

EEL 3701: Digital Logic and Computer Systems

***Fundamentals of Computer Engineering:
Logic Design and Microprocessors***
Chapters 1-7

by
Herman Lam, John O'Malley, and A. Antonio Arroyo

Eric M. Schwartz: "This document was generously made available by the author Herman Lam, after it was found to be illegally available on other websites for a fee."

Chapters 8-12 have
been intentionally
removed from this
packet.

CONTENTS

CHAPTER 1	NUMBER SYSTEMS AND REPRESENTATIONS	1
1.1	Introduction	1
1.2	Binary Number System	2
1.2.1	Addition and Subtraction	3
1.2.2	Multiplication and Division	4
1.3	Positional Number Systems	6
1.4	Binary-to-Decimal Conversion	7
1.5	Decimal-to-Binary Conversion	8
1.6	Octal Number System	10
1.7	Hexadecimal Number System	12
1.8	Representing Negative Binary Numbers	14
1.9	Binary-Coded Decimal (BCD) Representation	15
CHAPTER 2	BOOLEAN ALGEBRA	19
2.1	Introduction	19
2.2	Boolean Variables and Logic Values	19
2.3	Fundamental Operations	19
2.3.1	AND Operation	20
2.3.2	OR Operation	21
2.3.3	NOT Operation	22
2.3.4	Operation Hierarchy	22
2.3.5	Summary	23
2.4	Logic Expressions from Truth Tables	24
2.4.1	SOP Expression from a Truth Table	24
2.4.2	POS Expression from a Truth Table	26

2.5	Equivalent Expressions	27
2.6	Boolean Identities	28
2.7	Simplification Using Boolean Identities	30
2.8	Karnaugh Maps	32
	2.8.1 Obtaining an MSOP from a K-Map	34
	2.8.2 Obtaining an MPOS from a K-Map	40
2.9	Don't-Care Outputs	42
2.10	Minimization Summary	43
2.11	Other Common Logic Operations	43
	2.11.1 NAND Operation	43
	2.11.2 NOR Operation	44
	2.11.3 Exclusive OR Operation	45
	2.11.4 Equivalence Operation	46

CHAPTER 3 DIGITAL DESIGN WITH SMALL-SCALE INTEGRATED CIRCUIT ELEMENTS 52

3.1	Introduction	52
3.2	Transistor-Transistor Logic	52
3.3	Logic Conventions	53
	3.3.1 74'00	59
	3.3.2 74'02	62
	3.3.3 74'04	63
	3.3.4 SSI Basic Gate Summary	66
3.4	Synthesis of Digital Circuits	67
	3.4.1 Synthesis Based on the Positive-Logic Convention	67
	3.4.2 Synthesis Based on the Negative-Logic Convention	70
	3.4.3 Synthesis Based on the Mixed-Logic Convention	71
3.5	Summary—Logic Conventions	77

CHAPTER 4 COMBINATIONAL MSI CIRCUIT ELEMENTS 87

4.1	Introduction	87
4.2	Binary Adder and Subtractor	88
	4.2.1 Half Adder and Full Adder	88
	4.2.2 Parallel Adder	90
	4.2.3 MSI Parallel Adders	91
	4.2.4 Binary Subtractor	92
4.3	Magnitude Comparator	94
4.4	Decoder	97
	4.4.1 BCD-to-7-Segment Decoder	99
4.5	Encoder	102
	4.5.1 Priority Encoder	103
4.6	Multiplexer	104
	4.6.1 Three-State Logic Element	106
4.7	Demultiplexer	110

4.8 Design Considerations for Integrated Circuit (IC) Elements	111
4.8.1 TTL Digital Logic Family	113
4.8.2 Parameters for Static Characteristics	114
4.8.3 Parameters for Switching Characteristics	117
CHAPTER 5 SEQUENTIAL MSI CIRCUIT ELEMENTS	123
5.1 Introduction	123
5.2 The Clock Signal	123
5.3 Flip-Flops	125
5.3.1 The J-K Flip-Flop	125
5.3.2 The D Flip-Flop	128
5.3.3 The T Flip-Flop	132
5.3.4 Flip-Flop Conversion	133
5.4 The Unclocked S-R Flip-Flop	139
5.5 Realization of Flip-Flops	141
5.5.1 J-K Flip-Flops	143
5.6 Counters	144
5.6.1 The Design and Realization of Synchronous Counters	146
5.6.2 MSI Counters	156
5.7 Registers	157
5.7.1 Storage Registers	157
5.7.2 Shift Registers	160
5.8 Synchronous versus Asynchronous Designs	163
CHAPTER 6 LSI CIRCUIT ELEMENTS	174
6.1 Introduction	174
6.2 Arithmetic Logic Unit	175
6.3 Look-Ahead Carry Circuits for Adders and ALUs	177
6.3.1 Modified Look-Ahead Carry Approaches	180
6.4 Programmable Logic Array (PLA) and Programmable Array Logic (PAL)	183
6.4.1 Programmable Logic Array	184
6.4.2 Programmable Array Logic	191
6.5 Memories	197
6.5.1 Static RAM	199
6.5.2 Read-Only Memory	204
6.5.3 Dynamic RAM	209
CHAPTER 7 DIGITAL CIRCUIT DESIGN	222
7.1 Introduction	222
7.2 A Model for Digital Circuit Design	223
7.3 Digital Circuit Design Process	224
7.4 Algorithmic State Machine (ASM)	225
7.5 Translation from ASM Chart to Hardware Realization	228

7.5.1 Code Assignment	228
7.5.2 Traditional Method with D Flip-Flops	229
7.5.3 PLA/PAL Method of ASM Realization	233
7.5.4 ROM Method of ASM Realization	233
7.6 An Additional Controller Design	238
7.7 Traditional State Machines	241
7.7.1 Mealy State Machine	242
7.7.2 Moore State Machine	243
7.8 Design Examples	244
7.8.1 Simplified Dynamic RAM Controller	245
7.8.2 Modified Counter	255
7.8.3 Alternative Design for the Modified Counter	260
7.8.4 Hardware Multiplier	262

Number Systems and Representations

1.1 INTRODUCTION

The major topics of this chapter are the number systems and the number representations that are useful in the study of computer engineering. First introduced is the binary number system, which is the number system used in digital computers and other digital applications. After this introduction to the binary number system we will consider the foundation for a positional number system, such as the binary number system and the more familiar decimal number system. Then we will relate the binary and decimal number systems. Next we will briefly consider the octal and hexadecimal number systems since they are used as shorthand systems for the binary number system. After that we will consider three different ways of representing signed binary numbers. Finally, we will consider the binary-coded decimal (BCD) representation.

We are well familiar with the decimal number system. Probably its popularity results from humans having ten fingers. If God had endowed us with, say, eight fingers instead, then perhaps the octal number system would have been the one we studied in grade school. In other words, the extensive use of the decimal number system is mostly a matter of chance and not because it is best for calculations.

Only one number system—the binary number system—is well suited, at present, for direct use in digital applications, including digital computers. In an electronic digital computer application, each different symbol of a number system is represented by a different physical quantity such as a different voltage level. Consequently, the direct use of, say, the decimal number system requires components that utilize ten different physical quantities. Other considerations require these components to be small, light, inexpensive, and also very fast in operation. Components satisfying these latter requirements seldom possess the necessary ten distinct physical quantities. In fact, many of these components have just two distinct physical quantities and therefore are binary components.

To better appreciate the need for the binary number system for digital applications, consider some of the important components used in electronic digital systems. The flip-flop, a storage circuit, is popular because its output voltages can be changed very rapidly. Inherently, a flip-flop produces just two voltage levels and thus is a binary component. Another even more basic component, the transistor, is usually operated in digital applications in just two conditions—saturation and cutoff—and hence it is a binary device. In a magnetic storage element, typically there are just two physical conditions—the two possible directions of residual magnetic flux.

Although the binary number system is best for digital systems, it has the disadvantage of requiring more digits than the decimal number system—generally, more than three times as many. Fortunately, though, having more binary digits is not a serious problem, and can be avoided for hand calculations by using an octal or hexadecimal shorthand. Using octal reduces the number of digits by a factor of approximately three, and using hexadecimal reduces it by a factor of approximately four. As will be seen, using either of these number systems for a shorthand is convenient because it is very easy to convert a binary number into either its octal or hexadecimal equivalent.

1.2 BINARY NUMBER SYSTEM

The binary number system is simpler, in most respects, than the decimal number system because it has only two distinct symbols, 0 and 1, which is much fewer than the ten distinct symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 of the decimal number system. Table 1.1 has the first 18 nonnegative binary integers and the corresponding decimal integers.

TABLE 1.1 DECIMAL-BINARY EQUIVALENCE

Decimal	Binary	Decimal	Binary
0	0	9	1001
1	1	10	1010
2	10	11	1011
3	11	12	1100
4	100	13	1101
5	101	14	1110
6	110	15	1111
7	111	16	10000
8	1000	17	10001

We can derive the binary contents of this table by starting with 0, and continually adding 1, using the rules for binary addition given in Table 1.2. Starting with 0, we add 1 to get the next binary entry of Table 1.1:

$$\begin{array}{r} 0 \\ + 1 \\ \hline 1 \end{array}$$

Then, add 1 to get the next number.

$$\begin{array}{r} 1 \\ +1 \\ \hline 2 \end{array} \quad (\text{wrong!})$$

Wrong! In the binary system, the only valid digits are 0 and 1. From the rules for binary addition in Table 1.2, we see that the next number should be

$$\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array} \quad (\text{sum of 0 and a carry of 1})$$

Continuing counting in this manner, we obtain more table entries.

$$\begin{array}{r} 10 \\ + 1 \\ \hline 11 \\ + 1 \\ \hline 100 \\ + 1 \\ \hline 101 \end{array}$$

And so forth.

TABLE 1.2 BINARY ADDITION

<i>a</i>	<i>b</i>	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

In order to avoid any ambiguity, we will often use a subscript to specify the number system of a particular number. As an illustration,

$$13_{10} = 1101_2$$

The subscript 2 is used for the binary system, 10 for the decimal system, 8 for the octal system, and 16 for the hexadecimal system. Alternatively, a letter B, D, O, or H can be used, respectively, instead of a subscript.

1.2.1 Addition and Subtraction

Addition in binary is quite simple, as is evident from Table 1.2. Some examples follow.

$$\begin{array}{r} 1010 (10_{10}) \\ + 100 (4_{10}) \\ \hline 1110 (14_{10}) \end{array} \quad \begin{array}{r} 1 \\ 101 (5_{10}) \\ + 101 (5_{10}) \\ \hline 1010 (10_{10}) \end{array} \quad \begin{array}{r} 11 \\ 111 (7_{10}) \\ + 11 (3_{10}) \\ \hline 1010 (10_{10}) \end{array} \quad \begin{array}{r} 11 \\ 110 (6_{10}) \\ + 111 (7_{10}) \\ \hline 10000 (16_{10}) \end{array} \quad \begin{array}{r} 11111 \\ 11011.101 (27.625_{10}) \\ + 1010.111 (10.875_{10}) \\ \hline 100110.100 (38.5_{10}) \end{array}$$

TABLE 1.3 BINARY SUBTRACTION

a	b	Difference	Borrow	
0	0	0	0	
0	1	1	1	a (minuend)
1	0	1	0	$-b$ (subtrahend)
1	1	0	0	

Notice in the third example that the second column has the addition of two 1s plus a 1 carry from the first column of addition. The sum of $11_2 = 3_{10}$ is shown as a 1 in the sum plus a 1 carry in the third column from the right. The fourth example shows that one difficulty in adding binary numbers is the large number of carries that often occur. In this example, the carry from the second column of addition ($1 + 1 + 1 + 1 = 100$) extends over two places. Such an extension is not unusual in the addition of more than two binary numbers, but is rare in the addition of decimal numbers. The last example shows that the addition rules are the same for numbers that are not integers. In other words, the presence of the binary point (not decimal point) has no effect on the addition rules.

Binary subtraction is more difficult than addition, but no more so than decimal subtraction is more difficult than decimal addition. Table 1.3 specifies the rules for binary subtraction. A simple example of binary subtraction is the following:

$$\begin{array}{r} 1110 \text{ (} 14_{10}\text{)} \\ - 100 \text{ (} 4_{10}\text{)} \\ \hline 1010 \text{ (} 10_{10}\text{)} \end{array}$$

Just as in decimal subtraction, the principal difficulty in binary subtraction occurs when a borrow is required. This happens in the subtraction of a 1 from a 0. Then, it is necessary to borrow a 1 from the "next" digit. If this digit is a 1, then it is changed to a 0, and the borrow taken, as in the following example:

$$\begin{array}{r} 1101 \text{ (} 13_{10}\text{)} \\ - 10 \text{ (} 2_{10}\text{)} \\ \hline 1011 \text{ (} 11_{10}\text{)} \end{array} \quad \rightarrow \quad \begin{array}{r} 10^1 01 \text{ borrow} \\ - 10 \\ \hline 1011 \text{ (} 11_{10}\text{)} \end{array}$$

If the "next" digit in the minuend is a 0, then the borrow must extend over several digits, just as in decimal subtraction. As an illustration,

$$\begin{array}{r} 1100001 \\ - 100011 \\ \hline 111110 \end{array} \quad \rightarrow \quad \begin{array}{r} 10111^1 01 \text{ borrow} \\ - 1000 11 \\ \hline 1111 10 \end{array}$$

Note that the second, third, fourth, and fifth digits of the minuend change in the generation of the borrow for the sixth digit.

1.2.2 Multiplication and Division

Table 1.4 specifies the rules for binary multiplication. As is evident, 0 times a digit is 0, and 1 times a digit is that digit. Not shown by the table is the fact that the rule for

TABLE 1.4 BINARY MULTIPLICATION

<i>a</i>	<i>b</i>	Product
0	0	0
0	1	0
1	0	0
1	1	1

$$\begin{array}{r} a \\ \times b \\ \hline \end{array}$$

positioning the binary point in a product is the same as that for the decimal point in the decimal number system.

EXAMPLE 1.1

Multiply 1101.1 by 1010.1.

Solution.

$$\begin{array}{r} 1101.1 \text{ multiplicand} \\ \times 1010.1 \text{ multiplier} \\ \hline 11011 \\ 00000 \\ 11011 \\ 00000 \\ 11011 \\ \hline 10001101.11 \text{ product} \end{array}$$

■ ■

From Example 1.1, we see that, in general, for each 1 in the multiplier we write the multiplicand and then shift left one position before writing again. For each 0 in the multiplier, we just shift left.

Division in binary is much simpler than in decimal. The process of binary division is best illustrated by an example. We will check the result of the last example by dividing the multiplier into the product to see whether the quotient is the original multiplicand.

EXAMPLE 1.2

Divide 10001101.11 by 1010.1.

Solution.

$$\begin{array}{r} \text{divisor } 1010.1 \overline{) 10001101.11} \\ \underline{10101} \\ 11100 \\ \underline{10101} \\ 11111 \\ \underline{10101} \\ 10101 \\ \underline{10101} \\ 00000 \end{array}$$

■ ■

In general, the first step in binary division is to mark off, starting from the left in the dividend, a number of digits equal to the number of digits in the divisor. If the number marked off is larger, then the divisor divides into this number exactly once. If the number marked off is smaller, we include another digit of the dividend. Then, the divisor always divides into this number exactly once. We subtract, and then continue the process just as in decimal division. We position the binary point of the quotient in the same way as in decimal division.

1.3 POSITIONAL NUMBER SYSTEMS

Now that we have studied the binary number system, we will provide a foundation for what we have done by briefly considering *positional number systems*, starting with the familiar decimal number system.

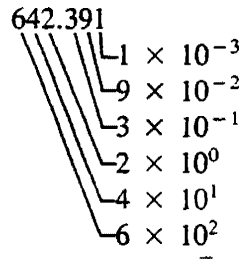
As we know, the decimal number system has ten different symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The number of symbols of a number system is the *base* or *radix*. So, the base of the decimal system is 10. Note that the symbol for the base is a combination of the first two symbols; there is no single symbol for the 10 base of the decimal system.

In a *positional* number system (and all the number systems we will study are positional number systems), the value of a number depends not only on the symbols used but also on the *positions* of the symbols. For example, 3674_{10} and 3746_{10} contain the same symbols but have entirely different values. The significance of the positions is that they correspond to different powers of the base, even though we do not show these powers because we write these numbers in contracted form. As an illustration, 3674_{10} is a contraction of $3 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 4 \times 10^0$. Graphically, it can be shown as follows:

$$\begin{array}{r} 3674 \\ \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \\ 3 \times 10^3 \\ 6 \times 10^2 \\ 7 \times 10^1 \\ 4 \times 10^0 \end{array}$$

Although we do not show the radix point (e.g., decimal point) for integer numbers, we assume it to be just to the right of the rightmost digit. Then, the first position to the *left* of the radix point corresponds to the base raised to the power of 0, the second position corresponds to the base raised to the power of 1, the third position corresponds to the base raised to the power of 2, and so on.

This concept of powers also applies to digits positioned to the *right* of a radix point, except the powers are *negative* powers of the base. The first position to the right corresponds to the base raised to the power of -1 , the second position to the base raised to the power of -2 , and so on. So, a radix point in a number designates the separation of the negative powers of the base from the nonnegative powers. As an illustration, consider the number 642.391_{10} , which is a contraction of $6 \times 10^2 + 4 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 9 \times 10^{-2} + 1 \times 10^{-3}$. Graphically, it can be shown as follows:



The decimal point in the contracted number is between digits 2 and 3—the multipliers of 10^0 and 10^{-1} , respectively. The digit just to the left of the decimal point is the multiplier of the zero power of the base, and the digit just to the right is the multiplier of the negative one power of the base.

This number representation applies not only to the decimal number system but also to all number systems we will consider. In general, a number $a_n a_{n-1} \cdots a_1 a_0 . a_{-1} a_{-2} \cdots$, in which the a_i 's are digits of a positional number system with radix r , is representable as

$$a_n r^n + a_{n-1} r^{n-1} + \cdots + a_2 r^2 + a_1 r^1 + a_0 r^0 + a_{-1} r^{-1} + a_{-2} r^{-2} + \cdots$$

in which $0 \leq a_i \leq r - 1$. In using this representation in our following considerations of the binary, octal, and hexadecimal number systems, we will, for convenience, always express the r 's and their powers in decimal.

Let us now see how this representation applies to the binary number system. The base or radix, r , is 2. Thus, this system has just two distinct digits 0 and 1 since

$$0 \leq a_i \leq r - 1$$

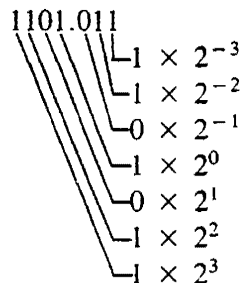
$$0 \leq a_i \leq 2 - 1$$

$$0 \leq a_i \leq 1$$

An example of a binary number is 1101.011, which is a contraction of

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}$$

Graphically, it can be shown as



1.4 BINARY-TO-DECIMAL CONVERSION

In our study we will often want to convert a binary number into its decimal equivalent. One easy way of doing this is to express the binary number in expanded form as powers of the base 2 expressed in decimal, and then add in decimal. For example, 11101.011_2

is in decimal: $1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 29.375$. Graphically,

$$\begin{array}{r}
 11101.011 \\
 \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \quad \swarrow \\
 1 \times 2^{-3} = 1 \times \frac{1}{8} = 0.125 \\
 1 \times 2^{-2} = 1 \times \frac{1}{4} = 0.25 \\
 0 \times 2^{-1} = 0 \times \frac{1}{2} = 0 \\
 1 \times 2^0 = 1 \times 1 = 1 \\
 0 \times 2^1 = 0 \times 2 = 0 \\
 1 \times 2^2 = 1 \times 4 = 4 \\
 1 \times 2^3 = 1 \times 8 = 8 \\
 1 \times 2^4 = 1 \times 16 = 16 \\
 \hline
 29.375
 \end{array}$$

Although some convenient algorithms exist for rapid conversion from binary to decimal, they are not worth the effort to learn unless we do a considerable amount of binary-to-decimal conversion.

1.5 DECIMAL-TO-BINARY CONVERSION

Frequently, we will need to convert a decimal integer into an equivalent binary integer. To do this, we repeatedly divide in decimal by the base 2, saving the remainders which will form the desired binary number. Specifically, using decimal division, we divide the decimal integer by 2. Next, we place the remainder of 0 or 1 to one side, and then divide the integer part of the quotient by 2. Again, we place the remainder from this second division to one side, and divide 2 into the integer part of the second quotient, and so on. We repeat this process until we obtain a zero quotient. Then, by arranging the remainders in reverse order, we obtain the desired binary equivalent.

EXAMPLE 1.3

Convert the decimal number 117 into its binary equivalent.

Solution.

	remainder
2 $\overline{)117}$	
2 $\overline{)58}$	1
2 $\overline{)29}$	0
2 $\overline{)14}$	1
2 $\overline{)7}$	0
2 $\overline{)3}$	1
2 $\overline{)1}$	1
0	1

By arranging the remainders in the reverse order in which we obtained them, we have the result that 1110101 is the binary equivalent of decimal 117. ■ ■

The justification for this rule is that in converting a decimal integer into its binary equivalent, we are finding the a_i 's of

$$\cdots a_3 \times 2^3 + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$$

in which each a_i is either 0 or 1. The first division by 2 results in

$$\cdots a_3 \times 2^2 + a_2 \times 2^1 + a_1 + \overbrace{a_0/2}^{\text{remainder}}$$

With a little thought, we see that the part of the quotient including a_1 and to the left is an integer, and a_0 is the remainder of the division. Similarly, when we divide this integer part by 2, the remainder from this second division is a_1 , and so on. So, the remainders from the repeated divisions are the a_i 's with an order beginning with the least significant, which means that we must reverse this order to get the equivalent binary number.

We must use a different rule for converting the *fractional* part of a decimal number into an equivalent binary fraction. For it, we multiply by 2, instead of divide, and we save the integer parts of the products. Specifically, using decimal multiplication, we multiply the decimal fraction by 2. Next, we place the integer part of the product, which is 0 or 1, to one side, and then multiply the fractional part of the product by 2. Again, we place the resulting integer part to one side, and multiply the fractional part by 2, and so on. We repeat this procedure until the fractional part of the product is zero or until the number of binary digits is that desired. The integer parts of the products form the corresponding binary fraction, with the integers arranged in the order in which we obtained them.

EXAMPLE 1.4

Convert decimal 0.8125 to binary.

Solution.

	integer
$2 \times 0.8125 = 1.625$	1
$2 \times 0.625 = 1.25$	1
$2 \times 0.25 = 0.5$	0
$2 \times 0.5 = 1$	1

So, 0.8125 in decimal is 0.1101 in binary. ■ ■

This procedure is easy to justify. We convert a decimal fraction into binary by finding the a_i 's of

$$a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + a_{-3} \times 2^{-3} + \cdots$$

in which each a_i is either 0 or 1. Multiplying this expression by 2 results in

$$a_{-1} + \underbrace{a_{-2} \times 2^{-1} + a_{-3} \times 2^{-2} + \cdots}_{\text{fraction}}$$

Note that the portion of the product to the right of a_{-1} is less than 1 and so is a fraction. Also, a_{-1} is the integer part of the product. If the product has an integer part of 1, then a_{-1} is 1. If the integer part is 0, then a_{-1} is 0.

To find a_{-2} , we use only the fractional part of the first product:

$$a_{-2} \times 2^{-1} + a_{-3} \times 2^{-2} + \dots$$

Multiplying this by 2 gives

$$a_{-2} + \underbrace{a_{-3} \times 2^{-1} + \dots}_{\text{fraction}}$$

The portion of the product to the right of a_{-2} is the fraction part of the second product, and a_{-2} is the integer part. Repetition of this process gives the remainder of the binary digits. Incidentally, a terminating fraction in decimal may not be terminating in binary (for example, $0.6_{10} = 0.10011001100110011 \dots_2$), but a terminating fraction in binary is always terminating in decimal.

For a decimal number with both integer and fraction parts, we use the integer rule on the integer part and the fraction rule on the fraction part. Then, we combine parts into one number. For example, for the binary equivalent of 27.875_{10} , we apply the integer rule to obtain $11011_2 = 27_{10}$ and the fraction rule to obtain $0.111_2 = 0.875_{10}$, and then combine the binary parts to obtain $27.875_{10} = 11011.111_2$.

1.6 OCTAL NUMBER SYSTEM

Since the base of the octal number system is 8, this system has eight symbols, which are 0, 1, 2, 3, 4, 5, 6, and 7. Table 1.5 has the correspondences among the decimal, binary, and octal number systems for the first 20 nonnegative integers.

As to be expected, to count in the octal number system, we simply add 1 to the current number to obtain the next number:

$$\dots 5 + 1 = 6, 6 + 1 = 7, 7 + 1 = 10, 10 + 1 = 11, \dots, 16 + 1 = 17, 17 + 1 = 20, 20 + 1 = 21, \dots$$

TABLE 1.5 DECIMAL-BINARY-OCTAL EQUIVALENCE

Decimal	Binary	Octal	Decimal	Binary	Octal
0	0	0	10	1010	12
1	1	1	11	1011	13
2	10	2	12	1100	14
3	11	3	13	1101	15
4	100	4	14	1110	16
5	101	5	15	1111	17
6	110	6	16	10000	20
7	111	7	17	10001	21
8	1000	10	18	10010	22
9	1001	11	19	10011	23

Note that $7 + 1 = 10$. Since this is a base 8 system, adding 1 to 7 generates a carry to the next digit in an addition of two octal numbers. With this in mind, we can readily add and subtract in octal. In our use of octal as a binary shorthand, we will never have to multiply or divide in octal.

In octal-to-binary conversion, we replace each octal digit with its three-digit binary equivalent. In the resulting binary number, we can, of course, ignore any leading zeros in the integer part or trailing zeros in the fraction part.

EXAMPLE 1.5

Convert 345.5602_8 into its binary equivalent.

Solution.

$$\begin{array}{ccccccc} 3 & 4 & 5 & \cdot & 5 & 6 & 0 & 2 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 011 & 100 & 101 & \cdot & 101 & 110 & 000 & 010 \end{array}$$

$$\text{So, } 345.5602_8 = 11100101.10111000001_2. \quad \blacksquare \blacksquare$$

The conversion from binary to octal is just the reverse of the above. Specifically, we group the binary digits (called *bits* for short) by threes from the right and from the left of the binary point. Then, we replace each group by its octal equivalent. In the binary fraction part, we add zeros, if needed, to complete the rightmost group.

EXAMPLE 1.6

Determine the octal equivalent of 11001110.0101101_2 .

Solution. We group the bits by threes as follows:

$$011 \ 001 \ 110 \ . \ 010 \ 110 \ 100$$

We do not really need to add the leading zero in the integer part, but we do need to add the two trailing zeros to the fraction part to complete the last group. After making this grouping, we replace each group with its octal equivalent.

$$\begin{array}{ccccccc} 011 & 001 & 110 & \cdot & 010 & 110 & 100 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 3 & 1 & 6 & \cdot & 2 & 6 & 4 \end{array}$$

So, the equivalent octal number is 316.264 . If we had not added the two trailing zeros, we would have the erroneous 316.261 . $\blacksquare \blacksquare$

The justification for the conversion rules is that the grouping of the binary digits into groups of three forms, in effect, numbers times powers of 8. This is perhaps best understood from a specific example. Again, consider $011 \ 001 \ 110.010 \ 110 \ 100$, which is a contraction of

$$011 \times 2^6 + 001 \times 2^3 + 110 \times 2^0 + 010 \times 2^{-3} + 110 \times 2^{-6} + 100 \times 2^{-9}$$

Converting this to octal, term by term, results in

$$3 \times 8^2 + 1 \times 8^1 + 6 \times 8^0 + 2 \times 8^{-1} + 6 \times 8^{-2} + 4 \times 8^{-3}$$

or 316.264 in contracted form. So, the grouping of bits by threes allows the powers of the binary base to be directly converted into powers of the octal base.

As is evident, octal numbers are a convenient shorthand for representing binary numbers because the equivalent octal numbers have only approximately one-third as many digits, and the conversion between binary and octal is easy and fast. There is another justification for this shorthand. In some computers, the basic binary numbers operated on have parts that are integer multiples of 3 bits. In other words, each part is either 3 bits, 6 bits, 9 bits, or some other integer multiple of 3 bits. So, conversion of these binary numbers to octal provides an exact conversion for each part, which is a convenience.

1.7 HEXADEcimal NUMBER SYSTEM

In some computers the parts of binary numbers considered as units are multiples of 4 bits instead of 3 bits, making the octal shorthand unsuitable. But, the hexadecimal number system is useful because in converting from binary to hexadecimal, we group the bits by fours.

Since the base of the hexadecimal number system is 16, this system has 16 different symbols. These symbols are the ten decimal digits plus the first six letters of the alphabet: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Table 1.6 has the correspondences for the first 20 nonnegative integers of the decimal, binary, and hexadecimal number systems. By starting with 0 and adding 1 consecutively, we can generate the hexadecimal entries for this table. In doing this, note that since the hexadecimal number system is a base 16 system, a carry is not generated for the next digit until the digit sum exceeds F (15_{10}).

TABLE 1.6 DECIMAL-BINARY-HEXADEcimal EQUIVALENCE

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0	0	10	1010	A
1	1	1	11	1011	B
2	10	2	12	1100	C
3	11	3	13	1101	D
4	100	4	14	1110	E
5	101	5	15	1111	F
6	110	6	16	10000	10
7	111	7	17	10001	11
8	1000	8	18	10010	12
9	1001	9	19	10011	13

The conversion between binary and hexadecimal is similar to that for octal except that the grouping of bits is by fours instead of by threes. Justification for this conversion follows from that for the octal-binary conversion.

EXAMPLE 1.7

Convert 11100101101.1111010111_2 to hexadecimal.

Solution. We group the bits by fours to the right and to the left of the binary point, and then substitute the hexadecimal equivalent for each group.

$$\begin{array}{cccccccc} 0111 & 0010 & 1101 & . & 1111 & 0101 & 1100 & \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\ 7 & 2 & D & . & F & 5 & C & \end{array}$$

So, the equivalent hexadecimal number is 72D.F5C. ■ ■

EXAMPLE 1.8

Convert $B9A4.E6C_{16}$ to binary.

Solution. In the conversion from hexadecimal to binary, we convert each hexadecimal digit into its 4-bit equivalent.

$$\begin{array}{cccccccc} B & 9 & A & 4 & . & E & 6 & C \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 1011 & 1001 & 1010 & 0100 & . & 1110 & 0110 & 1100 \end{array}$$

So, $B9A4.E6C_{16} = 1011100110100100.1110011011_2$. ■ ■

The easiest way to convert a hexadecimal number to a decimal number is to express the hexadecimal number in expanded form as powers of the base 16 and then add in decimal.

EXAMPLE 1.9

Convert $B63.4C_{16}$ to decimal.

Solution.

$$\begin{aligned} B63.4C_{16} &= 11 \times 16^2 + 6 \times 16^1 + 3 \times 16^0 + 4 \times 16^{-1} + 12 \times 16^{-2} \\ &= 2816 + 96 + 3 + 0.25 + 0.046875 \\ &= 2915.296875_{10} \end{aligned}$$

The decimal-to-hexadecimal conversion parallels that of Examples 1.3 and 1.4 for decimal-to-binary conversion. But, of course, the division and multiplication are by 16 instead of 2, and each decimal intermediate result must be converted to a hexadecimal integer.

EXAMPLE 1.10

Convert $43,976.3046875_{10}$ to hexadecimal.

Solution.

	remainder	
16	<u>43,976</u>	
16	<u>2748</u>	8
16	<u>171</u>	12→C
16	<u>10</u>	11→B
0		10→A

So, $43,976_{10} = ABC8_{16}$. Continuing,

	integer
$16 \times 0.3046875 = 4.875$	4
$16 \times 0.875 = 14.0$	14→E

So,

$$43,976.3046875_{10} = ABC8.4E_{16} \quad \blacksquare \blacksquare$$

1.8 REPRESENTING NEGATIVE BINARY NUMBERS

In a digital computer, or any other digital system, there are no positive (+) or negative (−) signs for representing positive and negative binary numbers. Instead, the signs of binary numbers must somehow be represented by 1s and 0s within the forms of the binary numbers. Three forms of signed binary numbers are popular: *signed magnitude*, *1s complement*, and *2s complement*. We will consider these three forms as related to integer-type or fixed-point numbers in which the first bit in a number is the sign bit and each binary point is assumed to be immediately to the right of the least-significant digit. But the concepts apply as well to numbers that are not integers.

In the signed-magnitude form, a positive or negative binary number is represented by a sign bit followed by the magnitude in binary. For example, for an 8-bit representation, the decimal number 13 is represented by the binary number 00001101 and −5 by 10000101, with, as is conventional, a 0 bit representing a positive sign and a 1 bit a negative sign. Incidentally, there are two representations for the number zero, a positive zero and a negative zero, which for 8 bits are 00000000 and 10000000, respectively.

In the 1s-complement representation, positive numbers are the same as in the signed-magnitude representation, but negative numbers differ for they are represented in 1s-complement form. To find the 1s-complement representation of a negative number, all we have to do is to consider the number to be positive, and then change all 0s to 1s, including the sign bit, and all 1s to 0s. For example, to find the 8-bit representation of decimal number −13, we first find the 8-bit binary equivalent of +13, which is 00001101. Then we change the 0s to 1s and the 1s to 0s, and obtain 11110010, which is the desired complement representation.

In the 2s-complement representation, positive numbers are the same as in the signed-magnitude representation, but negative numbers differ; they are represented in 2s-complement form. The 2s complement of a negative number is simply the 1s complement plus 1. As an illustration, the 8-bit 1s complement of -13 is 11110010, as has been shown. Therefore, the 2s-complement representation is $11110010 + 1 = 11110011$. Applications of the 2s-complement representation, along with 1s-complement and signed-magnitude representations, will be demonstrated throughout this text.

1.9 BINARY-CODED DECIMAL (BCD) REPRESENTATION

The *binary-coded decimal (BCD)* representation is a code for decimal numbers. It is an alternative to converting a decimal number to its binary equivalent. Many applications require the inputting and/or displaying of decimal digits. In these applications, it is often convenient to store the data in the BCD representation.

In the BCD representation, each decimal digit of a decimal number is coded into binary. Since there are ten decimal digits, a binary representation of these digits requires 4 bits. Only 10 of the 16 combinations of the 4 bits are required for a BCD code. Of the many possible such codes, the most popular is the 8421 code illustrated in Table 1.7. In this code, each decimal digit is converted into its 4-bit binary equivalent, and then the groups of four bits are concatenated.

EXAMPLE 1.11

What is the representation for the decimal number 7963 in the 8421 code?

Solution. In the coding of 7963, we replace each digit by the corresponding 4 bits from Table 1.7. The result is

7	9	6	3	decimal number
↓	↓	↓	↓	
0111	1001	0110	0011	8421 code

Finally, we concatenate the four groups of four bits:

0111100101100011

■ ■

The 8421 code is so popular as a BCD code that when a reference is made to the BCD code, we should assume that it is the 8421 code unless another BCD code is specified.

