Figure 10.39[*a*] shows a control circuit that generates a signal that remains high for a period of 16 clock periods. The four-bit IC counter and the *SR* FF are initially CLEARed. The BEGIN signal

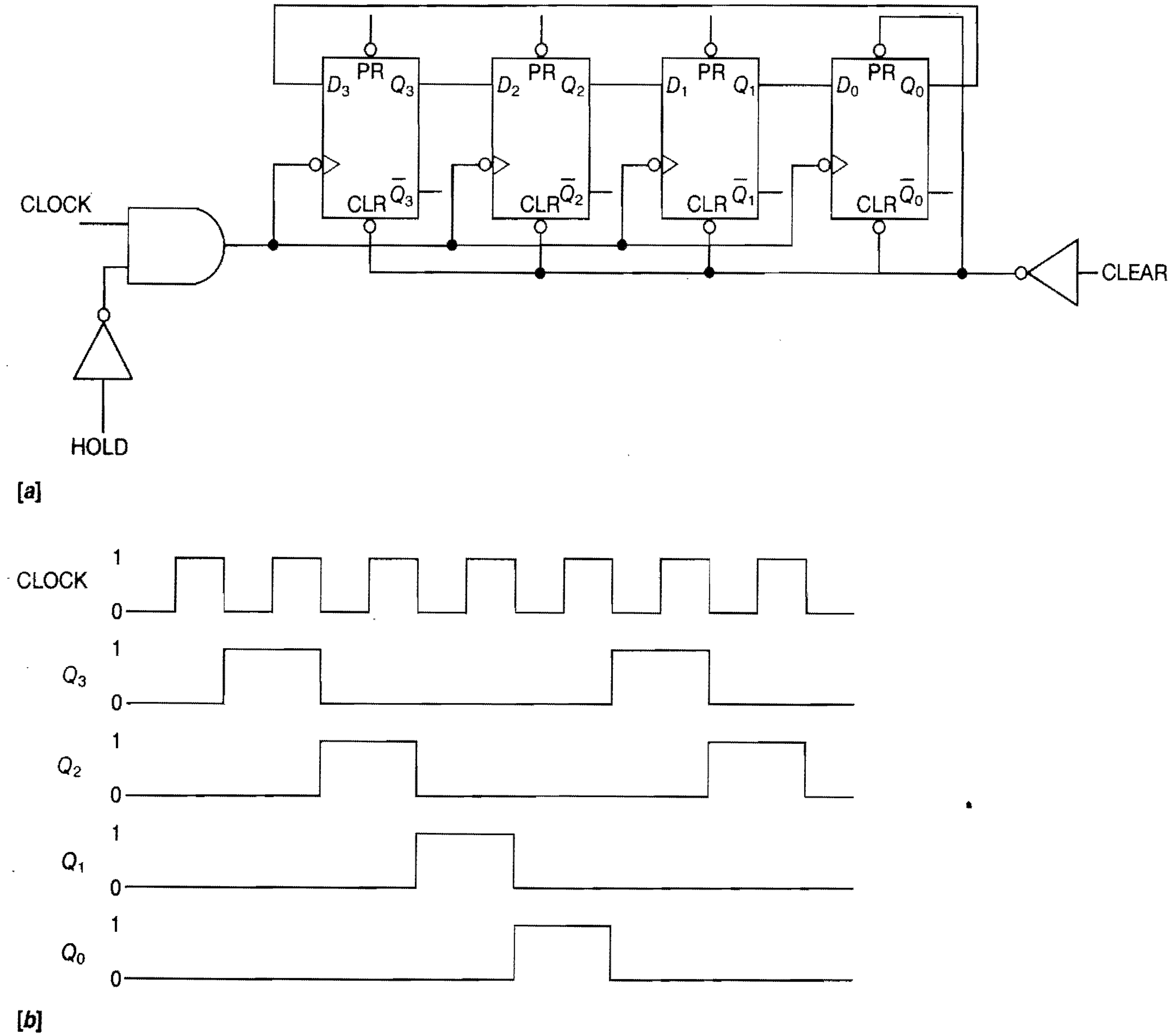
FIGURE 10.39 **Generation of Timing Sequences: [*a*] Circuit and [*b*] Timing Diagram.**



sets the *SR* FF, which in turn enables the counter. The FF output *Q* remains high for 16 pulses, as shown in the accompanying timing diagram of Figure 10.39[*b*]. When the counter reaches count 1111, HALT is activated, which in turn resets the FF. The BEGIN signal is synchronized with the clock and is made to stay on for one clock period. It could be made to stay on for a longer period; however, if it is made to last for more than 15 clock periods, the circuit will not function as expected. This HALT signal might be used in another similar circuit to generate a BEGIN pulse.

In a parallel mode of operation, a single pulse is generally used to specify the time at which an operation should be executed. Shift registers may be used to realize such a timing circuit when connected as a *ring counter.* A shift-right register used as a ring counter is shown in Figure 10.40. A feedback path is provided from the serial output to the serial input of the shift register. A shift register con-

FIGURE 10.40 **Four-Bit Ring Counter: [*a*] Circuit and [*b*] Timing Diagram.**



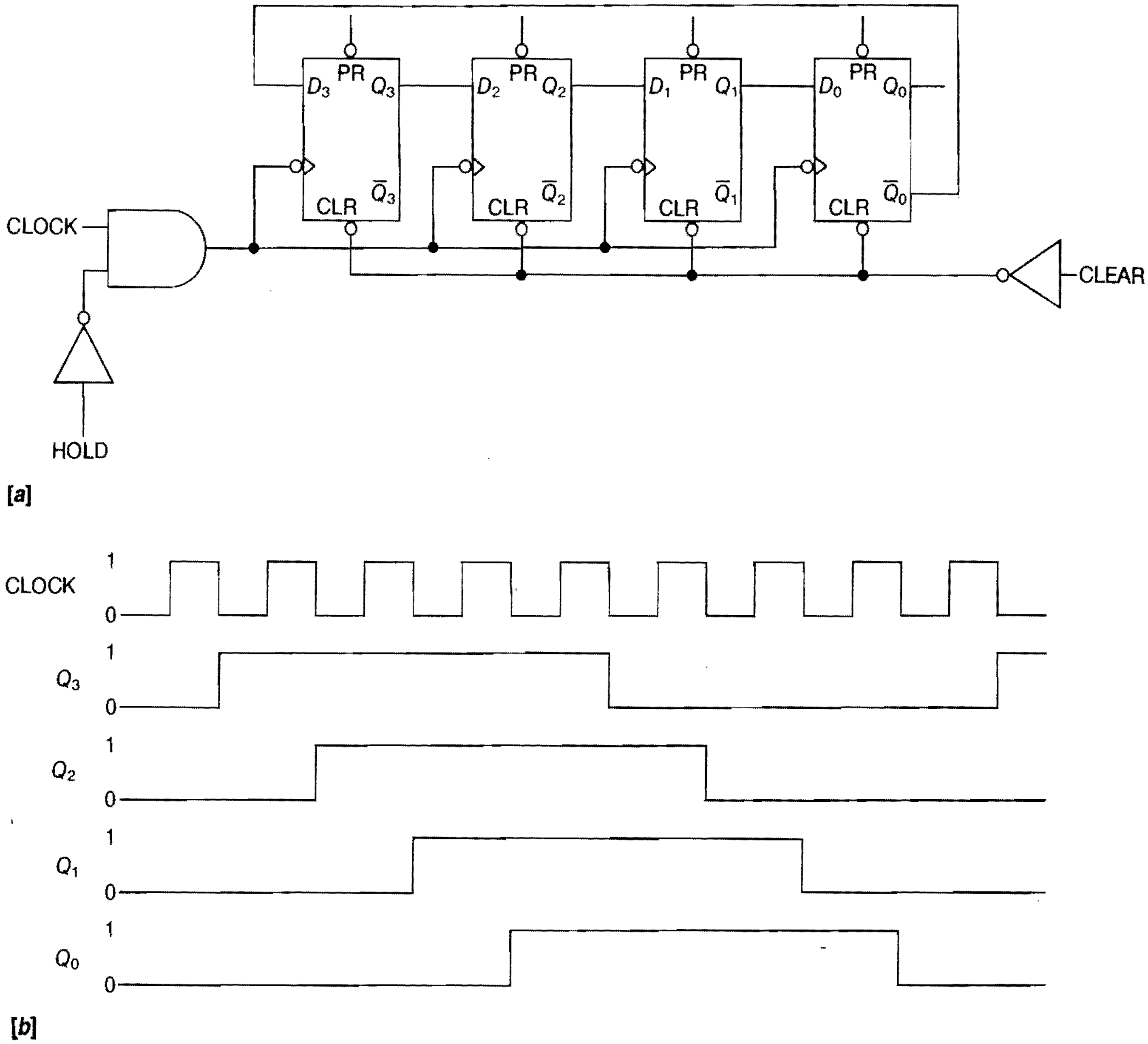
nected as such circulates the register contents. Initially the CLEAR input is set to a 1. This presets the right-most FF to a 1 and clears the others. The starting output word, therefore, is 0001, and as the clock pulses occur the output word becomes 1000, 0100, 0010, 0001, and so on. Only four unique states are possible. This circuit would allow timing four sequential operations by tying each of the four operation's initiating lines to one bit of the ring counter. Each operation would then be active one-fourth of the time.

A four-bit binary counter has 16 counting states, whereas the ring counter has only four states. Thus the ring counter makes an extremely uneconomical use of FFs, which is more true for systems requiring a large number of timing signals. An excellent alternative to using an uneconomical ring counter is to use an n-bit binary counter and an n-to-2^n line decoder. This combination is often referred to as a *Moebius* or *Johnson counter* (no relation to the coauthor). It also involves a shift-right register like that of a ring counter, but it is connected in a *switch-tail* configuration.

As stated earlier, an n-bit ring counter provides only n distinguishable states. The number of states can be doubled if the shift register is connected in a switch-tail configuration, as shown in the four-bit Johnson counter of Figure 10.41. Here $\overline{Q}_0$ instead of Q_0 is fed back as the D_3 input. The register shifts one bit to the right with every clock pulse, and at the same time the complement value of the fourth FF is transferred to the left-most FF. Consequently, this would result in eight different counting states: 0000, 1000, 1100, 1110, 1111, 0111, 0011, and 0001. These eight states must be decoded to give eight distinct timing sequences. Unlike the previous circuit, the least significant FF need not be preset. For minimum chip count, a ring counter or a Johnson counter is the best choice for a timing circuit.

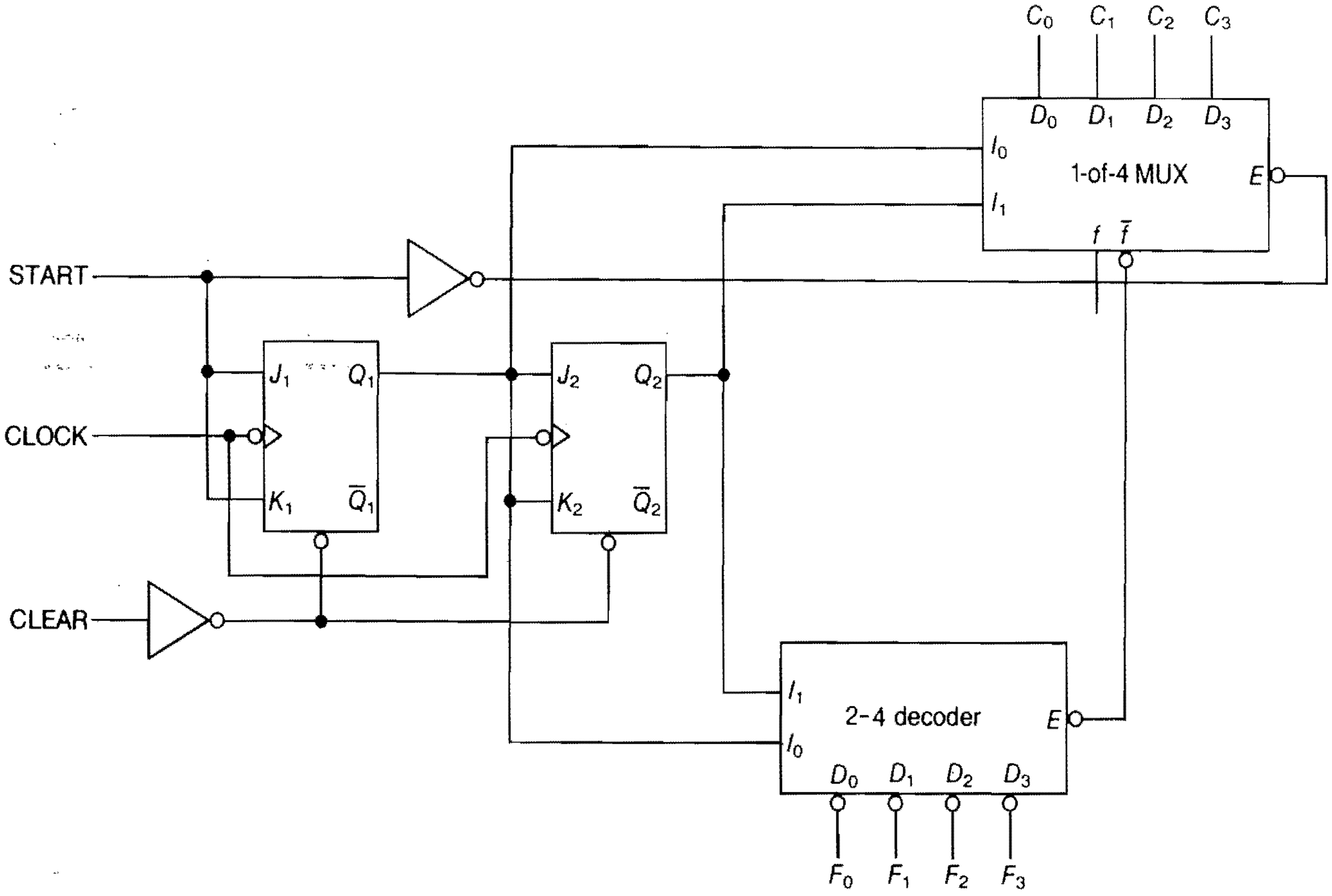
We have emphasized that counters are often used for sequencing various arithmetic and/or logic operations. And in most instances these operations are executed only if certain conditions are met. Figure 10.42 shows a four-state controller that can control four distinct operations. The sequencer circuit consists of a two-bit synchronous counter, a 1-of-4 MUX, and a 2-4 line decoder. Each of these four operations is activated by a low on the respective sequencer output. The sequencer output, D_n, will become low during its allocated time only if C_n (representing the corresponding condition) is a 1. The counter is first CLEARed. Consequently, it STARTs at the 00 address when the function F_0 is performed provided the condition C_0 is met. The function corresponds to a specific operation, and the condition C_0 may be the result of one or several test results. When the counter reaches the 01 value the function F_1 is executed, provided that the condition C_1 is met. The synchronous count allows the operations to be executed only at regular intervals. Once the sequencer is STARTed, the test conditions determine whether or

FIGURE 10.41 Four-Bit Johnson Counter: [*a*] Circuit and [*b*] Timing Diagram.



not the function is selected by the decoder. If the condition is not met, the decoder remains disabled, and as a result the corresponding function is not executed. The significance of a sequencer circuit, such as that of Figure 10.42, becomes meaningful only when it is allowed to control a complex digital system. Prior to the sequencer implementation, concepts of RTL (Register Transfer Language) will be introduced in Section 10.11.

FIGURE 10.42 Two-Bit Operation Sequencer.

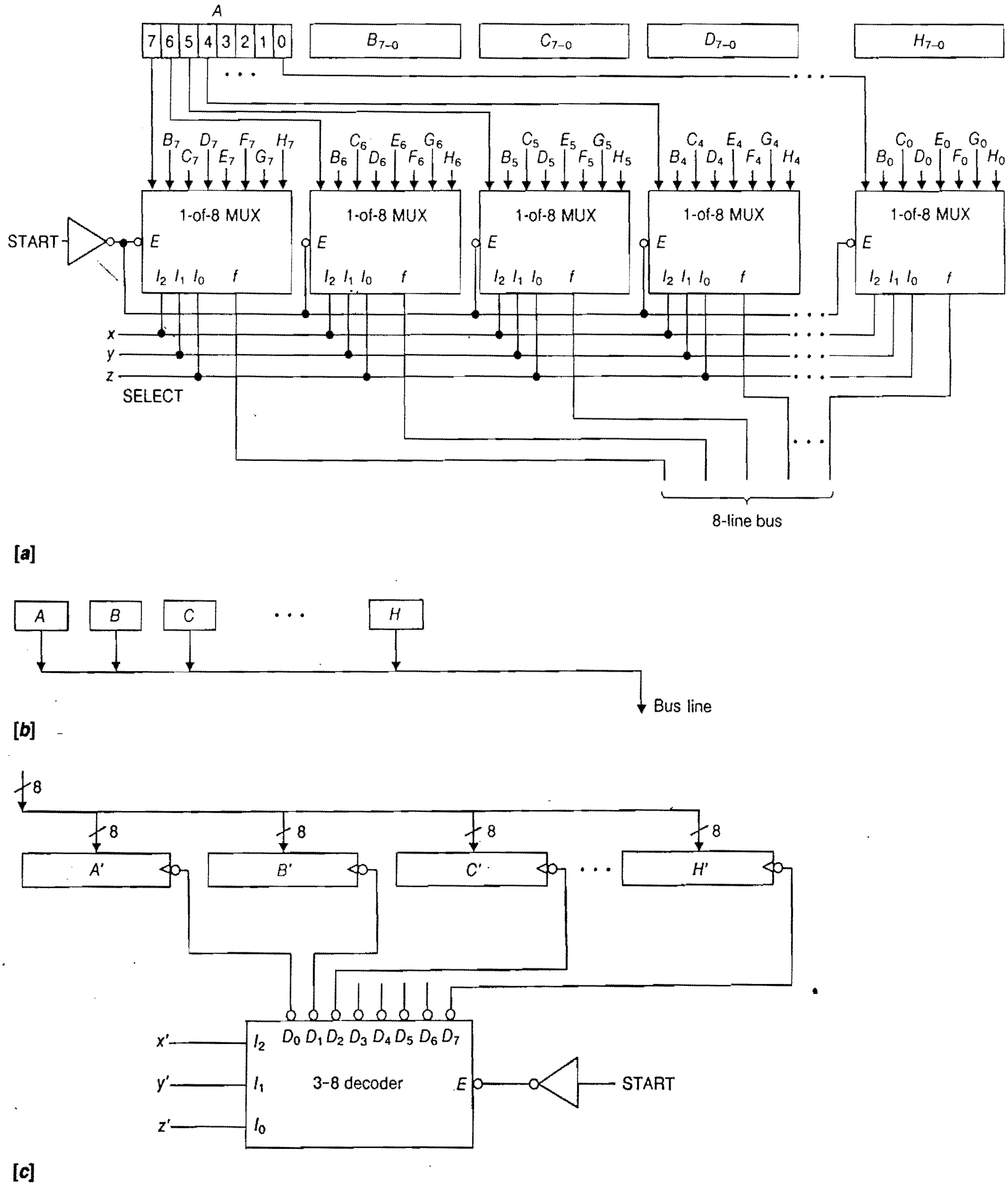


10.10 Bus Concept

Digital systems have large numbers of registers, and as part of the computational process it is often required to transfer data from one register to another. Consider the case of m registers with n bits in each. To allow direct inter-register transfer it would be necessary to have a total of $(m!)n$ data paths, which would require an awesome number of wires running between registers. Such a data transfer problem is solved by a group of wires called a *bus*. The bus concept is analogous to a mass transport system where each commuter waits in line until the transport becomes available (in this analogy the bus can carry only one passenger). For a parallel transfer of n bits the bus consists of only n lines, as shown in Figure 10.43[*a*].

Each of the 1-of-8 MUXs is equipped with eight input lines, three select lines, and one output line. The least significant inputs of each MUX are connected to the respective FFs of register A. The FF A_7 is connected to the first MUX, A_6 is connected to the second MUX, A_5 to the third MUX, and so on. The next significant input of each MUX is connected to the respective FFs of register B. This process is continued until all eight registers are connected. Figure

FIGURE 10.43 [*a*] Register-to-Bus Transfer Circuit for Eight Registers, [*b*] Block Diagram of Figure 10.43[*a*], and [*c*] Bus-to-Register Transfer Circuit for Eight Registers.


A
7 6 5 4 3 2 1 0
B_{7-0}
C_{7-0}
D_{7-0}
H_{7-0}
B_7 C_7 D_7 E_7 F_7 G_7 H_7
B_6 C_6 D_6 E_6 F_6 G_6 H_6
B_5 C_5 D_5 E_5 F_5 G_5 H_5
B_4 C_4 D_4 E_4 F_4 G_4 H_4
B_0 C_0 D_0 E_0 F_0 G_0 H_0
1-of-8 MUX
E
I_2 I_1 I_0 f
START
x
y
z
SELECT
8-line bus
[a]
A
B
C
H
Bus line
[b]
8
A'
B'
C'
H'
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7
x'
y'
z'
I_2
I_1
I_0
3-8 decoder
E
START
[c]

10.43[*a*] shows the necessary connections. When the select lines, x, y, and z, are all low the least significant input to each of the MUXs is selected, and consequently the bus is loaded with the contents of register A. Each combination of the x, y, and z inputs selects the contents of a particular register and the contents are then attached to the bus. The simplified block diagram is shown in Figure 10.43[*b*]. It is possible to design a bus system without the MUXs if the registers have tri-state outputs or are connected to the bus using tri-state buffers. Finally, the contents of the bus are required to reach a certain destination register. This requirement is accomplished by the circuit arrangement of Figure 10.43[*c*] where the select lines x', y', and z' determine the particular destination register by means of a 3-8 line decoder. Upon receiving the proper select inputs, the contents of the bus are loaded into the selected register.

Another example of the use of a bus involves a simple memory device. Registers often are assembled together to form a larger storage array. This arrangement of registers is referred to as a *scratch-pad* memory. Figure 10.44[*a*] shows an arrangement of registers that can store up to four four-bit words. The device consists of four registers that in turn have four FFs each. When the WRITE ENABLE (WE) is low, the four data inputs, I_0, I_1, I_2, and I_3, are routed to a particular location of each register as specified by the entries in the WRITE SELECT (WS) lines. A 00 on the WS lines will store the input bits in the respective 0th cell of the registers. Similarly, 01, 10, and 11 applied at the WS lines would respectively select the first, second, and third bit of each register. Stored data from the scratch-pad memory may be retrieved through the four output lines by applying a low to the READ ENABLE (RE) and necessary address bits to the READ SELECT (RS) lines. Figure 10.44[*b*] shows the logic diagram of the memory formed using gated D latches.

Scratch-pad memory, although very fast, is not extensively used. It is not particularly suitable for LSI because too many pin-outs

FIGURE 10.44 **16-Bit Scratch-Pad Memory: [*a*] Block Diagram**

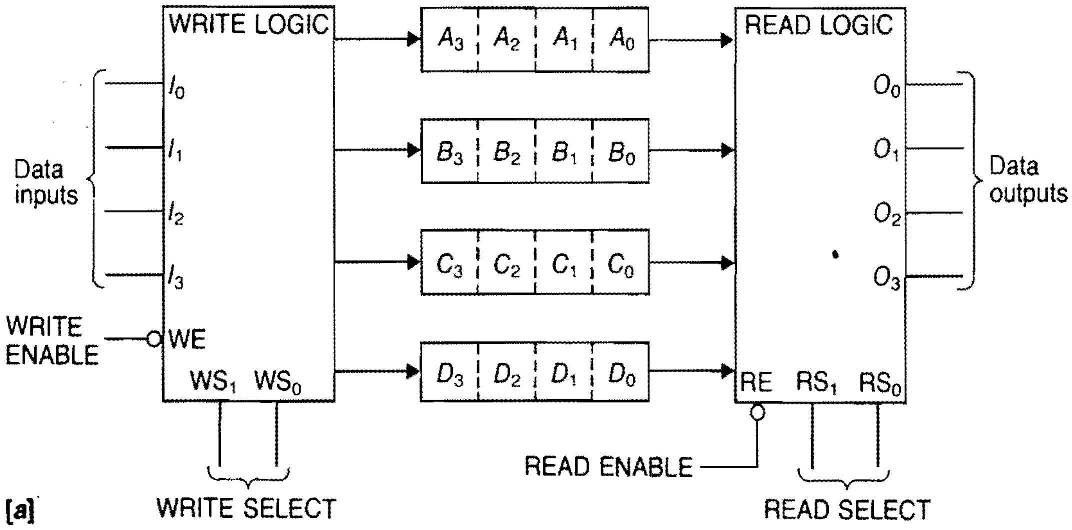
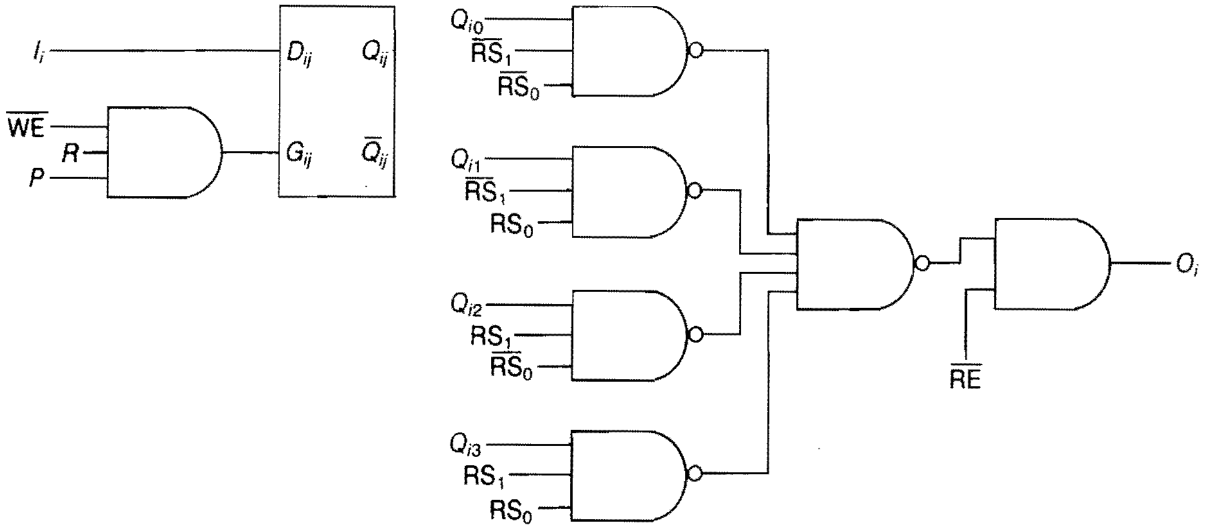


FIGURE 10.44 (*Continued*) **16-Bit Scratch-Pad Memory: [*b*] Circuit Showing *D* Latch at the *i*th Row and *j*th Column and the *i*th Output.**



$0 \leq i(= \text{row \#}) \leq 3$

$0 \leq j(= \text{column \#}) \leq 3$

$$R = \begin{cases} \overline{WS}_1 & \text{for } j = 0,1 \\ WS_1 & \text{for } j = 2,3 \end{cases}$$

$$P = \begin{cases} \overline{WS}_0 & \text{for } j = 0,2 \\ WS_0 & \text{for } j = 1,3 \end{cases}$$

RS lines: RS_1, RS_0
WS lines: WS_1, WS_0

[*b*]

make such a chip economically unattractive. Again, being fabricated from several gates/FFs, scratch-pad uses significant power and chip area. We shall look at some of the alternative memory sources in Chapter 12.

The applications of shift registers and counters are limited only by the imagination of the designer. Their importance will become more obvious as the design of digital systems continues in this and subsequent chapters.

EXAMPLE 10.6

Design a four-bit register capable of performing the 2's complement operation on its content.

SOLUTION

One of the procedures for performing the 2's complement operation (see Section 1.4 for details) requires that all bits on the left of the least significant nonzero bit be complemented. Using this scheme, a register can be built for loading the numbers in parallel and then performing the 2's complement operation using additional logic.