TABLE 1.7. 8421 BCD CODE

zero	0000	five	0101
one	0001	six	0110
two	0010	seven	0111
three	0011	eight	1000
four	0100	nine	1001

SUPPLEMENTARY READING (see Bibliography)

[Bartee 85], [Hill 81], [Mano 84], [McCluskey 75], [Roth 85]

PROBLEMS

- 1.1. Perform the indicated operations on the following binary numbers:
- | | |
|--|---|
| (a) $1101 + 110$ | (b) $\begin{array}{r} 111.011 \\ 1111.11 \\ + 111001.1 \\ \hline \end{array}$ |
| (c) $10011 - 110$ | (d) $1001.1101 - 11100.001$ |
| (e) $\begin{array}{r} 1111.1 \\ 1011.11 \\ + 1101.111 \\ \hline \end{array}$ | (f) $11001.01 - 111.1$ |
- 1.2. Repeat Problem 1.1 for
- | | |
|---|--|
| (a) $1101.1 + 111.101$ | (b) $\begin{array}{r} 111.01 \\ 1011.11 \\ + 11110.01 \\ \hline \end{array}$ |
| (c) $11011 - 101$ | (d) $1001.01 - 10110.101$ |
| (e) $\begin{array}{r} 111.101 \\ 1110.11 \\ 11111.01 \\ + 101110.011 \\ \hline \end{array}$ | (f) $1100.01 - 111.1$ |
- 1.3. Perform the indicated binary multiplications and divisions.
- | | |
|---------------------------------|------------------------------|
| (a) 101×1101 | (b) 11011×1100.1 |
| (c) $111011.01 \times 1111.001$ | (d) $10100 \div 100$ |
| (e) $100011.11 \div 101.1$ | (f) $101010001.1 \div 11011$ |
- 1.4. Repeat Problem 1.3 for
- | | |
|------------------------------------|---------------------------------------|
| (a) 1101×110011 | (b) 11010.1×1101.01 |
| (c) $1011101.011 \times 10110.101$ | (d) $1001000 \div 110$ |
| (e) $10000000.01 \div 1101.1$ | (f) $110101001.11101 \div 110011.101$ |
- 1.5. Convert the following binary numbers to decimal:
- | | | |
|------------|---------------|------------------|
| (a) 100111 | (b) 10101.101 | (c) 10001001.001 |
|------------|---------------|------------------|
- 1.6. Repeat Problem 1.5 for
- | | | |
|-------------|--------------|---------------------|
| (a) 1100111 | (b) 1110.111 | (c) 1100110101.0111 |
|-------------|--------------|---------------------|
- 1.7. Convert the following decimal numbers to binary:
- | | |
|----------|---------------------|
| (a) 146 | (b) 1928.875 |
| (c) 14.6 | (d) $19\frac{2}{3}$ |
- 1.8. Repeat Problem 1.7 for
- | | | |
|----------|--------------|-----------|
| (a) 1689 | (b) 1430.625 | (c) 178.3 |
|----------|--------------|-----------|
- 1.9. Perform the indicated operations on the following octal numbers:
- | | |
|-----------------|-------------------------|
| (a) $324 + 564$ | (b) $710.145 + 217.633$ |
|-----------------|-------------------------|

- (c) $352 - 163$ (d) $1236.47 - 765.2377$
 (e) $40002.3 - 675.77$ (f) 46.23
 327.12
 $+ 4652.327$

(g) $473.63 - 754.321$

1.10. Repeat Problem 1.9 for

- (a) $472 + 326$ (b) $410.23 + 367.55$
 (c) $456 - 324$ (d) $4723.636 - 724.647$
 (e) 47.653 (f) $5346.72 - 600321.1$
 327.734
 $+ 467.772$

1.11. Convert the following octal numbers to decimal:

- (a) 110011 (b) 1234.54 (c) 7006.302

1.12. Repeat Problem 1.11 for

- (a) 30076 (b) 6403.2 (c) 400602.244

1.13. Convert the following decimal numbers to octal:

- (a) 3094 (b) 2906.5625
 (c) 250.3 (d) $37\frac{1}{8}$

1.14. Repeat Problem 1.13 for

- (a) 1000 (b) 100
 (c) 2345.46875 (d) $46\frac{3}{13}$

1.15. Convert the following binary numbers to octal:

- (a) 101101 (b) 110010111011.01011011
 (c) 1000111101.01

1.16. Repeat Problem 1.15 for

- (a) 1101011 (b) 10111111011.0110111
 (c) 110111.0111

1.17. Convert the following octal numbers to binary:

- (a) 76002 (b) 773406.245

1.18. Repeat Problem 1.17 for

- (a) 23077 (b) 432.7066

1.19. Convert the following hexadecimal numbers to decimal:

- (a) 4A6 (b) ABD.C8

1.20. Repeat Problem 1.19 for

- (a) CAB (b) FE6.8C4

1.21. Convert the following binary numbers to hexadecimal:

- (a) 110111001 (b) 110001101.10011

1.22. Repeat Problem 1.21 for

- (a) 1001110101 (b) 1100111001.1000101

1.23. Convert the following hexadecimal numbers to binary:

- (a) CD3F (b) 8F2A.5EF

1.24. Repeat Problem 1.23 for

- (a) ACD8 (b) 3FD2.ACD

1.25. Convert the following decimal numbers to hexadecimal:

- (a) 329.5 (b) 7495.34 (c) $17\frac{1}{7}$

- 1.26. Repeat Problem 1.25 for
 (a) 492.25 (b) 9294.85 (c) $73\frac{13}{21}$
- 1.27. Determine the 1s complements of the following binary numbers:
 (a) 01011010 (b) 01110011 (c) 01111111 (d) 0011110100111101
- 1.28. Repeat Problem 1.27 for
 (a) 00110111 (b) 01111011 (c) 01010101 (d) 0111101101111111
- 1.29. Determine the 2s complements of the numbers of Problem 1.27.
- 1.30. Determine the 2s complements of the numbers of Problem 1.28.
- 1.31. Determine the decimal equivalents of the following binary numbers in each of the three representations: signed magnitude, 1s complement, and 2s complement.
 (a) 101010 (b) 111111
 (c) 100000 (d) 101101
 (e) 111100 (f) 110011
- 1.32. Repeat Problem 1.31 for
 (a) 1101111 (b) 0000000
 (c) 1111111 (d) 1000000
 (e) 1110111 (f) 1110100
- 1.33. For 8 bits, show the three binary representations of signed magnitude, 1s complement, and 2s complement for the following decimal numbers:
 (a) -23 (b) +0 (c) -0
 (d) -41 (e) -54 (f) -37
- 1.34. Repeat Problem 1.33 for
 (a) -63 (b) -18 (c) -3
 (d) -60 (e) -49 (f) -34
- 1.35. Code the decimal numbers 954, 672, 1394, and 67,942 into the 8421 BCD code.
- 1.36. Repeat Problem 1.35 for the decimal numbers 876, 4594, and 43,298.
- 1.37. Given the following BCD representations, find the corresponding decimal numbers.
 (a) 10000110 (b) 001101000010 (c) 1001100001010111
- 1.38. Repeat Problem 1.37 for
 (a) 10010111 (b) 100000101001 (c) 0010010001010111

Boolean Algebra

2.1 INTRODUCTION

This chapter presents *Boolean algebra*, which is the basic mathematics required for the study of the design of digital circuits. Since these circuits are also called switching circuits, the Boolean algebra used in their design is sometimes called *switching algebra*. Boolean algebra provides a systematic method for describing and simplifying the logic processes at the basic component level of digital design.

2.2 BOOLEAN VARIABLES AND LOGIC VALUES

A *Boolean variable*, unlike an ordinary algebraic variable, has only one of two values: true or false, called *logic values*. A Boolean variable can be viewed as representing a statement that can be only true or false. For example, the Boolean variable A may represent the statement "Frank has red hair." Obviously, variable A can have only the logic value of either true or false. In other words, the statement "Frank has red hair" is either true (Frank does have red hair) or false (Frank does not have red hair). If the statement is false, Frank has another color hair or is bald.

A Boolean variable may be a function of other Boolean variables. Then, as will be seen in the next chapter, we will find it convenient to call the function variable an *output variable* and the other variables *input variables*. The value of the output variable depends not only on the values of the input variables, but also on the operations of the function.

2.3 FUNDAMENTAL OPERATIONS

Boolean algebra has three fundamental operations: AND, OR, and NOT. We will now consider each of them.

2.3.1 AND Operation

The AND operation is represented by the symbol “ \cdot ” as in

$$Z = A \cdot B$$

which in words is “Z is equal to A AND B.” This equation specifies that the Boolean variable Z is true if both A is true *and* B is true. Otherwise, Z is false.

The AND operation can be precisely defined in a *truth table* (also called a *logic table*), as follows:

A	B	Z = A · B
F	F	F
F	T	F
T	F	F
T	T	T

A truth table is simply a listing of all possible values of the input variables (here, A and B) with the corresponding output variable values resulting from the operation (or operations). It is apparent from this table that Z is true (T) if and only if both A is true (T) *and* B is true (T). Otherwise, Z is false (F).

For a physical aid to the understanding of the AND operation, consider the circuit of Fig. 2.1, containing a voltage source, two switches A and B, and a light bulb Z. For this circuit, a variable A might represent the statement that “switch A is closed,” a variable B the statement that “switch B is closed,” and a variable Z the statement that “the light bulb Z is lit.” Clearly, if either statement A or B is false, then statement Z is false, since both switches must be closed for the voltage source to energize the light bulb.

The AND operation of Boolean algebra is not related to the multiplication operation of ordinary arithmetic, although the symbols are the same. Generally, Boolean algebra has many of the same elements as ordinary algebra, and has many of the same symbols and terminology. But there are some important differences. Therefore, in working with Boolean algebra, we should not rely on our past study of ordinary algebra but rather rely solely on the definitions given here.

The AND operation applies to any number of input variables. For three input variables the AND truth table is

A	B	C	Z = A · B · C
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	T

Note that Z is false unless all input variables are true, which is a general property of the AND operation, regardless of the number of input variables.

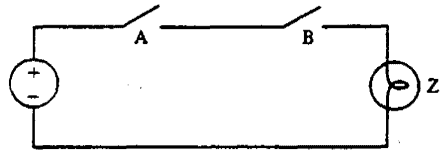


Figure 2.1 Illustrating AND.

For this truth table, as for all truth tables, there is a standard way of assigning the T's and F's in the input variable columns. All columns start with F. Then, in the rightmost input variable column, the F's and T's alternate. In the next column they alternate by twos, and in the first column they alternate by fours. If there were a fourth column, they would alternate by eights, and so on.

To have all possible combinations of input variable values, a truth table must have a number of rows equal to 2^n , in which n is the number of input variables. So, a two-variable truth table has 4 rows, a three-variable one has 8 rows, a four-variable one has 16 rows, and so on.

2.3.2 OR Operation

The OR operation is represented by the symbol “+” as in

$$Z = A + B$$

which in words is “Z is equal to A OR B.” This equation specifies that Z is true if *either* A or B is true. Otherwise, Z is false. In a truth table, this specification is

A	B	Z = A + B
F	F	F
F	T	T
T	F	T
T	T	T

Although the symbols are the same, the OR operation of Boolean algebra is not related to the addition operation of ordinary arithmetic.

For an illustration of the OR operation, consider the circuit of Fig. 2.2. As with Fig. 2.1, let variable A correspond to the statement that “switch A is closed,” variable B to the statement that “switch B is closed,” and variable Z to the statement that “light bulb Z is lit.” Obviously, if either statement A or B is true, then statement Z is true, since the closing of either switch completes a circuit from the voltage source to the light bulb.

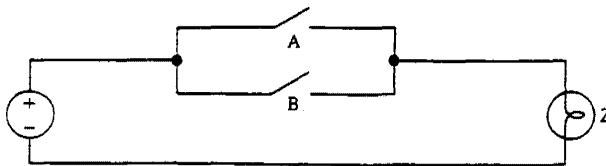


Figure 2.2 Illustrating OR.

The OR operation can extend to any number of input variables. For three input variables the OR truth table is

A	B	C	Z = A + B + C
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	T
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	T

Note that Z is true unless all input variables are false, which is a general property of the OR operation, regardless of the number of input variables.

2.3.3 NOT Operation

The NOT operation is designated by an overline, as in

$$Z = \bar{A}$$

which in words is that "Z is equal to A NOT." The NOT operation is defined as follows:

A	Z = \bar{A}
F	T
T	F

The NOT operation is a *complement* operation. Since a Boolean variable can have only one of two values, the complement of one logic value is the other value: $\bar{F} = T$ and $\bar{T} = F$.

The NOT operation can apply to more than one variable, and in fact to an entire expression. Then, the overline extends over more than one variable. As an illustration, the complement of $A \cdot \bar{B} + C$ has the designation $\overline{A \cdot \bar{B} + C}$.

2.3.4 Operation Hierarchy

To evaluate an expression, we have to know the priority of the operations. Suppose, for example, we want the value of the expression $\bar{A} + B \cdot C$ for $A = F$, $B = T$, and $C = F$. Since all three operations are present, we need to know the order in which to perform them. The priority or hierarchy rule is: perform the *individual* variable NOT operations first, the AND operations second, and the OR operations last. Therefore, this expression evaluates to

$$\bar{A} + B \cdot C = \bar{F} + T \cdot F = T + T \cdot F = T + F = T$$

We can override the priority rule by using parentheses, as in

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

The parentheses cause the OR operation to be performed before the AND operation.

If an expression is complemented, we must evaluate the expression before complementing; that is, we do not complement first.

The hierarchy of operations and the evaluation of expressions can best be understood from some examples.

EXAMPLE 2.1

Evaluate the following expressions for $A = T$, $B = F$, $C = T$, and $D = T$: (a) $A \cdot \bar{B} + C$, (b) $(\bar{A} + B) \cdot (C + \bar{B} \cdot D)$, and (c) $A \cdot \bar{B} \cdot (C + D \cdot \bar{E})$.

Solution.

$$(a) A \cdot \bar{B} + C = T \cdot \bar{F} + T = T \cdot T + T = T + T = T$$

We could have obtained the T result immediately after inserting the values, because T (from the C) OR anything is equal to T.

$$\begin{aligned} (b) (\bar{A} + B) \cdot (C + \bar{B} \cdot D) &= (\bar{T} + F) \cdot (T + \bar{F} \cdot T) \\ &= (F + F) \cdot (T + T \cdot T) \\ &= (F + F) \cdot (T + T) \\ &= (F) \cdot (T) = F \end{aligned}$$

We could have stopped after finding that one factor is false, since F AND anything is F.

$$\begin{aligned} (c) A \cdot \bar{B} \cdot (C + D \cdot \bar{E}) &= T \cdot \bar{F} \cdot (T + T \cdot \bar{E}) \\ &= T \cdot T \cdot (T + T \cdot \bar{E}) = T \cdot T \cdot T = T \end{aligned}$$

Note that the quantity in parentheses is true because $T + \text{anything} = T$. ■■

2.3.5 Summary

A summary of the three basic Boolean operations is given in Table 2.1. Remember that a Boolean variable can have only one of two logic values: true (T) or false (F). For convenience, though, we will often represent the logic value true (T) by the symbol 1, and the logic value false (F) by the symbol 0, as in Table 2.1. *Important:* Although the symbol 1 looks like the numeric 1 and the symbol 0 looks like the numeric 0, they are logic values representing true and false, respectively. So we cannot add or subtract or perform any other ordinary arithmetic operations on the logic values 0 and 1.

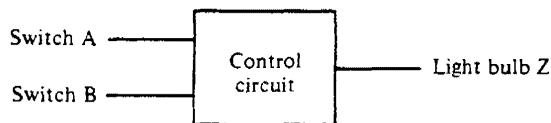
TABLE 2.1

A	B	$A \cdot B$	$A + B$	\bar{A}	\bar{B}
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

2.4 LOGIC EXPRESSIONS FROM TRUTH TABLES

We have used truth tables in defining the AND, OR, and NOT operations. Truth tables are also useful in designing digital circuits. Specifically, in a design we form a truth table from the design specifications and then determine a logic expression from the truth table.

Consider the design of a simple digital circuit for controlling the operation of a light bulb Z with two switches A and B. Suppose the light bulb is to be on when both switches are on (closed), and also when both switches are off (open). Otherwise (one switch off and the other on), the light bulb is to be off.



We can easily represent this description of the circuit operation with a truth table in which a 0 implies that a switch or the light bulb is off, and a 1 implies that a switch or the light bulb is on.

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

The next step is to obtain a logic expression from this truth table. There are two methods we can use, and they give different types of logic expressions. One type is a sum-of-products (SOP) expression, and the other is a product-of-sums (POS) expression. An SOP expression comprises several “product” (AND) terms “summed” (ORed) together. For example, the following expression is in SOP form:

$$A \cdot \bar{B} + \bar{A} \cdot B + A \cdot B$$

A POS expression comprises several “sum” (OR) factors “multiplied” (ANDed) together, as in

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

2.4.1 SOP Expression from a Truth Table

To obtain an SOP expression for the truth table of the light control circuit, we need to derive a logic expression for Z in the form of

$$Z = f(A, B)$$

that will express the condition for which Z is true (the light bulb is on). In other words, what are the combinations of the inputs A and B for which Z is true (1)? From the truth

table, we see that Z is true when A is false *and* B is false, *or* when A is true *and* B is true. Written in a formalized manner, this is

$$Z = \bar{A} \cdot \bar{B} + A \cdot B$$

We can generalize from this example and derive the following rules for obtaining an SOP expression from a truth table:


1. Identify each row in which the output function is *true* (1).
2. For each of these rows, write an AND term of all the input variables, applying complements such that the term evaluates to true for the variable values of that row. Incidentally, each of these terms is commonly called a *minterm*. (In general, a minterm is a term that contains all the input variables.)
3. OR together all the AND terms (minterms) found in step 2.

The resulting SOP expression is called a *canonical sum-of-products expression* (CSOP) or, more simply, a *minterm expansion*. Note that in a CSOP each minterm contributes one and only one true (1) value. Examples will help us better understand these rules.

EXAMPLE 2.2

Given the following truth table, find a CSOP expression for Z.

A	B	Z
0	0	1
0	1	0
1	0	1
1	1	1



Solution.


1. Rows 1, 3, and 4 have true (1) outputs.
2. The minterms corresponding to these rows are $\bar{A} \cdot \bar{B}$, $A \cdot \bar{B}$, and $A \cdot B$, respectively.
3. ORing these minterms results in $Z = \bar{A}\bar{B} + A\bar{B} + AB$.

Note that the AND operator symbol “ \cdot ” is omitted in the final result. The AND operator is implied by the adjacent placement of the variables. We will often omit this symbol when its omission does not cause any confusion. ■ ■

EXAMPLE 2.3

Find a CSOP expression for the Z specified in the following truth table.

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Solution.

1. Rows 1, 3, 7, and 8 have true (1) outputs.
2. The corresponding minterms are, respectively, $\overline{A}\overline{B}\overline{C}$, $\overline{A}B\overline{C}$, $A\overline{B}\overline{C}$, and ABC .
3. ORing the minterms results in $Z = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$. ■■

2.4.2 POS Expression from a Truth Table

From the same truth table for the described light control circuit, we can also obtain a POS expression for Z. Recall that this truth table is

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1



The rules for obtaining a POS expression from a truth table are as follows:

1. Identify each row in the truth table for which the output Z is *false* (0).
2. For each of these rows, write an OR factor of all the input variables, using complements such that the factor evaluates to false for the variable values of that row. Incidentally, each of these factors is commonly called a *maxterm*. (In general, a maxterm is a factor that contains all the input variables.)
3. AND together all the OR factors (maxterms) found in step 2.

The resulting POS expression is called a *canonical product-of-sums expression* (CPOS) or, more simply, a *maxterm expansion*. Note that in a CPOS, each maxterm contributes one and only one false (0) value.

We will now apply these rules to the light control circuit example.

1. Z is false (0) for rows 2 and 3.
2. The corresponding maxterms are $A + \overline{B}$ and $\overline{A} + B$.
3. ANDing the maxterms, we obtain $Z = (A + \overline{B})(\overline{A} + B)$.

EXAMPLE 2.4

Find a CPOS expression for Z.

A	B	Z
0	0	1
0	1	0
1	0	1
1	1	1

Solution.

1. Row 2 is the only row with a false (0) output.
2. The maxterm corresponding to this row is $A + \bar{B}$.
3. Since there is only one maxterm, Z is equal to it: $Z = A + \bar{B}$. ■ ■

EXAMPLE 2.5

Find a CPOS expression for Z.

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Solution.

1. Rows 2, 4, 5, and 6 have false (0) outputs.
2. The corresponding maxterms are, respectively, $A + B + \bar{C}$, $A + \bar{B} + \bar{C}$, $\bar{A} + B + C$, and $\bar{A} + B + \bar{C}$.
3. By ANDing the maxterms, we obtain

$$Z = (A + B + \bar{C})(A + \bar{B} + \bar{C})(\bar{A} + B + C)(\bar{A} + B + \bar{C}) \quad \blacksquare \blacksquare$$

2.5 EQUIVALENT EXPRESSIONS

Two expressions are *equivalent* if for every set of values of the input variables, the two expressions evaluate to the *same* value. As an illustration, for the light control circuit we obtained a CSOP expression $\bar{A}\bar{B} + AB$ and an equivalent CPOS expression $(A + \bar{B})(\bar{A} + B)$. We can verify that these two expressions are equivalent by using a truth table.

A	B	$\bar{A} \cdot \bar{B} + A \cdot B$	$(A + \bar{B})(\bar{A} + B)$
0	0	$\bar{0} \cdot \bar{0} + 0 \cdot 0 = 1$	$(0 + \bar{0})(\bar{0} + 0) = 1$
0	1	$\bar{0} \cdot \bar{1} + 0 \cdot 1 = 0$	$(0 + \bar{1})(\bar{0} + 1) = 0$
1	0	$\bar{1} \cdot \bar{0} + 1 \cdot 0 = 0$	$(1 + \bar{0})(\bar{1} + 0) = 0$
1	1	$\bar{1} \cdot \bar{1} + 1 \cdot 1 = 1$	$(1 + \bar{1})(\bar{1} + 1) = 1$

Since the two expressions evaluate to the same values for each combination of input variable values, they are equivalent, or

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

For another illustration, recall the truth table for the OR operation:

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

The CSOP expression for Z is $AB + \bar{A}\bar{B} + A\bar{B}$, and the CPOS expression is $A + B$. Obviously, the two expressions are equivalent. In other words,

$$Z = \bar{A}\bar{B} + A\bar{B} + AB = A + B$$

Again, we can show this equivalence with a truth table. But there is another method of proof. We can *reduce* the expression $\bar{A}\bar{B} + A\bar{B} + AB$ to the expression $A + B$ by using *Boolean identities*:

$$\begin{aligned} \bar{A}\bar{B} + A\bar{B} + AB &= \bar{A}\bar{B} + (\bar{A}\bar{B} + A\bar{B}) = \bar{A}\bar{B} + A(\bar{B} + B) \\ &= \bar{A}\bar{B} + A(1) = \bar{A}\bar{B} + A = B + A = A + B \end{aligned}$$

Obviously, some of us may not fully comprehend this manipulation at this point since Boolean identities are discussed in the next section.

2.6 BOOLEAN IDENTITIES

Generally, we want to reduce any logic expression to its simplest form since a simpler expression usually requires fewer hardware components for its implementation. Boolean identities are often useful in a reduction.

Table 2.2 contains the Boolean identities most important to digital design. Identities 1–5 are the fundamental relations of Boolean algebra and provide the foundation for

TABLE 2.2 BOOLEAN IDENTITIES

	(a)	(b)
1.	$\bar{\bar{A}} = A$	$\bar{\bar{A}} = A$
2.	$A + \text{false} = A \quad (A + 0 = A)$	$A \cdot \text{true} = A \quad (A \cdot 1 = A)$
3.	$A + \text{true} = \text{true} \quad (A + 1 = 1)$	$A \cdot \text{false} = \text{false} \quad (A \cdot 0 = 0)$
4.	$A + \bar{A} = 1$	$A \cdot \bar{A} = 0$
5.	$A + \bar{A} = \text{true} \quad (A + \bar{A} = 1)$	$A \cdot \bar{A} = \text{false} \quad (A \cdot \bar{A} = 0)$
6.	$A + B = B + A$	$A \cdot B = B \cdot A$
7.	$A + B + C = (A + B) + C = A + (B + C)$	$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$
8.	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + B \cdot C = (A + B)(A + C)$
9.	$\overline{A + B} = \bar{A} \cdot \bar{B}$	$\overline{A \cdot B} = \bar{A} + \bar{B}$
10.	$A \cdot B + \bar{A} \cdot \bar{B} = A \oplus B$	$(A + B)(A + \bar{B}) = A$
11.	$A + A \cdot B = A$	$A(A + B) = A$
12.	$A(\bar{A} + B) = A \cdot B$	$A + \bar{A} \cdot B = A + B$
13.	$A \cdot B + \bar{A} \cdot C + B \cdot C = A \cdot B + \bar{A} \cdot C$	$(A + B)(\bar{A} + C)(B + C) = (A + B)(\bar{A} + C)$

Boolean manipulation. Identities 6, 7, and 8 are, respectively, the commutative, associative, and distributive laws of Boolean algebra. Except for 8(b), they are very similar to the corresponding laws of ordinary algebra. Identity 8(b) is unfamiliar only if we think in terms of “+” as addition and “·” as multiplication operators. Remember that “+” in Boolean algebra is the OR operator and so has properties that are not the same as the addition operator of ordinary algebra. One of these properties enables us to “multiply through” or “factor out,” as in

$$A + BCD = (A + B)(A + C)(A + D)$$

Identities 9(a) and (b) are the well-known DeMorgan’s laws. Note that both forms of DeMorgan’s laws have the complement of an entire expression, and the effect of this complementing is to change each “+” to a “·” and each “·” to a “+” and to complement each *literal*. (A literal is a generic term for a variable or its complement.) Identities 10 through 13 are other identities frequently used in digital design. Identities 10(a) and (b) are the bases for several systematic Boolean simplification methods, one of which we will study in a following section. In each of these identities a variable may be replaced by an expression, and the identity will still be valid.

Using truth tables, we can prove all these Boolean identities, exhaustively. As an illustration, we will use this method to prove DeMorgan’s law of 9(a) for three variables. The result is

A	B	C	$\overline{A + B + C}$	$\overline{A} \cdot \overline{B} \cdot \overline{C}$
0	0	0	1	1
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

Since the two output columns are identical, then $\overline{A + B + C} = \overline{A} \cdot \overline{B} \cdot \overline{C}$.

Another method of proving these identities is with Boolean manipulation, using other proven identities. As an illustration, we will prove identity 12(a) in this manner, with the assumption that the other identities we use have been proved.

$$\begin{aligned} A(\overline{A} + B) &= A\overline{A} + AB && \text{by the distributive law 8(a)} \\ A\overline{A} + AB &= 0 + AB && \text{by 5(b)} \\ 0 + AB &= AB && \text{by 2(a)} \end{aligned}$$

So, we have proved that $A(\overline{A} + B) = AB$ by using Boolean manipulation to obtain the second expression from the first.

EXAMPLE 2.6

Using Boolean manipulation, prove identity 11(a), assuming that the preceding identities are valid.

Solution.

$$\begin{aligned} A + AB &= (A \cdot 1) + AB && 2(b) \\ (A \cdot 1) + AB &= A \cdot (1 + B) && 8(a) \\ A \cdot (1 + B) &= A \cdot (1) && 3(a) \\ A \cdot (1) &= A && 2(b) \end{aligned}$$

■ ■

To gain facility with the use of these identities, you should use Boolean manipulation to work through the proofs of some of the other identities of Table 2.2.

Note in Table 2.2 that the Boolean identities are divided into two columns. The identities in column (a) and the corresponding identities in column (b) are *duals* of each other. A dual of an identity is formed by replacing each AND operator with an OR operator, each OR operator with an AND operator, each true (1) with false (0), and each false with true. This procedure for finding a dual of an expression is identical to the application of DeMorgan's laws, except that there is no complementing of literals. In general, in Boolean algebra a dual of a theorem is another valid theorem.

2.7 SIMPLIFICATION USING BOOLEAN IDENTITIES

The primary use of Boolean manipulation is for simplifying Boolean expressions to obtain simpler expressions for digital design. A simpler expression usually results in a simpler implementation with digital devices.

In using Boolean manipulation we are usually trying to obtain either a *minimum sum of products* (MSOP) or a *minimum product of sums* (MPOS) that is equivalent to the original expression. An MSOP is an SOP that is equivalent to the original expression, and has no more terms or literals than any other equivalent SOP. In other words, in an MSOP, the number of terms and the number of literals are the minimum possible for any equivalent SOP. There may be several equivalent MSOP expressions. Once in a while, though, the requirements for a minimum number of terms and a minimum number of literals are mutually exclusive—they conflict. Then, there is no clear meaning of what an MSOP is.

The MSOP definition applies to an MPOS with the substitution of “factors” for “terms.” An MPOS and an equivalent MSOP may have a different number of literals, as we will see for the expression of the next example. Also, the number of factors of the MPOS may be different than the number of terms of the MSOP.

EXAMPLE 2.7

Find an MSOP for $F = \bar{X}W + Y + \bar{Z}(Y + \bar{X}W)$.

Solution.

$$\begin{aligned} F &= \bar{X}W + Y + \bar{Z}(Y + \bar{X}W) \\ &= \bar{X}W + Y + \bar{Z}Y + \bar{Z}\bar{X}W && 8(a) \\ &= (\bar{X}W + \bar{Z}\bar{X}W) + (Y + \bar{Z}Y) && 6(a), 7(a) \\ &= \bar{X}W(1 + \bar{Z}) + Y(1 + \bar{Z}) && 8(a) \end{aligned}$$

$$= \overline{X}W(1) + Y(1) \quad 3(a)$$

$$= \overline{X}W + Y \quad 2(b)$$

So, $\overline{X}W + Y$ is an MSOP expression for F .

Sometimes we want an MPOS even though obtaining an MSOP is easier. We can often obtain an MPOS from an MSOP by taking the dual of the MSOP, multiplying out, and then taking the dual again, making obvious simplifications in multiplying out. Taking the dual twice gives us an equivalent expression. Doing this for the $\overline{X}W + Y$ of this example, we obtain $(\overline{X} + W)Y$ from taking the dual. Then, we multiply, getting $\overline{X}Y + WY$. Finally, we take the dual again, obtaining $(\overline{X} + Y)(W + Y)$, which is an MPOS expression for F . Note that this MPOS has one more literal than the MSOP. ■ ■

EXAMPLE 2.8

Find an MSOP for $F = \overline{W}XY + \overline{W}XZ + (\overline{Y} + \overline{Z})$.

Solution.

$$F = \overline{W}XY + \overline{W}XZ + (\overline{Y} + \overline{Z})$$

$$= \overline{W}X(Y + Z) + (\overline{Y} + \overline{Z}) \quad 8(a)$$

$$= \overline{W}X + (\overline{Y} + \overline{Z}) \quad 12(b)$$

$$= \overline{W}X + \overline{Y}\overline{Z} \quad 9(a) \quad \blacksquare \blacksquare$$

EXAMPLE 2.9

Find an MSOP for $F = (\overline{X} + WY + \overline{Z})(\overline{X} + WY + Z)$.

Solution.

$$F = (\overline{X} + WY + \overline{Z})(\overline{X} + WY + Z)$$

$$= (\overline{X} + WY) + (\overline{Z} \cdot Z) \quad 8(b)$$

$$= (\overline{X} + WY) + 0 \quad 5(b)$$

$$= \overline{X} + WY \quad 2(a)$$

Or, *directly* use 10(b) and obtain $F = \overline{X} + WY$ in one step. ■ ■

EXAMPLE 2.10

Find an MSOP for $F = V\overline{W}XY + VWYZ + V\overline{X}YZ$.

Solution.

$$F = V\overline{W}XY + VWYZ + V\overline{X}YZ$$

$$= VY(\overline{W}X + WZ + \overline{X}Z) \quad 8(a)$$

$$= VY(\overline{W}X + Z(W + \overline{X})) \quad 8(a)$$

Since $(W + \overline{X}) = \overline{\overline{W}X} = \overline{W}X \quad 1(a), 9(a)$

$$F = VY(\overline{W}X + Z(\overline{W}X))$$

$$= VY(\overline{W}X + Z) \quad 12(b)$$

$$= VY\overline{W}X + VYZ \quad 8(a) \quad \blacksquare \blacksquare$$

EXAMPLE 2.11

Find an MSOP for $F = W\bar{X}\bar{Y} + \bar{W}XY + W\bar{X}Z + WYZ + XYZ$.

Solution. Where do we begin? By using identity 13(a), we can obtain

$$W\bar{X}Z + XYZ + WYZ = W\bar{X}Z + XYZ \quad (1)$$

$$W\bar{X}\bar{Y} + WYZ + W\bar{X}Z = W\bar{X}\bar{Y} + WYZ \quad (2)$$

$$WYZ + \bar{W}XY + XYZ = WYZ + \bar{W}XY \quad (3)$$

We can use equation (1) to eliminate the WYZ term. But with it eliminated, we cannot use equations (2) and (3) since this term is needed in these equations. Therefore, we should not use equation (1) and eliminate WYZ . Instead, we will use equations (2) and (3). Note that to do this, we use the same term (WYZ) twice, but that is acceptable because, from identity 4(a), $WYZ = WYZ + WYZ$. By using equations (2) and (3) we eliminate the $W\bar{X}Z$ and XYZ terms, obtaining $F = W\bar{X}\bar{Y} + \bar{W}XY + WYZ$. ■ ■

The difficulties with simplification we had in this last example should make us think that there must be a better way to simplify an expression than by using Boolean manipulation. With it, we have difficulty determining where to begin, how to proceed, and when to know that we are finished and have an MSOP or an MPOS. There are simply no definitive rules for Boolean manipulation. Consequently, we will use Boolean manipulation only for reducing simple Boolean expressions, or Boolean expressions that we cannot conveniently simplify with other methods. For simplifying expressions of up to six variables, we will usually prefer to use the graphical method of the next section.

2.8 KARNAUGH MAPS

A Karnaugh map (K-map) is a convenient graphical method for obtaining an MSOP or MPOS of a Boolean expression of three, four, or five variables—and possibly, though not conveniently, six or seven variables. All the information of a truth table of a function is contained in a K-map. In other words, there is a one-to-one correspondence between a truth table and its K-map.

A K-map contains squares, one for each row of a corresponding truth table. So, a two-variable K-map has 4 squares, a three-variable K-map has 8 squares, a four-variable K-map has 16 squares, and so on. The values of the input variables are arranged in a certain order along two edges of a K-map, and from these values we can determine the corresponding truth table row for each square. The function values are inserted into the squares.