Define a control input, LOAD, to be used for selecting the parallel load operation, and its complement, $\overline{\text{LOAD}}$, for selecting the 2's complement operation. The loading excitation corresponding to a register consisting of only T FFs is given by

$$T_i = \text{LOAD} \cdot I_i + \overline{\text{LOAD}} \cdot (Q_{i-1} + Q_{i-2} + \cdots + Q_0) \quad \text{for } i > 0$$

where I_i is the parallel data input to the ith FF. The first term corresponds to loading the parallel input and the remaining terms correspond to performing a complement operation if at least one of the less significant bits is a 1. The equation is valid as long as the FFs have been CLEARed initially.

Note that in this complementing scheme Q_0 remains unchanged. Accordingly, the excitation equation for the least significant FF is given by

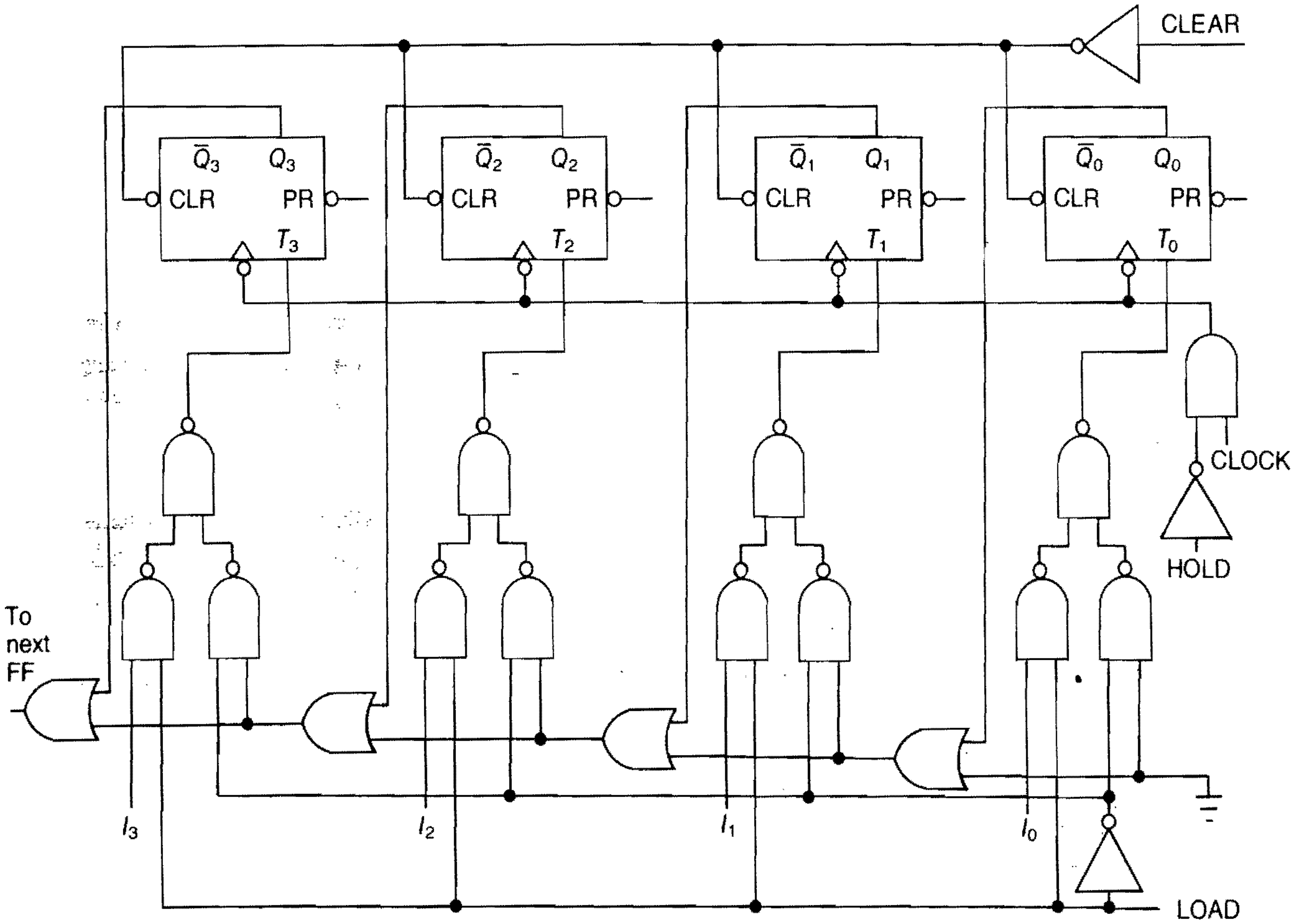
$$T_0 = \text{LOAD} \cdot I_0 + \overline{\text{LOAD}} \cdot (Q_0)$$

The overall excitation equation for the ith FF is now obtained as follows:

$$T_i = \text{LOAD} \cdot I_i + \overline{\text{LOAD}} \cdot (Q_{i-1} + (Q_{i-2} + (Q_{i-3} + (\cdots + (Q_0 + 0)\ldots))))$$

Consequently, if at least one of the bits on the right is a 1, the bit in question is complemented. The equation may be used to obtain the register circuit shown in Figure 10.45. The register should be LOADed with the data

FIGURE 10.45



only after the FFs have been CLEARed. The corresponding 2's complement is then obtained by supplying a low LOAD input. Note that a high HOLD could be used at any time to keep the register content unchanged indefinitely.

EXAMPLE 10.7

Design a digital wristwatch that displays the month, day, and time accurate up to 1 second using BCD–to–seven-segment LED display devices. All display except for seconds should be adjustable by the corresponding external ADJUST switches. Assume that the ADJUST switches can be used only if the external DISABLE switch has been turned on. The external BEGIN switch may be used to resume the operation. Assume further that the MINUTE (MIN) ADJUST automatically resets the seconds and you have a 60 Hz quartz crystal to run your device.

SOLUTION

A detailed examination of the problem reveals that this device can be made to function in the prescribed way in several steps:

Step 1. Sixty of the 60 Hz clock pulses may be counted in sequence to indicate 1 second.

Step 2. Sixty seconds may be counted in sequence to indicate 1 minute.

Step 3. Sixty minutes may be counted in sequence to indicate 1 hour.

Step 4. Twelve hours may be counted in sequence to indicate one-half day.

Step 5. Two sequences of 12 hours may be counted to indicate 1 day.

Step 6. Days are counted, and once the count equals the maximum number of days in a given month, the month count should be made to go up by 1 and the day count should be set to 1.

Step 7. At the completion of 12 months, the month count should be set to 1.

From the steps listed it appears that this problem can be solved by interconnecting several counters and decoders. A list of the components needed for the wristwatch includes the following:

scale-of-sixty counters,

up-counter that counts 1 through 12,

binary-to-BCD converters,

decoding circuit to determine the last day of a month,

multiplexers,

T FF,

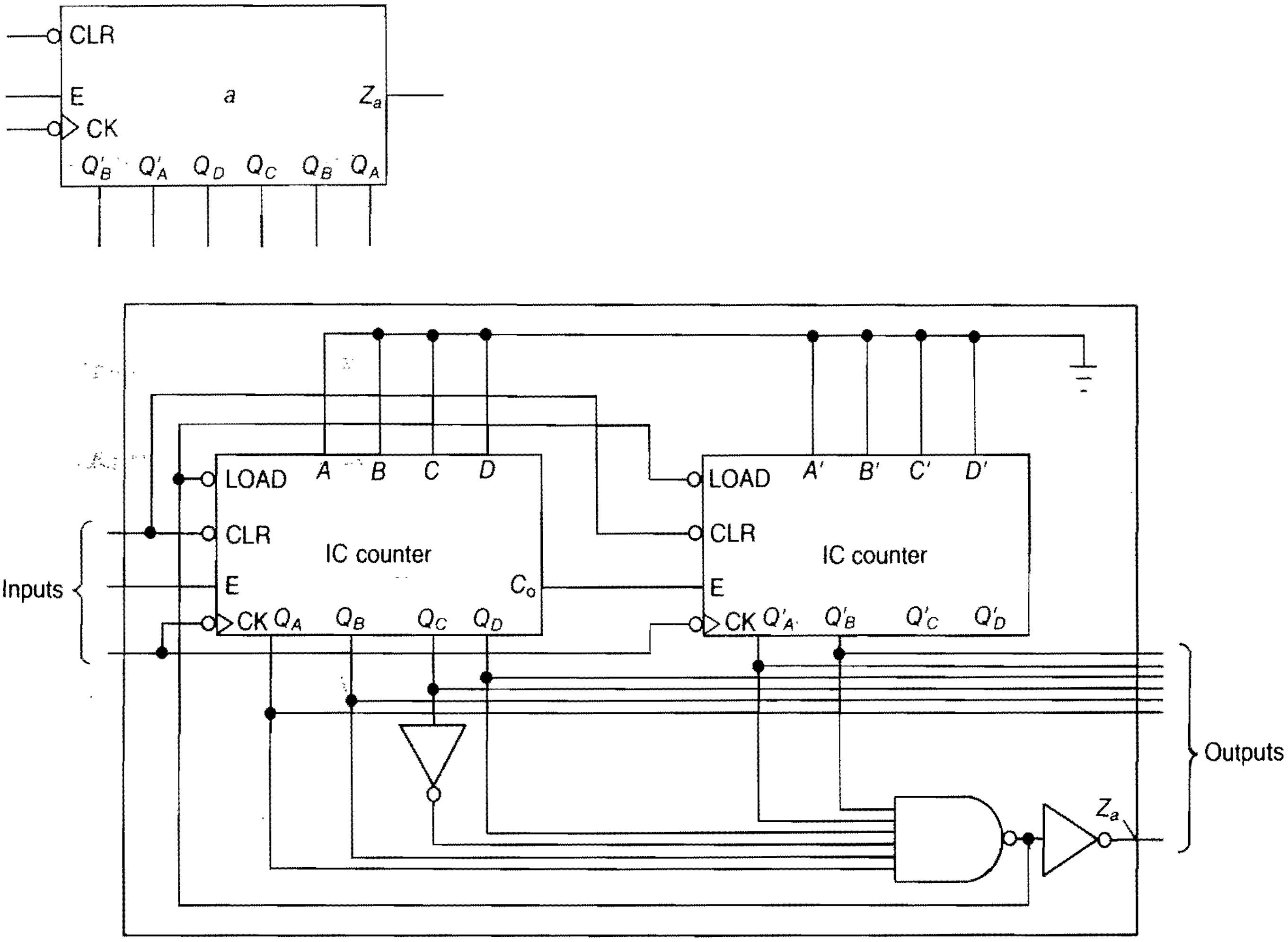
seven-segment display devices.

Steps 1–3 can be realized using three modulo-60 counters. Two modulo-12 counters could be used to implement Steps 4–7. Steps 5 and 6 can be implemented, respectively, using a *T* FF and last-day decoder circuit. BCD converters and seven-segment displays will be used for the purpose of display, and MUXs will be used for routing the data.

The IC counter module of Figure 10.18 will be used as the basic unit for producing the counter modules *a* and *b* as shown, respectively, in Figures 10.46 and 10.47.

Internally, module *a* consists of two four-bit IC counters cascaded together. The carry-out of the first is made to enable the second counter (see Figure 10.19 for a similar circuit). When the count reaches 59

FIGURE 10.46

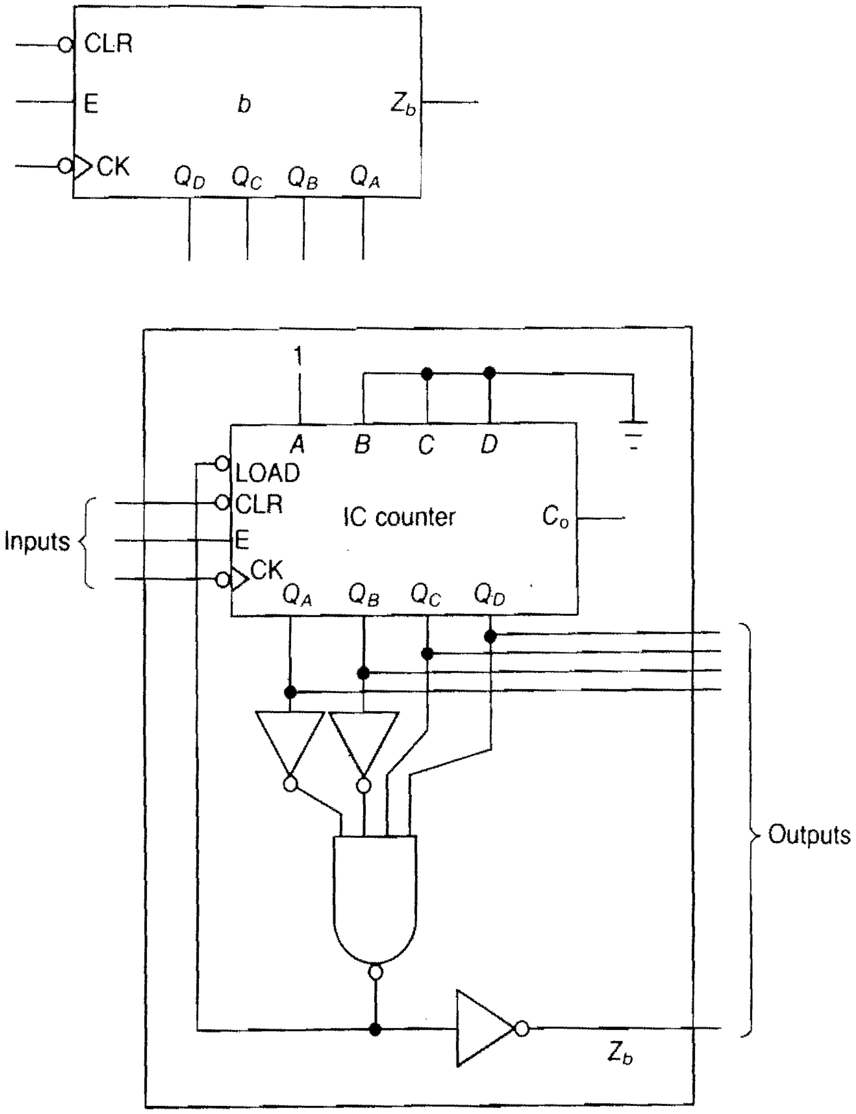

CLR
E
a
Z_a
CK
Q'_B Q'_A Q_D Q_C Q_B Q_A
LOAD
A B C D
CLR
IC counter
E
C_o
CK Q_A Q_B Q_C Q_D
LOAD
A' B' C' D'
CLR
IC counter
E
CK Q'_A Q'_B Q'_C Q'_D
Inputs
Outputs
Z_a

(111011), zero count is LOADed. However, module *b* consists of a single four-bit IC counter that is made to LOAD 0001 every time the count reaches 1100.

Module *c* was designed already in Example 5.7 (Figure 5.24). The only other module that remains to be designed is the decoding circuit for identifying the last day of a month. In terms of their lengths, the months may be classifed into four groups, respectively having a total of 28, 29, 30, and 31 days. The month of February has 28 days normally but has 29 days in a leap year. The module *d* may be designed accordingly, as shown in Figure 10.48, using a 1-of-4 MUX and two four-bit IC counters. The end-of-month is decided upon by the four values of *AB*. The MUX output becomes a 1 at the end of 28, 29, 30, and 31 input pulses when *AB* is 00, 01, 10, and 11, respectively. The decoding circuit is set to 1 each time an end-of-month has been located.

The modules may now be assembled to yield the wrist watch using the steps described earlier. In Step 1 the 60 Hz oscillator output is introduced into module *a*, as shown in Figure 10.49. A high SEC output indicates that an integral multiple of 60 input pulses has been counted. This is followed by Step 2, as shown in Figure 10.50, where a BEGIN input would allow another module *a* to count SEC pulses. For every integral multiple of 60

FIGURE 10.47



SEC pulses, a high will be generated at the minute output, MIN. The outputs of this module *a* are then converted to equivalent BCD numbers by means of three binary-to-BCD converter modules as shown in the circuit. For an explanation of this multi-bit, binary-to-BCD conversion scheme, review Example 5.8.

The BCD output is displayed by means of BCD–to–seven-segment LED display devices. The display will become 00 whenever the MIN ADJUST input is activated. This allows for the fact that the second display is cleared automatically whenever the minute display needs to be adjusted. Note, however, that the DISABLE input must always be activated prior to MIN ADJUST operation. The resulting low at *M* output would be used for all of the remaining ADJUST operations.

In Step 3, as shown by the circuit of Figure 10.51, the output of Figure 10.50 is fed into another module *a*. This configuration is made possible by means of a 1-of-2 MUX selectable by *M*. A high *M* would cause the circuit to count MIN pulses. In comparison, a low *M* would allow the adjustments of MIN counts. The HOUR output becomes a 1 corresponding to every integral multiple of 60 MIN pulses.

FIGURE 10.48

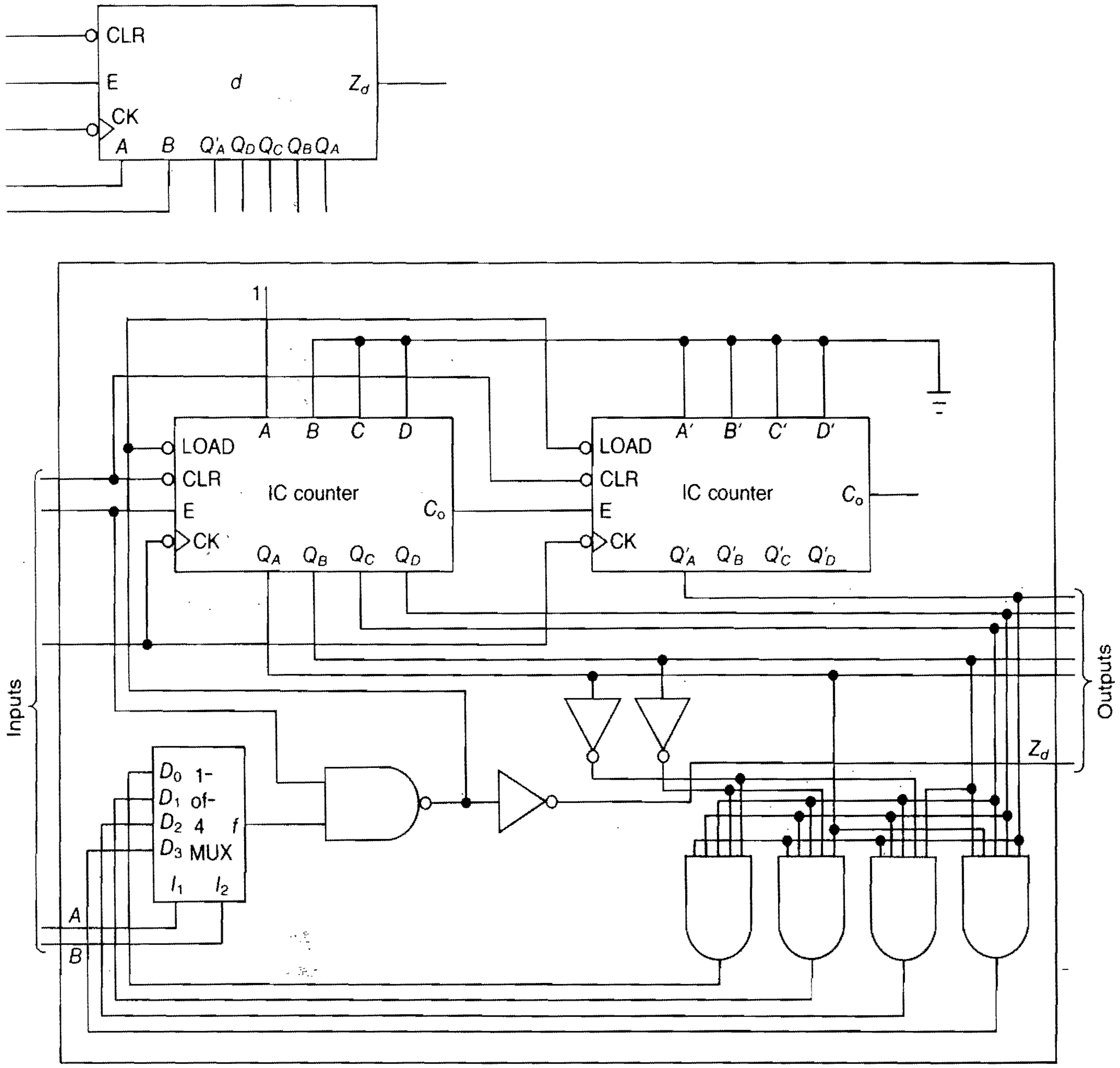


FIGURE 10.49

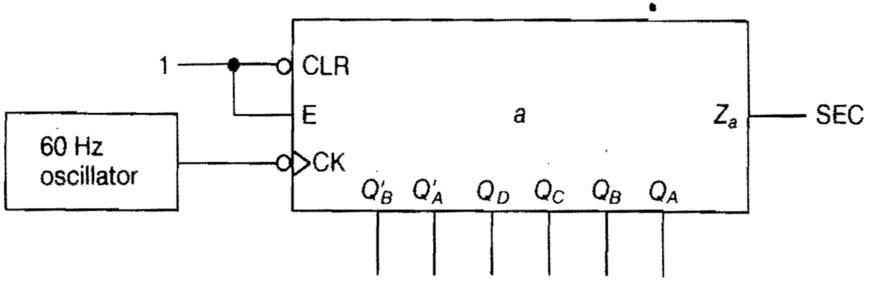


FIGURE 10.50

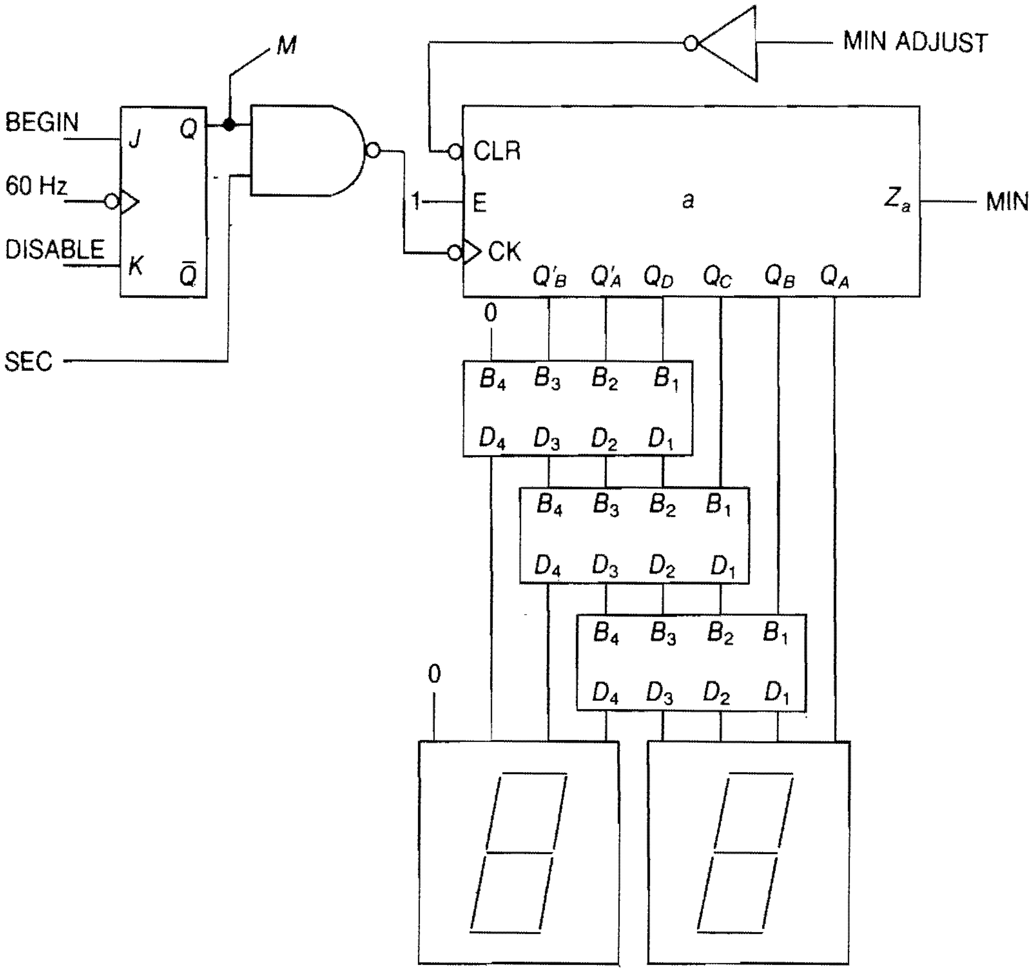
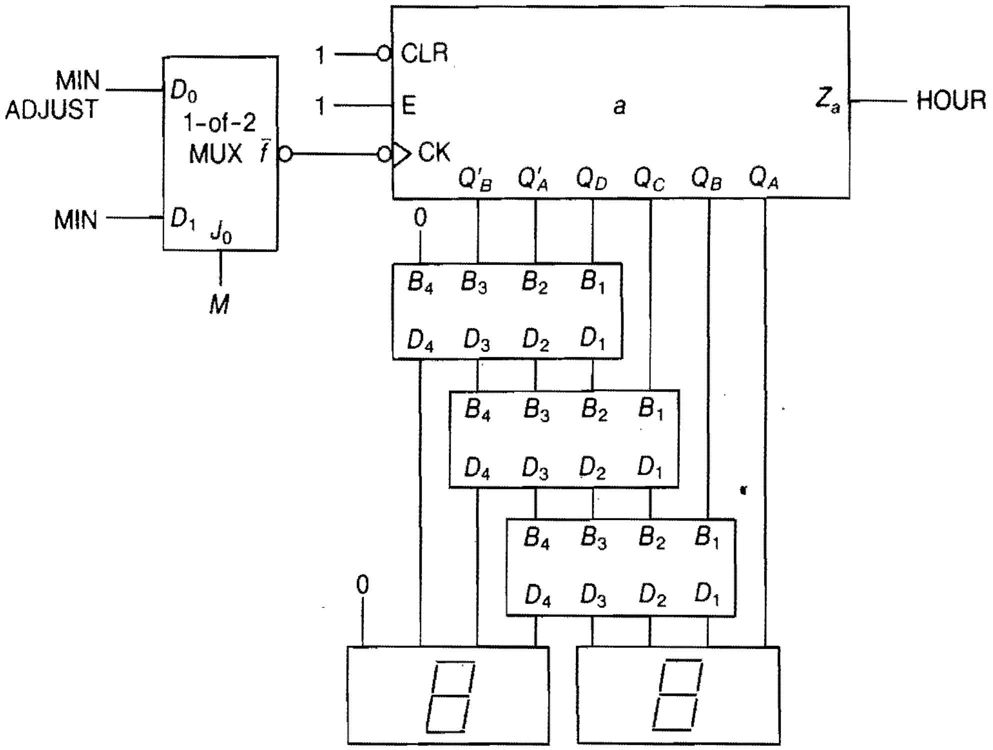


FIGURE 10.51



The circuitry in Step 4 is constructed using a module b, as shown in Figure 10.52. Its working principle is very similar to that of Figure 10.51. The 1/2-DAY output becomes a 1 corresponding to every integral multiple of 12 HOUR pulses.

The circuit of Figure 10.53 corresponds to both Steps 5 and 6. The counting circuit goes up by a 1 corresponding to every other 1/2-DAY pulse. Every time the end-of-month is located by means of AB inputs, the circuit of module d is set to day 1. The MONTH output becomes high each time the end-of-month is identified.