Figure 2.3 illustrates truth tables and corresponding K-maps for some two-variable, three-variable, and four-variable functions. Note, for example, in Fig. 2.3(a), that the top left square corresponds to the first truth table row since both the A and B input values are 0 for this square. And, from the 1 inside the square, we see that $Z = 1$ for $A = 0$ and $B = 0$. Similarly, the bottom left square corresponds to the second row of the truth

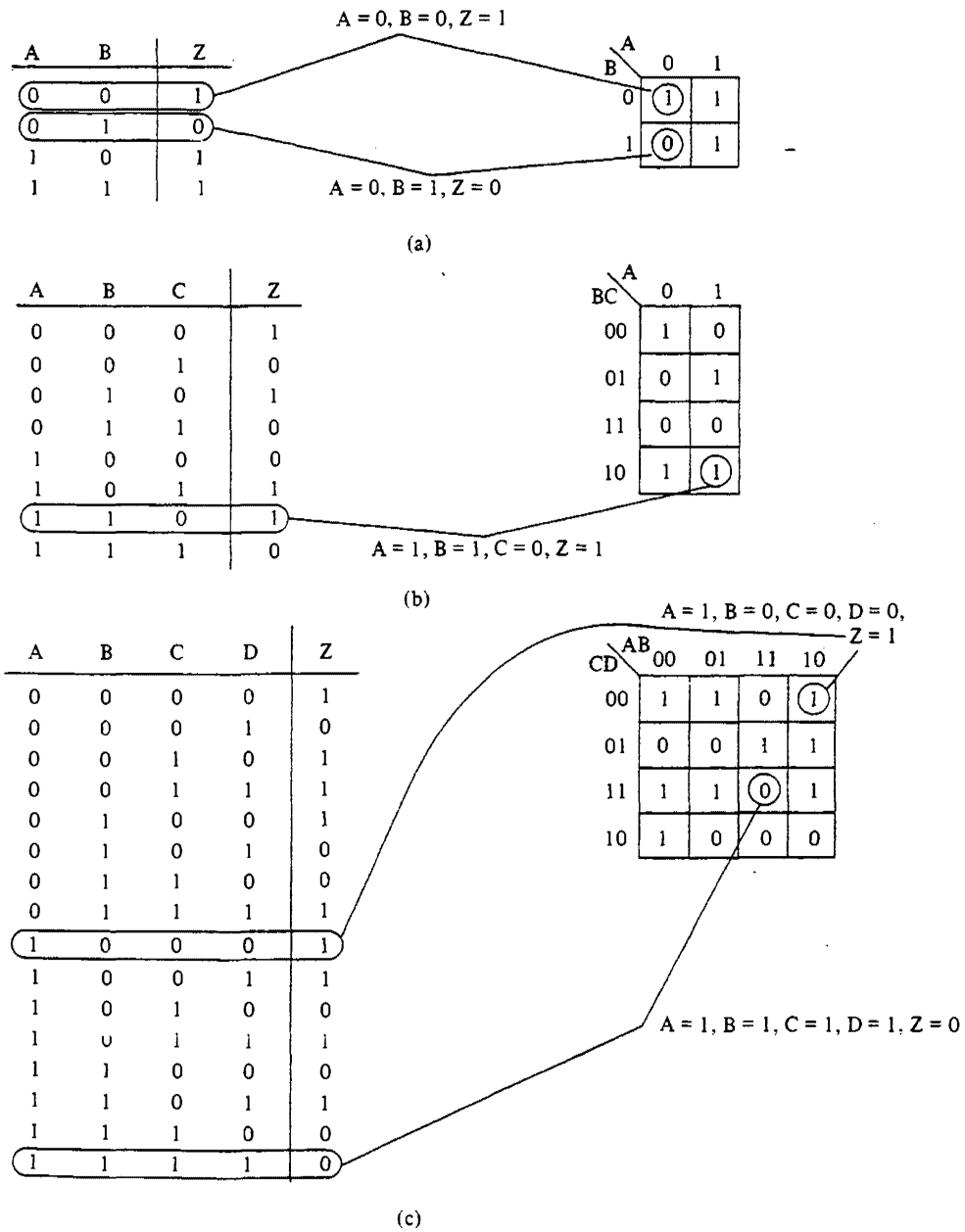


Figure 2.3 Truth tables and corresponding K-maps.

table, where $A = 0, B = 1, Z = 0$, and so on. In the three-variable K-map of Fig. 2.3(b), the bottom right square, for example, corresponds to the seventh row of the truth table, where $A = 1, B = 1, C = 0$, and $Z = 1$. In the four-variable K-map of Fig. 2.3(c), the top right corner square corresponds to the ninth row of the corresponding truth table, where $A = 1, B = 0, C = 0, D = 0$, and $Z = 1$. Also, the square in the third row and column of the K-map corresponds to the last row of the truth table, where $A = 1, B = 1, C = 1, D = 1$, and $Z = 0$, and so on. You should verify the other entries for each K-map.

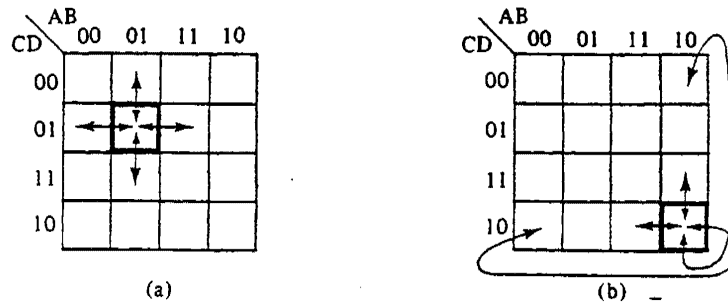


Figure 2.4 Adjacent squares.

We can consider a K-map to be an alternative representation of a truth table. Given a truth table, we can draw the corresponding K-map, or given a K-map we can form the corresponding truth table. But there is one very important difference: In a K-map the values of the input variables are arranged such that for any two physically adjacent squares, only *one* of the input variables has a different value. For example, in Fig. 2.4(a), consider the square corresponding to $A = 0, B = 1, C = 0, D = 1$, and any adjacent square. For the square on top the values are $A = 0, B = 1, C = 0, D = 0$, which differs only in the value of the variable D . For the square on the left, only variable B has a different value, and so on.

Adjacent squares are more difficult to see for a square at an edge. As an illustration, in Fig. 2.4(b), for the square corresponding to $A = 1, B = 0, C = 1, D = 0$, at the bottom right, there are two physically adjacent squares, one at the top and one on the left. Although there are no other physically adjacent squares, there are two other squares that differ in only one variable value. The square for $A = 1, B = 0, C = 0, D = 0$, which is in the top row and last column, though not physically adjacent, is also adjacent from the point of view that only one variable has a different value—the variable C . So we consider these two squares to be “adjacent” squares even though they are not physically adjacent. Similarly, in the last row the square on the left (corresponding to $A = 0, B = 0, C = 1, D = 0$) has only one variable with a different value—the variable A —and so is an “adjacent” square. Generalizing, for the purposes of adjacency in the sense that only one variable has a different value, we see that the squares in the top row are “adjacent” to the corresponding squares in the bottom row, and the squares in the leftmost column are “adjacent” to the corresponding squares in the rightmost column. If ever we have a doubt whether two squares are “adjacent,” all we have to do is to check the variable values for the two squares. If only one variable has a different value, then the two squares are “adjacent.”

In Fig. 2.4, note the numbering of the variable values. If we interpret these values as being binary numbers, then the first column is numbered zero, the second column one, but the third column is numbered three, and the fourth column two. So, from a numeric sense, the numberings of the third and fourth columns are interchanged. The same is true for the third and fourth rows. It is this interchanging of numbering that gives the adjacencies needed to simplify Boolean functions entered on K-maps.

2.8.1 Obtaining an MSOP from a K-Map

As mentioned, a K-map is a specific graphical representation of a truth table. Consequently, we can obtain a canonical SOP expression (CSOP) directly from a K-map, just

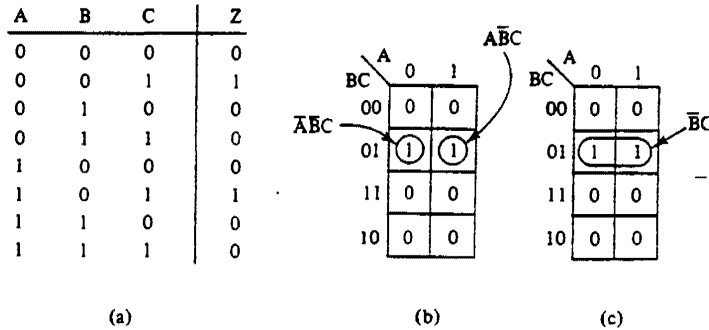


Figure 2.5 Using a K-map.

as we can from a truth table. We will do this for the K-map of Fig. 2.5(b), which corresponds to the truth table of Fig. 2.5(a). Recall from Sec. 2.4.1, that to obtain a CSOP from a truth table, we identify each row for which the function (Z here) is 1. With this K-map, then, we must identify the squares in which 1s are entered for Z, and, for convenience, circle the 1s as shown in Fig. 2.5(b). Next, we obtain the minterm corresponding to each of these squares. Here, these minterms are $\bar{A}\bar{B}C$ and $\bar{A}BC$. Finally, we OR the minterms to obtain the CSOP for Z. For the function Z of Fig. 2.5, the result is $Z = \bar{A}\bar{B}C + \bar{A}BC$.

We can algebraically simplify this result by using Boolean identity 10(a) of Table 2.2:

$$Z = \bar{A}\bar{B}C + \bar{A}BC = \bar{B}C$$

Better yet, we can simplify *graphically* using the K-map, as illustrated in Fig. 2.5(c). For this, we just circle the two adjacent 1-squares and read the simplified expression directly from the K-map by keeping the literals that do not change and dropping the variable that changes in value—in this case, variable A. This graphical simplification is possible because of the K-map feature that for any two adjacent squares, only *one* input variable has a different value. So, this K-map simplification is a graphical application of Boolean identity 10(a): $AB + A\bar{B} = A$.

In a similar fashion, we can group four adjacent 1-squares, arranged in the form of a rectangle, to eliminate two variables, as illustrated in Fig. 2.6(a). For this group of four squares notice that two variables (A and B)—and only two—have different values. These are the variables that drop out. So, we can read the simplified expression for the

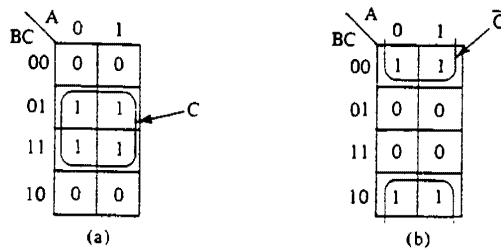


Figure 2.6 Grouping four 1-squares.

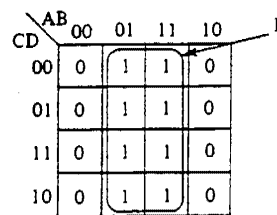


Figure 2.7 Grouping eight 1-squares.

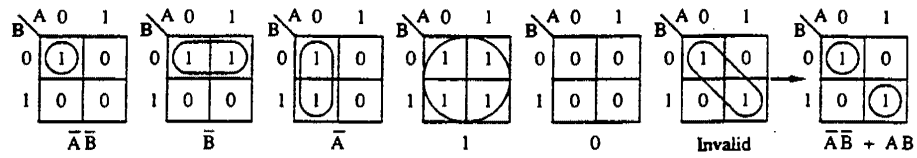


Figure 2.8 Two-variable K-map grouping examples. -

circled group directly from the K-map and obtain $Z = C$. This grouping of four squares corresponds to applying Boolean identity 10(a) twice:

$$Z = (\bar{A}\bar{B}C + \bar{A}BC) + (\bar{A}BC + ABC) = \bar{B}C + BC = C$$

For this variation of variable values that allows dropping of two or more variables, the circled squares must be in the shape of a rectangle; however, the rectangle can extend from the top of the K-map to the bottom, as in Fig. 2.6(b). Also, the number of squares grouped must be a power of two: 2^n . Then, n variables drop out.

Figure 2.7 shows a rectangular grouping of eight adjacent squares. We can read the simplified expression directly from the K-map as $Z = B$. The justification for this simplification is that this graphical application corresponds to applying Boolean identity 10(a) three times, as you can prove to yourself.

The terms we have been obtaining from K-maps are called *prime implicants*. For every set of variable values that makes a prime implicant 1, the corresponding function is also 1. This is rather obvious, because to obtain a prime implicant, we circle only 1s of a function. Another feature of a prime implicant is that no literal can be deleted from it and yet have it remain a valid term of the function. Thus there is a sense of minimalness about the number of literals in a prime implicant.

Figure 2.8 shows some common groupings for two-variable K-maps. However, for the simplification of expressions of just two variables we will usually find it easier to use the identities of Table 2.2. Note the invalid grouping of the next-to-last K-map. We cannot validly circle the two 1s because both variables have different values for the two 1-squares. For a valid grouping of two 1s only one variable can and must have different values. Graphically, we should know that the grouping is invalid from the fact that the sides of the squares are not physically adjacent. Since we cannot group the two 1s, we cannot simplify the CSOP.

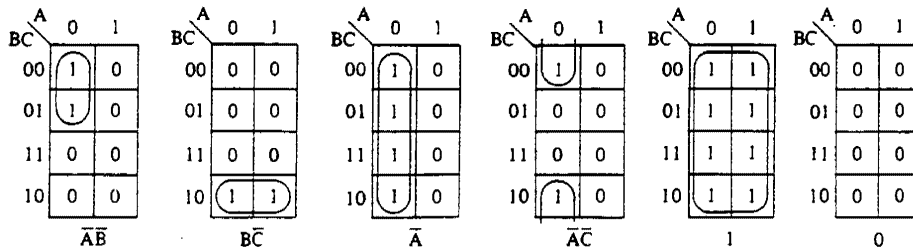


Figure 2.9 Three-variable K-map grouping examples.

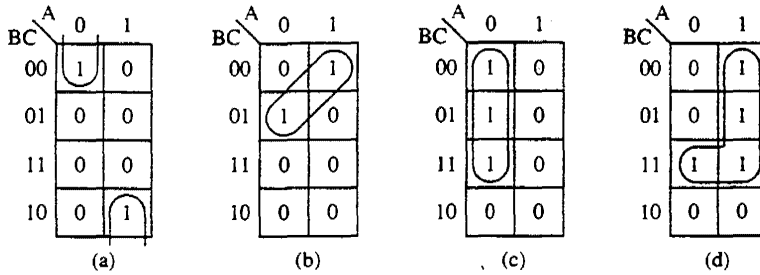


Figure 2.10 Invalid groupings.

Figure 2.9 shows some common groupings for three-variable K-maps, whereas Fig. 2.10 shows some invalid groupings for three-variable K-maps. In Fig. 2.10(a) the grouping is invalid because two variables (A and B), and not just one, have different values for the two 1-squares. We do not, however, have to check the variable values to know this. It is obvious from the lack of physical adjacency, even with the K-map rolled up to make the top edge and bottom edge join. Also, the grouping does not form a rectangle. For similar reasons, the grouping of Fig. 2.10(b) is invalid. The grouping of Fig. 2.10(c) is invalid because the number of grouped 1-squares (three) is not a power of two. The grouping of Fig. 2.10(d) is invalid, graphically speaking, because the grouped 1-squares do not form a rectangle. From an analytical point of view, the grouping is invalid because all three variables have different values for the grouped 1-squares, while for a valid grouping of four 1-squares, two and only two variables must have different values.

Figure 2.11 shows some common groupings for four-variable K-maps.

So far we have considered only a single grouping on each K-map. Almost always, though, more than one grouping is required to completely specify the function represented by the K-map, as shown by the following example.

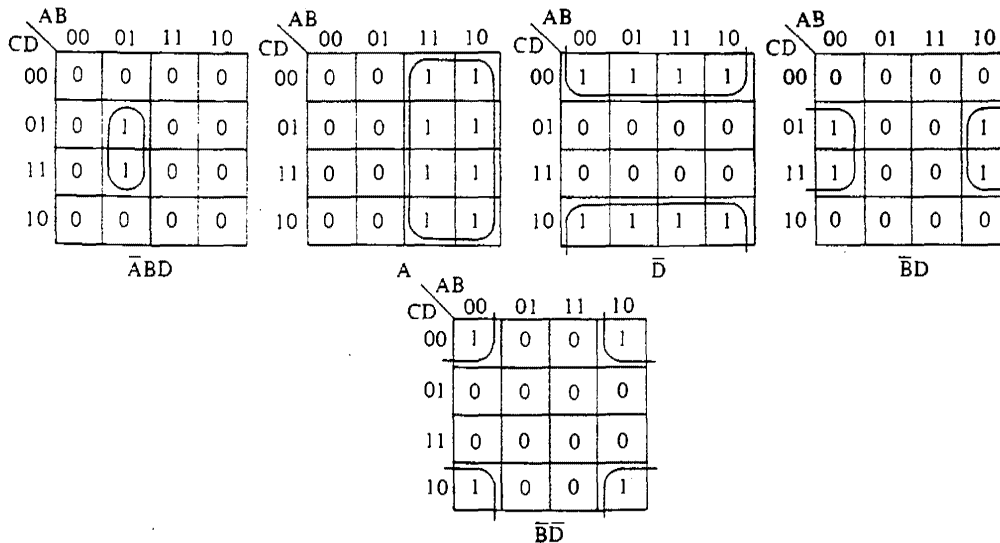


Figure 2.11 Four-variable K-map grouping examples.

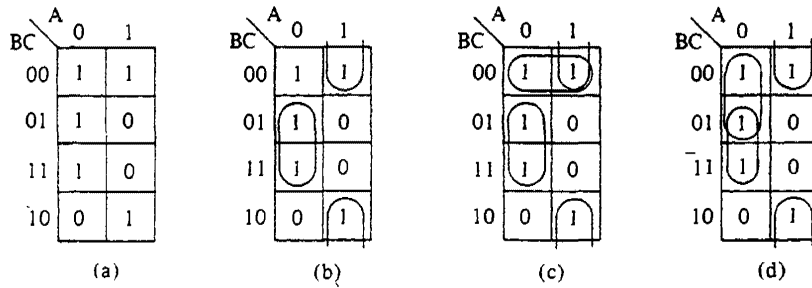


Figure 2.12 K-map illustration for Example 2.12.

EXAMPLE 2.12

Use a K-map to find an MSOP expression for Z.

A	B	C	Z
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Solution. Our first step is to enter the function on a three-variable K-map, as shown in Fig. 2.12(a).

The next step is to find the 1-squares that we can circle only once. There are two of them: $A = 1, B = 1, C = 0$ and $A = 0, B = 1, C = 1$. We form groupings with them, as shown in Fig. 2.12(b). The corresponding prime implicants ($\overline{A}C$ and $\overline{A}C$) are called *essential prime implicants* since they must be included in an MSOP. Only one 1-square remains uncircled, that of $A = 0, B = 0, C = 0$. For it, we have two grouping choices. We can group it with the 1-square of $A = 1, B = 0, C = 0$, as shown in Fig. 2.12(c), to obtain prime implicant $\overline{B}C$. Or, we can group it with the 1-square of $A = 0, B = 0, C = 1$, as shown in Fig. 2.12(d), to obtain prime implicant $\overline{A}\overline{B}$. Since the number of literals in these prime implicants is the same, we can use either one, which means there is no unique MSOP. From the Fig. 2.12(c) K-map we obtain $Z = \overline{A}C + \overline{A}C + \overline{B}C$, and from the Fig. 2.12(d) K-map we obtain the equivalent $Z = \overline{A}C + \overline{A}C + \overline{A}\overline{B}$. ■■

Figure 2.13 shows some three-variable K-maps and corresponding MSOP expressions. Remember, in finding an MSOP, we want to use as few groupings as possible in order to have the fewest number of terms. Also, we want each grouping to be as large as possible because the larger a grouping, the fewer the number of literals in the corresponding prime implicant. Note the redundant dotted grouping in the last K-map. If we used this grouping also, the expression would be $\overline{A}\overline{B} + \overline{A}C + \overline{B}C$. This is not an MSOP because it has more terms and literals than the MSOP $\overline{A}\overline{B} + \overline{A}C$, which covers all the 1s.

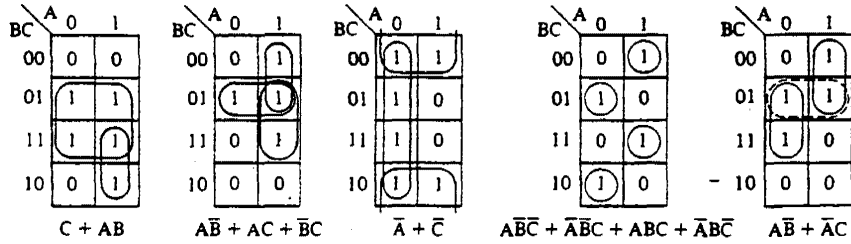


Figure 2.13 Three-variable K-map MSOP examples.

Obtaining an MSOP for a four-variable function is only slightly more difficult than for a three-variable function. Additionally, we must remember that 1-squares in the leftmost column are “adjacent” 1-squares in the corresponding rows in the rightmost column. Also, all four corner squares are “adjacent.”

EXAMPLE 2.13

Use a K-map to find an MSOP expression for Z.

A	B	C	D	Z
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Solution. Our first step is to enter the function on a four-variable K-map, as shown in Fig. 2.14(a). The next step is to find the essential prime implicant group-

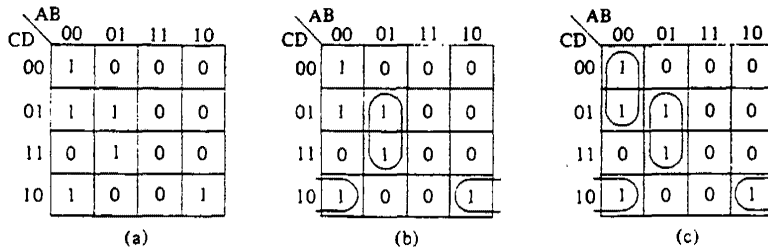


Figure 2.14 K-map illustration for Example 2.13.

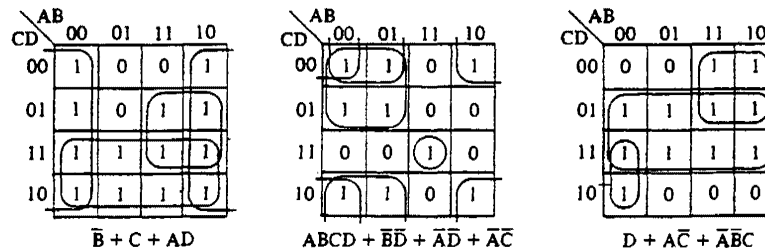


Figure 2.15 Four-variable K-map MSOP examples.

ings. There are just two of them since the only 1-squares we cannot circle more than once are those for $A = 1, B = 0, C = 1, D = 0$ and $A = 0, B = 1, C = 1, D = 1$. Figure 2.14(b) shows the essential prime implicant groupings. Two 1-squares remain to be circled. Since they are adjacent, we should obviously group them and obtain the complete grouping of Fig. 2.14(c). From it we obtain $Z = \overline{BCD} + \overline{ABD} + \overline{ABC}$, which is a unique MSOP even though one of the terms is not an essential prime implicant. ■ ■

Figure 2.15 shows some four-variable K-maps and the corresponding MSOP expressions.

In summary, finding an MSOP from a K-map is an art. Although there are no definitive rules to follow to guarantee obtaining an MSOP, the following guidelines are helpful.

1. Circle every 1 at least once.
2. Circle a 1 more than once if this circling helps in making larger groupings, but do not circle any more times than necessary to circle all the 1s.
3. Make the groups as large as possible.
4. Use no more groups than necessary.
5. Start the circling with 1s that can be circled only once. In general, start with the 1s that are most difficult to group.

We could extend our study of K-maps to five-variable and six-variable K-maps. The principles are the same, but the adjacencies become progressively more difficult to visualize as the number of variables increases. Besides, there are computer programs for finding MSOPs for functions of many more variables than we can use K-maps for. These computer programs are not based on graphical techniques such as K-maps, but on tabular methods such as modifications of the tabular Quine-McCluskey method.

2.8.2 Obtaining an MPOS from a K-Map

We can use K-maps for finding MPOSs just as readily as for finding MSOPs. The rules for grouping are the same except that we circle 0s instead of 1s. From these groups we form factors instead of terms. Also, we complement variables that have 1 values and do not complement those that have 0 values—just the opposite as for finding literals of MSOPs.

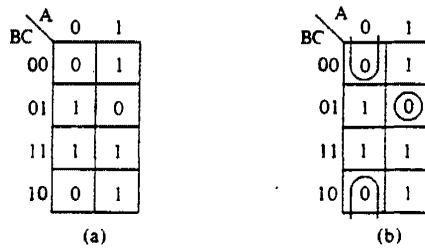


Figure 2.16 K-map illustration for Example 2.14.

EXAMPLE 2.14

Use a K-map to find an MPOS expression for Z.

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Solution. As usual, our first step is to enter the function on a K-map, as shown in Fig. 2.16(a). Then, we group the 0s, as shown in Fig. 2.16(b). Next, we find factors corresponding to the groups, remembering to complement if a variable has a 1 value. These factors are $(A + C)$ and $(\bar{A} + \bar{B} + \bar{C})$. Finally, we AND these factors. The result is $Z = (A + C)(\bar{A} + \bar{B} + \bar{C})$. ■ ■

Figure 2.17 shows some four-variable K-maps and corresponding MPOS expressions obtained from the shown groupings of 0s. Since K-maps can be used to find MSOPs or MPOSs, there arises the question of which type of minimum expression to solve for in a digital design. Actually, we may want to solve for both to determine which, possibly, is simpler. As an illustration, if for the function entered on the K-map of Fig. 2.16(a) we had solved for an MSOP, we would have obtained either $Z = \bar{A}C + A\bar{C} + BC$ or

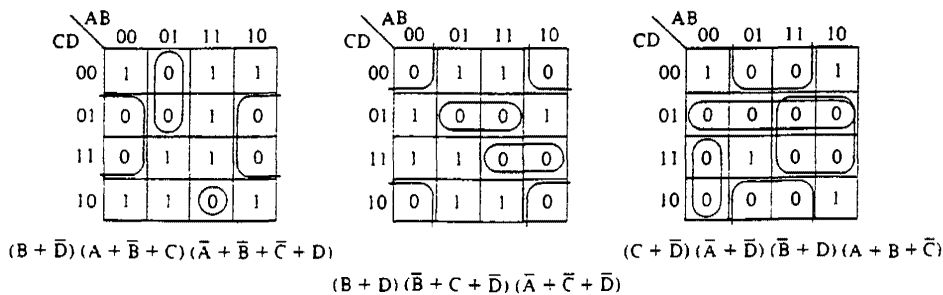


Figure 2.17 Four-variable K-map MPOS illustrations.

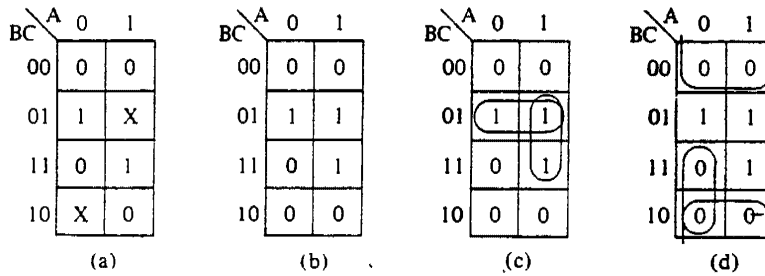


Figure 2.18 Three-variable don't-care illustrations.

$Z = \bar{A}C + A\bar{C} + AB$. Both MSOPs have three terms and six literals, while the MPOS $Z = (A + C)(\bar{A} + B + \bar{C})$ we found has just two factors and five literals, and so is simpler. Often, though, the MSOP and MPOS expressions are equally minimum. Another consideration is the type of hardware that will be used to implement the function. As shown in the next chapter, we want to use the minimum expression that more closely corresponds to the available hardware.

2.9 DON'T-CARE OUTPUTS

Often, in designing a digital system, a designer will know that certain components of the system will never have all possible combinations of inputs. A component with, say, inputs of A, B, and C, may never have inputs of $A = 1, B = 1, C = 0$ and $A = 1, B = 0, C = 0$, for example. And yet there are rows in the truth table and squares in the K-map for these input values. What then does the designer insert for the output for each of these inputs? The answer is a "don't care," identified by the symbol X. Clearly, for those inputs it does not matter what the outputs are, at least with regard to system operation. But it does make a difference with regard to minimization.

Since for don't-care outputs we are free to select either 0s or 1s, we select the values that are best for minimizing the output expression. In this, the selection for one don't care does not restrict us from selecting a different value for another don't care. In other words, the don't-care assignments do not have to be the same. By using K-maps we can readily determine which values for the don't cares result in minimum expressions.

Consider the K-map of Fig. 2.18(a), which has two don't cares. For an MSOP, obviously we want to replace the don't care of $A = 1, B = 0, C = 1$ with a 1 and the

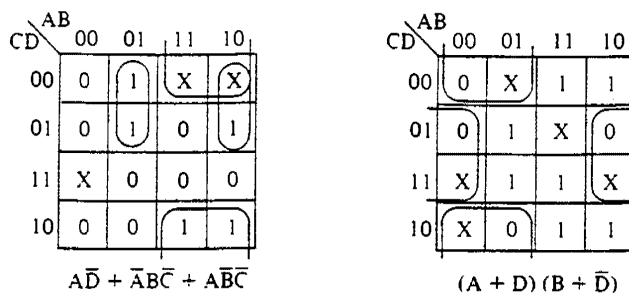


Figure 2.19 Four-variable don't-care illustrations.

don't care of $A = 0, B = 1, C = 0$ with a 0, as shown in Fig. 2.18(b). Then, with the grouping shown in Fig. 2.18(c), we obtain the MSOP expression $AC + \overline{B}C$ for Z . This don't care selection is also clearly best for an MPOS, which from Fig. 2.18(d) we see is $Z = C(A + \overline{B})$. But, the best don't-care selection for an MSOP is not always best for an MPOS. Figure 2.19 gives some additional don't-care illustrations.

2.10 MINIMIZATION SUMMARY

As has been shown, the K-map is a very convenient tool for simplifying logic functions of up to four variables—and, as mentioned, the K-map method can be extended to the simplifying of logic functions of five or more variables. However, the K-map method becomes increasingly more difficult to use with an increase in the number of variables, and eventually becomes unmanageable. Other systematic Boolean simplification methods have been developed to avoid this problem. These methods were important when hardware devices were the dominant cost of digital systems. In recent years, however, advances in microelectronic technology have been such that hardware costs are far from being the dominant costs. Consequently, our time will be better devoted to the study of concepts that are relevant to the enhancement of the overall digital system at the design level rather than in the study of sophisticated methods for eliminating a few logic gates.

2.11 OTHER COMMON LOGIC OPERATIONS

To complete our study of Boolean algebra, we will consider some other logic operations, that though not as fundamental as AND, OR, and NOT, are still important.

For two variables A and B , we can systematically generate 16 possible logic functions, as illustrated in Table 2.3. The three fundamental operations AND, OR, and NOT are represented by F_1 , F_7 , and F_{10} (and F_{12}), respectively. All other logic functions are combinations of these three fundamental operations. We will consider the most common of these.

2.11.1 NAND Operation

The NAND operation, represented by F_{14} in Table 2.3, is defined again as follows:

A	B	F_{14}
0	0	1
0	1	1
1	0	1
1	1	0

Note from the first three rows of the truth table that the Boolean variable F_{14} is true if *not* both A and B are true. From the last row of the truth table we get $F_{14} = \overline{A + B}$, which by DeMorgan's laws also has the form $F_{14} = \overline{A \cdot B}$. From this second form we

TABLE 2.3

A	B	F ₀	F ₁	F ₂	F ₃	F ₄	F ₅	F ₆	F ₇	F ₈	F ₉	F ₁₀	F ₁₁	F ₁₂	F ₁₃	F ₁₄	F ₁₅
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
		↑	↑		↑		↑	↑	↑	↑	↑	↑		↑		↑	↑
		0	A · B		A		B	A ⊕ B	A + B	$\overline{A + B}$	A ⊖ B	\overline{B}		\overline{A}		$\overline{A \cdot B}$	1

see that NAND is a combination of NOT and AND—hence the name NAND. More specifically, the NAND operation is the complement of the AND operation; it is the AND operation followed by the NOT operation.

The NAND operation applies to any number of variables. For the three variables A, B, and C, the NAND truth table is

A	B	C	$\overline{A \cdot B \cdot C}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Again, the only 0 output is for the last row, which is always the case, regardless of the number of input variables. In other words, the output variable is 1 unless all input variables are 1, in which case the output variable is 0.

2.11.2 NOR Operation

The NOR operation, represented by F₈ in Table 2.3, is defined again as follows:

A	B	F ₈
0	0	1
0	1	0
1	0	0
1	1	0

Note from the first row of the truth table that the Boolean variable F₈ is true only if *neither A nor B* is true. Also, from this row we get that $F_8 = \overline{A \cdot B}$, which by DeMorgan's laws also has the form $F_8 = \overline{A + B}$. From this second form we see that NOR is a combination of NOT and OR—hence the name NOR. More specifically, the NOR operation is the complement of the OR operation; it is the OR operation followed by the NOT operation.

The NOR operation applies to any number of variables. For the three variables A, B, and C, the NOR truth table is

A	B	C	$\overline{A + B + C}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Again, the only 1 output is for the first row, which is always the case, regardless of the number of input variables. In other words, the output variable is 0 unless all input variables are 0, in which case the output variable is 1.

Although the NAND and NOR operations are by far the most popular of the nonfundamental Boolean operations, there are other important ones we should consider.

2.11.3 Exclusive OR Operation

The Exclusive OR (XOR) operation, represented by F_6 in Table 2.3, is commonly designated by the symbol \oplus as in $F_6 = A \oplus B$. The XOR operation is defined by the following table:

A	B	F_6
0	0	0
0	1	1
1	0	1
1	1	0

Note that the Boolean variable F_6 is true if A is true or B is true, but not both. We can also view F_6 as being true if and only if $A \neq B$. Consequently this operation is useful in digital design for comparison purposes. Note also that this truth table differs from the two-variable OR truth table only in the last row, for both inputs of 1. From the second and third rows of the truth table we find that $F_6 = \overline{A}B + A\overline{B}$.

The XOR operation applies to any number of variables. For the three variables A, B, and C, the XOR truth table is

A	B	C	$A \oplus B \oplus C$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Note that for an odd number of 1 inputs, the output is 1. But for an even number of 1 inputs, the output is 0. This is generally true. Because of this, the XOR pattern on a K-map is a checkerboard, and so no terms combine.

2.11.4 Equivalence Operation

The equivalence operation, also called the coincidence operation, is represented by F_9 in Table 2.3. This operation is commonly designated by the symbol \odot , as in $F_9 = A \odot B$, and worded "F₉ is equal to A equivalence B." The equivalence operation is defined by the following table:

A	B	F ₉
0	0	1
0	1	0
1	0	0
1	1	1

From the first and last rows, we see that the Boolean variable F_9 is true if and only if $A = B$. So $F_9 = \overline{A}B + A\overline{B}$. Also, from a comparison of this truth table and that for XOR, we see that *for two variables*, the equivalence and XOR operations are complements of one another, which means that the equivalence operation, like the XOR operation, is useful in digital design for comparison purposes.

SUPPLEMENTARY READING (see Bibliography)

[Bartee 85], [Boole 54], [Blakeslee 79], [Hill 81], [Karnaugh 53], [Mano 84], [McCluskey 75], [Peatman 80], [Roth 85], [Shannon 38]

PROBLEMS

2.1. Evaluate the following expressions for $A = F$, $B = T$, $C = F$, and $D = T$:

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

(b) $(A + \overline{BC})(\overline{AD} + \overline{AD})$

(c) $\overline{AD}(A + \overline{BC} + \overline{BD})$

(d) $\overline{AB}(\overline{AC} + \overline{BD} + \overline{ABCD})$

(e) $\overline{\overline{AB} + C + \overline{CD}}$

(f) $(A + \overline{BC} + \overline{BCD} + \overline{B} + C)(\overline{ACD} + \overline{BD})$

2.2. Evaluate the following expressions for $A = F$, $B = F$, $C = T$, and $D = T$:

(a) $\overline{AB} + \overline{BCD}(\overline{AB} + \overline{CD} + \overline{AD})$

(b) $\overline{ABCD} + \overline{ACD} + \overline{BC}(A + \overline{CD})$

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

(d) $\overline{(A + B)}(A + \overline{CD} + \overline{CD})$

(e) $\overline{\overline{ABC} + \overline{CD}}(\overline{AC} + \overline{D} + \overline{BCD})$

(f) $\overline{(A + \overline{BC})CD} + \overline{BCD}$

- 2.3. Repeat Problem 2.1 for $A = 1, B = 1, C = 0,$ and $D = 1.$
 2.4. Repeat Problem 2.2 for $A = 1, B = 0, C = 1,$ and $D = 0.$
 2.5. Find the CSOP expressions for the Z functions defined in the following truth tables:

ABC	Z ₁	ABC	Z ₂	ABC	Z ₃	ABC	Z ₄
000	0	000	1	000	1	000	0
001	1	001	1	001	0	001	0
010	1	010	1	010	1	010	1
011	0	011	0	011	1	011	1
100	1	100	0	100	0	100	1
101	1	101	1	101	0	101	0
110	0	110	1	110	1	110	1
111	1	111	0	111	1	111	0
(a)		(b)		(c)		(d)	

- 2.6. Find the CPOS expressions for the Z functions of Problem 2.5.
- 2.7. A function Z of four variables A, B, C, and D is 1 if and only if two of the four variables are 1. Express the function Z as a CSOP and also as a CPOS.
- 2.8. A function Z of four variables A, B, C, and D is 1 if and only if an odd number of the four variables is 1. Express the function Z as a CSOP and also as a CPOS.
- 2.9. A function Z of five variables A, B, C, D, and E is 1 if and only if an even number of the five variables is 1. Express the function Z as a CSOP and also as a CPOS.
- 2.10. Use truth tables to determine whether the following pairs of expressions are equivalent:
- (a) $\overline{A}\overline{B} + \overline{A}\overline{B}\overline{C} + AB + \overline{A}BC$ and $\overline{A}\overline{B}\overline{C} + \overline{A}C + \overline{A}C$
 - (b) $\overline{A}\overline{C} + BC + \overline{A}\overline{B}\overline{C}$ and $\overline{B}\overline{C} + ABC + \overline{A}B$
 - (c) $AB + AC$ and $(A + \overline{C})(A + C)(B + C)$
 - (d) $(A + \overline{B} + C)(B + \overline{C})(\overline{A} + \overline{B})$ and $\overline{A}BC + \overline{B}\overline{C}$
 - (e) $(\overline{A} + C)(A + \overline{B} + \overline{C})(\overline{A} + \overline{B})$ and $(\overline{A} + \overline{B} + C)(\overline{B} + \overline{C})$
- 2.11. Repeat Problem 2.10 for the following:
- (a) $\overline{B}\overline{C} + BCD + \overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B} + \overline{A}\overline{B}\overline{C} + \overline{A}BCD + ACD$
 - (b) $A\overline{D} + \overline{A}CD + C\overline{D}$ and $\overline{A}C + AC\overline{D} + \overline{A}B\overline{C} + \overline{A}\overline{B}\overline{C}\overline{D}$
 - (c) $BD + \overline{B}\overline{D}$ and $(A + B + \overline{D})(\overline{B} + D)(\overline{A} + B + \overline{D})$
 - (d) $\overline{B}D(\overline{A} + \overline{C})$ and $(\overline{B} + C)(B + D)(\overline{B} + \overline{C})$
 - (e) $B(\overline{A} + B + C + D)(\overline{B} + C)$ and $C(A + B)(\overline{A} + B + \overline{C})$
- 2.12. Repeat Problem 2.10 for the following:
- (a) $\overline{A}\overline{C}\overline{D} + \overline{A}\overline{D}E + ABD + \overline{A}\overline{B}\overline{C}\overline{D}E + ADE$ and $\overline{A}\overline{C}\overline{D}E + \overline{A}\overline{C}\overline{D}\overline{E} + AD + \overline{A}\overline{C}\overline{D}E$
 - (b) $(A + E)(D + \overline{E})(A + \overline{D})(C + D)(B + C)$ and $A(C + BD)(D + \overline{C}\overline{E})$
- 2.13. Use truth tables to determine whether the following pairs of expressions are complements of each other:
- (a) $\overline{A}C + ABC + \overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{B} + \overline{A}C + \overline{A}\overline{B}\overline{C}$
 - (b) $\overline{C} + \overline{A}\overline{B}\overline{C}$ and $\overline{A}C + AB$
 - (c) $AB + \overline{A}BC + \overline{A}\overline{B}\overline{C}$ and $\overline{A}\overline{C} + \overline{B}C + \overline{A}\overline{B}\overline{C}$

- (d) $(A + B)(\bar{A} + C)$ and $\bar{A}\bar{B} + \bar{B}\bar{C}$
 (e) $(\bar{A} + \bar{B})(A + \bar{C})$ and $(A + \bar{B} + C)(A + B + C)(\bar{A} + B)$
 (f) $(\bar{A} + B)(\bar{A} + \bar{B} + \bar{C})(A + B)$ and $(A + \bar{B})(\bar{A} + \bar{B} + C)$
- 2.14.** Repeat Problem 2.13 for
 (a) $\bar{A}\bar{B} + CD + \bar{B}C$ and $\bar{B}\bar{D} + \bar{B}\bar{C} + A\bar{B}\bar{C}$
 (b) $\bar{A}\bar{B}C + A\bar{D}$ and $\bar{A}\bar{B} + AD + \bar{A}\bar{C}$
 (c) $\bar{A}\bar{C} + AC$ and $(\bar{A} + \bar{C})(A + B + C + D)(A + \bar{B} + C)$
 (d) $(B + \bar{C})(A + D)(\bar{A} + B)$ and $(\bar{A} + \bar{B})(\bar{B} + \bar{D})$
 (e) $(\bar{A} + \bar{C})(\bar{A} + B)$ and $(A + D)(\bar{B} + C)(A + B)$
- 2.15.** Using Boolean identities, simplify the following expressions:
 (a) $A\bar{B}C + (\bar{A} + B)C\bar{D}$
 (b) $A\bar{C}(B + D) + \bar{B}\bar{D} + C$
 (c) $(A\bar{B} + C + D)(\bar{A} + B + C + D + E)$
 (d) $(A\bar{B} + C\bar{D} + E + F)(\bar{A} + B + E + F)(\bar{C} + D + E + F)$
- 2.16.** Repeat Problem 2.15 for
 (a) $A + \bar{C}D + CDE + \bar{A}B$
 (b) $AC\bar{D} + A\bar{B}\bar{C} + A\bar{B}D$
 (c) $(A + C + D)(A + C + \bar{D} + E)$
 (d) $(A + B + \bar{C})(A + \bar{C} + D)(\bar{B} + \bar{D})C$
- 2.17.** Repeat Problem 2.15 for
 (a) $ABC + A\bar{B}C + \overline{(\bar{A} + \bar{B} + \bar{C})(A + B + \bar{C})}$
 (b) $(A + B + C)(A + \bar{B}C)(A + \bar{B} + C)(A + B + \bar{C})$
 (c) $ABCD + A\bar{B}\bar{C}D + \bar{A}BCD + \bar{A}\bar{B}\bar{C}D$
 (d) $(A + \bar{B})(\bar{B} + C)(\bar{B} + \bar{C})$
- 2.18.** Find complements of the following expressions by using DeMorgan's laws and then simplify until DeMorgan's laws cannot be applied further.
 (a) $A\bar{B} + C(\bar{D} + E)$
 (b) $A\bar{B} + \bar{C}\bar{D}(E + \bar{F}G)$
 (c) $A\bar{B}(C + \bar{D}E) + \bar{A}\bar{B}C + \bar{B}(C + D)$
- 2.19.** Repeat Problem 2.18 for
 (a) $(A + \bar{C})(D + \bar{E}F) + A\bar{B}$
 (b) $\overline{AB(C + \bar{D}E)} + A\bar{B}(C + D)$
 (c) $A\bar{C}\bar{D}(E + \bar{F}G) + HI\bar{J}(K + \overline{LM})$
- 2.20.** Using Boolean identities, convert each of the following expressions to a CSOP, and also to a CPOS:
 (a) $\bar{A}\bar{B} + \bar{B}C$ (b) $A + \bar{B}C$
 (c) $(A + B)(\bar{A} + \bar{B})$ (d) $(A + B)(\bar{B} + \bar{C})$
- 2.21.** Repeat Problem 2.20 for
 (a) $A\bar{B} + C$ (b) $A\bar{B} + \bar{C}D$
 (c) $(A + \bar{B}C)(A + \bar{B}D)$ (d) $(A + \bar{B})(C + D)$

2.22. Find the K-maps for each function $Z_1, Z_2, Z_3,$ and Z_4 defined as follows:

ABC	Z_1	Z_2	ABCD	Z_3	Z_4
000	0	1	0000	0	1
001	1	1	0001	1	0
010	1	0	0010	0	1
011	0	0	0011	1	1
100	0	1	0100	1	0
101	1	1	0101	0	0
110	1	0	0110	0	1
111	1	0	0111	1	0
			1000	1	1
			1001	0	1
			1010	0	0
			1011	0	1
			1100	1	1
			1101	1	0
			1110	1	0
			1111	0	1

2.23. Find the truth tables corresponding to the functions defined by the K-maps of Fig. 2.20.

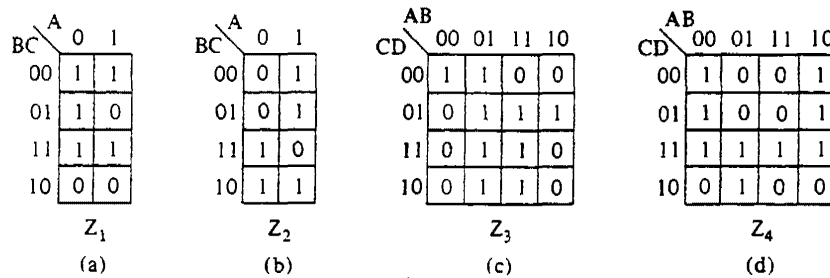


Figure 2.20 K-maps for Problem 2.23.

2.24. Use K-maps to find an MSOP and an MPOS for each of the functions defined by the truth tables of Problem 2.22.

2.25. Find an MSOP and an MPOS for each of the functions defined by the K-maps of Fig. 2.20.

2.26. Obtain an MSOP and an MPOS for each of the functions defined by the K-maps of Fig. 2.21.

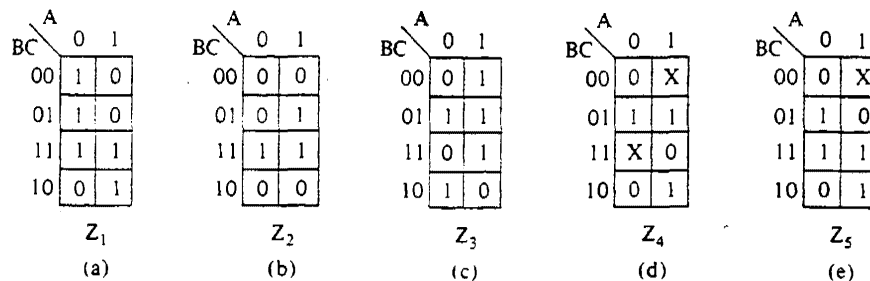


Figure 2.21 K-maps for Problem 2.26.

2.27. Obtain an MSOP and an MPOS for each of the functions defined by the K-maps of Fig. 2.22.

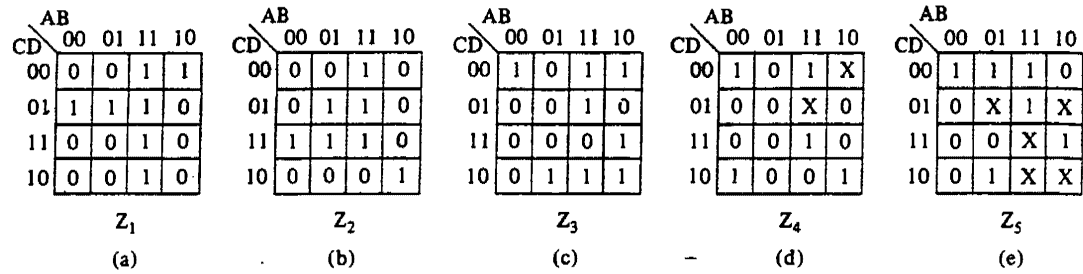


Figure 2.22 K-maps for Problem 2.27.

2.28. Use K-maps to obtain an MSOP and an MPOS for each of the following functions:

- (a) $Z_1 = \overline{A}\overline{B}\overline{C} + ABC + \overline{A}\overline{B}C + \overline{A}B\overline{C}$
 (b) $Z_2 = \overline{A}\overline{B}\overline{C} + ABC + \overline{A}B\overline{C} + \overline{A}B\overline{C}$
 (c) $Z_3 = \overline{B}C + \overline{A}\overline{B}\overline{C} + ABC$
 (d) $Z_4 = (A + B + C)(\overline{A} + \overline{B} + \overline{C})(\overline{A} + \overline{B} + C)$
 (e) $Z_5 = (\overline{A} + B + \overline{C})(A + B + \overline{C})(A + \overline{B} + C)$
 (f) $Z_6 = (A + B + C)(\overline{A} + \overline{C})(A + \overline{B} + C)$

2.29. Repeat Problem 2.28 for

- (a) $Z_1 = \overline{A}\overline{B}\overline{C} + ABC + \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C}$ with a don't care for $ABC = 101$
 (b) $Z_2 = \overline{A}\overline{B}\overline{C} + ABC + \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C}$ with don't cares for $ABC = 001$ and 011
 (c) $Z_3 = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + ABC$ with don't cares for $ABC = 101$ and 010
 (d) $Z_4 = (A + B + C)(\overline{A} + \overline{B} + \overline{C})$ with don't cares for $ABC = 100, 011,$ and 110
 (e) $Z_5 = (\overline{A} + B + \overline{C})(A + \overline{B} + C)$ with don't cares for $ABC = 001, 111,$ and 110
 (f) $Z_6 = (\overline{A} + \overline{B})(B + \overline{C})(A + B + C)$ with a don't care for $ABC = 011$

2.30. Repeat Problem 2.28 for

- (a) $Z_1 = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}B\overline{C}\overline{D}$
 (b) $Z_2 = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D$
 (c) $Z_3 = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C\overline{D} + BCD + \overline{A}B\overline{C}\overline{D} + \overline{A}C\overline{D}$
 (d) $Z_4 = \frac{(A + B + \overline{C} + D)(A + \overline{B} + C + D)(\overline{A} + B + \overline{C} + \overline{D})(A + \overline{B} + \overline{C} + D)}{(A + \overline{B} + C + D)}$
 (e) $Z_5 = \frac{(\overline{A} + B + \overline{C} + D)(A + B)(C + D)(\overline{B} + C + \overline{D})(\overline{A} + B + C + \overline{D})}{(\overline{A} + B + \overline{C} + D)}$
 (f) $Z_6 = \frac{(\overline{A} + B + \overline{D})(\overline{B} + C + D)(\overline{A} + B + C + \overline{D})(\overline{A} + \overline{B} + \overline{C} + D)}{(\overline{A} + B + \overline{C} + D)}$

2.31. Repeat Problem 2.28 for

- (a) $Z_1 = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}B\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}\overline{D}$ with a don't care for $ABCD = 0101$
 (b) $Z_2 = \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}\overline{B}\overline{C}\overline{D}$ with don't cares for $ABCD = 0001$ and 1111
 (c) $Z_3 = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}\overline{C}D$ with don't cares for $ABCD = 0010$ and 0101

(d) $Z_4 = (\bar{A} + B)(A + \bar{B} + \bar{C})(A + \bar{B} + C + D)$ with a don't care for $ABCD = 0101$

(e) $Z_5 = (C + D)(A + \bar{B})(\bar{A} + \bar{C} + D)$ with don't cares for $ABCD = 1001, 1101,$
and 1111

(f) $Z_6 = (C + D)(\bar{B} + D)(C + \bar{D})$ with don't cares for $ABCD = 0111$ and 1111

- 2.32. For two variables it has been stated that the equivalence and XOR operations are complements of one another. What about three variables? Compare the truth table of $A \odot B \odot C$ with that for $A \oplus B \oplus C$.

Digital Design with Small-Scale Integrated Circuit Elements

3.1 INTRODUCTION

In Chapter 2 we considered the concept of Boolean variables (variables that can have only the logic values of true and false) and the three fundamental operations of Boolean algebra: AND, OR, and NOT. We proceeded to derive complex expressions based on these operations and discussed various methods of manipulating these expressions. With this background in mind, in this chapter we will study the design of digital systems using physical devices that actually perform Boolean operations, and with extraordinary speed. We will find that we can physically implement any imaginable digital system that is consistent with Boolean logic. Furthermore, the implementations will perform the basic Boolean operations in a nanosecond time frame. It is this fact that elevated Boolean algebra from an interesting concept derived by George Boole in the nineteenth century into the world of wizardry of modern-day digital computers.

Specifically, in this chapter we will study the design of digital systems with physical components called *small-scale integrated circuits* (SSI circuits). The term “small-scale” refers to the relatively small number of electronic gates in a single *integrated circuit*, often referred to as a *chip*. (A *gate* is an individual logic implementer.) Generally, integrated circuits with 12 or fewer equivalent logic gates per semiconductor chip are considered to be SSI circuits. Other types of integrated circuits include *medium-scale integrated circuits* (MSI, 13 to 99 logic gates), *large-scale integrated circuits* (LSI, 100 to 1000 logic gates), and *very large-scale integrated circuits* (VLSI, more than 1000 logic gates).

3.2 TRANSISTOR-TRANSISTOR LOGIC

For all our SSI implementations, we will use the SSI circuit family that has been most popular for years: the transistor-transistor logic (TTL) family. Currently, there are many

series within the TTL family, including standard TTL, high-speed TTL (H-TTL), low-power TTL (L-TTL), Schottky TTL (S-TTL), low-power Schottky TTL (LS-TTL), advanced S-TTL (AS-TTL), and advanced LS-TTL (ALS-TTL).

Each TTL chip has a standard 74XY identifier label. The 74 specifies a commercial grade TTL product. The X identifies the series within the TTL family, and the Y, which is two or more digits, identifies the particular member of the series. For example, the 74ALS04 chip is a commercial grade TTL product in the advanced LS (ALS) series with part number 04. Under these labels, integrated-circuit manufacturers publish chip descriptions in their TTL data books. Included in each description is the functional operation of the chip components. We will use this information (in the form of voltage tables) in our design of digital systems.

The corresponding circuit elements of each of the TTL series are functionally identical, in regard to logic operations. For example, a 7400, a 74LS00, and a 74ALS00 are all two-input NAND gates. The differences among them are physical characteristics such as speed of operation, power dissipation, and so forth, all of which are unrelated to logic operations. Since in this chapter only the logic operations of the chips are of concern, we will adopt an apostrophe notation, such as in 74'00, rather than specify an actual series within the TTL family. A more detailed discussion of the TTL family is given in Sec. 4.8.

3.3 LOGIC CONVENTIONS

This section contains a unifying view of the various logic conventions that apply to digital circuits in general. In the discussion we will view a digital circuit as a network of digital *elements* energized by interconnecting digital *signals*. Each digital signal corresponds to a logic variable, and the voltage level of the signal represents the logic value of the variable. Also, each digital element, in the form of an integrated circuit, performs some logic function on input signals and produces corresponding output signals. We will begin our consideration of logic conventions with some definitions, terminology, and notational standards that will be used in this text.

For a digital signal, the logic values of true (T) and false (F) are represented by one of two voltage levels: high (H) or low (L). Therefore, there are just two possible assignments:

	Assignment	Terminology
(a)	$\left\{ \begin{array}{l} T \leftrightarrow H \\ F \leftrightarrow L \end{array} \right\}$	Active-high
(b)	$\left\{ \begin{array}{l} T \leftrightarrow L \\ F \leftrightarrow H \end{array} \right\}$	Active-low

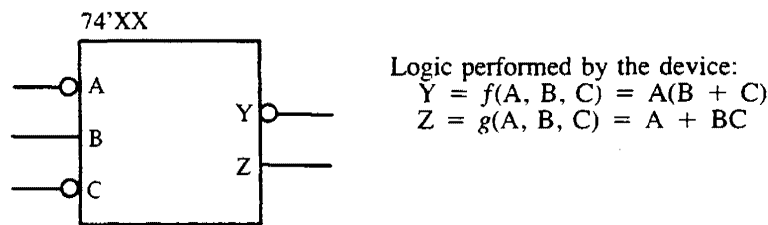
For each signal of a digital circuit, we need to assign a voltage representation of either active-high or active-low. In the *positive-logic convention* of logic/voltage assignment, we assign all signals in the digital circuit to be active-high. Conversely, in the *negative-logic convention*, we assign all signals to be active-low. Finally, in the *mixed-logic convention*, we individually assign the voltage representation of each signal, which

means that there can be a mixture of active-high and active-low signals in a single digital circuit. But if in the mixed-logic convention we happen to assign all signals to be active-high, then it is equivalent to the positive-logic convention. In other words, the positive-logic convention is a subset of the mixed-logic convention—as is the negative-logic convention.

To understand the concepts of active-high and active-low more fully, consider Fig. 3.1, which shows a fictitious device 74'XX with three input terminals (A, B, and C) and two output terminals (Y and Z). The small circles at terminals A and C indicate that they are active-low input terminals. In other words, the signals applied at terminals A and C will be interpreted by the 74'XX device as being active-low signals. The absence of a circle at terminal B indicates that it is an active-high input terminal and so any signal applied there will be interpreted by the device as being an active-high signal. Note the effects of such interpretations as illustrated by the “input voltage values” columns and the “input logic values” columns of the table shown in Fig. 3.1(b).

The device performs the logic function f on the signals applied at input terminals A, B, and C, and produces an output signal at output terminal Y. The circle at terminal Y designates that it is an active-low output terminal. Consequently, the signal that is produced there is active-low, as illustrated by the “output logic values” columns and the “output voltage values” columns of the table shown in Fig. 3.1(b). Similarly, the device also performs the logic function g and produces an active-high signal at the active-high output terminal Z (absence of a circle).

Generally, in order to make use of the function that is defined for a particular device, we must apply an active-low signal to an active-low input terminal, and an active-



(a) Fictitious device

Input voltage values			Input logic values			Output logic values		Output voltage values	
A	B	C	A	B	C	Y	Z	Y	Z
L	L	L	T	F	T	T	T	L	H
L	L	H	T	F	F	F	T	H	H
L	H	L	T	T	T	T	T	L	H
L	H	H	T	T	F	T	T	L	H
H	L	L	F	F	T	F	F	H	L
H	L	H	F	F	F	F	F	H	L
H	H	L	F	T	T	F	T	H	H
H	H	H	F	T	F	F	F	H	L

(b) Voltage and logic table

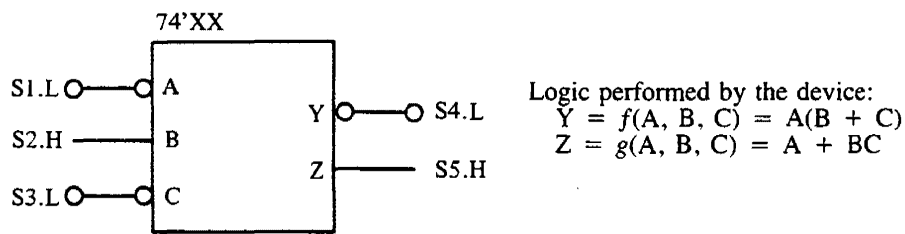
Figure 3.1 Illustration of active-high and active-low concepts.

high signal to an active-high input terminal. Also, we must accept the output signals as being either active-high or active-low, as defined for the output terminals. These facts are illustrated by the following example.

EXAMPLE 3.1

For the circuit shown in Fig. 3.2(a), find the logic expressions for S4 and S5 as functions of S1, S2, and S3.

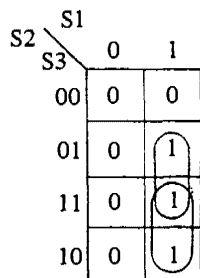
Solution. Note from Fig. 3.2(a) that each signal, besides having a name label (e.g., S1), also has a voltage representation label of either .H for active-high or .L for active-low. We will consistently use this notation for the labeling of signals.



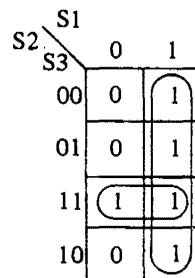
(a) Fictitious device

Input logic values			Input voltage values S1 S2 S3 (A B C)			Input logic values			Output logic values		Output voltage values Y Z (S4 S5)		Output logic values	
S1	S2	S3	A	B	C	A	B	C	Y	Z	S4	S5	S4	S5
F	F	F	H	L	H	F	F	F	F	F	H	L	F	F
F	F	T	H	L	L	F	F	T	F	F	H	L	F	F
F	T	F	H	H	H	F	T	F	F	F	H	L	F	F
F	T	T	H	H	L	F	T	T	F	T	H	H	F	T
T	F	F	L	L	H	T	F	F	F	T	H	H	F	T
T	F	T	L	L	L	T	F	T	T	T	L	H	T	T
T	T	F	L	H	H	T	T	F	T	T	L	H	T	T
T	T	T	L	H	L	T	T	T	T	T	L	H	T	T

(b) Voltage and logic table



$$S4 = S1 \cdot S3 + S1 \cdot S2 = S1(S2 + S3)$$



$$S5 = S1 + S2 \cdot S3$$

(c) Results

Figure 3.2 Illustration for Example 3.1.

Thus a signal is incompletely specified unless it is labeled with a name *and* a voltage representation. Also, to emphasize graphically that S1 is an active-low signal, a circle is associated with it (as with S3 and S4).

For this example, the input signals S1 and S3 and the output signal S4 are all active-low, whereas the input signal S2 and the output signal S5 are both active-high. Consequently, the logic/voltage assignments of these signals match those of the terminals of the device.

The desired logic expressions can be obtained in a systematic manner by using the table shown in Fig. 3.2(b). The “input logic values” columns show all the possible logic values for input signals S1, S2, and S3. And the “input voltage values” columns contain the corresponding voltage values for S1, S2, and S3 that are actually applied at the input terminals A, B, and C. These voltage values are interpreted by the input terminals of the device, resulting in the logic values shown in the “input logic values” columns for A, B, and C.

The device then performs the functions f and g on the input logic values, and produces the output logic values for Y and Z, as shown in the “output logic values” columns for Y and Z. Next, the device outputs the actual voltage values corresponding to these logic values, as is shown in the “output voltage values” columns. Note for active-low output terminal Y that the device outputs a low voltage value for a true logic value and a high voltage value for a false logic value. Finally, the voltage values generated by the device are interpreted by the output signals S4 and S5. The result is shown in the “output logic values” columns for S4 and S5.

With this table completed, it is a simple matter to derive the logic expressions for S4 and S5 by using the techniques presented in Chapter 2. The results are shown in Fig. 3.2(c). Note that they agree with the device logic expressions of Fig. 3.2(a), with S1, S2, and S3 substituted for A, B, and C, respectively. It is much easier, of course, to make these substitutions than to use the voltage and logic table approach.

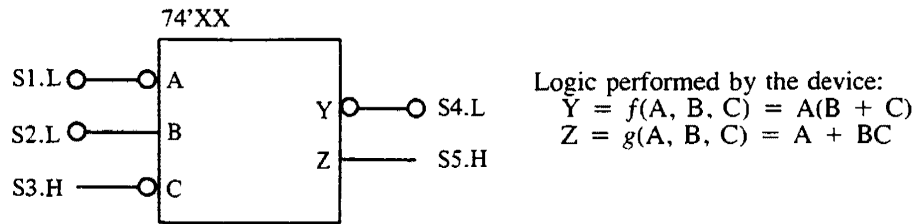
Generalizing, we conclude from this example that to make use of the function that is defined for a particular device, we must make certain that the logic/voltage assignments of the input and output signals match those of the input and output terminals of the device. ■ ■

EXAMPLE 3.2

For the circuit shown in Fig. 3.3(a), find the logic expressions for S4 and S5 as functions of S1, S2, and S3. Note that the logic/voltage assignments of input signals S2 and S3 do not match those of input terminals B and C, respectively, of the device.

Solution. As in Example 3.1, the solution for this problem can be obtained systematically by using a table, as is shown in Fig. 3.3(b). Note specifically the difference between the “input logic values” columns for S2 and S3 and the “input logic values” columns for B and C, resulting from the difference in the interpretation of the same input voltage values.

With this table completed, it is again a simple matter to derive the logic expressions for S4 and S5 by using the techniques presented in Chapter 2. The



(a) Fictitious device

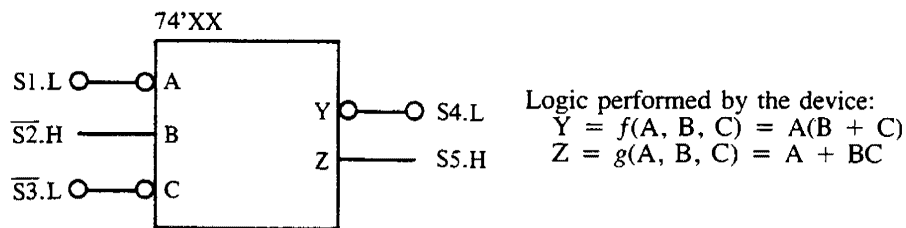
Input logic values			Input voltage values S1 S2 S3 (A B C)			Input logic values			Output logic values		Output voltage values Y Z (S4 S5)		Output logic values	
S1	S2	S3	A	B	C	A	B	C	Y	Z	S4	S5	S4	S5
F	F	F	H	H	L	F	T	T	F	T	H	H	F	T
F	F	T	H	H	H	F	T	F	F	F	H	L	F	F
F	T	F	H	L	L	F	F	T	F	F	H	L	F	F
F	T	T	H	L	H	F	F	F	F	F	H	L	F	F
T	F	F	L	H	L	T	T	T	T	T	L	H	T	T
T	F	T	L	H	H	T	T	F	T	T	L	H	T	T
T	T	F	L	L	L	T	F	T	T	T	L	H	T	T
T	T	T	L	L	H	T	F	F	F	T	H	H	F	T

(b) Voltage and logic table

$$S4 = S1 \cdot \overline{S2} + S1 \cdot \overline{S3} = S1(\overline{S2} + \overline{S3})$$

$$S5 = S1 + \overline{S2} \cdot \overline{S3}$$

(c) Results



(d) Circuit diagram with transformed input signals

Figure 3.3 Illustration for Example 3.2.

result is shown in Fig. 3.3(c). From comparing these results with the logic expressions for Y and Z in Fig. 3.3(a), observe that in effect we have transformed the input signal S2.L to $\overline{S2.H}$ so that it matches the logic/voltage assignment of input terminal B. Similarly, S3.H has been transformed to $\overline{S3.L}$ to match the logic/voltage assignment of input terminal C. These transformations are shown in Fig. 3.3(d).

The conclusion that we can draw from this example is that when there is a mismatch of the logic/voltage assignment between an input signal and an input terminal, we need to transform the logic/voltage assignment of the signal to make

it match the logic/voltage assignment of the input terminal by using the following identities.

$$X.H = \bar{X}.L \quad \text{and} \quad \bar{X}.H = X.L$$

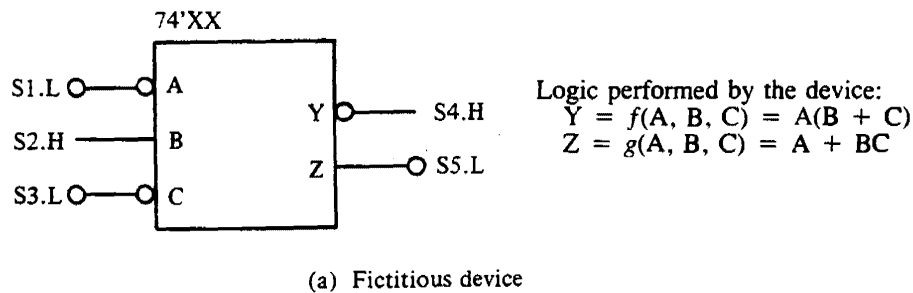
The proof of these identities can be obtained by using the following table.

Voltage value X	Logic value	
	X.H	X.L
L	F	T
H	T	F

Expressed in words, what this table shows is that for a low voltage (L), X.H “interprets” it as being false (F) and X.L “interprets” it as being true (T). Conversely, for a high voltage, X.H “interprets” it as being true and X.L “interprets” it as being false. Then from columns 2 and 3 of the table, we can determine that $X.H = \bar{X}.L$ and $\bar{X}.H = X.L$. ■ ■

EXAMPLE 3.3

For the circuit shown in Fig. 3.4(a), determine the logic expressions for S4 and S5 as functions of S1, S2, and S3. Note that the logic/voltage assignments of the output signals S4 and S5 do not match those of the output terminals Y and Z, respectively, of the device.



$$\begin{aligned} \bar{S4} &= S1(S2 + S3) & \bar{S5} &= S1 + S2 \cdot S3 \\ S4 &= \overline{S1(S2 + S3)} & S5 &= \overline{S1 + S2 \cdot S3} \end{aligned}$$

(b) Results

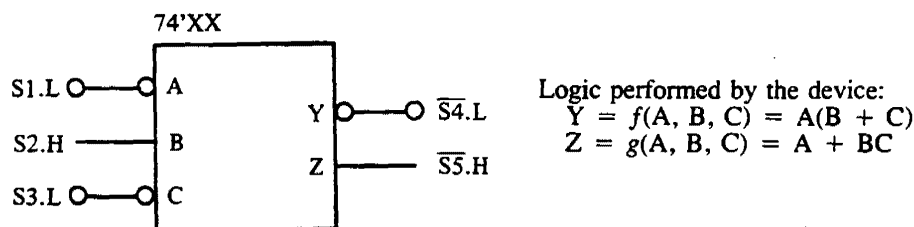


Figure 3.4 Illustration for Example 3.3.

Solution. As in Examples 3.1 and 3.2, the solution for this example [shown in Fig. 3.4(b)] can be systematically obtained by using a table like the one shown in Fig. 3.3(b). The solution, however, is left to the reader. (See Problem 3.3.) Observe that we can get the same result by transforming the output signal S4.H to $\overline{S4.L}$ and S5.L to $\overline{S5.H}$ to obtain matches with the logic/voltage assignments of output terminals Y and Z, respectively, as is shown in Fig. 3.4(c).

The conclusion that we can draw from this example is that when there is a mismatch of the logic/voltage assignment between an output signal and an output terminal, we need to transform the logic/voltage assignment of the signal to match the logic/voltage assignment of the output terminal by again using the following identities: $X.H = \overline{X.L}$ and $\overline{X.H} = X.L$. ■ ■

We will now use the concepts of this section in a consideration of some specific SSI devices.

3.3.1 74'00

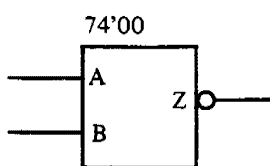
The 74'00 has two input terminals, which we will call terminals A and B, and one output terminal, which we will call terminal Z. From a TTL data book, we can determine the following voltage table for the 74'00.

A	B	Z
L	L	H
L	H	H
H	L	H
H	H	L

This table specifies how the 74'00 actually works, physically speaking. But what it does, logically speaking, depends on our logic/voltage assignment. Consider the assignment of

A—active-high B—active-high Z—active-low

For this assignment, the corresponding graphical representation is that of Fig. 3.5(a), and the 74'00 voltage table translates into the logic table of Fig. 3.5(b). From this table we see that $Z = A \cdot B$, and so the 74'00 is an AND gate for this specific logic/voltage assignment.



(a)

A	B	Z
F	F	F
F	T	F
T	F	F
T	T	T

(b)

Figure 3.5 74'00 AND gate.

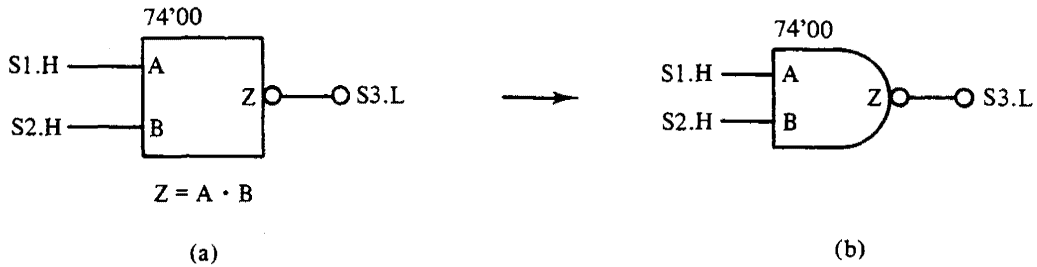
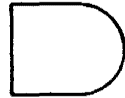


Figure 3.6 Use of the AND graphic symbol.