FIGURE 10.52

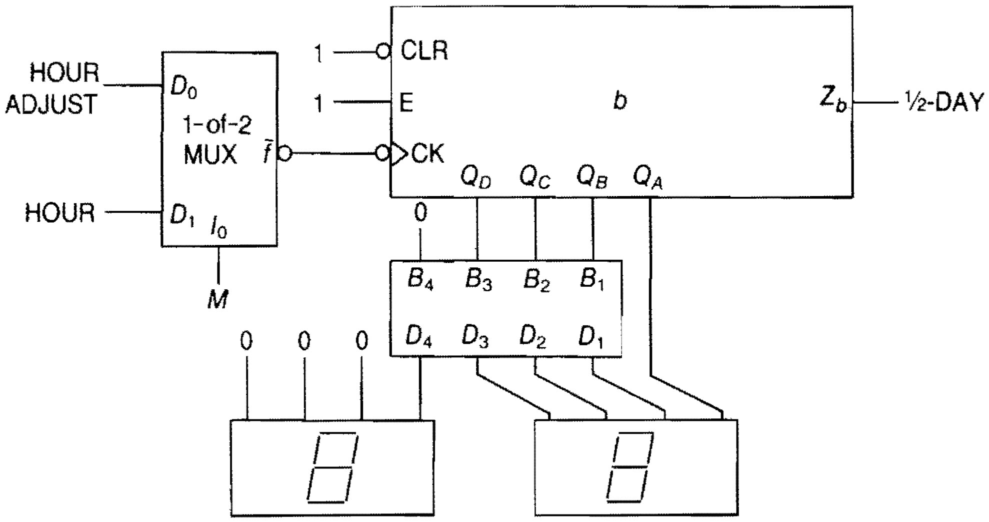
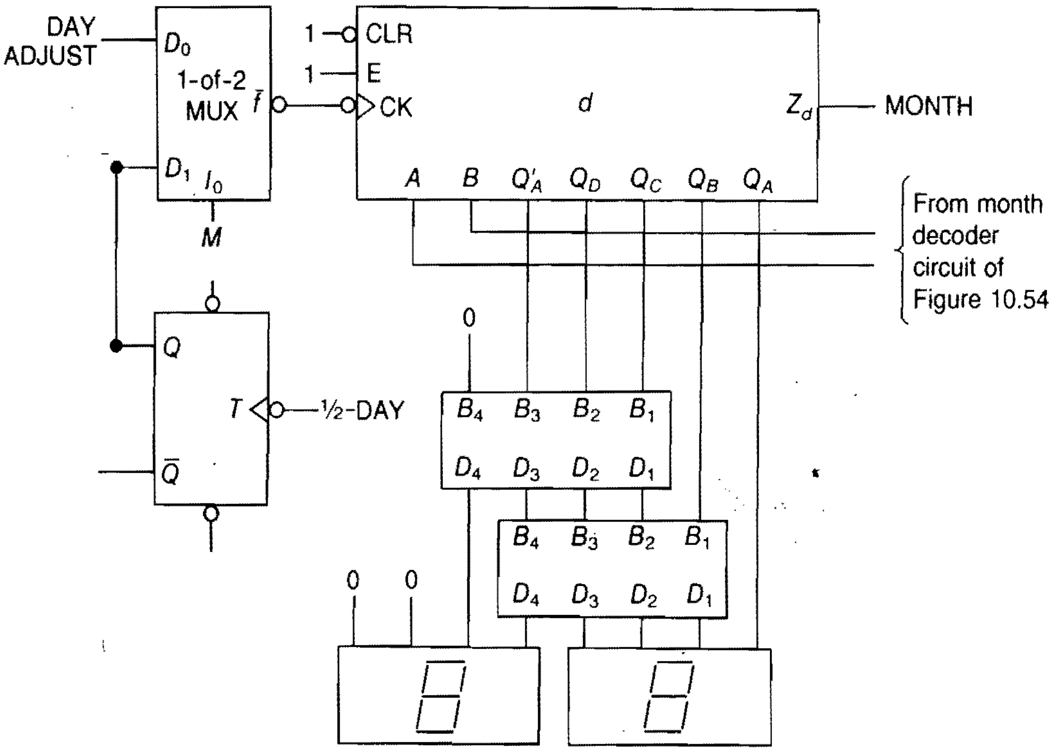


FIGURE 10.53

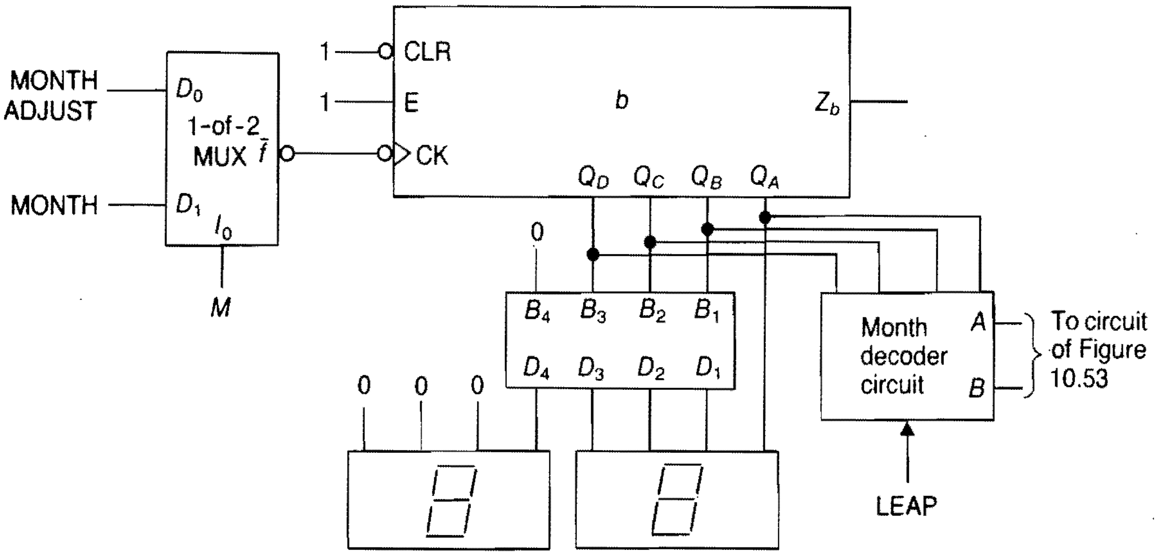


In the last step, as shown in Figure 10.54, the MONTH output is fed into a module *b*. The FF outputs, Q_D, Q_C, Q_B, and Q_A, and the external input, LEAP, are decoded to generate *A* and *B*. These outputs, *A* and *B*, are fed into module *d* of Figure 10.53. Note that LEAP is the extra input required for indicating that the current year is a leap year. This input needs to be entered prior to February 29. The month decoder circuit of Figure 10.54 examines the four FF outputs of module *b*. For the months January, March, May, July, August, October, and December, both *A* and *B* are set high. For all other months, except February, $A = 1$ and $B = 0$. In February $A = B = 0$ if LEAP $= 0$, and $A = 0$ and $B = 1$ if LEAP $= 1$. Consequently, the decoder has the following Boolean equations:

$$A = \overline{Q_D + Q_C + Q_A}$$

$$B = Q_D \oplus Q_A + \bar{Q}_D\bar{Q}_C \cdot \text{LEAP}$$

FIGURE 10.54



The design is now complete. After counting the chips involved, let's call this a table clock. Note that BCD counters instead of binary counters could have been used in this design, which would have reduced the total chip count. However, the incentives behind this particular design were to demonstrate the use of binary counters, to demonstrate the use of binary-to-BCD converter modules, and to demonstrate the thought processes involved in digital circuit designs.

10.11 Register Transfer Language Operations

The diversity of register types and applications points to the need for concise language to describe the flow of information (and processing enroute) of bits between registers. The most commonly used way to attain this goal is by an informal scheme called *register transfer language,* RTL, which was introduced first by I. S. Reed. A thorough investigation would reveal that a complete register transfer description is made up to two units: data and control. The *data unit* consists of registers, data paths, and logic necessary to implement a set of register transfers. The *control unit*, on the other hand, generates necessary signals in a specific sequence to regulate the register transfers within the data unit. The RTL scheme has the ability to specify the hardware involved in both units.

The simplest of all RTL operations is represented by $P \leftarrow Q$, which indicates that the data in register P are replaced by the data in register Q. It is also understood that both of these registers have the same number of bits. Such an operation can be completed during a single clock period, and thus it corresponds to a single-state transition of a sequential machine. One clock period, referred to as the *cycle time*, may be taken as the basic unit of time at the MSI complexity level. For uniformity, the following standardization is essential:

1. The contents of the registers should be denoted by one or more letters with the first always in uppercase. A concatenated register is the result of joining two or more registers in a string (represented by the register symbols separated only by commas) so that the LSB of the first-mentioned register is one bit to the left of the MSB of the second one, and so on.
2. Transfer between the registers will be considered parallel. In other words, all of the bits will transfer at the same instant of time.
3. The bits of each register shall be numbered from right to left. A_0 and A_{n-1} respectively represent the LSB and the MSB of an n-bit register named A. Note also that $A_{1,4}$ represents the bits 1 and 4 of register A, A_{1-4} represents bits 1 through 4 of register A, and A_M represents the subset of bits, M, of the register A.

Table 10.1 lists some of the most important RTL examples that include arithmetic, bit-by-bit logic, shift, rotate, scale, and conditional operations. In order to differentiate between the arithmetic and the logic operations, the following convention is maintained. The arithmetic addition is represented by a + symbol, the logical OR operation by a $\vee$ symbol, and the logical AND operation by a $\wedge$ symbol. The shift, rotate, and scale operations are generally represented by two lowercase letters. For shift and rotate, the first letter indicates the type of operation (r for rotate and s for shift) and the second letter indicates the particular direction (r for right and l for left). Furthermore, in the rotate operation the LSB and the MSB are considered to be adjacent. For all shift operations a 0 will be assumed to occupy the vacant bit. For the scale operations scl indicates scale left and scr indicates scale right.

TABLE 10.1 **Examples of RTL**

Type of Operation	Meaning	Register Bits after Operation
General		
$A_3 \leftarrow A_2$	Bit 2 of A to bit 3 of A	$A = 11110$
$A_3 \leftarrow B_4$	Bit 4 of B to bit 3 of A	$A = 11110$
$A_{1\text{-}3} \leftarrow B_{1\text{-}3}$	Bits 1 through 3 of B to bits 1 through 3 of A	$A = 11000$
$A_{1,4} \leftarrow B_{1,4}$	Bits 1 and 4 of B to bits 1 and 4 of A	$A = 10100$
$A_{1\text{-}3} \leftarrow B_z$	Groups of bit Z of B to bits 1 through 3 of A	$A = 11000$
Arithmetic		
$B \leftarrow 0$	Clear B	$B = 00000$
$A \leftarrow B + C$	Sum of B and C to A	$A = 11001$
$A \leftarrow B - C$	Difference $B - C$ to A	$A = 10111$
$C \leftarrow C + 1$	Increment C by 1	$C = 00010$
Logic		
$A \leftarrow B \wedge C$	Bit-by-bit AND result of B and C to A	$A = 00000$
$A \leftarrow B \vee C_4$	OR operation result of B with bit 4 of C to A	$A = 11000$
$C \leftarrow \overline{C}$	Complement C	$C = 11110$
$B \leftarrow \overline{B} + 1$	2's complement of B	$B = 01000$
$B \leftarrow A \oplus C$	X-OR operation result of A and C to B	$B = 10111$
Serial		
$B \leftarrow \text{sr } B$	Shift right B one bit	$B = 01100$
$B \leftarrow \text{sl } B$	Shift left B one bit	$B = 10000$
$B \leftarrow \text{sr2 } B$	Shift right B two bits	$B = 00110$
$B \leftarrow \text{rr } B$	Rotate right B one bit	$B = 01100$
$B \leftarrow \text{rl2 } B$	Rotate left two bits	$B = 00011$
$B \leftarrow \text{scr } B$	Scale B one bit (shift right with sign bit unchanged)	$B = 11100$
$B \leftarrow \text{scl } B$	Scale B one bit (shift left with sign bit unchanged)	$B = 10000$
$B,C \leftarrow \text{sr2 } B,C$	Shift right concatenated B and C two bits	$B,C = 0011000000$
Conditional		
IF $(B_4 = 1)$ $C \leftarrow 0$	If bit 4 of B is a 1, then C is cleared	$C = 00000$
IF $(B \geq C)$ $B \leftarrow 0,\ C_1 \leftarrow 1$	If B is greater than or equal to C, then B is cleared and C is set to 1	$B = 00000$ $C = 00011$

Initial values: $A = 10110$, $B = 1\underbrace{100}_{z}0$ and $C = 00001$.

The operations as described in Table 10.1 may now be combined to write complex functions or a sequence of operations. The control conditions are included along with the operation state to distinguish one set of executions from another. Recall from the circuits of Figure 10.43 that the execution of an operation and the transfer of data are usually regulated by one or several control conditions. For example, the loading of the contents of A into the bus may be expressed as follows:

$\bar{x} \cdot \bar{y} \cdot \bar{z}$: bus $\leftarrow A$;

The control condition $\bar{x}\bar{y}\bar{z}$ is separated from the corresponding operation by the ":" sign, while the sign ";" indicates the end of the operation. This RTL statement indicates that when $\bar{x}\bar{y}\bar{z} = 1$, the content of register A should be transferred to the bus. When more than one operation, $A \leftarrow B$ and $B \leftarrow$ sr B, are to be performed under the same control condition $S = 1$, the operation is expressed as follows:

S: $A \leftarrow B$; $B \leftarrow$ sr B ;

The operations that are performed simultaneously follow the same control condition. Accordingly, when S is true, B is transferred to A and is also restored after being shifted right one bit. Similarly, the RTL operations describing the function of the circuit of Example 10.6 may be summarized as follows:

CLEAR: $R \leftarrow 0$;

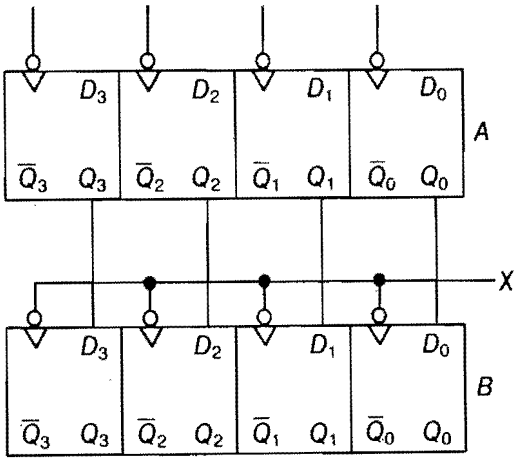
LOAD: $R \leftarrow I$;

$\overline{\text{LOAD}}$: $R \leftarrow \bar{R} + 1$;

where R is the register receiving the input bits from register I. These three RTL statements describe the action of a significant amount of hardware. In this and the next chapter we shall be discussing several of the complex circuits and their correlation with the corresponding RTL statements. Such one-to-one correspondence will make us appreciate the simplification that results from the use of RTL.