The commonly accepted graphic symbol for an AND gate is



As illustrated in Fig. 3.6, we will use this symbol to replace the rectangular box for the 74'00. Then there is no need to specify the logic operation.

To recapitulate, from Fig. 3.6(b) we see that the 74'00 has two input terminals (A and B) and one output terminal (Z). Having no circles, both input terminals expect active-high signals. And, because of the circle at terminal Z, the output signal is active-low. As indicated by the shape of the symbol, the logic function performed on the signals applied at terminals A and B is the AND operation $Z = A \cdot B$, provided that the signals applied are active-high and provided that we interpret the output signal to be active-low. In this particular case, $S3 = S1 \cdot S2$, and the signal S3 is active-low.

Now, consider the logic/voltage assignment of

A—active-low B—active-low Z—active-high

For this assignment, the corresponding graphical representation is that of Fig. 3.7(a), and the 74'00 voltage table translates into the logic table of Fig. 3.7(b). Rearranging this logic table, we obtain the standard logic table of Fig. 3.7(c). From it we see that $Z = A + B$, and so the 74'00 is an OR gate for this specific logic/voltage assignment.

The commonly accepted graphic symbol for an OR gate is



As illustrated in Fig. 3.8, we will use this symbol to replace the rectangular box for the 74'00. Then we do not have to specify the logic operation.

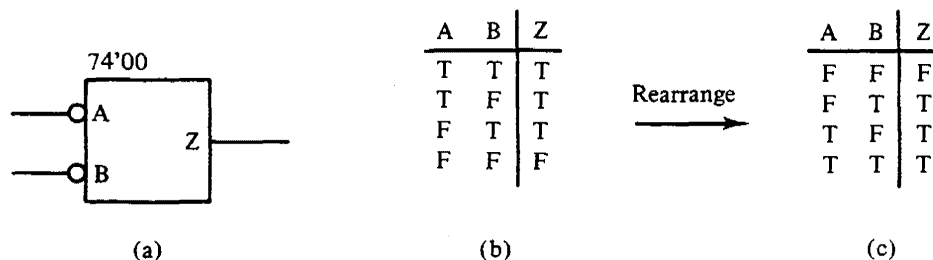


Figure 3.7 74'00 OR gate.

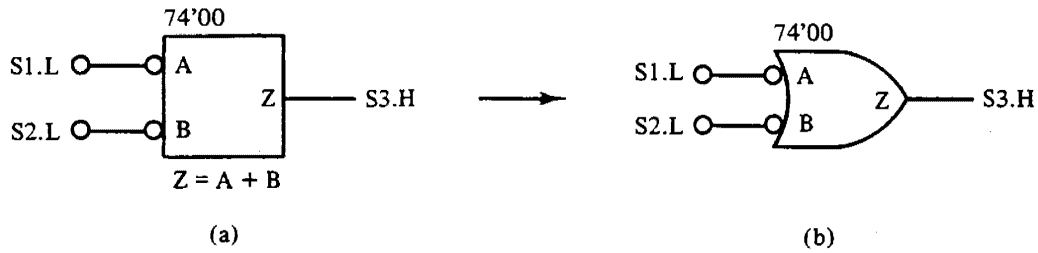


Figure 3.8 Use of the OR graphic symbol.

In Fig. 3.8(b) the two circles at both input terminals indicate that these terminals expect active-low input signals. And, because there is no circle at terminal Z, the output signal is active-high. As indicated by the shape of the symbol, the logic function performed on the signals applied at terminals A and B is the OR operation $Z = A + B$, provided that the applied signals are active-low and provided that we interpret the output signal to be active-high. In this particular case, $S3 = S1 + S2$, and the output signal S3 is active-high.

We have considered two of the eight possible logic/voltage assignments for the three terminals of a 74'00. All the possible assignments are

Assignments	Signals			
	A	B	Z	
(1)	.L	.L	.L	
(2)	.L	.L	.H	← OR
(3)	.L	.H	.L	
(4)	.L	.H	.H	
(5)	.H	.L	.L	
(6)	.H	.L	.H	
(7)	.H	.H	.L	← AND
(8)	.H	.H	.H	

We have shown that the 74'00 implements an OR gate for assignment (2), and an AND gate for assignment (7). The remaining six assignments do not implement any of the three fundamental Boolean operations (AND, OR, NOT), but rather a combination of them.

Of the remaining six assignments, the most popular one for the 74'00 is assignment (8), in which all signals are active-high. For this assignment, the corresponding graphical representation is that of Fig. 3.9(a), and the 74'00 voltage table translates into the logic table of Fig. 3.9(b). From this table we see that for an all active-high assignment, the

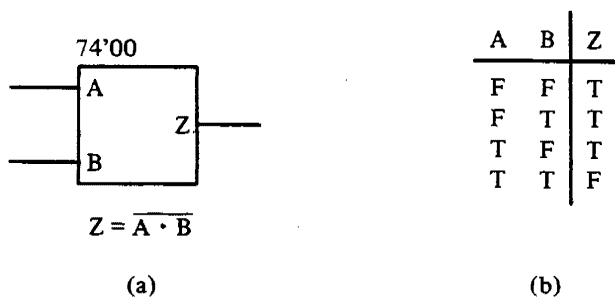
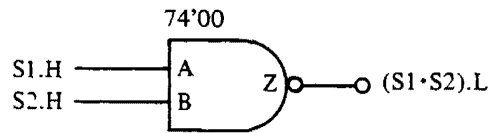


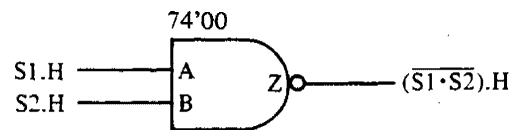
Figure 3.9 74'00 NAND gate.

74'00 performs the NAND operation, which is the AND operation followed by the NOT operation. So, $Z = \overline{A \cdot B}$. As a result, in the popular positive-logic convention in which all signals are active high, the 74'00 is always a NAND gate.

To indicate the use of the 74'00 as a NAND gate, consider the following illustration. In Fig. 3.6 we have indicated the use of the 74'00 as an AND gate as follows:



From the identity $X.L = \overline{X}.H$, this is equivalent to



Therefore a mismatch of the voltage representation between the output terminal and the output signal results in a logic inversion of the function performed by the 74'00, causing it to go from an AND operation to a NAND operation.

We should remember that logic values (T, F) and the voltage representations of these logic values are two separate concepts. Also, although the voltage table of a device describes its physical behavior, the logic operation that the device performs in a particular digital circuit depends on our assignment of voltage representations to the logic values.

3.3.2 74'02

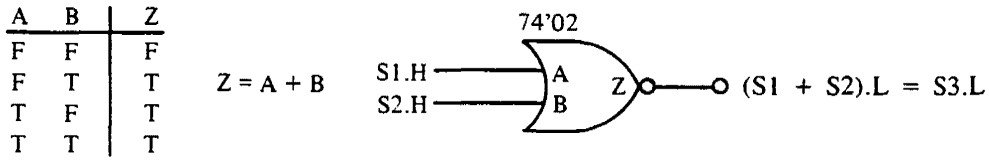
In a similar fashion we can consider the 74'02, which also has three terminals. But, we will omit some of the obvious details. The voltage table for the 74'02 is

A	B	Z
L	L	H
L	H	L
H	L	L
H	H	L

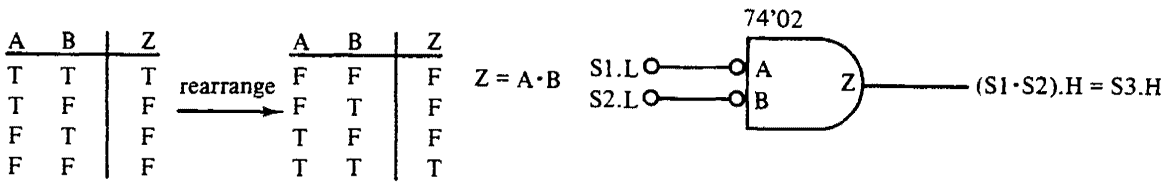
All possible logic/voltage assignments are, of course,

Assignments	Signals		
	A	B	Z
(1)	.L	.L	.L
(2)	.L	.L	.H
(3)	.L	.H	.L
(4)	.L	.H	.H
(5)	.H	.L	.L
(6)	.H	.L	.H
(7)	.H	.H	.L
(8)	.H	.H	.H

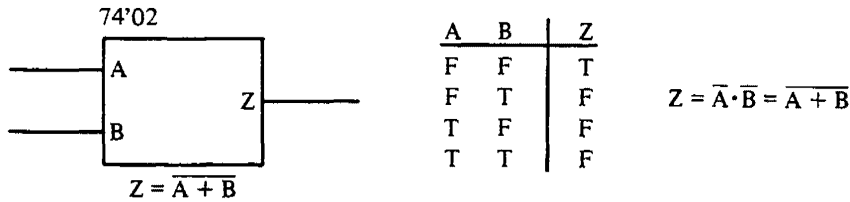
For assignment (7) of active-high input terminals and an active-low output terminal, we obtain the following logic table. From it we can see that for this assignment, the 74'02 is an OR gate.



For assignment (2) of active-low input terminals and an active-high output terminal, we obtain the following logic table. From it we see that for this assignment, the 74'02 is an AND gate.

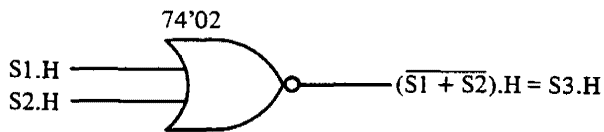


For assignment (8), that of all active-high terminals, we obtain the following block diagram and NOR logic table.



So, with the positive-logic convention in which all signals are active-high, the 74'02 is always a NOR gate.

The 74'02 performing as a NOR gate results from a mismatch of the voltage representation between the output terminal and the output signal, causing the operation to change from OR to NOR.



3.3.3 74'04

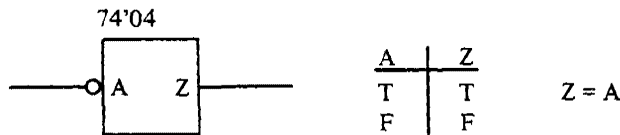
We will now consider the 74'04, which has only one input terminal and one output terminal, which we will label A and Z, respectively. The 74'04 voltage table is

A	Z
L	H
H	L

Since the 74'04 has only two terminals, there are just four possible logic/voltage assignments:

Assignments	Signals	
	A	Z
(1)	.L	.L
(2)	.L	.H
(3)	.H	.L
(4)	.H	.H

For assignment (2) the graphical representation and logic table are as follows:



With this assignment, the 74'04 implements an IDENTITY logic operation, which means it does not do anything *logically*. But, of course, it does change the voltage representation from active-low to active-high, and so is useful in digital circuits that have mismatched voltage representations.

The commonly accepted symbol for an IDENTITY gate, also known as a *buffer*, is



We will use this symbol to replace the rectangular box for the 74'04, as illustrated in Fig. 3.10.

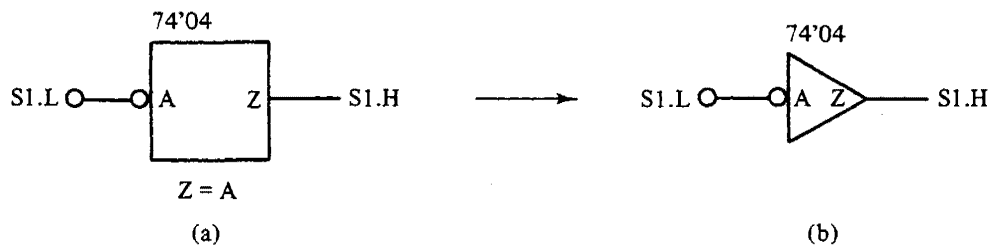


Figure 3.10 Use of the buffer graphic symbol.

In Fig. 3.10(b) the shape of the symbol tells us that the 74'04 performs the IDENTITY logic operation on A, making $Z = A$. Also, with a circle at the input but none at the output, the gate functions as a *voltage* inverter, transforming S1.L to S1.H for this assignment (2).

For assignment (3) the graphical representation and logic table are



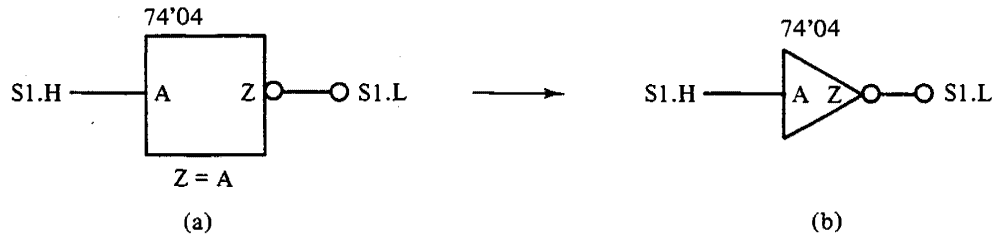
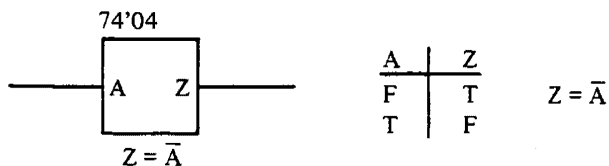


Figure 3.11 Active-high input voltage inverter.

Again, the 74'04 implements an IDENTITY operation. In this case, though, it changes the voltage representation from active-high to active-low. Also, as shown in Fig. 3.11, we can again use the buffer symbol instead of the rectangular box.

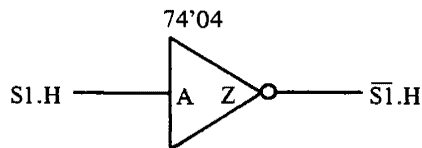
In Fig. 3.11(b) the shape of the symbol tells us that for assignment (3) the 74'04 performs the IDENTITY operation on A, making $Z = A$. Also, with no circle as the input but with one at the output, the gate functions as a *voltage* inverter, transforming S1.H to S1.L.

For assignment (4) of an active-high input and also output, the block diagram and the logic table are

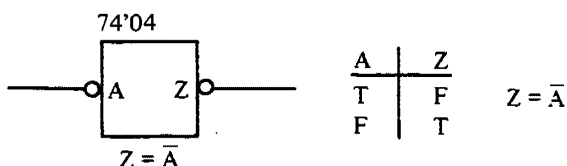


Therefore for the assignment of both signals active-high, the 74'04 functions as a *logic* inverter, but not a *voltage* inverter. In the positive-logic convention in which all signals are active-high, the 74'04 is always a logic inverter—a NOT gate.

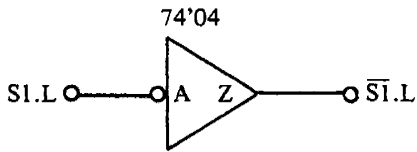
In this case the logic inversion action of the 74'04 results from a mismatch of the voltage representation between the output terminal and the output signal, as follows:



Finally, for assignment (1) of active-low input and also output, the block diagram and the logic table are



from which we see that the 74'04 performs the logic NOT operation. So, with both signals active-low, the 74'04 again functions as a logic inverter, but not a voltage inverter. With the negative-logic convention, in which all signals are active-low, the 74'04 is again always a logic inverter. This logic inversion action of the 74'04 also results from a mismatch of the voltage representation between the output signal and output terminal, as follows:



3.3.4 SSI Basic Gate Summary

Figure 3.12 summarizes the most important interpretations of some of the popular SSI devices described in TTL data books.

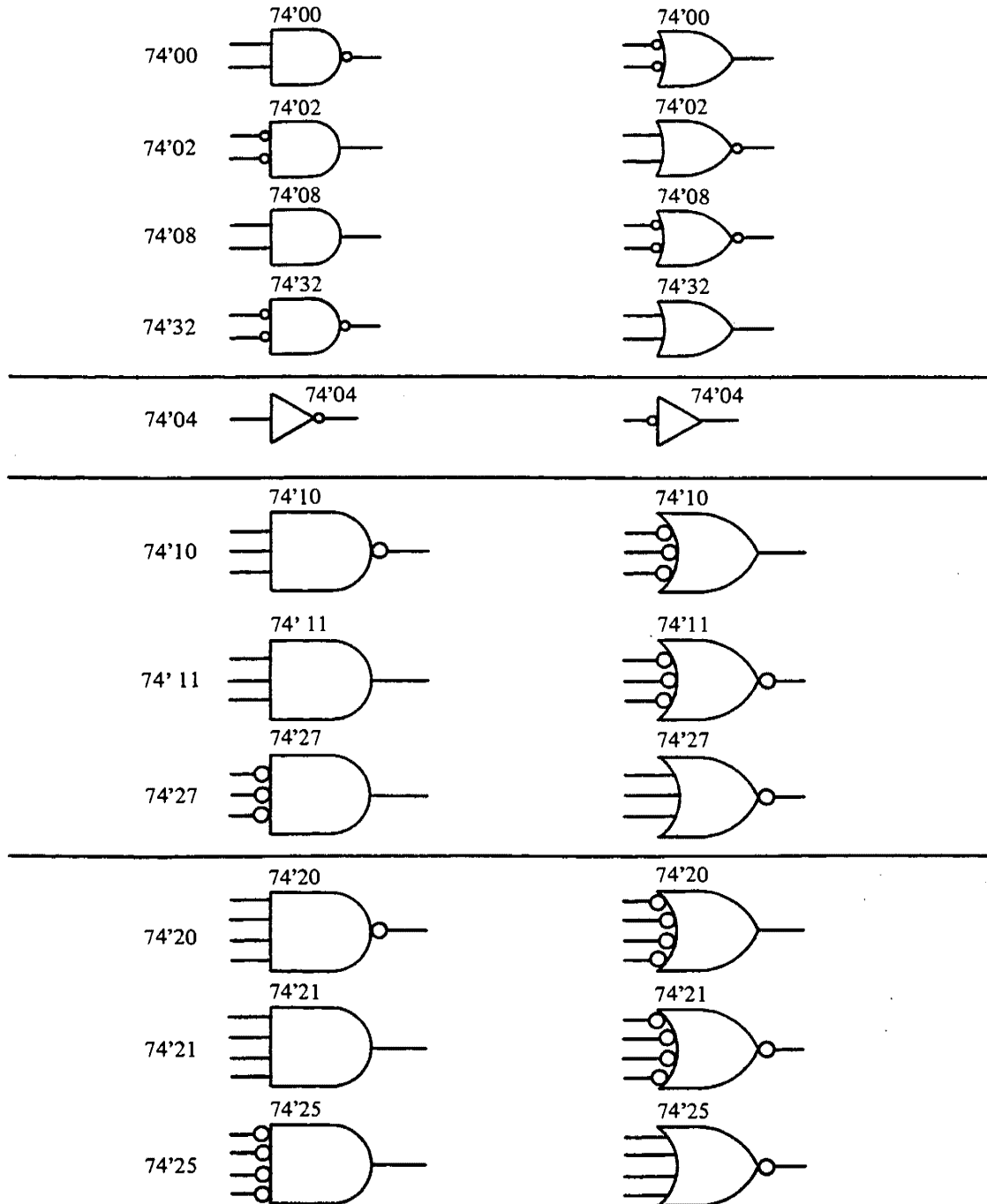


Figure 3.12 TTL basic gate summary.

3.4 SYNTHESIS OF DIGITAL CIRCUITS

Our main goal in this chapter is to learn how to synthesize (design) digital circuits with SSI components, starting with truth table or K-map specifications of the Boolean functions to be implemented. For this design process we must learn how to select the proper SSI components and how to interconnect them such that the resulting digital circuit performs the logic function specified by the logic expression (or truth table). Part of the design process is to make certain that the voltage representations are consistent among the signals. Specifically, we must connect active-high signals to active-high terminals, and connect active-low signals to active-low terminals.

3.4.1 Synthesis Based on the Positive-Logic Convention

One sure way to guarantee that the voltage representations among signals are consistent is to assign them all to be active high. As mentioned, this approach is commonly called the *positive-logic convention*. Currently, it is the most popular approach.

For the positive-logic convention, we can assume in Fig. 3.12 that all the device terminal output circles correspond to logic inversion. (The same is true for the input circles.) In other words, these circles at the outputs have the same effect as NOT gates (logic inverters). This observation follows from the facts that there are no active-low signals in the positive-logic convention and, as we proved in Sec. 3.3, $X.L = \bar{X}.H$. Thus, we see from Fig. 3.12 that the 74'00, 74'10, and 74'20 are positive-logic NAND gates; the 74'02, 74'27, and 74'25 are positive-logic NOR gates; and the 74'04 is a logic inverter. These positive-logic descriptions are the ones that digital-circuit manufacturers specify in their data books along with the 74 labels.

We will study the positive-logic convention approach by way of examples. In them, we will use only MSOPs since the extension to designing with MPOSs should be apparent.

EXAMPLE 3.4

Using the positive-logic convention, design a digital circuit based on the following truth table:

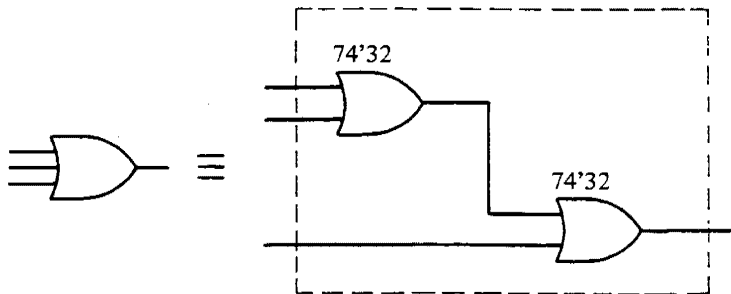
A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Solution. Our first step is to obtain an MSOP, as follows:

	A	0	1	
BC		0	1	
00		0	1	
01		1	1	
11		0	0	
10		1	0	

$$Z = \overline{A}B + \overline{B}C + \overline{A}B\overline{C}$$

As should be apparent, to implement this MSOP we need 2 two-input AND gates, 1 three-input AND gate, 1 three-input OR gate and some inverters. Of course, the implementation would be very straightforward if we had the following gates: 74'08 (two-input AND), 74'11 (three-input AND), 74'04 (inverter), and a three-input OR gate. Since there is no three-input TTL OR gate, we could, instead, use two 74'32 gates (two-input OR), as follows:



A problem arises, though, if, say, we have only gates 74'00, 74'02, 74'04, 74'10, and 74'27, since in this list there are no AND and OR gates. Because we fixed our voltage representation (all signals are active-high), our interpretations of these gates are also fixed. From Fig. 3.12 they are

74'00—two-input NAND	74'10—three-input NAND
74'02—two-input NOR	74'27—three-input NOR
74'04—NOT	

So all we have available are NAND and NOR gates and some inverters. Consequently, to implement Z we must transform the MSOP (which is in terms of AND, OR, and NOT) into an equivalent expression in terms of NAND, NOR, or NOT. To change an OR expression into a NAND expression, we can double complement (which gives an equivalent expression), and in so doing, use one complement to change the ORs to ANDs as required for NAND, and use the other complement for the inversion part of the NAND operation:

$$\begin{aligned}
 Z &= \overline{A}B + \overline{B}C + \overline{A}B\overline{C} \\
 &= \overline{\overline{\overline{A}B} + \overline{\overline{\overline{B}C} + \overline{\overline{\overline{A}B\overline{C}}}}} \\
 &= \overline{\overline{A}B} \cdot \overline{\overline{B}C} \cdot \overline{\overline{A}B\overline{C}} \quad (\text{DeMorgan's law})
 \end{aligned}$$

By using a pair of two-input NAND gates (74'00) and a three-input NAND gate (74'10) at the input level, we can obtain $\overline{\overline{A}B}$, $\overline{\overline{B}C}$, and $\overline{\overline{A}B\overline{C}}$. Then, with a single three-input NAND gate (74'10) at the second level, we can obtain the NAND

of these three terms. Figure 3.13 shows the resulting digital circuit. Note that the literal inputs at the first level are the literals appearing in the original MSOP. This is generally true for a two-level NAND implementation. (Similarly, a two-level NOR realization has as inputs the literals appearing in an MPOS.) ■ ■

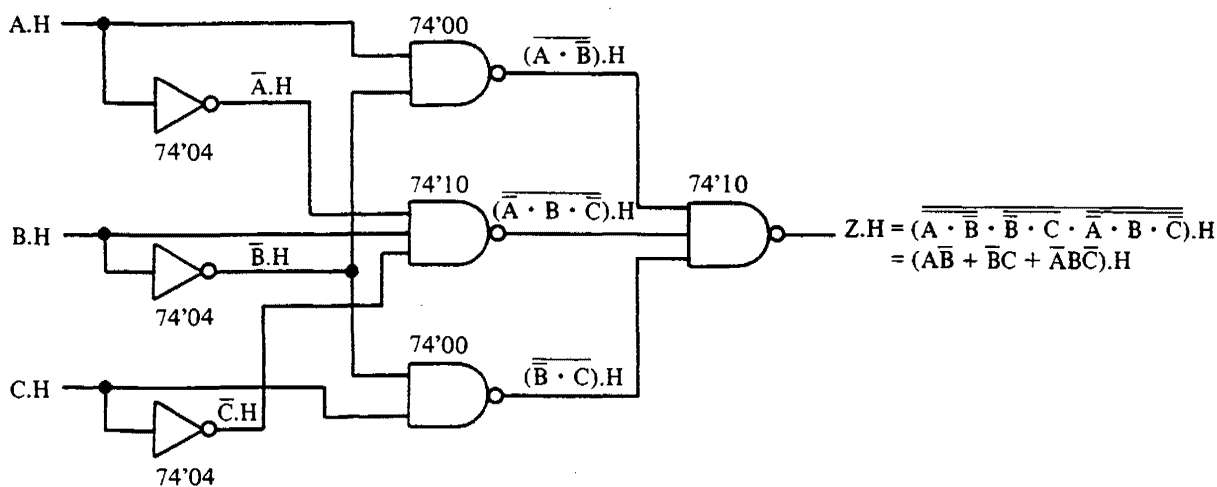


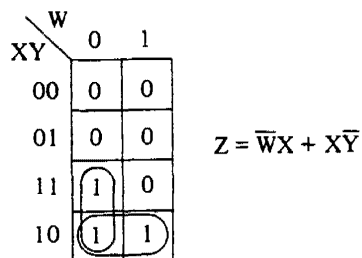
Figure 3.13 Implementation for Example 3.4.

EXAMPLE 3.5

Using positive logic, design a digital circuit for producing the function Z defined in the following truth table.

W	X	Y	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Solution. As before, our first step is to find an MSOP:



For our implementation of $Z = \bar{W}X + X\bar{Y}$, again assume that we have only the following gates with the corresponding interpretations (because all signals are active-high):

- 74'00—two-input NAND
- 74'10—three-input NAND

74'02—two-input NOR 74'27—three-input NOR
74'04—NOT

Since all we have available are NAND and NOR gates and inverters, we must transform the MSOP expression into an expression in terms of these operations. As before, we will double complement, and then apply one of DeMorgan's laws:

$$Z = \overline{\overline{WX}} + \overline{\overline{XY}} = \overline{\overline{\overline{WX}} + \overline{\overline{XY}}} = \overline{\overline{WX} \cdot \overline{\overline{XY}}}$$

This expression is in a form requiring 3 two-input NAND gates and some inverters for the implementation. For the sake of illustration, we will make another assumption, which is that we have only 2 two-input NAND gates. In this case, then, we must eliminate one of the NAND operations. We can do this by applying one of DeMorgan's laws to one of the terms:

$$Z = \overline{\overline{WX} \cdot \overline{\overline{XY}}} = \overline{\overline{WX} \cdot (\overline{\overline{X}} + \overline{\overline{Y}})} = \overline{\overline{WX} \cdot (\overline{\overline{X}} + \overline{\overline{Y}})}$$

The double complementing of the OR term, besides giving an equivalent term, corresponds to NOR followed by NOT. Now we can implement the expression with a pair of two-input NAND gates, 1 two-input NOR gate, and some inverters, as shown in Fig. 3.14. ■ ■

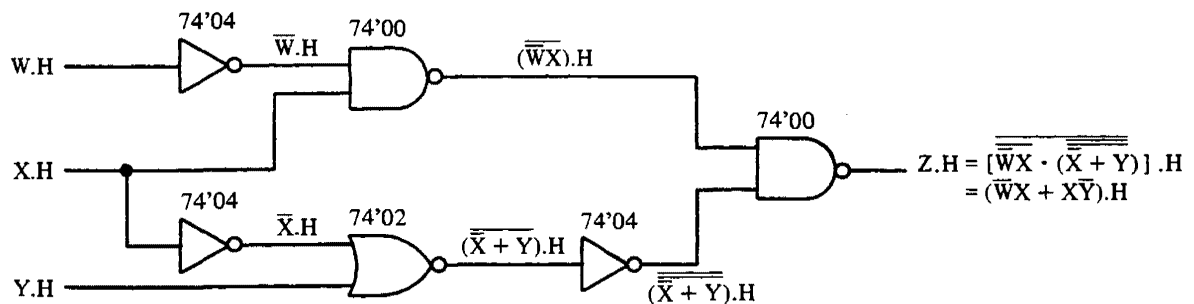


Figure 3.14 Implementation for Example 3.5.

As is evident, a major advantage of the positive-logic approach is the consistency of signal voltage representations since all signals are active-high. A disadvantage is that we sometimes have to transform the original logic expression into some initially obscure expression in order to implement it with the available gates. For this, DeMorgan's laws are very helpful.

3.4.2 Synthesis Based on the Negative-Logic Convention

Another sure way to guarantee that the voltage representations are consistent is to assign them all to be active-low. This approach, commonly called the *negative-logic convention*, is analogous to the approach based on the positive-logic convention. Since the voltage representations are fixed (all signals are active-low), we are, for example, limited to the use of only one of the eight logic/voltage assignments for a two-input, single-output gate, such as the 74'00. Consequently, we must still transform the original logic expression to "fit" that interpretation of the gate. Because of the similarity to the positive-logic convention, we will not consider any examples here.

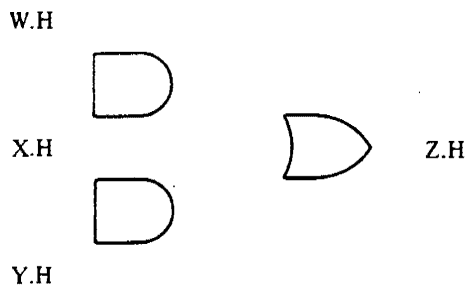
3.4.3 Synthesis Based on the Mixed-Logic Convention

The third approach to digital circuit synthesis is based on the *mixed-logic convention* in which both active-high and active-low signals are allowed. Since we are not restricted to a particular logic/voltage assignment (where *all* the signals are either active-high or active-low), we can choose the voltage assignment of each signal at implementation time. As a result, we can use all eight logic/voltage assignments of a gate such as the 74'00. Also, we can base our designs on the AND, OR, and NOT operations, and so directly implement an MSOP or an MPOS without any algebraic manipulation. With this approach, we must, of course, take care to ensure that the voltage representations are consistent between signals and terminals. Perhaps the mixed-logic approach can be best introduced by way of an example.

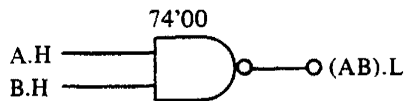
EXAMPLE 3.6

Using the mixed-logic convention, directly implement $Z = \overline{W}X + X\overline{Y}$ from Example 3.5 in Sec. 3.4.1. The available SSI gates are the 74'00, 74'02, 74'04, 74'10, and 74'27. The input signals are W.H, X.H, and Y.H, and the output signal is to be Z.H.

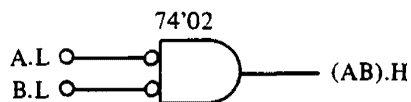
Solution. To implement the logic expression directly for Z , we need a pair of two-input AND gates, a two-input OR gate, and some inverters. (Note that we do *not* make any transformation of the original expression.) We will make a preliminary sketch of the circuit diagram, showing the AND and OR gate placements.



Now, do we have any AND and OR gates among our available gates? Looking at Fig. 3.12, we see that the 74'00 is a two-input AND gate if we use the following voltage assignments:



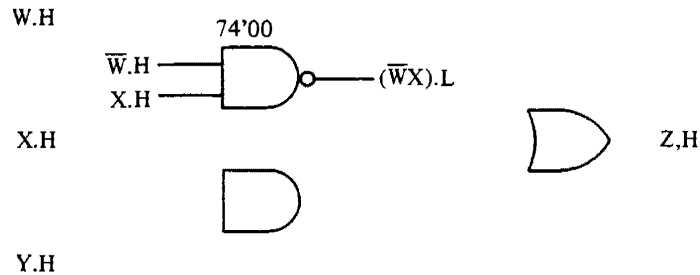
Furthermore, the 74'02 is also a two-input AND gate if we use the following voltage assignments:



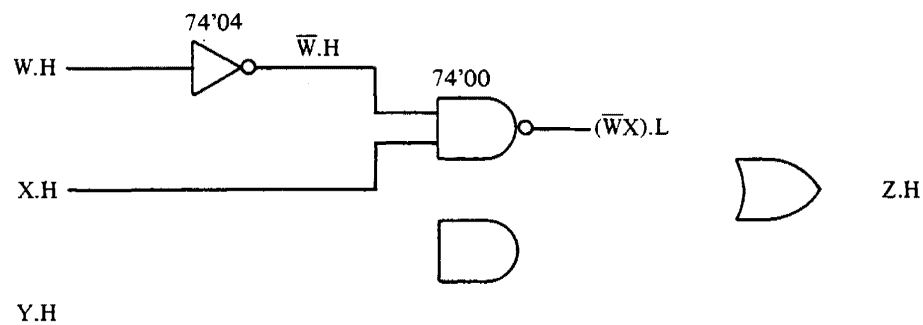
So, in our list of available gates, we have not one, but two types of two-input AND gates! Similarly, we have two types of two-input OR gates:



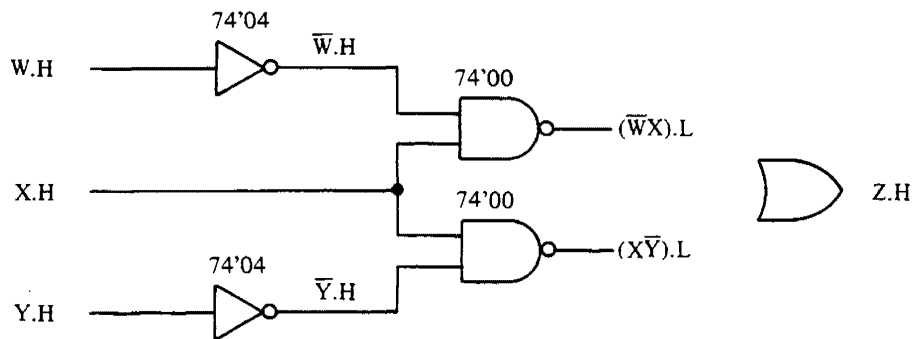
Both these AND and OR gates are available for our selection. Now, back to the example. Do we use the 74'00 or the 74'02 for our two-input AND gates? Since at this point it really doesn't matter, we will arbitrarily select the 74'00 for the first two-input AND gate.



With the operation specified, we have fixed the voltage representations of all the signals of this 74'00. For the AND operation, the inputs are active-high and the output is active-low. Since we want this gate to generate the term $\bar{W}X$, we need inputs of X.H and $\bar{W}.H$. Of course, X.H is available. In addition, we can generate $\bar{W}.H$ from the available W.H by using a 74'04 as a logic inverter. This is possible because $\bar{W}.H = W.L$.



Similarly, we can generate the term $X\bar{Y}$.



There is no special reason to choose the 74'00 for this second AND gate other than to have the voltage representation (active-low) of the output of this gate the same as the voltage representation of the output of the other AND gate.

Finally, we need to OR $\bar{W}X$ and $X\bar{Y}$. For this, we should use the 74'00 version of the OR gate, rather than the 74'02 version, since these two signals are active-low, and the 74'00 OR gate requires active-low inputs. An added bonus is

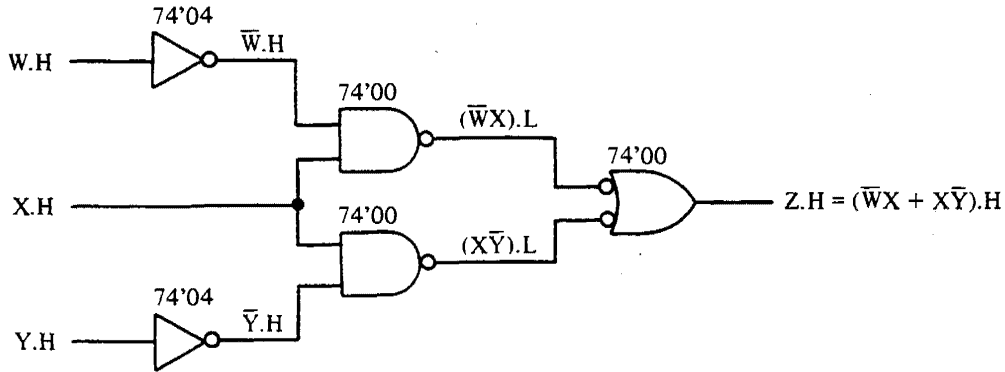


Figure 3.15 Mixed-logic implementation for Example 3.6.

that the output of the 74'00 OR gate is active-high, as is required for the output signal. The complete circuit is shown in Fig. 3.15.

This implementation of $Z = \overline{W}X + XY$ is functionally equivalent to that of Fig. 3.14 which, although for the same function, is based on a different approach—that of positive logic. Even with the same logic convention, we can obtain different implementations, some of which may be better than others. Figure 3.16 shows another mixed-logic implementation of $Z = \overline{W}X + XY$. ■ ■

The functionally equivalent implementations of Figs. 3.15 and 3.16 have the same number of gates. Usually, though, different implementations have different numbers of gates, primarily of inverters. Optimally, of course, we want an implementation with the least number of inverters, and of gates in general. There are no definitive rules for obtaining such a “best” implementation. It is a skill that can be enhanced through practice. The following guidelines are helpful, however.

1. Determine the framework of the digital circuit by defining all the input and output signals and sketching in the required AND and OR gates.
2. Pick one of the gates, preferably at the input level, and select a specific TTL gate for it. This selection fixes the voltage assignments for the gate input and output signals.
3. Obtain the required input signals for this gate from the available signals, using inverters and/or the “other label” ($A.H = \overline{A}.L$), when necessary.
4. Repeat steps 2 and 3 for the remaining gates until the circuit is complete. To minimize the number of inverters, you may, from time to time, have to back up and reconsider your choice of specific TTL gates when it is obvious that you have made a poor choice.

We will illustrate these guidelines with an example.

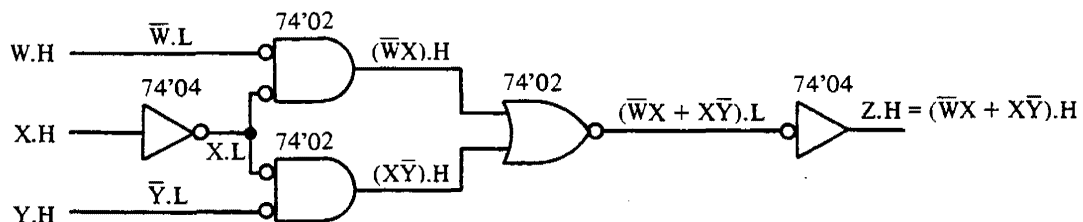
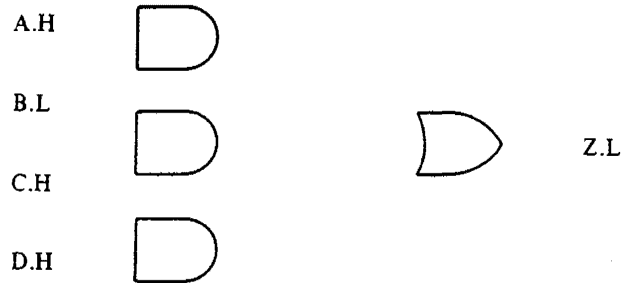


Figure 3.16 Another mixed-logic implementation for Example 3.6.

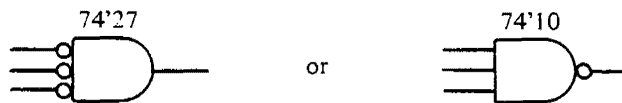
EXAMPLE 3.7

Implement $Z = \bar{A}BC + CD + A\bar{C}$ using any gates of Fig. 3.12. The inputs are A.H, B.L, C.H, and D.H, and the output is to be Z.L.

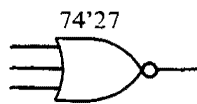
Solution. Determine the circuit framework (step 1).



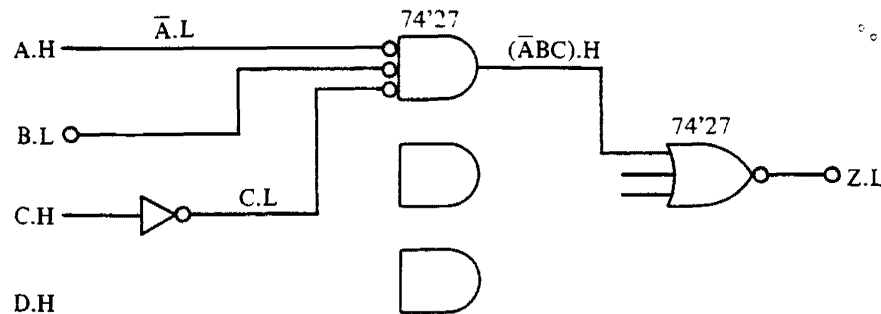
Now (step 2) pick a gate at the first level and select the specific TTL gate for it. We will start with the first three-input AND gate. Which TTL gate should we use?



Normally, at this point in the design it would not matter which TTL gate we selected. But we know that at the second level we want a three-input OR gate with an active-low output to generate the output Z.L, and in Fig. 3.12 the only such OR gate is the 74'27, which requires active-high inputs.



Therefore, for the first AND gate we prefer the 74'27, at least temporarily, since it has an active-high output. Now obtain the required input signals (step 3).



In accordance with step 4, repeat the process until the complete circuit is designed, as shown in Fig. 3.17.

Note in Fig. 3.17 that the second AND gate is a 74'00 rather than a 74'02 version. If we had selected the 74'02 instead, then we would have needed two inverters for the two inputs and no inverter for the output. But with the 74'00, we need no inverters for the inputs and just one inverter for the output. Sometimes by making the appropriate choice of gates, we can reduce the number of inverters

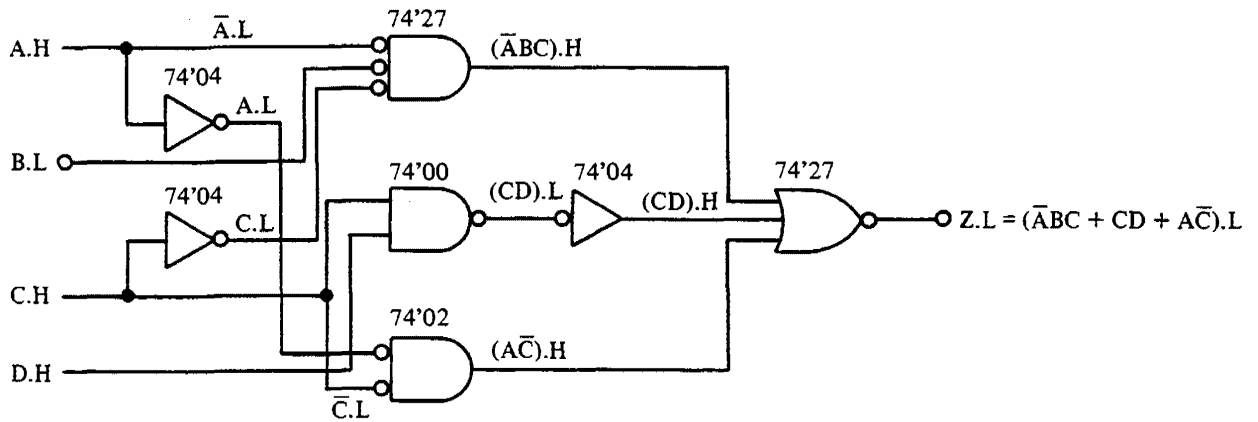


Figure 3.17 Mixed-logic implementation for Example 3.7.

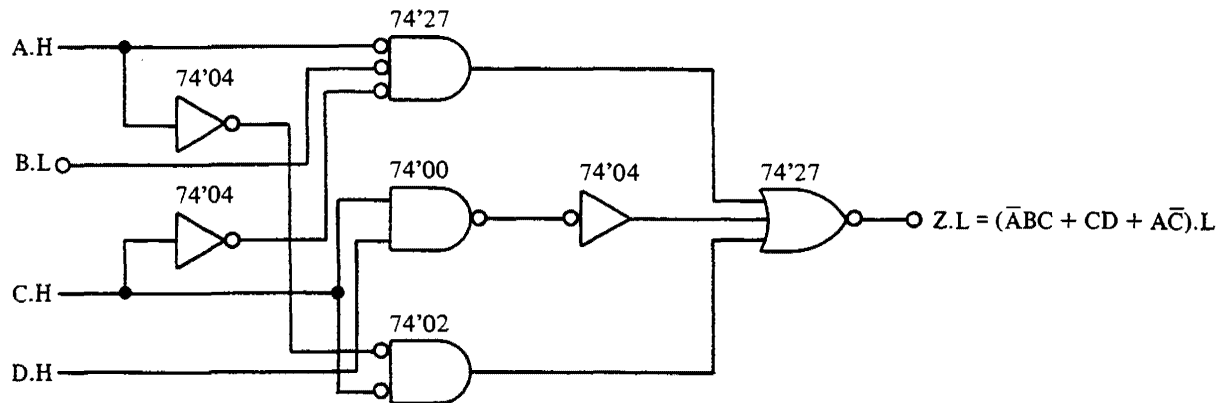


Figure 3.18 Preferred circuit diagram for Example 3.7.

required. Saving inverters is desirable, especially when board space is limited. But that should not be an overriding concern. Having a clear and structured design is more important.

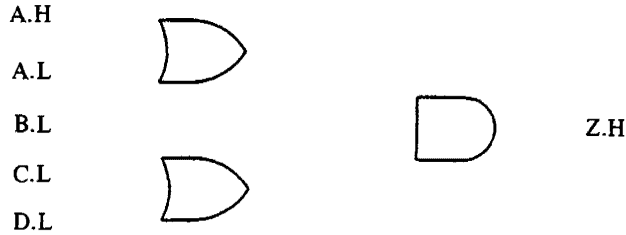
In Fig. 3.17 all the intermediate terms are labeled. Usually, though, this is not desirable, and they can be omitted, as shown in Fig. 3.18.

When we finish our design, we should check our circuit diagram. In analyzing a circuit diagram we should realize that the circle at an input or output terminal can implement a voltage inversion or a logic inversion operation, depending on the context. If there is a mismatched logic/voltage assignment between a signal and a terminal, then it is a logic inversion operation. Otherwise, it is a voltage inversion operation and can be ignored in the analysis of the circuit. With this in mind, we can readily see in Fig. 3.18 that the output of the three-input 74'27 AND gate is \overline{ABC} , and the output of the 74'02 AND gate is $A\overline{C}$, even though these intermediate terms are not labeled. ■ ■

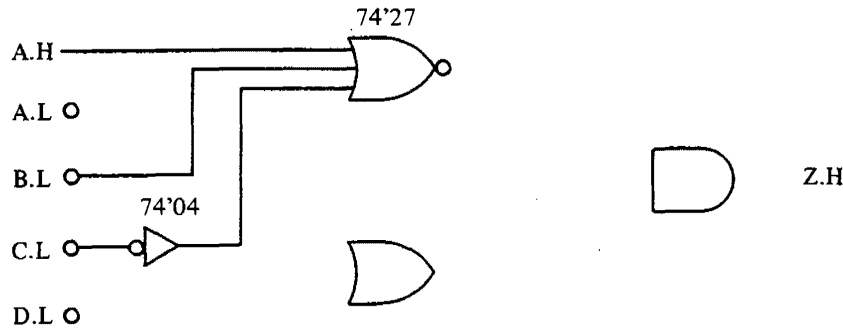
EXAMPLE 3.8

Directly implement $Z = \overline{(A + \overline{B} + C)} \cdot (A + \overline{D})$. The inputs are A.H, A.L, B.L, C.L, and D.L, and the output is to be Z.H.

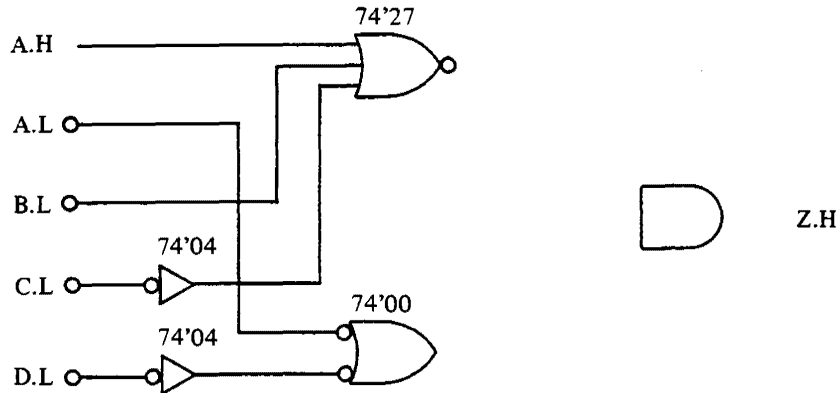
Solution. Step 1: Determine the framework of the circuit.



Steps 2 and 3: Select a TTL gate for one of the first-level gates. Then, generate the gate input signals.



Step 4: Do the same for the other first-level gate.



Continuing for the second-level gate, we obtain the circuit of Fig. 3.19. Note how the term $(A + \bar{B} + C)$ is complemented to $\overline{(A + \bar{B} + C)}$.

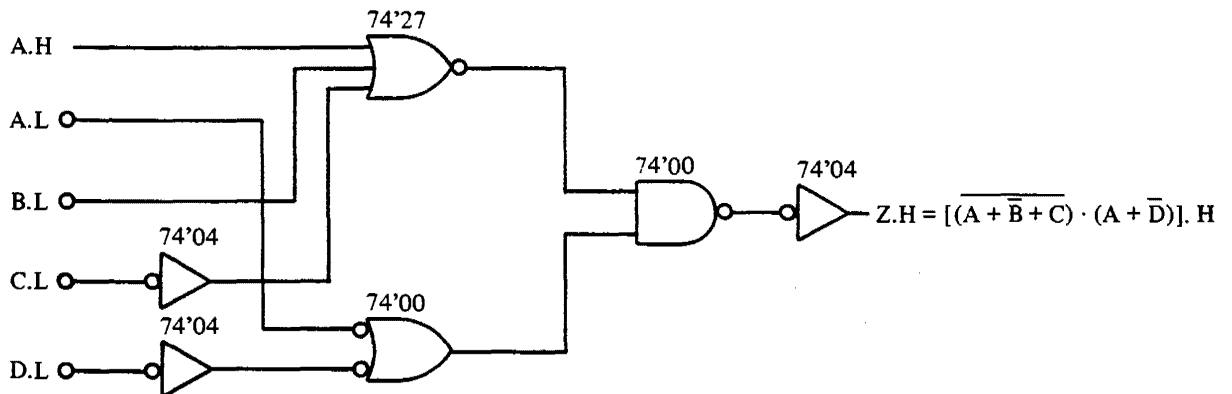


Figure 3.19 Mixed-logic implementation for Example 3.8.

As a check, and also as an exercise, you should analyze the circuit diagram of Fig. 3.19 to determine whether the output is $Z = (\overline{A} + \overline{B} + C) \cdot (A + \overline{D})$. As mentioned, to do this you should ignore the circles if they are voltage inverters and consider each mismatched logic/voltage assignment as producing a logic NOT.

Incidentally, with different choices for the AND and OR gates, the number of inverters required can be reduced from three to one. We will leave it to the reader to try it. ■ ■

3.5 SUMMARY—LOGIC CONVENTIONS

The positive- and negative-logic conventions are simply special cases of the mixed-logic convention. An advantage of the positive-logic approach (and also of the negative-logic approach) is the certainty of matching between signals and gate terminals, since all are active-high (or active-low). The trade-off is a loss of some flexibility. For example, for a gate such as the 74'00 with three signals, there are $2^3 = 8$ different possible logic/voltage assignments. Yet, with the positive-logic approach we can select only one of the eight assignments—that of all signals being active-high. Consequently, we are forced to transform the original logic expression to “fit” *that* interpretation of the gate. This lack of flexibility is also true of the negative-logic approach.

With the mixed-logic approach we do not assign the logic/voltage assignment until implementation time. Therefore we have the freedom to use all possible assignments for a gate. Consequently, we do not need to transform the original expression to fit the gates. Rather, we change the interpretations of the gates *to fit the logic expression*. The trade-off for the mixed-logic approach is the necessity of making certain that the voltage representations between signals and gate input requirements are consistent (sometimes using inverters, sometimes using the “other label”).

For the positive- and negative-logic approaches, circuit synthesis generally requires Boolean algebraic manipulation of logic expressions. For the mixed-logic approach, however, circuit synthesis generally requires graphical manipulation. For the design of simple circuits that are implemented with SSI devices, the selection of the easiest approach is a matter of personal preference. But the advantages of the mixed-logic approach become more apparent in the design of more complex circuits with MSI and LSI components. Also, the mixed-logic approach becomes increasingly valuable as the clarity of the digital design becomes more important.

SUPPLEMENTARY READING (see Bibliography)

[Boole 54], [Fletcher 80], [Kintner 71], [Mano 84], [Motorola], [Prosser 87], [Taub 82], [Texas Instruments]

PROBLEMS

- 3.1. Explain the differences among the positive-logic, negative-logic, and mixed-logic conventions.
- 3.2. What is the difference between logic values and voltage levels?

3.3. Complete the solution for Example 3.3 in Sec. 3.3.

3.4. Consider the device of Fig. 3.20.

(a) Complete the following table:

Input voltage values			Output logic values (0 = false, 1 = true)	
S1	S2	S3	SA	SB
L	L	L		
L	L	H		
L	H	L		
L	H	H		
H	L	L		
H	L	H		
H	H	L		
H	H	H		

(b) Find the logic equations for SA and SB as functions of S1, S2, and S3.

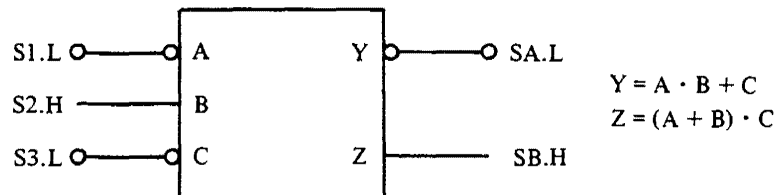


Figure 3.20 Device for Problem 3.4.

3.5. Repeat Problem 3.4 for the device of Fig. 3.21.

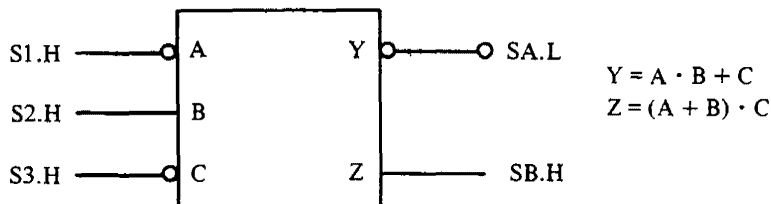


Figure 3.21 Device for Problem 3.5.

3.6. Repeat Problem 3.4 for the device of Fig. 3.22.

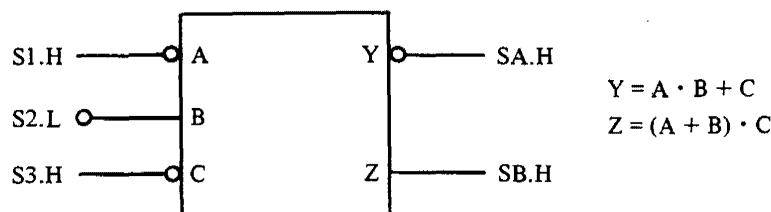


Figure 3.22 Device for Problem 3.6.

3.7. Given the following logic table and that the logic/voltage assignment is A active-low, B active-high, C active-high, and Z active-low, determine the corresponding voltage table. Arrange it in the conventional manner.

A	B	C	Z
F	F	F	F
F	F	T	T
F	T	F	T
F	T	T	F
T	F	F	T
T	F	T	F
T	T	F	T
T	T	T	T

- 3.8. (a) Show how a 74'00 gate can be made into an inverter.
 (b) Is it a logic inverter or a voltage inverter? Explain.
- 3.9. (a) By looking into a TTL data book, find the 74'Y component corresponding to the following logic symbol:



- (b) Determine the voltage table for it.
 (c) List the eight possible logic/voltage assignments.
 (d) For each of the eight assignments, determine (using the voltage and logic tables) the logic function that this component performs.
 (e) What logic function does it perform for the positive-logic convention?
 (f) What logic function does it perform for the negative-logic convention?
- 3.10. Repeat Problem 3.9 for the following logic symbol:



- 3.11. Find voltage tables for the devices of Fig. 3.23.

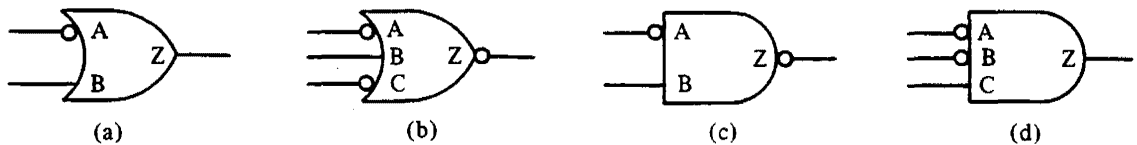


Figure 3.23 Devices for Problem 3.11.

- 3.12. Find the logic expressions for the Z outputs in Fig. 3.24.

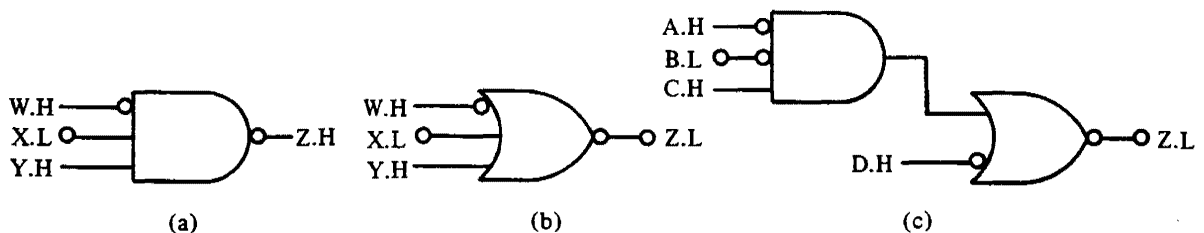


Figure 3.24 Devices for Problem 3.12.

- 3.13. Fill in all the intermediate signal names for the mixed-logic circuit diagram of Fig. 3.25. Also, find a logic expression for Z.

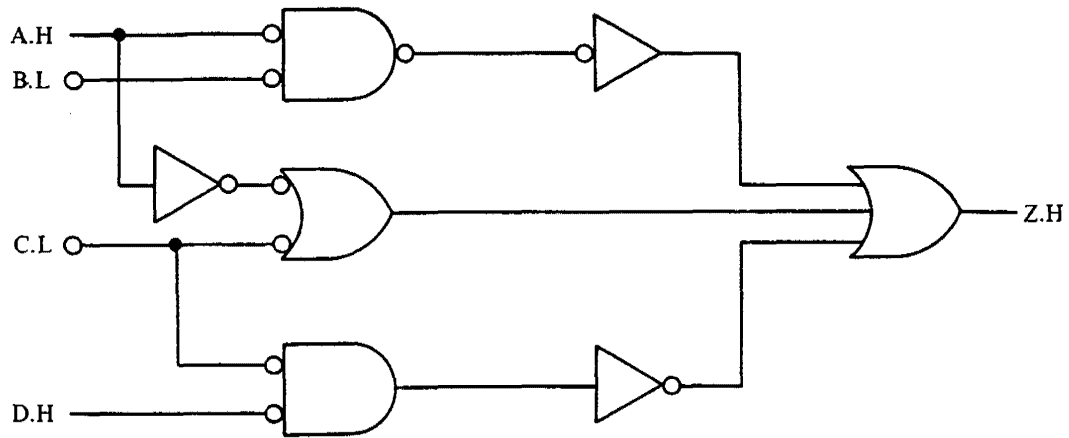


Figure 3.25 Mixed-logic circuit diagram for Problem 3.13.

- 3.14. Repeat Problem 3.13 for the circuit diagram of Fig. 3.26.

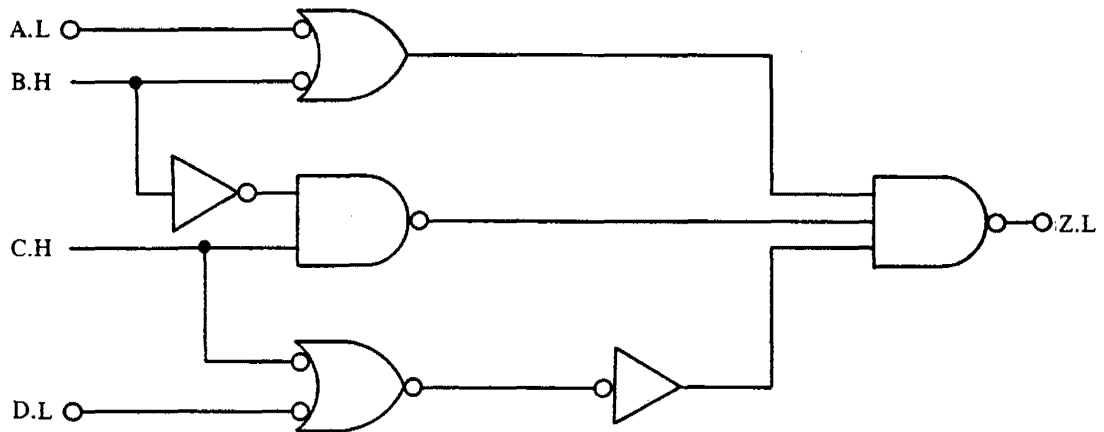


Figure 3.26 Mixed-logic circuit diagram for Problem 3.14.

- 3.15. Repeat Problem 3.13 for the circuit diagram of Fig. 3.27.

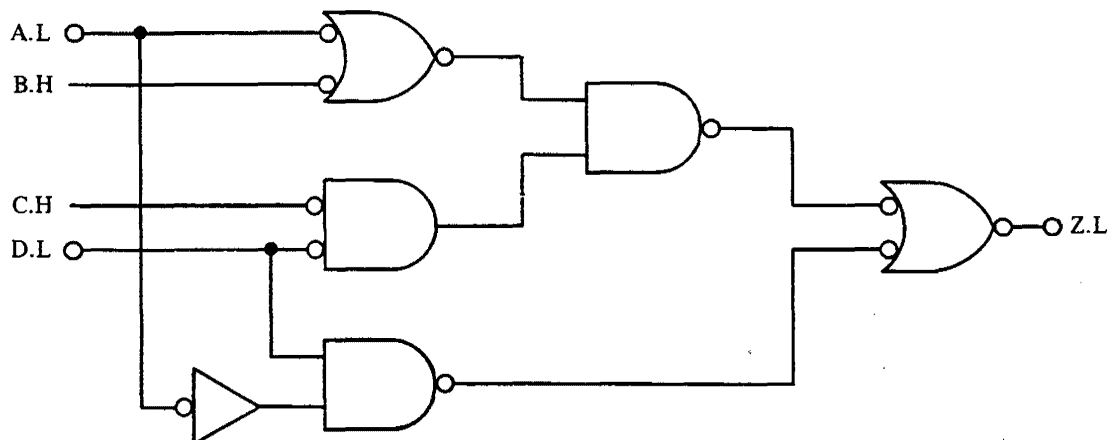


Figure 3.27 Mixed-logic circuit diagram for Problem 3.15.

- 3.16. Figure 3.28 shows a circuit based on the positive-logic convention. Fill in all the intermediate signal names and find a logic expression for Z. Compare with the answer to Problem 3.13.

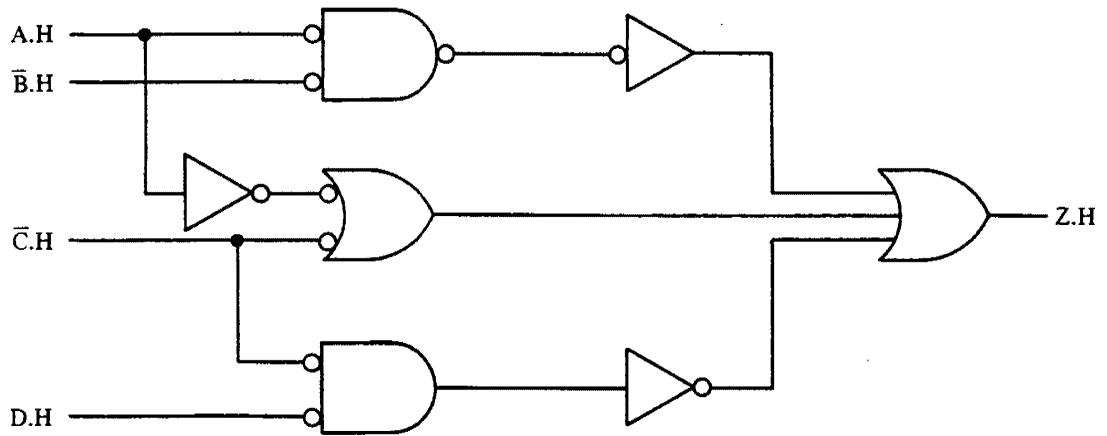


Figure 3.28 Positive-logic circuit diagram for Problem 3.16.

- 3.17. Repeat Problem 3.16 for the circuit of Fig. 3.29. Compare with the answer to Problem 3.14.

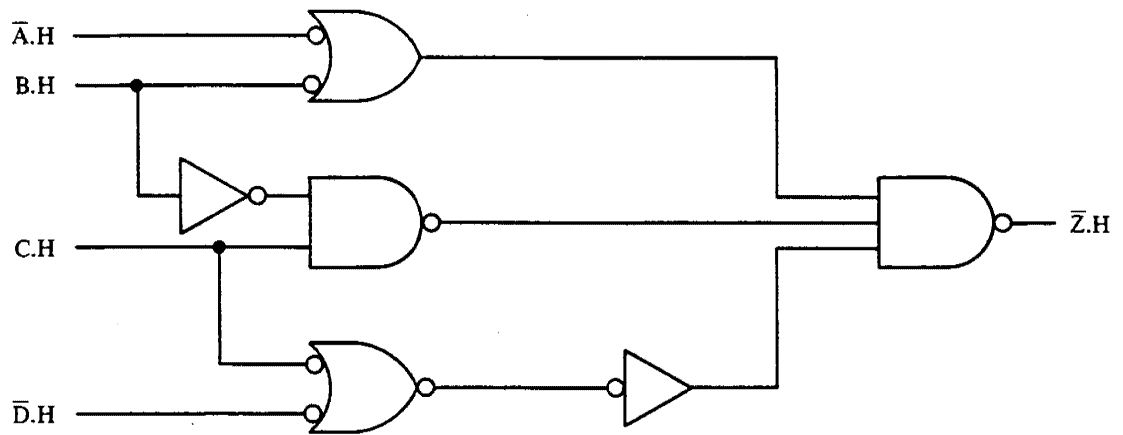


Figure 3.29 Positive-logic circuit diagram for Problem 3.17.

- 3.18. Repeat Problem 3.16 for the circuit of Fig. 3.30. Compare with the answer to Problem 3.15.

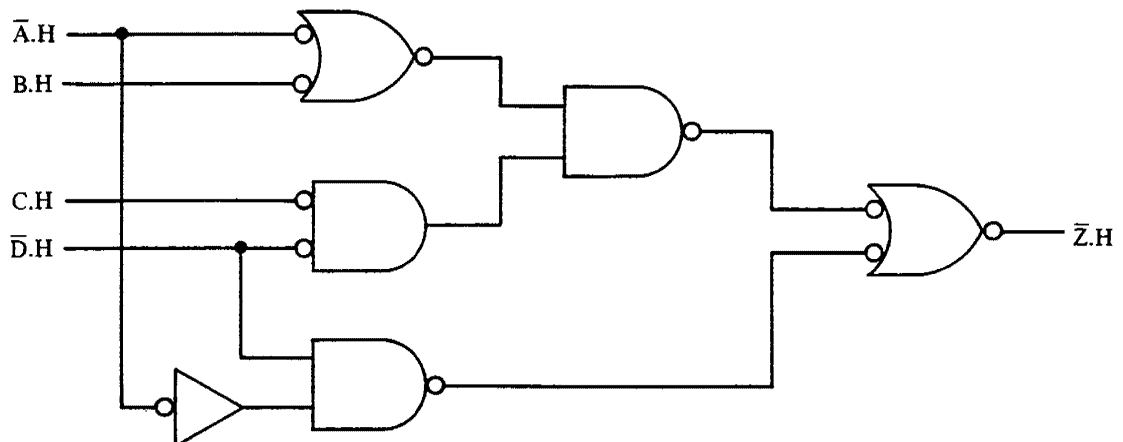


Figure 3.30 Positive-logic circuit diagram for Problem 3.18.

- 3.19. Analyze each circuit diagram of Fig. 3.31 and determine the logic equation for Z based on the positive-logic convention. Remember that in the positive-logic convention, all inverters and inverting circles perform a logic inversion.

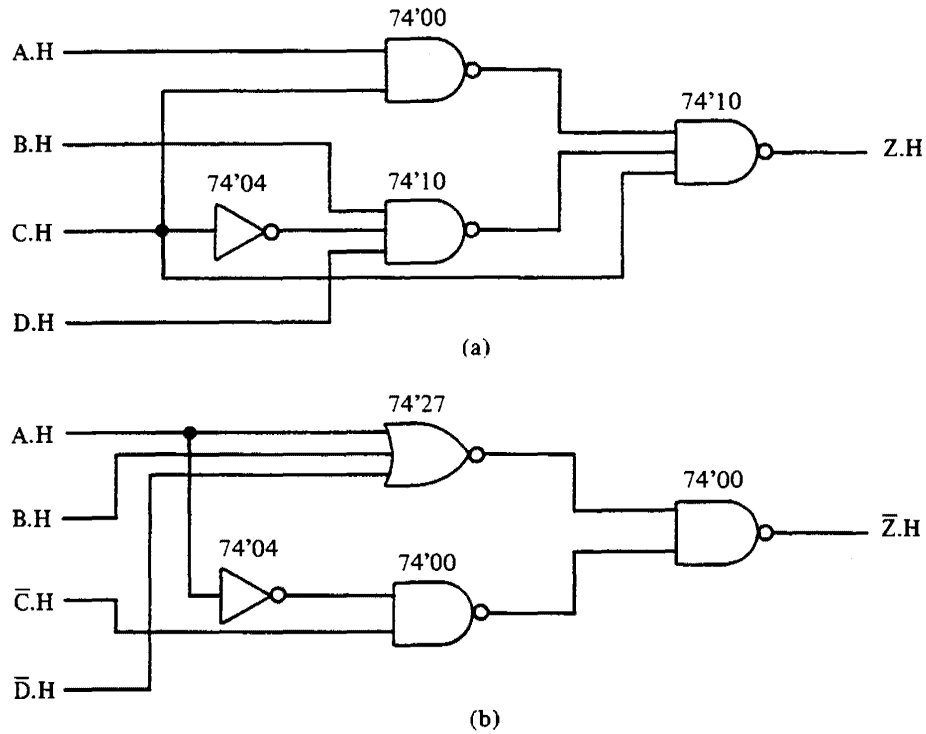


Figure 3.31 Circuit diagrams for Problem 3.19.