Consider the circuit shown in Figure 10.55. Here we have two registers, A and B, having four D FFs each. For simplicity the FFs

FIGURE 10.55 **Realization of a Register Copying Operation.**



within the registers are not internally cascaded together. The FF outputs of register A are connected to the respective D inputs of register B. Corresponding to the trailing edge of X input, the contents of register A are loaded into register B. This hardware operation will be represented by the following RTL statement:

X: $B \leftarrow A$;

This could also be written as follows:

X: $B_3 \leftarrow A_3$, $B_2 \leftarrow A_2$, $B_1 \leftarrow A_1$, $B_0 \leftarrow A_0$;

Both of these operations are equivalent; but we would prefer to use the first form since it is more concise.

Note that a transfer operation is really a copying operation where the contents of the source register remain unaltered. This type of RTL operation is the most common, but there are other possibilities. Consider the four situations of Figure 10.56. In each of these cases each of the registers is of four-bit length.

In Figure 10.56[*a*] the $\bar{Q}$ outputs of register A are connected to the respective D inputs of register B. A clock input P would result in the following transfer:

P: $B \leftarrow \bar{A}$;

Figure 10.56[*b*] shows register A where its MSB is connected to a 0. Each of the Q_n outputs of register A, except for the LSB, is fed to the D_{n-1} input of the same register. A pulse at R would cause the following transfer:

R: $A \leftarrow$ sr A ;

FIGURE 10.56 **Realization of: [*a*] a Complement Transfer Operation, [*b*] a Shift-Right Operation.**

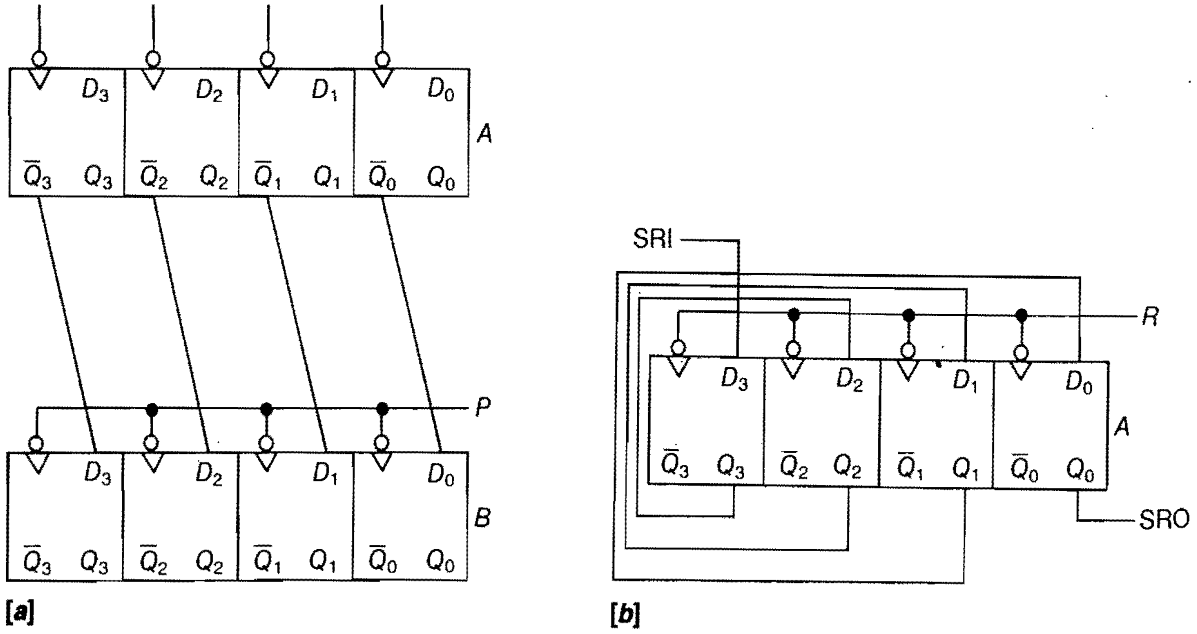
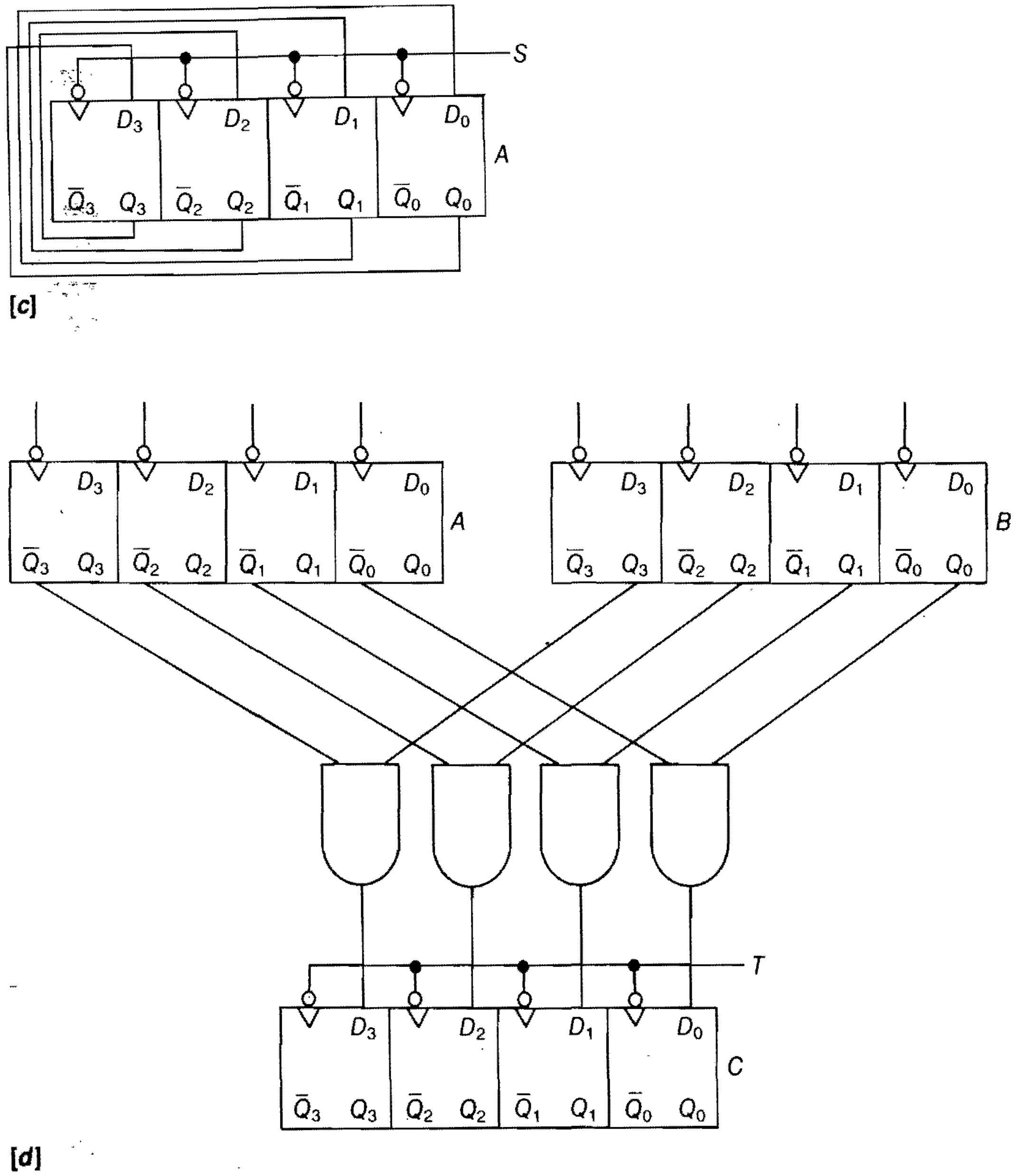


FIGURE 10.56 (*Continued*)
Realization of: [*c*] a Rotate-Right Operation, and [*d*] a Logical Operation and Transfer.



Similarly, Figures 10.56[*c*–*d*] respectively correspond to the following register transfers:

S: $A \leftarrow \text{rr}\ A$;

T: $C \leftarrow \bar{A} \wedge B$;

Note that in all of these valid RTL expressions, the source and destination registers have the same number of bits.

EXAMPLE 10.8

Design a typical stage for performing the following operations:

T_1: $A \leftarrow 0$;

T_2: $A \leftarrow A \vee B$;

T_3: $A \leftarrow A \wedge B$;

T_4: $A \leftarrow \bar{A}$;

where A and B are two multi-bit registers of equal bit size.

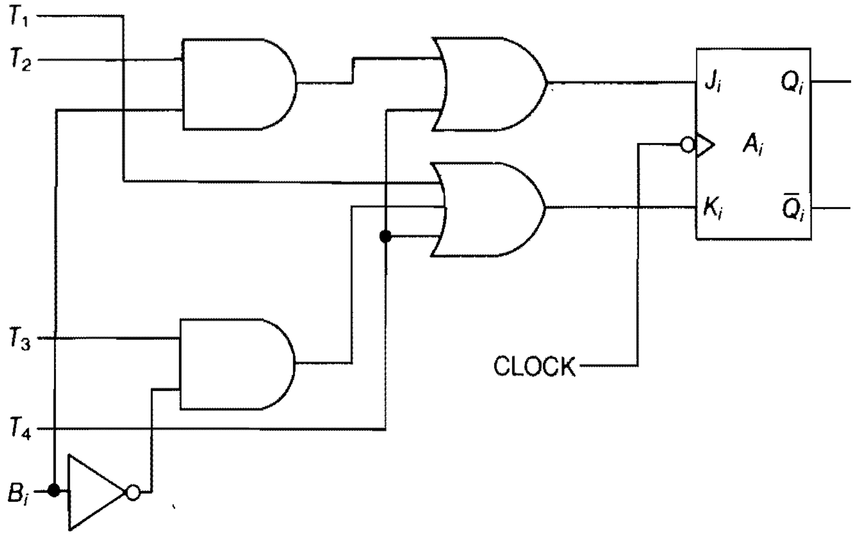
SOLUTION

You might decide to use *JK* FFs for the design of register A. Register B doesn't need to be designed because it is no different than a regular register with parallel outputs. The *JK* excitations necessary to turn on the *i*th FF for performing the required operations are obtained as follows.

$A \leftarrow 0$	is possible when $K_i = T_1$
$A \leftarrow A \vee B$	is possible when $J_i = T_2 \cdot B_i$ and $K_i = 0$
$A \leftarrow A \wedge B$	is possible when $J_i = 0$ and $K_i = T_3 \cdot \bar{B}_i$
$A \leftarrow \bar{A}$	is possible when $J_i = K_i = T_4$

The corresponding circuit is obtained as shown in Figure 10.57.

FIGURE 10.57



10.12 RTL Applications

The importance of RTL to describe the internal operations of a digital system is primarily due to the flexibility with which a design can be described and the direct way the data and control circuitry can be realized from RTL statements. Consider, for example, a system with a four-bit input, I, a four-bit output, O, and three four-bit registers, X, Y, and Z, in which the following algorithm is to be implemented:

A: $X \leftarrow I$;

B: $Z \leftarrow \text{sl2}\ X$;

C: $Y \leftarrow \bar{X}$;

D: $Y \leftarrow Y \wedge Z$;

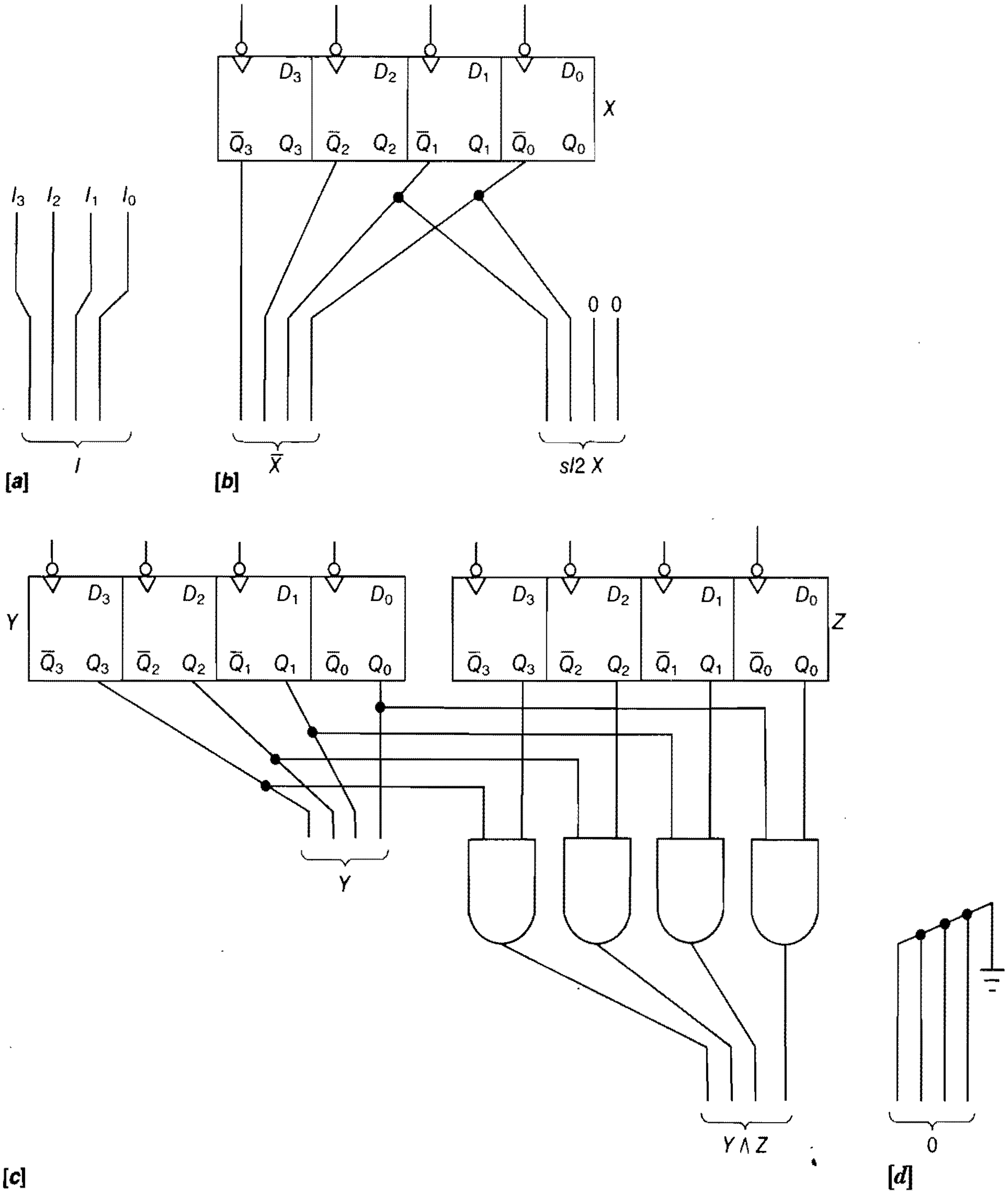
E: IF ($Y_3 = 1$) $O \leftarrow 0$ ELSE $O \leftarrow Y$;

The algorithm begins at state A and continues through state E. The input, I, is shifted left two bits and stored in Z, and the complemented input is stored in Y. This operation is followed by a bit-by-bit AND operation between the contents of Y and Z, and the result is stored subsequently in Y. If $Y_3 = 1$, only 0000 is placed on the output lines; otherwise, the contents of Y are placed on the output lines. In brief, this particular algorithm includes various logic, shift, and conditional transfers.