- 3.20. Analyze each circuit diagram of Fig. 3.32 and determine the logic equation for Z based on the mixed-logic convention. Remember that in the mixed-logic convention, a logic NOT occurs as a result of a "mismatched" logic/voltage assignment.

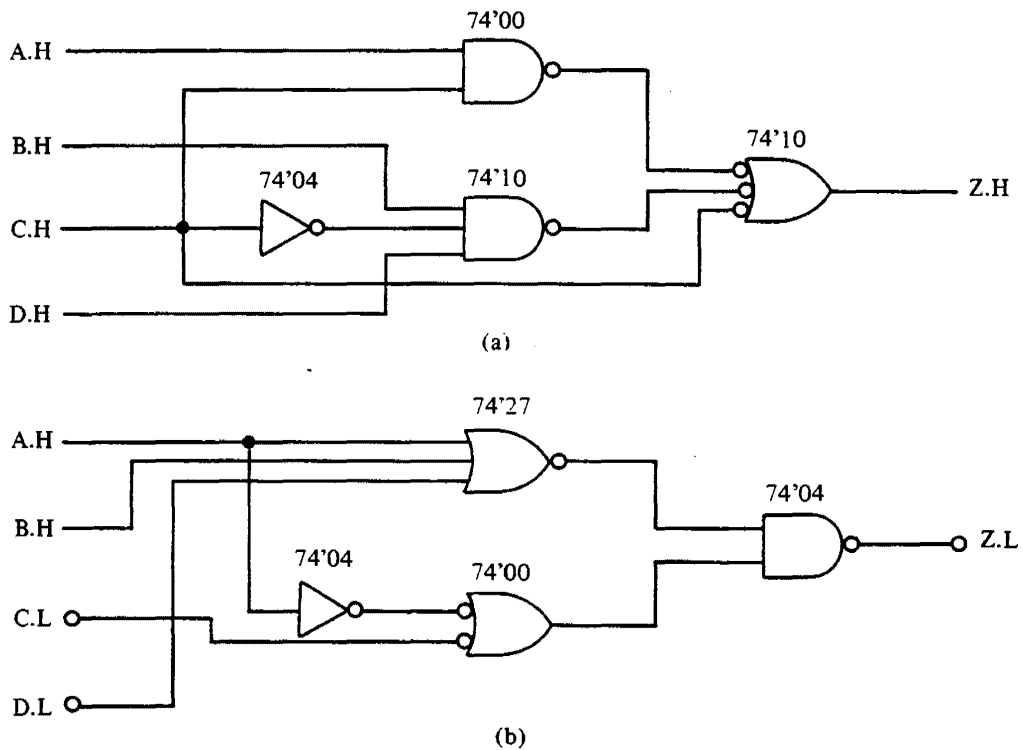


Figure 3.32 Circuit diagrams for Problem 3.20.

- 3.21. Figure 3.33 shows a two-level NAND logic diagram based on the positive-logic convention. (The number of levels is the maximum number of gates that signals must pass through.) Find an SOP expression for Z.

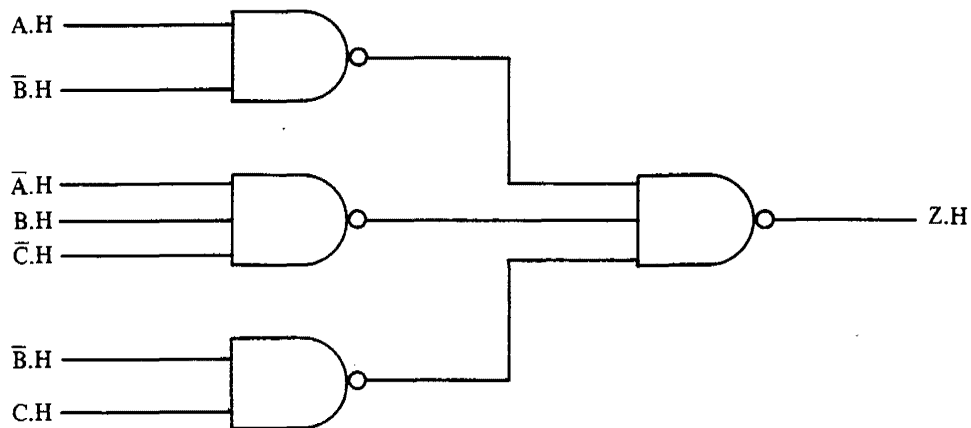


Figure 3.33 Logic diagram for Problem 3.21.

- 3.22. Implement each of the following using an optimum two-level positive-logic NAND realization. In other words, only NAND gates are available for the implementation, and they are to be used in no more than two levels. Assume that the variables and their complements are both available for inputs.

- (a) $Z = \overline{A}B + \overline{B}C$
- (b) $Z = (\overline{A} + B + \overline{C})(\overline{B} + C)$
- (c) $Z = \overline{A}CD + A\overline{C}\overline{D} + AB + B\overline{C}\overline{D}$
- (d) $Z = (B + \overline{D})(A + \overline{B})(\overline{B} + C + D)$

- 3.23. Figure 3.34 shows a two-level NOR logic diagram based on the positive-logic convention. Find a POS expression for Z.

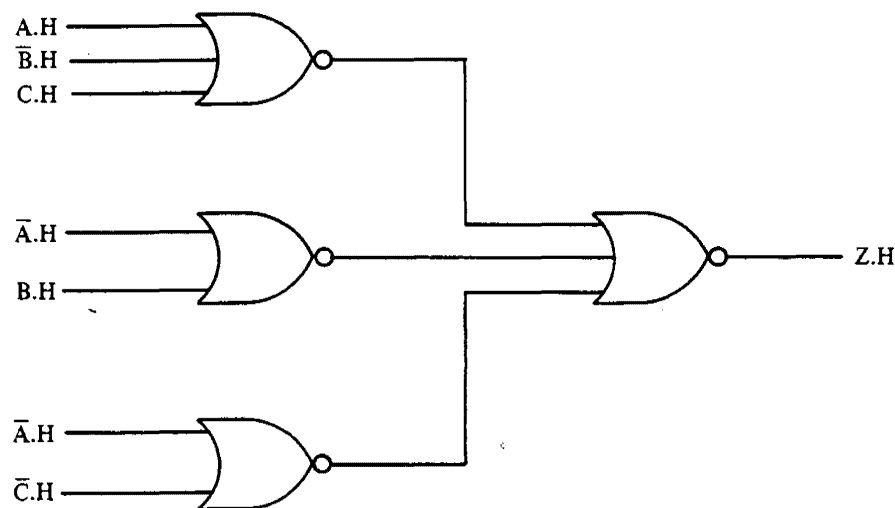


Figure 3.34 Logic diagram for Problem 3.23.

- 3.24. Implement each of the following using an optimum two-level positive-logic NOR realization. In other words, only NOR gates are available for the implementation, and they are to be used in no more than two levels. Assume that the variables and their complements are both available for inputs.

- (a) $Z = (\overline{A} + \overline{B})(\overline{B} + C)$

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

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

$$(d) Z = B\overline{D} + A\overline{B} + \overline{B}CD$$

3.25. Implement the following logic equations for Z, basing the implementations on the positive-logic convention. Assume that only the following gates are available: 74'00, 74'02, 74'04, 74'10, and 74'27. Also, for each implementation use a minimum number of gates and draw the circuit diagram with all gates labeled.

(a) $Z = AB + \overline{C} + \overline{A}\overline{B}\overline{D}$ for Z.H. The inputs are A.H, B.H, C.H, and D.L.

(b) $Z = (A + D)(\overline{A} + C + \overline{D})(\overline{C} + D)$ for Z.H. The inputs are A.H, B.L, C.H, and D.H.

(c) $Z = \overline{\overline{A}B} + \overline{C}D + C\overline{D}$ for Z.H. The inputs are A.L, B.H, C.H, and D.L.

(d) $Z = \overline{(A + D)(\overline{B} + C)(B + \overline{C})}$ for Z.H. The inputs are A.H, B.L, C.H, D.H, and D.L.

(e) $Z = \overline{\overline{A}C} + BD + \overline{B}\overline{D}$ for Z.L. The inputs are A.L, B.L, C.H, and D.H.

(f) $Z = A\overline{C} + BD + C\overline{D}$ for Z.L. The inputs are A.L, B.L, C.H, and D.L.

3.26. Repeat Problem 3.25 for implementations based on the mixed-logic convention.

3.27. Implement the following logic equations for Z, basing the implementations on the positive-logic convention. Draw the circuit diagrams with all gates labeled. Select any gates from a TTL data book, and the 74'86 Exclusive OR gate in particular. But, minimize the number of IC packages used. In other words, try to use the same types of gates when possible.

(a) $Z = AB + \overline{A} \odot \overline{C} + \overline{A}C$ for Z.L. The inputs are A.H, A.L, B.L, C.H, and C.L.

(b) $Z = \overline{A \oplus B} + \overline{B} \odot C + \overline{\overline{C}D}$ for Z.H. The inputs are A.H, B.H, C.L, and D.H.

(c) $Z = A\overline{B} + B \oplus D$ for Z.H. The inputs are A.H, B.L, C.L, and D.H.

3.28. Repeat Problem 3.27 for the mixed-logic convention.

3.29. Figure 3.35 shows a parity detector for detecting the parity of an input 4-bit number ABCD. The detector output PARITY is 0 (PARITY = 0) if ABCD has even parity, which means that ABCD contains an even number of 1s. And the output PARITY is 1 (PARITY = 1) if ABCD has odd parity, which means that it contains an odd number of 1s. For example, for ABCD = 0110, PARITY = 0. And for ABCD = 1101, PARITY = 1.

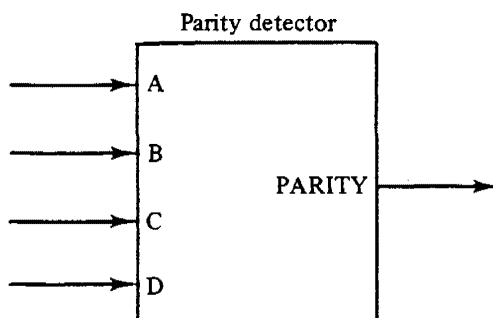


Figure 3.35 Parity detector for Problem 3.29.

(a) Make a truth table for the parity detector, with input columns of ABCD and an output column of PARITY.

(b) Determine an MSOP expression for PARITY.

(c) For inputs of A.H, B.H, C.H, and D.H, implement the MSOP logic equation for PARITY.H using the positive-logic convention.

- (d) Repeat part (c) for the mixed-logic convention.
- 3.30. A combinational circuit with inputs A, B, C, and D and an output Z is to be designed such that $Z = 1$ if and only if three or more of the inputs are 1.
- Make a truth table for the circuit.
 - Determine an MSOP expression for Z.
 - For inputs of A.H, B.H, C.H, and D.H, implement the MSOP logic equation for Z.H using the positive-logic convention.
 - Repeat part (c) for the mixed-logic convention.
- 3.31. Figure 3.36 shows an excess-3 code generator and table. An excess-3 code is a binary code for decimal numbers in which each *decimal digit* is represented by its binary equivalent *plus 3*. For example, the excess-3 code for the digit 0 is 0011, and for the digit 9 it is 1100. A characteristic of the excess-3 code is that every coded digit has at least one 1, which is important in some applications.
- Complete the truth table for the circuit, using don't cares for invalid inputs.
 - Determine MSOP expressions for the four outputs $X_3, X_2, X_1,$ and X_0 .
 - Implement the MSOP logic equations for the four outputs, basing the implementations on the positive-logic convention. Assume that all inputs and outputs are active-high.
 - Repeat part (c) for the mixed-logic convention, again assuming that all inputs and outputs are active-high.

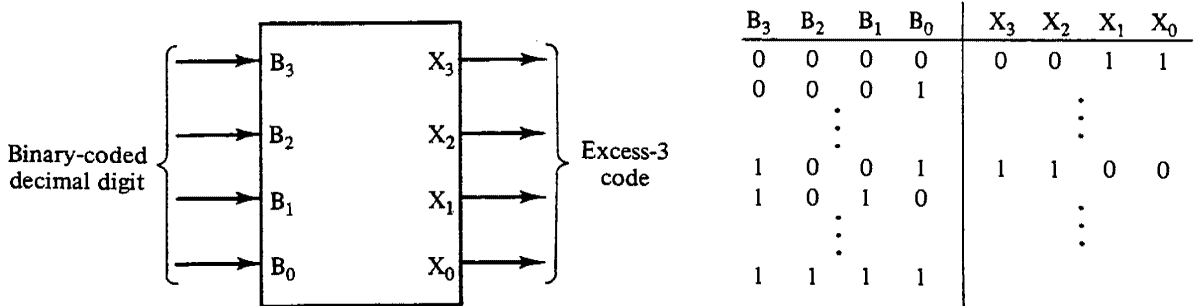


Figure 3.36 Excess-3 code generator and table for Problem 3.31.

- 3.32. A large room has three entrances, each with a light switch that controls an overhead light. A combinational circuit is to be designed with inputs A, B, and C from the individual light switches and an output LIGHT for controlling the energization of the overhead light. When all three switches are down (i.e., $A = 0, B = 0,$ and $C = 0$), then the light is to be off ($LIGHT = 0$). Also, a change in position of any switch will change the state of the light. Assume that only one switch can be changed at a time.
- Draw a block diagram of this circuit
 - Make a truth table for the circuit.
 - Determine an MSOP expression for LIGHT.
 - Implement the MSOP logic equation for LIGHT.H, basing the implementation on the positive-logic convention. Assume that the inputs are active-high.
 - Repeat part (d) for the mixed-logic convention, again assuming that all inputs are active-high.
- 3.33. The input to the combinational circuit of Fig. 3.37 is a 4-bit binary number $B_3B_2B_1B_0$. The function of this circuit is to convert this binary number into the corresponding negative number $N_3N_2N_1N_0$ in 2s-complement form.
- Complete the truth table for the circuit.
 - Determine MSOP expressions for the four outputs $N_3, N_2, N_1,$ and N_0 .

- (c) Implement the MSOP logic equations for the four outputs, basing the implementations on the positive-logic convention. Assume that all inputs and outputs are active-high.
- (d) Repeat part (c) for the mixed-logic convention, again assuming that all inputs and outputs are active-high.

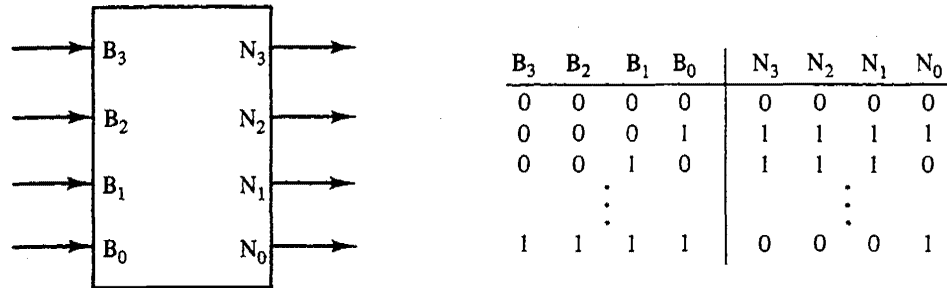


Figure 3.37 Combinational circuit and table for Problem 3.33.

- 3.34. What are the advantages and disadvantages of synthesis based on the positive-logic convention?
- 3.35. What are the advantages and disadvantages of synthesis based on the mixed-logic convention?

Combinational MSI Circuit Elements

4.1 INTRODUCTION

A number of logic functions are used in the design of a typical digital circuit. The circuit elements that realize these logic functions are classified as one of two types: *combinational* or *sequential*. For a combinational circuit element, the output values at any time are functions only of a combination of the present input values. In contrast, for a sequential circuit element, the output values are functions not only of the present inputs but also of the conditions of some internal states of the circuit element. And the conditions of these states are, in turn, functions of previous inputs. Consequently, for a sequential circuit element the outputs depend on both the present and past values of the inputs. Because past inputs affect present outputs, a sequential circuit element must have some type of memory capability.

In this and the next two chapters we will study the designs and applications of circuit elements that realize some of the commonly used logic functions. These circuit elements form the building blocks used in the design of a digital circuit, as will be discussed in Chapter 7. In the present chapter, combinational MSI (medium-scale integration) circuits will be considered. In Chapter 5 sequential MSI circuit elements will be considered; and in Chapter 6 we complete the study of digital building blocks by considering some of the commonly used LSI (large-scale integration) circuit elements.

In all the following sections of this chapter, the presentation format for each logic function is consistent. First, a functional description is given. Then, there is the design and realization of the logic function based on the SSI design methods of the preceding chapters. Next, commercially available MSI realizations are described. Finally, some applications of these MSI circuit elements are presented.

4.2 BINARY ADDER AND SUBTRACTOR

4.2.1 Half Adder and Full Adder

Binary addition was discussed in Chapter 1. As should be recalled, the binary addition of two bits (A_i and B_i) is represented by the *addition table* shown in Fig. 4.1(a). Here, 0 and 1 represent the binary bits zero and one. If we associate the binary bit 0 with the logic value false, and the binary bit 1 with the logic value true, then the table of Fig. 4.1(a) is *also* the *truth table* for a binary addition circuit element. Using the methods of Chapter 2, we can determine the logic expressions for SUM_i and $CARRY_{i+1}$, and formulate the corresponding circuit, as shown in Fig. 4.1(b). This circuit is called a *half adder*. (The upper graphic symbol represents Exclusive OR.)

The half-adder circuit does not suffice for general additions. To see this, consider the following addition of two multibit binary numbers:

carry	0	1	1	0
A		0	1	1
B	+	0	0	1
sum		1	0	0

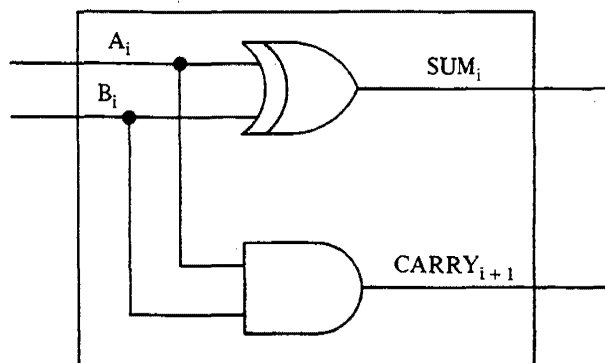
As is evident, when the binary numbers to be added are multibit, then we need to consider the carry that is generated from the preceding stage of addition. Consequently, except

A_i	B_i	SUM_i	$CARRY_{i+1}$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) Addition table

$$SUM_i = \bar{A}_i B_i + A_i \bar{B}_i = A_i \oplus B_i$$

$$CARRY_{i+1} = A_i B_i$$

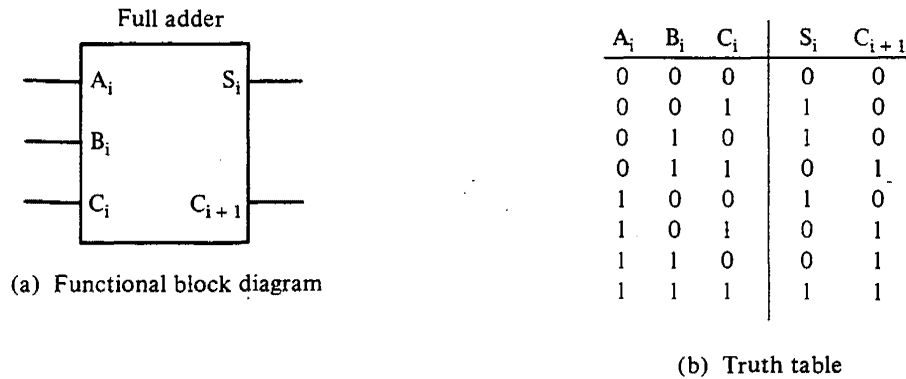


(b) Half-adder circuit

Figure 4.1 Half adder.

for the addition of the least significant bits, a half adder is not adequate. What is needed is a *full adder*.

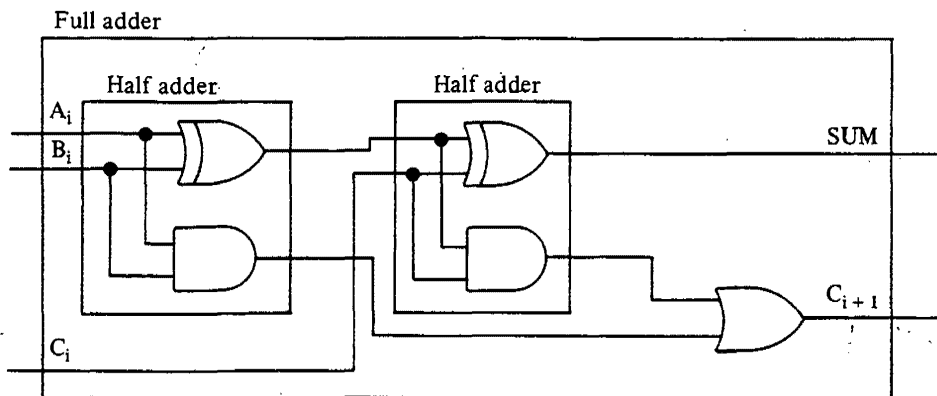
The functional block diagram and the truth table describing a full adder (FA) are given in Figs. 4.2(a) and (b), respectively. The inputs to the full adder are the current bits of the numbers to be added (A_i and B_i) and the carry-in (C_i) from the preceding addition stage. The outputs are the sum (S_i) and the carry-out (C_{i+1}) generated from the current stage of addition. This carry-out is the carry-in for the next stage.



$$\begin{aligned} \text{SUM} &= \bar{A}_i \bar{B}_i C_i + \bar{A}_i B_i \bar{C}_i + A_i \bar{B}_i \bar{C}_i + A_i B_i C_i \\ &= \bar{A}_i (\bar{B}_i C_i + B_i \bar{C}_i) + A_i (\bar{B}_i \bar{C}_i + B_i C_i) \\ &= \bar{A}_i (B_i \oplus C_i) + A_i (\overline{B_i \oplus C_i}) \\ &= A_i \oplus B_i \oplus C_i \end{aligned}$$

$$\begin{aligned} C_{i+1} &= A_i B_i + A_i \bar{B}_i C_i + \bar{A}_i B_i C_i \\ &= A_i B_i + C_i (A_i \bar{B}_i + \bar{A}_i B_i) \\ &= A_i B_i + C_i (A_i \oplus B_i) \end{aligned}$$

(c) Design



(d) Realization

Figure 4.2 Full adder.

Using the methods of Chapters 2 and 3 we can design and realize a full adder, as shown in Figs. 4.2(c) and (d). Note that the groupings of the 1s in the K-map for C_{i+1} do *not* produce the minimum SOP expression for C_{i+1} , which is

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i$$

This expression would result in the minimum amount of hardware used if C_{i+1} was the only output. However, since S_i is also an output, less *overall* hardware is required if we use the indicated “nonminimum” expression for C_{i+1} and allow C_{i+1} to share with S_i the common term $A_i \oplus B_i$. Consequently, with some clever maneuvering, we can realize a full adder with two half adders plus a single OR gate, as shown in Fig. 4.2(d).

4.2.2 Parallel Adder

Full adders can be connected together to perform addition in parallel of two multibit binary numbers. Shown in Fig. 4.3 are four full adders cascaded to form a 4-bit parallel adder. With it, two 4-bit numbers, inputted at A and B, can be added in parallel to produce a 4-bit sum S and a carry-out C_4 . Note that the carry-in C_0 of Stage 0 must be connected to “0” for the 4-bit adder to function properly. Alternatively, a half adder could be used for this stage. In general, an N -bit parallel adder can be realized by cascading N full adders in this manner.

If all the inputs are simultaneously applied to a physical parallel adder, the correct sum bits and carry-out bit do not appear simultaneously, but at a time that may be tens of nanoseconds later. The cause for the delay is the inherent propagation delay that every real digital element has. (Propagation delay is considered in Sec. 4.8.3.) For the parallel adder configuration of Fig. 4.3, the overall delay is aggravated by the cascade arrangement of the full adders. Note that, except for Stage 0, each carry-in is the carry-out of the preceding stage, and so is not stable until the preceding stage produces a stable carry-out output. Specifically, the carry-in for Stage 1 is not stable until Stage 0 produces a stable output at C_1 . Similarly the carry-in at Stage 2 is not stable until Stage 1 produces a stable output at C_2 , and so forth. As a result, the stage outputs become stable successively from right to left. And, in a manner very much as in performing binary addition manually, a carry “ripples” down the chain of full adders. For this reason, this type of parallel adder is commonly called a *ripple adder*.

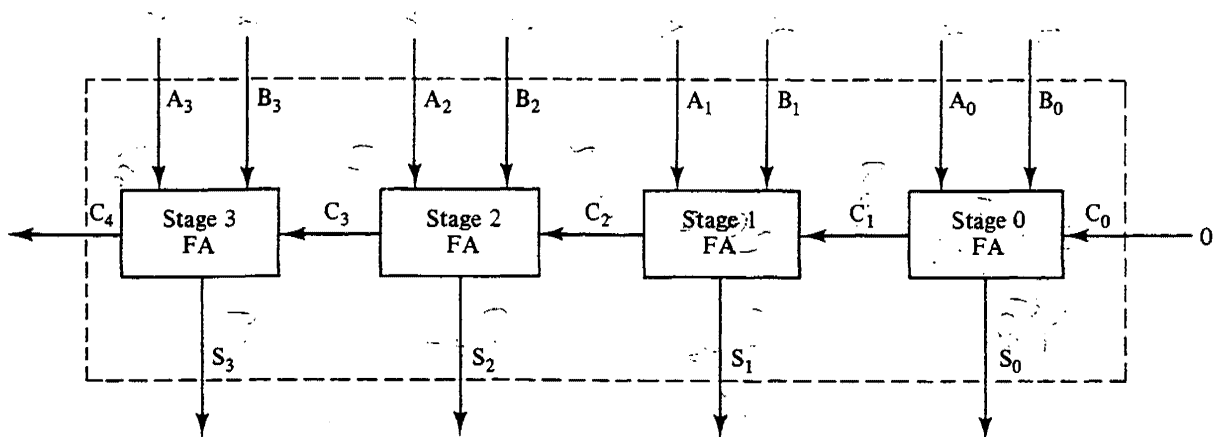


Figure 4.3 4-bit parallel adder.

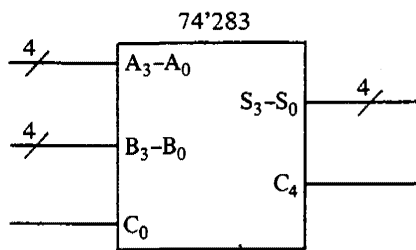
4.2.3 MSI Parallel Adders

Since binary addition is an important function in digital design, integrated-circuit manufacturers produce multibit parallel adders in the form of MSI chips. Two examples are the 74'83 and the 74'283. Functionally, both perform the same function as the 4-bit parallel adder described in the last section. In other words, each adds two 4-bit binary numbers with a carry-in, and produces a 4-bit sum and a carry-out. Additionally, both of these adders feature *look-ahead* circuitry to eliminate the relatively slow rippling effect of the carry bits of the ripple adder. Carry look-ahead circuitry is discussed in more detail in Chapter 6.

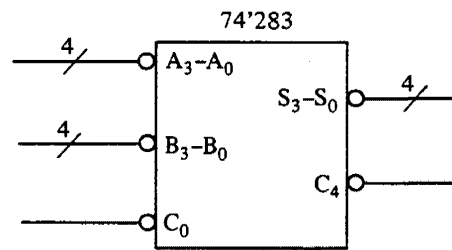
The voltage table for the 74'283 is shown in Fig. 4.4(a). Recall that a voltage table defines the physical behavior of a digital device. It shows how the device really

A_i	B_i	C_i	S_i	C_{i+1}
L	L	L	L	L
L	L	H	H	L
L	H	L	H	L
L	H	H	L	H
H	L	L	H	L
H	L	H	L	H
H	H	L	L	H
H	H	H	H	H

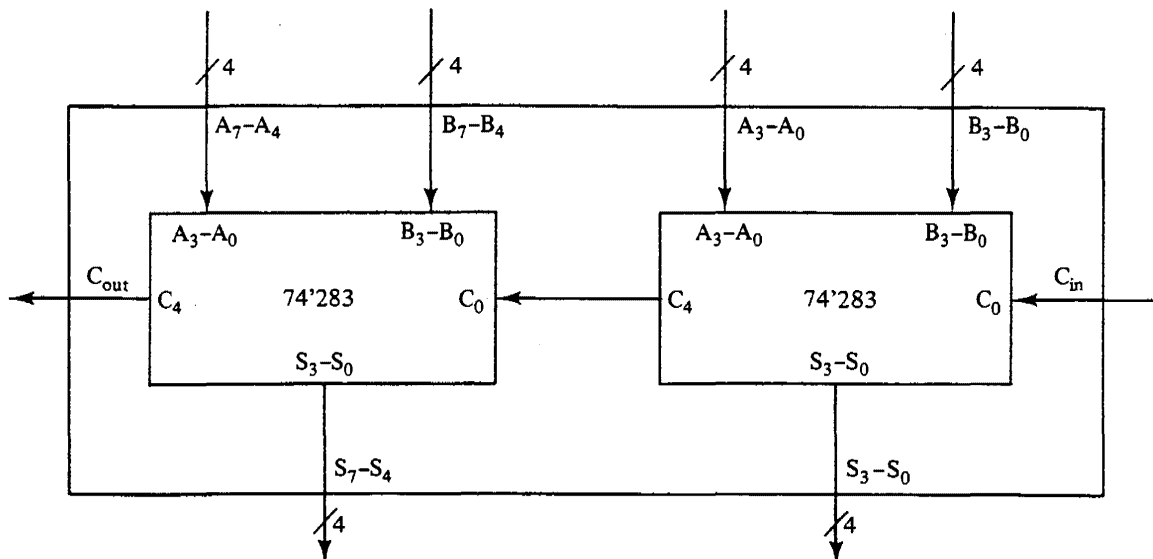
(a) Voltage table



(b) Active-high view



(c) Active-low view

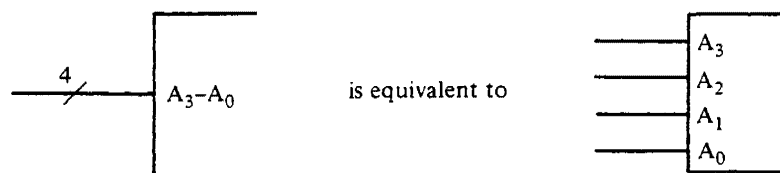


(d) An 8-bit adder constructed from two 74'283 adders

Figure 4.4 74'283 4-bit parallel adder.

works. In contrast, the logic operations that the device performs depend on the assignment of the voltage representation to the logic values. Due to the symmetry of the binary add function, the 74'283 can function as a 4-bit binary adder for two different voltage assignments. The functional block diagram for the active-high view is shown in Fig. 4.4(b), and that for the active-low view is shown in Fig. 4.4(c).

Figure 4.4 introduces a commonly used shorthand notation:



Often, it is convenient to group a set of related signals. Notationally, a slash indicates that a line represents more than one signal. And, the number associated with the slash indicates the number of signals.

The most common view of the 74'283 is the active-high view shown in Fig. 4.4(b). If, however, most of the signals that require the use of the adder are active-low, then the active-low view should be used. Forcing the active-high view in this case requires additional inverters. By mapping the voltage table of Fig. 4.4(a) into the respective logic tables, as done in Chapter 3, we can see that both views result in the binary addition function. (See Problem 4.1.)

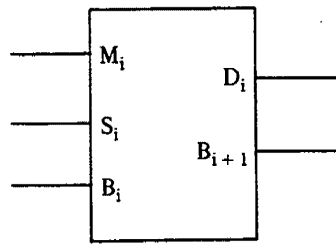
For an illustration, we will use the active-low view of a 74'283 adder and show the voltage-level representations for all signals for the addition of $9 + 3$ with a carry-in of 1. The sum is 13, of course. And, the carry-out is 0 because no more than four bits are required for the sum. So, for the active-low view, the 74'283 will have LLHL at the SUM outputs, representing the 13, and H at the C_4 output, representing the 0. Following is a summary of the input and output signals.

	A	B	C_0	SUM	C_4
decimal	9	3	1	13	0
binary	1001	0011	1	1101	0
74'283 voltage level	LHHL	HHLL	L	LLHL	H

Two 4-bit 74'283 (or 74'83) adders can be connected in cascade to produce an 8-bit parallel adder, as shown in Fig. 4.4(d). In general, N 4-bit adders can be cascaded to produce a parallel adder that can add binary numbers of up to $4 \times N$ bits each.

4.2.4 Binary Subtractor

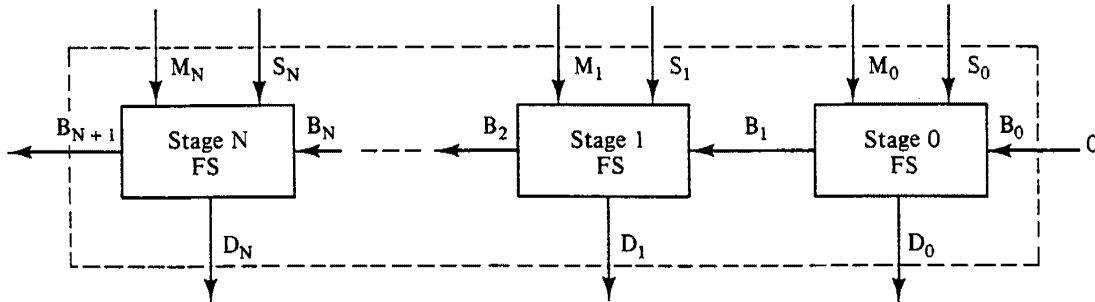
The functional block diagram and the truth table for a full subtractor (FS) are given in Figs. 4.5(a) and (b), respectively. Analogous to a full adder, the inputs to the full subtractor are the current bits of the minuend (M_i), the subtrahend (S_i), and the borrow-in (B_i) from the preceding subtraction stage. The outputs are the current difference bit (D_i) and the borrow-out (B_{i+1}) generated by the current subtraction stage. Full subtractors can also be cascaded for the subtraction in parallel of two multibit binary numbers, as shown in Fig. 4.5(c).



(a) Functional block diagram

M_i	S_i	B_i	D_i	B_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(b) Truth table



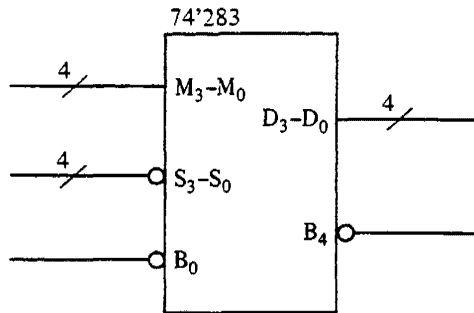
(c) (N + 1)-bit parallel subtractor

Figure 4.5 Binary subtractor.