We shall now attempt to develop the hardware necessary from this RTL algorithm. Each of these RTL statements has two significant parts. In each statement, the right side of the ← sign specifies the signals that must be generated for either storage in registers or transfer to output lines. Correspondingly, the left side of ← specifies the destination registers or output lines. For this example $\bar{X}$ can be obtained by taking lines from the $\bar{Q}$ outputs of register X. No logic gates would be necessary for the shifting operation, as evident from Figure 10.56[*b*]. However, four AND gates would be necessary in state D. Figure 10.58 shows the connections necessary to develop the required signals, I, $\bar{X}$, sl2 X, Y, $Y \wedge Z$, and 0.

The next step is to feed the generated signals to the respective destination register or output lines. Figure 10.59 shows the involved connections. The complete RTL algorithm requires five clock periods. Corresponding to the first RTL statement, the inputs, I, are loaded to the D inputs of register X at the trailing edge of clock A, a transfer pulse fed to the clock input of register X during the allocated time for the first RTL statement. Similarly, at the trailing edge of clock B, $A_1A_0$00 are loaded into register Z. This is equivalent to the transfer of A after it has been shifted left twice. The third RTL statement requires that $\bar{X}$ be transferred to Y. This could have been possible simply by loading $\bar{X}$ to register Y at the trailing edge of clock C. However, it can be seen that the fourth RTL statement also includes a load operation involving register Y. To allow for these load operations, two separate ports of AND gates are used to select the inputs to Y, and a port of OR gates is used to combine them. The $\bar{Q}$ outputs of register X are transferred through one of the AND ports by CL, a level signal that is high during the period available for the third RTL statement. Similarly, the corresponding outputs of both Y and Z are transferred through the other AND port by DL. This AND port functions as the bit-by-bit AND logic circuit and also as the select port for register Y. The outputs of these two AND ports are loaded into register Y at the trailing edge of clocks C and D, respectively. The conditional transfer of the fifth RTL statement is realized when EL is high by ANDing each of the Y bits with $\bar{Y}_3$ and EL. It is appropriate to point out that when data are loaded into a register, they remain there until the power is turned off or other data are loaded. However, when data are placed

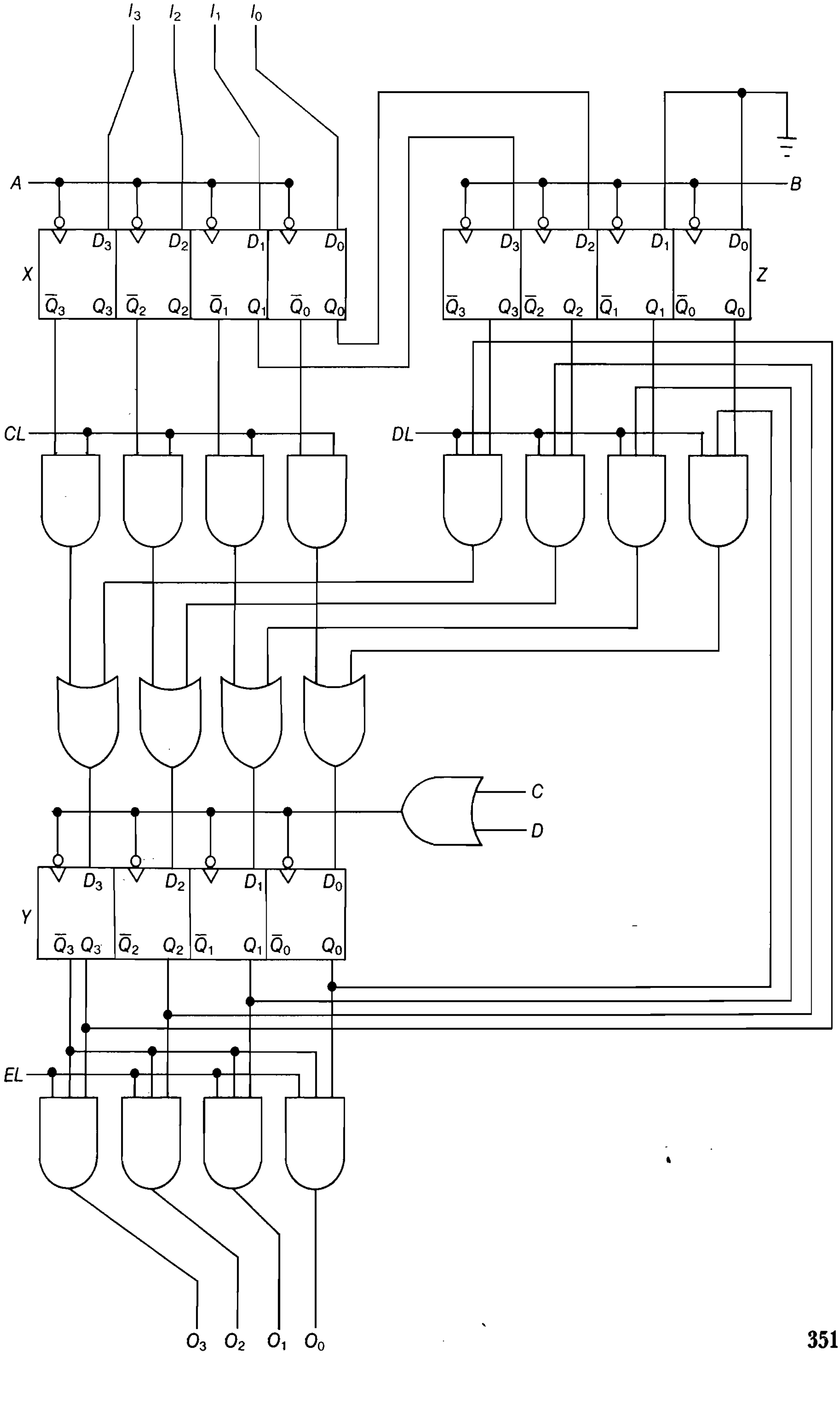
FIGURE 10.58 **Preliminary Step for the Development of Signals: [*a*] I, [*b*] $\overline{X}$ and sl2 X, [*c*] Y and $Y \wedge Z$, and [*d*] 0.**


I_3 I_2 I_1 I_0
I
[a]
D_3 D_2 D_1 D_0
X
$\overline{Q}_3$ Q_3 $\overline{Q}_2$ Q_2 $\overline{Q}_1$ Q_1 $\overline{Q}_0$ Q_0
0 0
$\overline{X}$
sl2 X
[b]
Y
D_3 D_2 D_1 D_0
$\overline{Q}_3$ Q_3 $\overline{Q}_2$ Q_2 $\overline{Q}_1$ Q_1 $\overline{Q}_0$ Q_0
D_3 D_2 D_1 D_0
Z
$\overline{Q}_3$ Q_3 $\overline{Q}_2$ Q_2 $\overline{Q}_1$ Q_1 $\overline{Q}_0$ Q_0
Y
$Y \wedge Z$
0
[c]
[d]

on the output lines, they remain there only during the steps for which they are valid.

The next step in the design is to regulate this algorithm by generating *A*, *B*, *CL*, *C*, *DL*, *D*, and *EL* signals in the proper order as specified by the algorithm. To run the system, a synchronizing sys-

FIGURE 10.59 Complete Data Unit.



tem clock signal is necessary, as shown by the timing diagram of Figure 10.60. The time during which each step is valid is indicated by the level signals. When a level signal is ANDed with the system clock, the corresponding transfer clock pulse is generated. Even though *AL* and *BL* are not used for any of the operations in Figure 10.59, they are considered necessary in the control unit since they are used to generate *A* and *B* clocks, respectively. However, there is no need to generate the *E* clock signal.

We have already seen in Section 10.9 how various register organizations can be used to produce an operation sequencer. This particular algorithm requires only a five-bit ring counter, as shown in Figure 10.61. Using CLR input, the LSB of this ring counter is set and the remaining FFs are reset asynchronously. Consequently the FF outputs start generating the respective level signals in the correct order. The corresponding transfer clock pulses are realized by ANDing the level signals with the system clock.

It is now appropriate to consider several practical aspects about the functioning of this control unit. In order to correctly change Q_5 from a 0 to a 1, the CLR operation must be done in such a way as to assure a full clock period for the first RTL statement. The circuit might be expected to HALT this algorithm at the end of the algo-

FIGURE 10.60 **Timing Diagram of the Control Unit.**

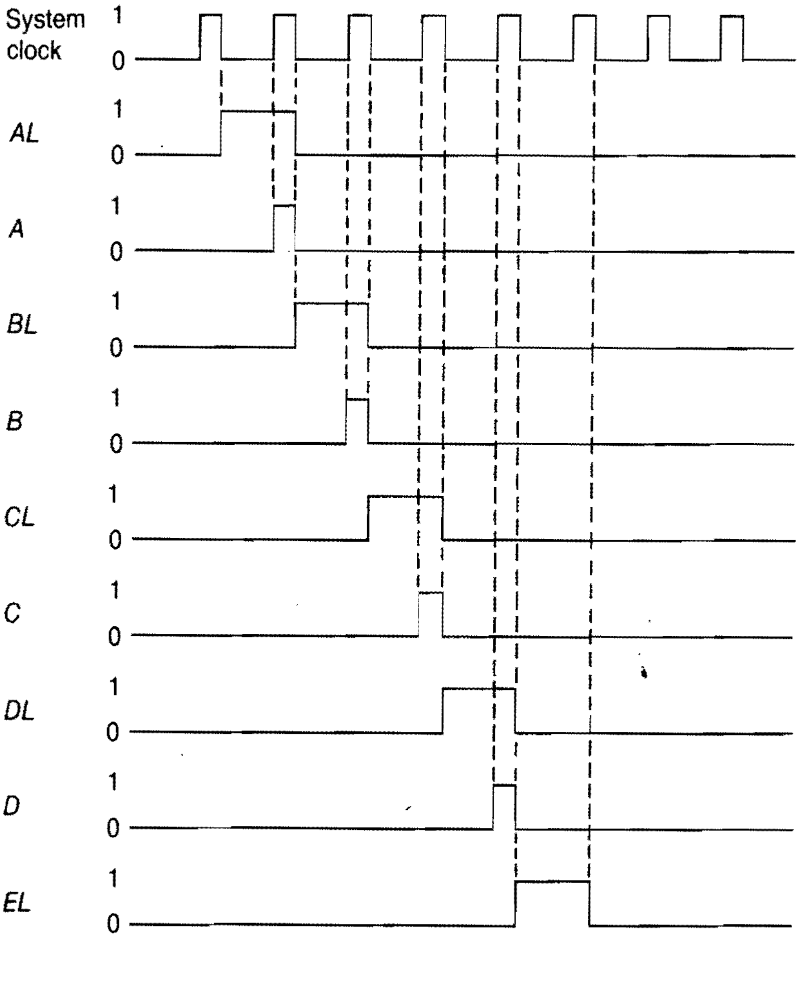
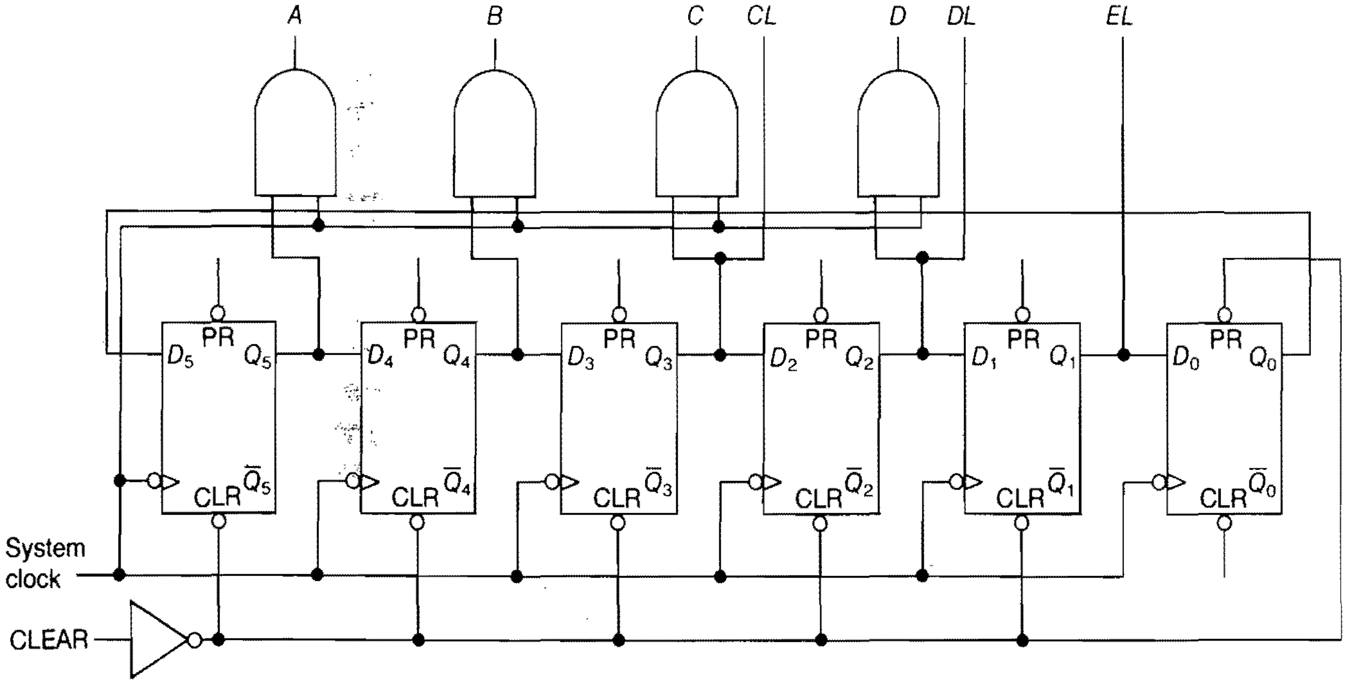


FIGURE 10.61 Controller for the Data Unit of Figure 10.59.



rithm, which could be accomplished by ANDing the system clock with $\bar{Q}_1$ prior to supplying it to various points of the circuit. This arrangement will disable the system clock when $Q_1 = 1$. *EL* becomes high and stops the control circuit from repeating the algorithm. Consequently the outputs would be available indefinitely. In many of the arithmetic circuits of the next chapter we will consider the HALT operation in more detail.

Throughout this section we have preferred to use negative edge-triggered FFs, primarily because transfer clock pulses are easily derived by ANDing the corresponding level signal with the system clock. Generating the transfer clock pulses would not be as easy if positive edge-triggered devices were used. Thus in the event a designer is faced with using a positive edge-triggered device, several modifications are necessary. These modifications involve inverting the system clock before feeding it to the clock input of positive edge-triggered devices. Furthermore, NAND gates rather than AND gates should be used for generating the transfer clock pulses.

10.13 Summary

In covering the application of what is known as the traditional sequential machines, our studies moved through synchronous and asynchronous counters; serial, parallel, and mixed-mode registers; and operation sequencers. An understanding of these functional units was subsequently applied to the development of RTL (register transfer language). This unique tool was then implemented in the design of both data and controller units. An understanding of the concepts presented is necessary, for much of the remaining material in this book is dependent on understanding this chapter.

Problems

1. Determine the state diagram of the three-bit programmed counter whose excitation equations are given as follows:

$$D_1 = Q_3Q_1 + Q_2\bar{Q}_1$$
$$D_2 = \bar{Q}_2(Q_1 + Q_3)$$
$$D_3 = \bar{Q}_3$$

2. Design a three-bit, Gray-code counter using (a) D FFs and (b) JK FFs.
3. Obtain a four-bit presettable asynchronous counter. Discuss its operation and significant characteristics.
4. Design a modulo-12 counter using a modulo-3 and a modulo-4 counter. Discuss its functions and characteristics.
5. The registers of Figure 10.38 store six bits each. A holds 010101 and B holds 001010. List the binary values in A and Q after each shift. Assume that Q was cleared initially.
6. You are given D FFs and several 1-of-4 MUXs and nothing else. Obtain the logic diagram for a three-bit register that is able to do the following: hold the present data, shift right, shift left, and load new data in parallel.
7. Design a four-bit circulate-right shift register using (a) D FFs and (b) JK FFs.
8. Verify the equations of the four-bit down-counter given in Section 10.2.
9. Design a four-bit shift register using JK FFs and a minimum number of assorted gates that performs the shift-left operation.
10. Explain how the unused sequences of Example 10.5 are obtained.
11. Design a divide-by-2048 counter using specific four-bit binary counters. The counter consists of a single FF, Q_A, followed by three cascaded FFs that form a divide-by-8 counter. It has two inputs, A and B, and four standard FF outputs, Q_A through Q_D. The two reset inputs, R_1 and R_2, clear the FFs when both are high. The counter counts in sequence when at least one of the reset inputs is low.
12. Obtain a circuit of as few FFs as possible to sequence 16 different operations.
13. Consider the truth table for a four-bit, Gray code–to–four-bit

binary equivalent conversion. It can be seen that if the MSB of the codes is disregarded, the numbers 8 through 15 will be found to be mirror images of the numbers 0 through 7. Design a parallel Gray code–to–parallel and serial binary converter using a four-bit shift register and only one FF. Explain the detailed functioning of the circuit.

14. Draw the logic diagram of a four-bit register with clocked *JK* FFs having control inputs for the increment, complement, and parallel transfer micro-operations. Show how the 2's complement can be implemented in this register.

15. Design a two-bit counter that counts up when control variable C is a 1 and counts down when C is a 0. No counting occurs when the control variable D is a 0.

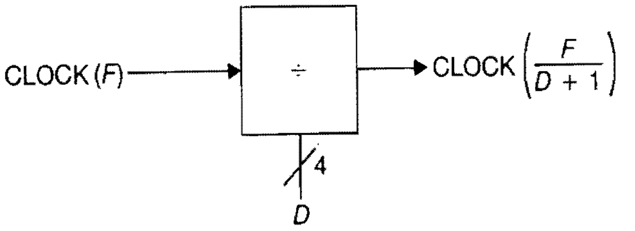
16. Using a four-bit binary counter with synchronous clear and asynchronous load and clock action on the leading edge, complete the necessary circuit to make a two-digit BCD counter. Make it as hardware-efficient as possible.

17. You have available a four-bit adder and a large assortment of gates, counters, multiplexers, and decoders. Design and draw the circuit for a device that will multiply a two-bit (plus sign) sign-magnitude quantity by 3. *Note*: You don't have to use an adder.

18. Using the four-bit adder and any additional circuits you want, design a circuit that will multiply a number $X_2X_1X_0$ (assume signed magnitude with the sign bit handled elsewhere) by 2.5. The result is to be rounded to the next highest integer if the product results in a fractional part.

19. Design the circuit whose block diagram is shown in Figure 10.P1, using decoders, counters, MUXs, FFs, and so on. The circuit has as inputs a clock and four select lines. The select lines determine the number by which the input clock frequency is divided. The output is a clock divided by the select line value. If F is the frequency of the input clock and D is the binary value of the select lines, the output frequency, f, is $F/(D + 1)$.

FIGURE 10.P1



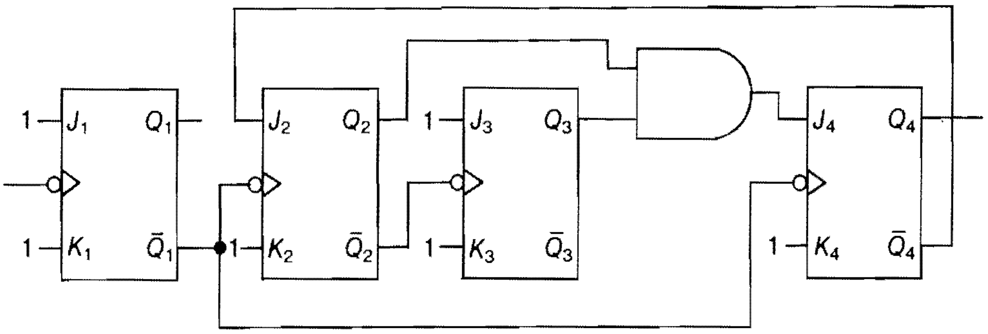
20. Design a two-bit counter that counts up one count at a time when $C = 0$ and counts up two counts per clock when $C = 1$.

21. Using a four-bit universal-shift register, design a 12-bit, serial-in, parallel-out, left-shift register.
22. Using a four-bit universal-shift register, design a 12-bit, parallel-in, serial-out, right-shift register.
23. Using a four-bit universal-shift register, design a 16-bit, universal-shift register.
24. Use four-bit binary counters in parallel and assorted gates to realize a divide-by-39 counter. Show the circuit configuration.
25. Design a typical stage that implements the following logic microoperations:

a.		b.	
	Q_1: $A \leftarrow A \vee \bar{B}$		P_1: $A \leftarrow \bar{A} \vee B$
	Q_2: $A \leftarrow \bar{A} \wedge B$		P_2: $A \leftarrow A \wedge \bar{B}$
	Q_3: $A \leftarrow \overline{A \vee B}$		P_3: $A \leftarrow \overline{A \vee \bar{B}}$
	Q_4: $A \leftarrow \overline{A \wedge B}$		P_4: $A \leftarrow \overline{\bar{A} \wedge B}$
			P_5: $A \leftarrow A \oplus B$

26. Describe the function and characteristics of the counter circuit of Figure 10.P2.

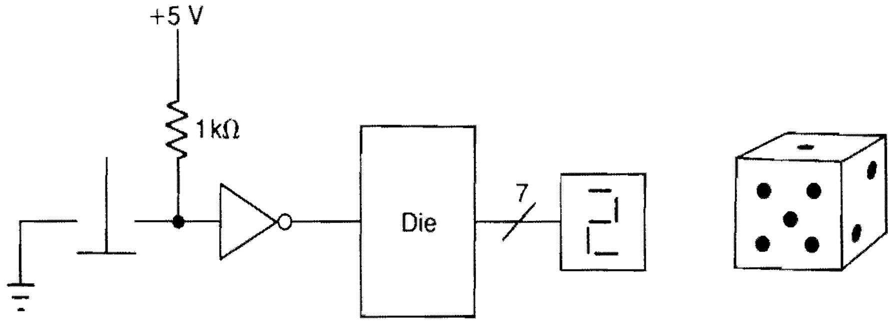
FIGURE 10.P2



27. Obtain the universal state diagram for a four-bit, shift-left feedback register.
28. Obtain the universal state diagram for a four-bit bidirectional feedback register.
29. Design the additional circuitry that would be necessary in the table clock design of Example 10.7 so that the external LEAP input would not be needed. The table clock circuitry would have to take into consideration the consequence of a leap year on its own.
30. Design a wristwatch using BCD counters instead of binary counters. Comment on this design after comparing it with that of Example 10.7.
31. Design the complete data and control units for the algorithm of Problem 25a.
32. Design the complete data and control units for the algorithm of Problem 25b. .

33. Using a seven-segment decoder and a display driver, design an electronic die as illustrated in Figure 10.P3. With equal probability the die should display the digits 1 to 6 on the seven-segment display when a switch is depressed and released. You may use a 1 MHz oscillator and a make/break push-button switch.

FIGURE 10.P3



Suggested Readings

Awwal, A. A. S., and Karim, M. A. "A digital pseudo-random number generator scheme." Paper presented at the Proceedings of the Twenty-Eighth Midwest Symposium for Circuits and Systems, Louisville, Kentucky, August 1985.

Barbacci, M. R. "A comparison of register transfer languages for describing computers and digital systems." *IEEE Trans. Comp.* vol. C-24 (1975): 137.

Bell, C. G.; Eggert, J. L.; Garson, J.; and Williams, P. "The description and use of register transfer modules (RTM's)." *IEEE Trans. Comp.* vol. C-21 (1972): 495.

Berndt, H. "Functional microprogramming as a logic design aid." *IEEE Trans. Comp.* vol. C-19 (1970): 902.

Chambers, W. G., and Jennings, S. M. "Linear equivalence of certain BRM shift-register sequences." *Elect. Lett.* vol. 20 (1984): 1018.

Dadda, L. "Composite parallel counters." *IEEE Trans. Comp.* vol. C-29 (1980): 942.

David, R. "Testing by feedback shift register." *IEEE Trans. Comp.* vol. C-29 (1980): 668.

Divan, D. M.; Hancock, G. C.; Hope, G. S.; and Burton, T. H. "Microprogrammable sequential controller." *IEE Proc. E., Comp. & Dig. Tech.* vol. 131 (1984): 201.

Dormido, S., and Canto, M. A. "Synthesis of generalized parallel counters." *IEEE Trans. Comp.* vol. C-30 (1981): 699.

Dormido, S., and Canto, M. A. "An upper bound for the synthesis of generalized parallel counters." *IEEE Trans. Comp.* vol. C-31 (1982): 802.

Downing, C. P. "Proposal for a digital pseudorandom number generator." *Elect. Lett.* vol. 20 (1984): 435.

Hayes, J. P. *Digital System Design and Microprocessors.* New York: McGraw-Hill, 1984.

Hill, F. J., and Peterson, G. R. *Digital Systems: Hardware Organization and Design.* 2d Ed. New York: Wiley, 1978.

Hill, F. J., and Peterson, G. R. *Digital Logic and Microprocessors.* New York: Wiley, 1984.

Kline, R. *Structured Digital Design Including MSI/LSI Components and Microprocessors.* Englewood Cliffs, N.J.: Prentice-Hall, 1983.

Lo, H. Y.; Lu, J. H.; and Aoki, Y. "Programmable variable-rate up/down counter for generating binary logarithms." *IEE Proc. E., Comp. & Dig. Tech.* vol. 131 (1984): 125.

Manning, F. B., and Fenichel, R. R. "Synchronous counters constructed entirely of J-K flip-flops." *IEEE Trans. Comp.* vol. C-25 (1976): 300.

Rhyne, V. T. "Serial binary-to-decimal and decimal-to-binary conversion." *IEEE Trans. Comp.* vol. C-19 (1970): 808.

Swartzlander, E. E. "Parallel counters." *IEEE Trans. Comp.* vol. C-22 (1973): 1021.

Tien, P. S. "Sequential counter design techniques." *Comp. Des.* vol. 10 (1971): 49.

Vasanthavada, N. S. "Group parity prediction scheme for concurrent testing of linear feedback shift registers." *Elect. Lett.* vol. 21 (1985): 67.

Winkel, D., and Prosser, F. *The Art of Digital Design—An Introduction to Top-Down Design.* Englewood Cliffs, N.J.: Prentice-Hall, 1980.