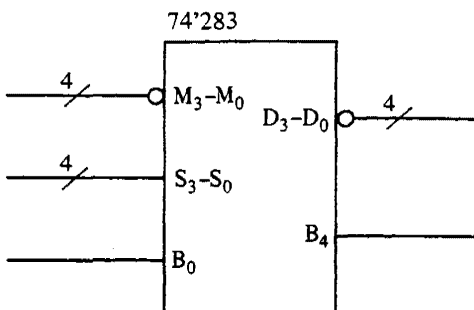
Binary subtractors are commercially available in the form of MSI circuit elements. One example is, interestingly enough, the 74'283. With a change in voltage assignment from that of Fig. 4.4(b) or (c), it can be made to subtract instead of add. In fact, as shown in Figs. 4.6(a) and (b), two additional views, both realizing 4-bit binary subtractors, for the 74'283 are possible. Consider the 74'283 for the voltage assignment of Fig. 4.6(a). For the voltage table for stage i of the 74'283 shown in Fig. 4.4(a), we obtain the following logic table:

VOLTAGE TABLE					LOGIC TABLE					LOGIC TABLE				
M_i	S_i	B_i	D_i	B_{i+1}	M_i	S_i	B_i	D_i	B_{i+1}	M_i	S_i	B_i	D_i	B_{i+1}
L	L	L	L	L	0	1	1	0	1	0	0	0	0	0
L	L	H	H	L	0	1	0	1	1	0	0	1	1	1
L	H	L	H	L	0	0	1	1	1	0	1	0	1	1
L	H	H	L	H	0	0	0	0	0	0	1	1	0	1
H	L	L	H	L	1	1	1	1	1	1	0	0	1	0
H	L	H	L	H	1	1	0	0	0	1	0	1	0	0
H	H	L	L	H	1	0	1	0	0	1	1	0	0	0
H	H	H	H	H	1	0	0	1	0	1	1	1	1	1

We see from the second logic table that the 74'283 realizes a 4-bit binary subtractor for an assignment of an active-high minuend and difference, and an active-low subtrahend, borrow-in, and borrow-out. Consider what needs to be added for the 4-bit subtractor to have an active-high minuend and subtrahend. (See Problem 4.3.) Similarly, a proof can be derived for the voltage assignment shown in Fig. 4.6(b). (See Problem 4.2.)



(a) Active-high minuend and difference



(b) Active-low minuend and difference

Figure 4.6 74'283 binary subtractor.

4.3 MAGNITUDE COMPARATOR

A magnitude comparator is a combinational circuit element that produces outputs that are functions of the relative magnitudes of two inputted binary numbers. A functional block diagram for a 2-bit magnitude comparator is shown in Fig. 4.7(a). For this comparator the inputs are two 2-bit numbers A and B. The outputs are signals indicating whether the binary number A is greater than B, equal to B, or less than B. The functional description of the 2-bit comparator, in the form of a truth table, is given in Fig. 4.7(b). As shown, the signal $(A = B)$ is true when A and B are both equal to 00, 01, 10, or 11. Otherwise, A is either less than B, as shown in rows 2, 3, 4, 7, 8, and 12 of the truth table; or A is greater than B, as shown in rows 5, 9, 10, 13, 14, and 15.

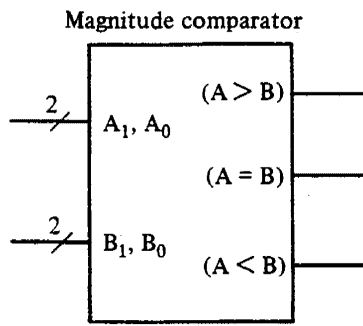
With this truth table, we can use the methods of Chapters 2 and 3 to design and realize the 2-bit comparator, as shown in Figs. 4.7(c) and (d). Note that we need to implement only two out of the three outputs directly since we can generate the third output more economically by using the fact that it is true only when both other outputs are false. So, we can generate one output indirectly by using one of the following equations:

$$(A < B) = \overline{(A > B)} \cdot \overline{(A = B)}$$

$$(A = B) = \overline{(A > B)} \cdot \overline{(A < B)}$$

or

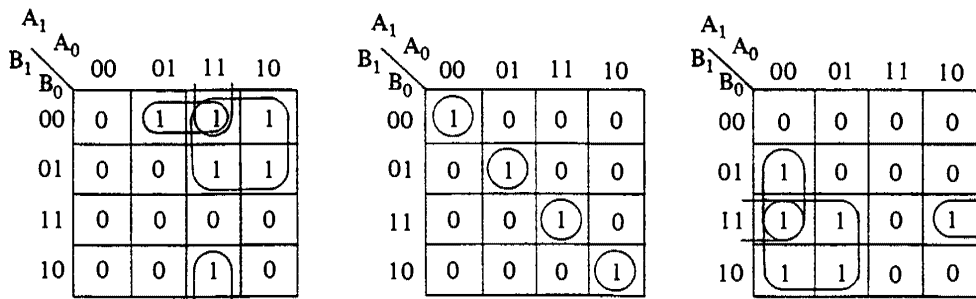
$$(A > B) = \overline{(A = B)} \cdot \overline{(A < B)}$$



(a) Functional block diagram

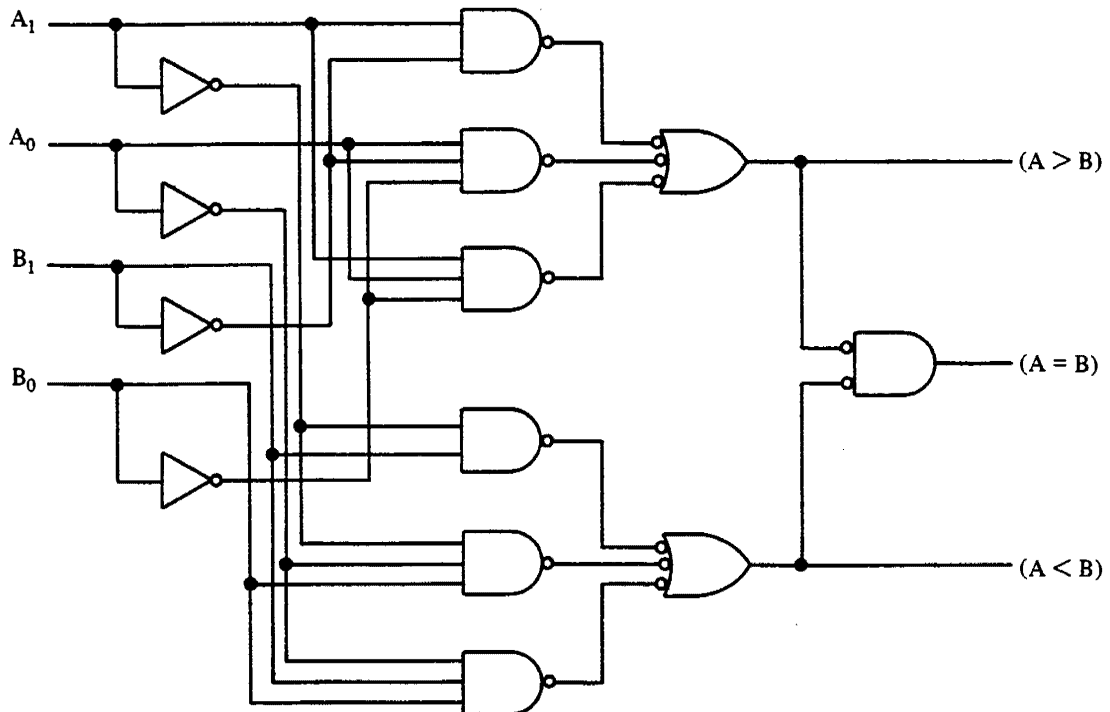
A ₁	A ₀	B ₁	B ₀	(A > B)	(A = B)	(A < B)
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

(b) Truth table.



$$\begin{aligned}
 (A > B) &= A_1 \bar{B}_1 + A_0 \bar{B}_1 \bar{B}_0 + A_1 A_0 \bar{B}_0 \\
 (A = B) &= \bar{A}_1 \bar{A}_0 \bar{B}_1 \bar{B}_0 + \bar{A}_1 A_0 \bar{B}_1 B_0 + A_1 A_0 B_1 B_0 + A_1 \bar{A}_0 B_1 \bar{B}_0 \\
 (A < B) &= \bar{A}_1 B_1 + \bar{A}_1 \bar{A}_0 B_0 + \bar{A}_0 B_1 B_0
 \end{aligned}$$

(c) Design

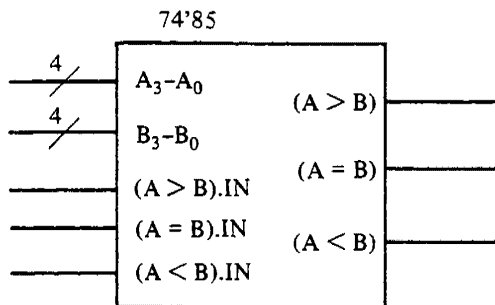


(d) Realization

Figure 4.7 2-bit magnitude comparator.

In Fig. 4.7(d) the signal $(A = B)$ is generated indirectly. Of course, this design can be generalized for the construction of a magnitude comparator of more than 2 bits. But, the design becomes more and more unwieldy as the number of inputs increases.

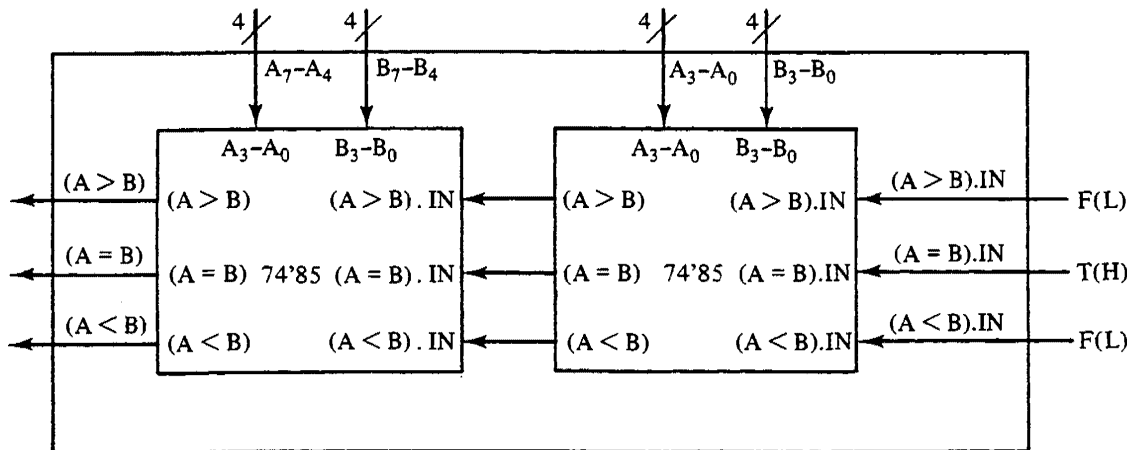
Four-bit magnitude comparators are commercially available as MSI circuit elements. An example of such a device is the 74'85, which has the block diagram and voltage table shown in Figs. 4.8(a) and (b), respectively. In addition to the inputs and outputs considered above, the 74'85 also has the following cascading inputs:



(a) Functional block diagram

Comparing inputs				Cascading inputs			Outputs		
A_3, B_3	A_2, B_2	A_1, B_1	A_0, B_0	$(A > B).IN$	$(A < B).IN$	$(A = B).IN$	$(A > B)$	$(A < B)$	$(A = B)$
$A_3 > B_3$	X	X	X	X	X	X	H	L	L
$A_3 < B_3$	X	X	X	X	X	X	L	H	L
$A_3 = B_3$	$A_2 > B_2$	X	X	X	X	X	H	L	L
$A_3 = B_3$	$A_2 < B_2$	X	X	X	X	X	L	H	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 > B_1$	X	X	X	X	H	L	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 < B_1$	X	X	X	X	L	H	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 > B_0$	X	X	X	H	L	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 < B_0$	X	X	X	L	H	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	H	L	L	H	L	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	L	H	L	L	H	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	X	X	H	L	L	H
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	H	H	L	L	L	L
$A_3 = B_3$	$A_2 = B_2$	$A_1 = B_1$	$A_0 = B_0$	L	L	L	H	H	L

(b) Voltage table



(c) 8-bit magnitude comparator

Figure 4.8 74'85 magnitude comparator.

$(A > B).IN$, $(A = B).IN$, and $(A < B).IN$. These inputs make it easy to cascade 74'85s to produce an N -bit magnitude comparator, where N is a multiple of 4. An 8-bit comparator is shown in Fig. 4.8(c). Note for this connection that if the most significant 4 bits of A have a value greater than those of B , then the signal $(A > B)$ is true regardless of the cascading inputs from the comparator of the preceding stage. Conversely, if the value of the most significant 4 bits of A is less than that of B , then the signal $(A < B)$ is true regardless of the inputs from the preceding stage. When, however, these two values are equal, then the comparator outputs depend on the cascading inputs from the comparator of the preceding stage. Again, in general, 4-bit magnitude comparators can be cascaded in this manner to produce a magnitude comparator of any reasonable length. Note that for the least significant stage of a chain of 74'85 comparators, the $(A = B).IN$ input must be connected to true, and the $(A > B).IN$ and $(A < B).IN$ inputs must be connected to false.

4.4 DECODER

A binary code of N bits can encode up to 2^N different elements of information. So, a 2-bit code can encode up to four elements. A 3-bit code can encode up to eight elements, and so forth. In the design of a digital circuit, it is often necessary to use a decoding function. A *decoder* is a combinational circuit element that will decode an N -bit code. It has up to 2^N output lines, and activates the output signals as a function of the N -bit code applied at the inputs.

Figure 4.9 shows a 3-to-8 decoder. Its functional block diagram is shown in Fig. 4.9(a), and its functional description, in the form of a truth table, is given in Fig. 4.9(b). The input to the decoder is a 3-bit code at the A_2 , A_1 , and A_0 inputs. Since a 3-bit code can encode up to eight different elements of information, this decoder has eight outputs, each of which represents one of the eight different elements. For example, if the input code is 000 (i.e., $A_2 = \text{false}$, $A_1 = \text{false}$, $A_0 = \text{false}$), then the Z_0 output is activated and the other outputs are all false. If the input code is 001, then only the Z_1 output is activated, and so forth. From the truth table of Fig. 4.9(b), the logic equations for the eight outputs can be determined in a straightforward manner. The result is the circuit shown in Fig. 4.9(c).

Decoders are commercially available as MSI circuit elements in the form of 2-to-4, 3-to-8, and 4-to-16 decoders. An example of a commercially available MSI 3-to-8 decoder is the 74'138 shown in Fig. 4.10. As shown in the functional block diagram of Fig. 4.10(a), the three inputs A_2 , A_1 , and A_0 are active-high and the eight outputs are active-low. Furthermore, there are two active-low enable inputs E_1 and E_2 , and an active-high enable input E_3 . As shown in the voltage table of Fig. 4.10(b), the 74'138 functions as a 3-to-8 decoder only if all three enable inputs are true: $E_1 = L$, $E_2 = L$, and $E_3 = H$. Otherwise all eight outputs are false (H).

The function of the enable inputs is to permit convenient expansion. Figures 4.10(c) and (d) show a 4-to-16 decoder, with an active-low enable, constructed from two 74'138 decoders without any additional logic. Note that the 74'138 decoder on the top is enabled only if A_3 is 0, and the bottom 74'138 decoder is enabled only if A_3 is 1. Consequently, each different element of the 4-bit code activates a unique output. In general, a decoder

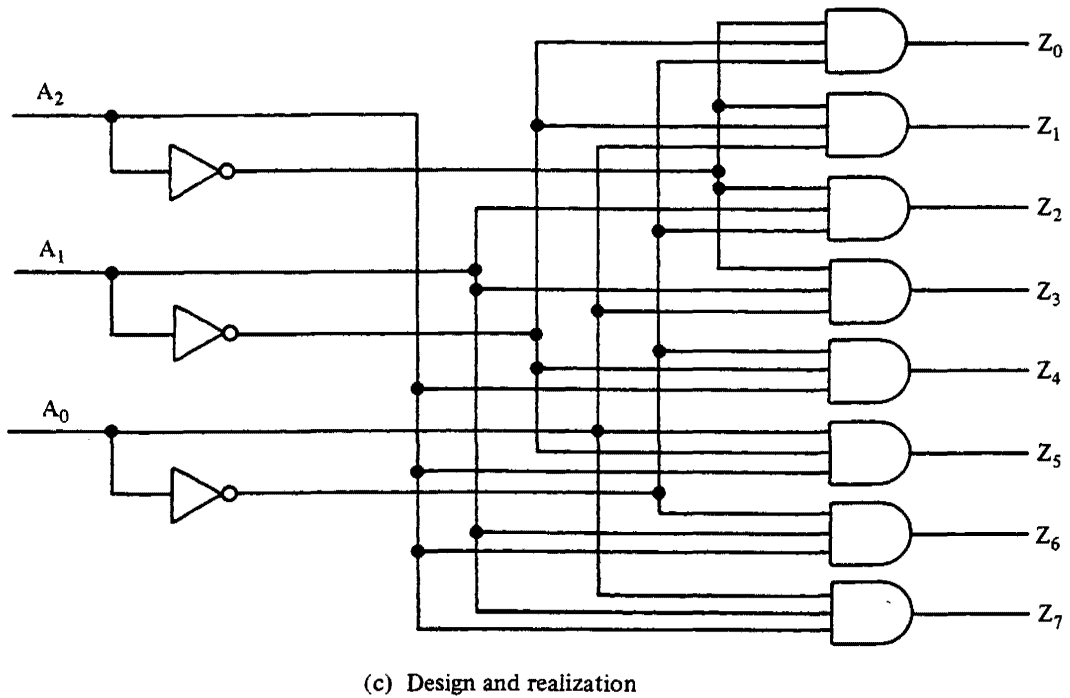
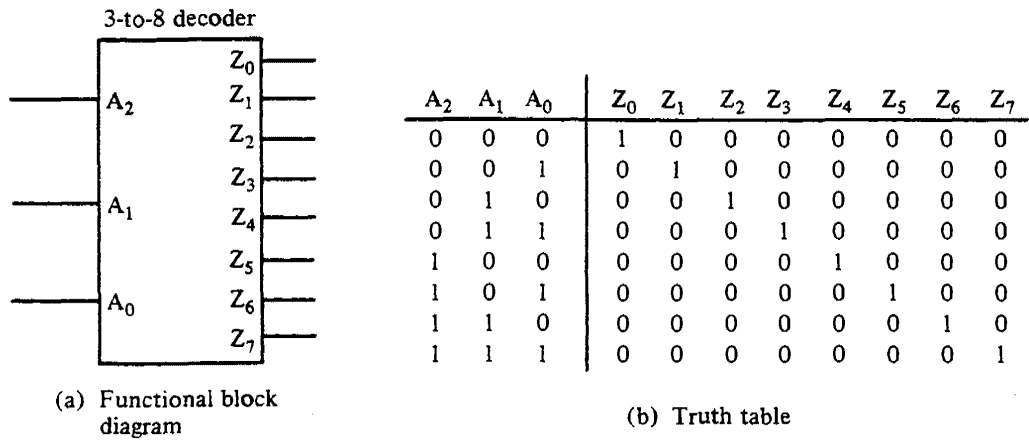


Figure 4.9 3-to-8 decoder.

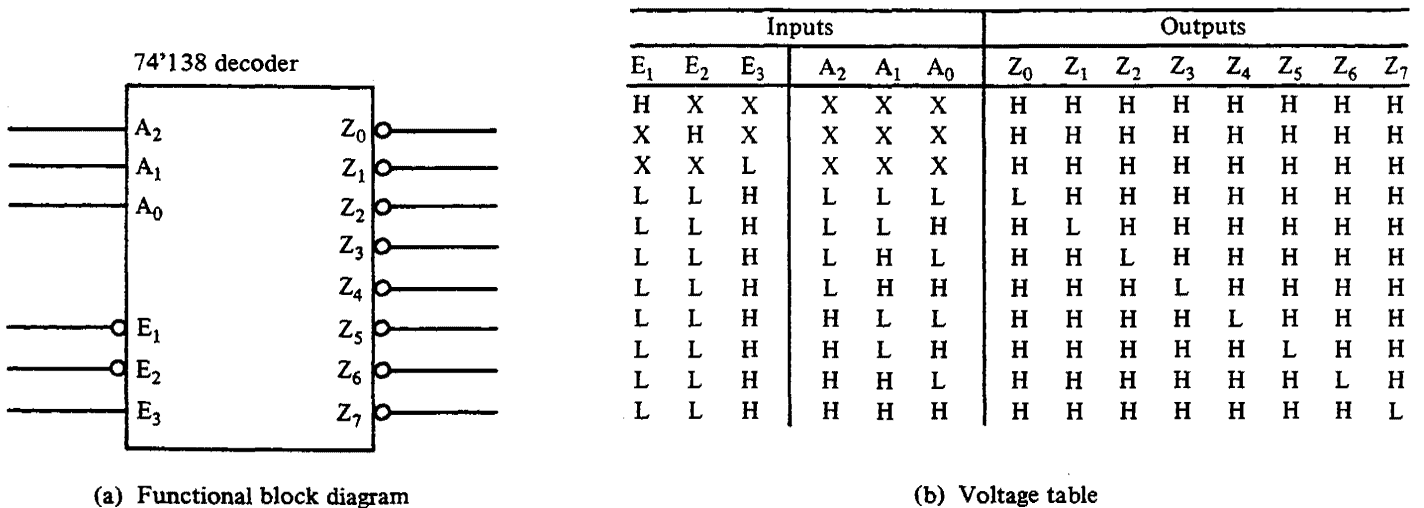


Figure 4.10 74'138 decoder.

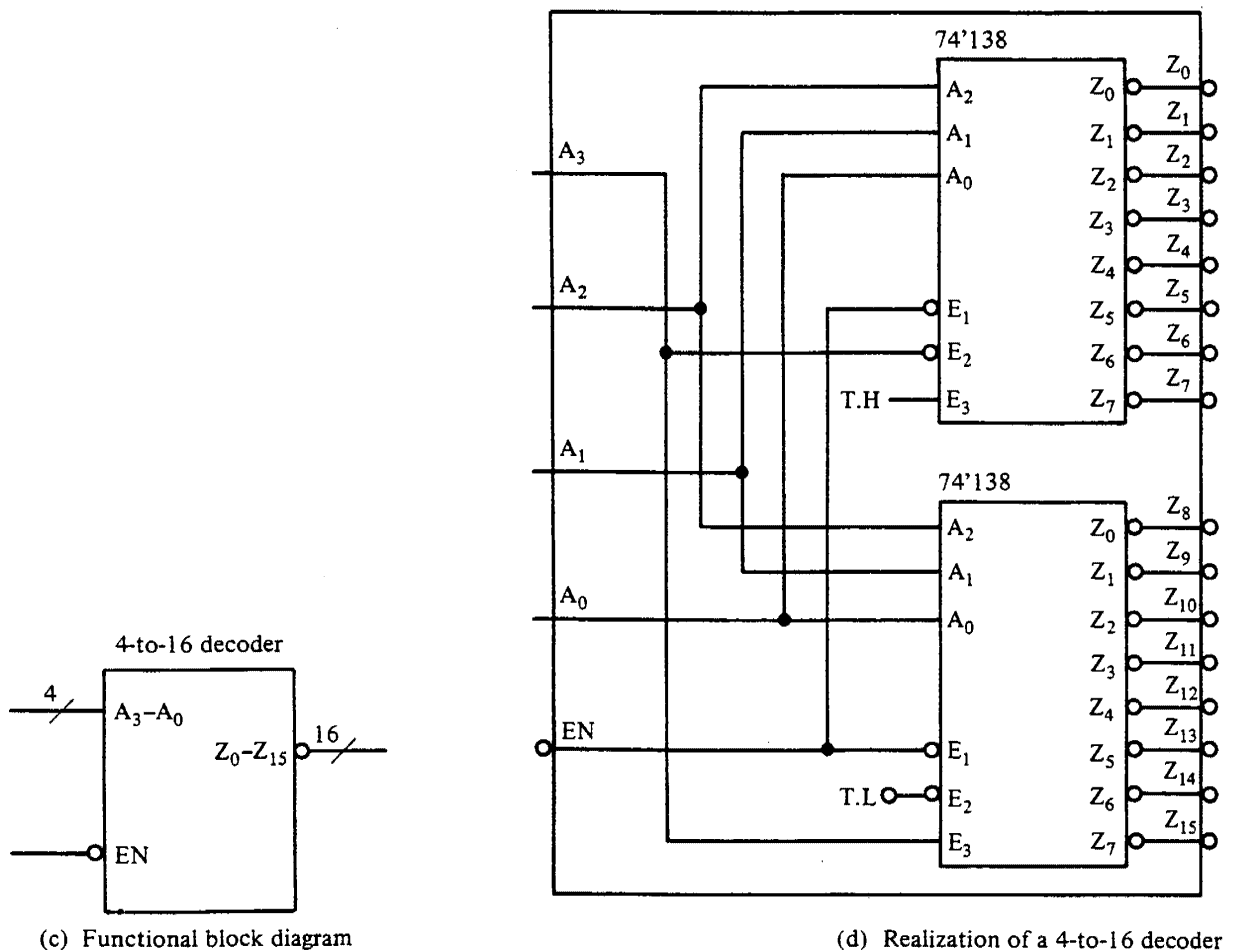


Figure 4.10 (cont.)

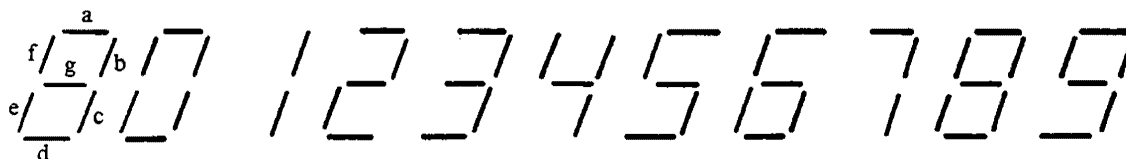
of a larger size can be constructed by using decoders of a smaller size along with some additional circuitry. Is such circuitry needed to construct a 5-to-32 decoder from four 74'138 decoders? (See Problem 4.8.)

4.4.1 BCD-to-7-Segment Decoder

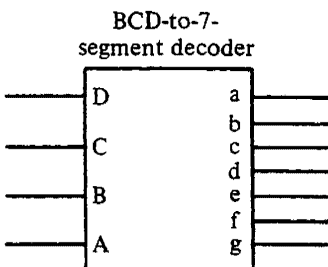
Outputs of a digital circuit are often displayed as decimal digits. The most common and simplest device for displaying a decimal digit is a 7-segment display, as shown in Fig. 4.11(a). Each of the segments is an LED (light-emitting diode) that will glow when a true signal is applied to it. By a proper selection of the segments to be lit, the decimal digits can be displayed as shown in Fig. 4.11(a).

Seven-segment displays are commercially available in two forms: common anode and common cathode. The common-anode display, with all the LED anodes connected together, is active-low. And, the common-cathode display, with all the LED cathodes connected together, is active-high.

A BCD-to-7-segment decoder is a combinational circuit element that converts a BCD number into the signals required for the display of the value of that number on a 7-segment display. The functional block diagram for such a decoder is shown in Fig. 4.11(b). The seven decoder outputs (a, b, c, d, e, f, g) correspond to the seven segments with the same labels of the 7-segment display. The functional description of the decoder,



(a) A 7-segment display



(b) Functional block diagram

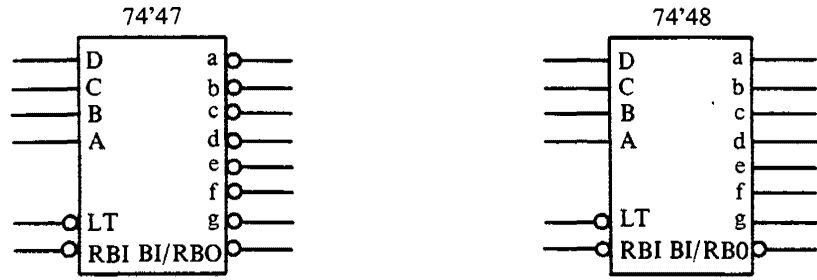
D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

(c) Truth table

Figure 4.11 BCD-to-7-segment decoder.

in the form of a truth table, is shown in Fig. 4.11(c). Observe from the first row of the truth table that for the display of the digit 0, segments a, b, c, d, e, and f have to be lit, as is evident from Fig. 4.11(a). For the display of the digit 1, the second row specifies that segments b and c have to be lit, and so forth. From another point of view, this truth table specifies that the output corresponding to segment a has to be true for the displays of digits 0, 2, 3, 5, 6, 7, 8, and 9. Also, the output corresponding to segment b has to be true for digits 0, 1, 2, 3, 4, 7, 8, and 9, and so forth. Note that since binary numbers 1010 through 1111 are not valid BCD representations, the outputs for these inputs are designated as don't cares in the truth table. The logic equations for the seven outputs can be determined in a straightforward manner. (See Problem 4.10.)

BCD-to-7-segment decoders are commercially available with either active-low or active-high outputs. Examples are the 74'47 (active-low) and the 74'48 (active-high). The functional block diagrams and voltage tables of both are shown in Figs. 4.12(a) and (b). Note from the voltage tables of Fig. 4.12(b) that the inverse in the output levels is



(a) Functional block diagrams for the 74'47 and 74'48



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

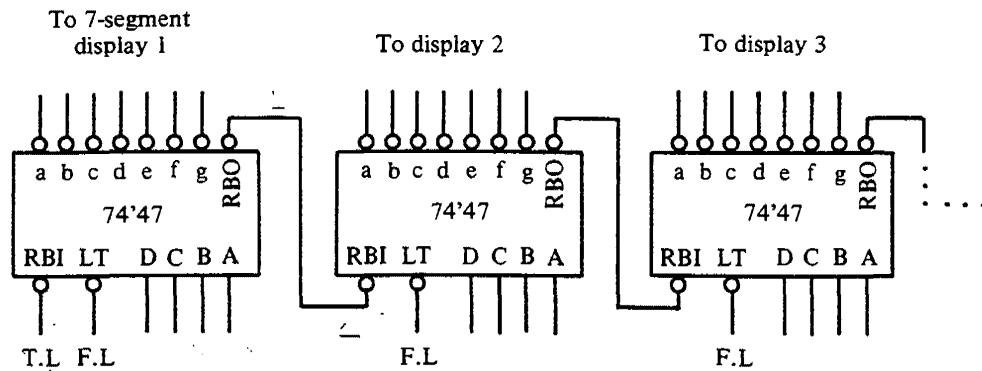
Numerical designations — resultant displays

Decimal or function	Inputs						Outputs							
	LT	RBI	D	C	B	A	BI/RBO	a	b	c	d	e	f	g
0	H	H	L	L	L	L	H	L	L	L	L	L	L	H
1	H	X	L	L	L	H	H	L	L	L	H	H	H	H
2	H	X	L	L	H	L	H	L	L	H	L	L	H	L
3	H	X	L	L	H	H	H	L	L	L	H	H	L	L
4	H	X	L	H	L	L	H	H	L	L	H	H	L	L
5	H	X	L	H	L	H	H	L	H	L	L	L	L	L
6	H	X	L	H	H	L	H	H	L	L	L	L	L	L
7	H	X	L	H	H	H	H	L	L	L	H	H	H	H
8	H	X	H	L	L	L	H	L	L	L	L	L	L	L
9	H	X	H	L	L	H	H	L	L	L	H	H	L	L
10	H	X	H	L	H	L	H	H	H	L	L	H	L	L
11	H	X	H	L	H	H	H	H	L	L	H	H	L	L
12	H	X	H	H	L	L	H	H	L	H	H	H	L	L
13	H	X	H	H	L	H	H	L	H	H	L	H	L	L
14	H	X	H	H	H	L	H	H	H	L	L	L	L	L
15	H	X	H	H	H	H	H	H	H	H	H	H	H	H
BI	X	X	X	X	X	X	L	H	H	H	H	H	H	H
RBI	H	L	L	L	L	L	L	H	H	H	H	H	H	H
LT	L	X	X	X	X	X	H	L	L	L	L	L	L	L

74'47

74'48

(b) Displays and voltage tables for the 74'47 and 74'48



(c) Use of RBI and RBO in a cascade of 7-segment displays

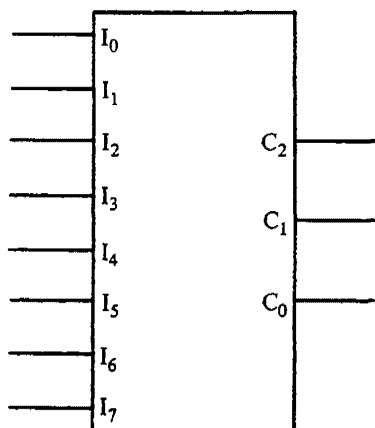
Figure 4.12 74'47 and 74'48 BCD-to-7-segment decoders.

the only difference in the operation of these decoders. Also, the outputs for 1010 through 1111 are *not* don't cares, in contrast to those shown in Fig. 4.11(c). Instead, the specified special characters will be displayed. In the display of BCD numbers, however, these outputs should never occur unless there is an error.

Observe that these BCD-to-7-segment decoders have additional features, which are provided by the inputs LT, RBI, and the input/output BI/RBO. The lamp test (LT) input can be used to test the LEDs of an attached 7-segment display. As shown in the last row of the voltage tables of Fig. 4.12(b), all segments of the display can be lit by making LT true (L) and BI/RBO false (H) for either the 74'47 or the 74'48. Now consider the BI/RBO input/output for which the BI is an abbreviation for blanking input, and RBO for ripple-blanking output. The BI/RBO input/output serves two functions. Used as an input, as shown in row 17 of either voltage table, a true (L) applied to the BI/RBO input will blank the display. As an output, the BI/RBO signal is used in conjunction with RBI (ripple-blanking input) to suppress the display of leading zeros in a cascade of 7-segment displays. As illustrated in Fig. 4.12(c), for decoder 1 the RBI input is connected to true (L). Then, if the digit to be displayed is 0, the output is a blank (all outputs false) and the RBO output is true (L), as indicated by row 18 (RBI) of the voltage tables. As a result, the RBI input of decoder 2 is true, and if the digit to be displayed on display 2 is 0, then its output is a blank and its RBO output is true (L), and so forth. In this manner, all the leading zeros of the 7-segment displays will be displayed as blanks. The first nonzero digit will cause its RBO to be false (H). Therefore, any subsequent embedded zeros (e.g., as in 430103) will be displayed as zeros and not blanks, as indicated by the first row of either voltage table. How should RBI and RBO be connected if leading zeros are desired? (See Problem 4.11.)

4.5 ENCODER

The inverse of the decoding function is the encoding function. For N different inputs, only one of which is activated, an *encoder* is a combinational circuit element that generates an M -bit binary code that uniquely identifies the activated input. Here, $2^M \geq N$. An example of an 8-to-3 encoder is shown in Fig. 4.13. Note that in this definition of an encoder, one and only one input can be activated at a time. Otherwise the circuit has



(a) Functional block diagram

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	C_2	C_1	C_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

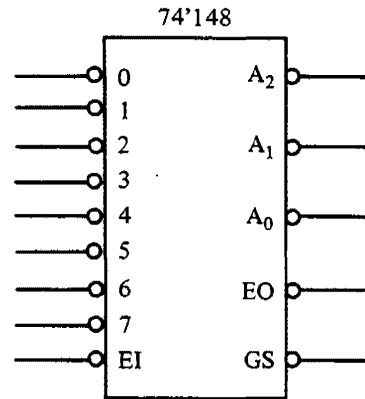
(b) Truth table

Figure 4.13 8-to-3 encoder.

no meaning. In other words, this circuit does not allow the case where several inputs are activated simultaneously or where no input is activated.

4.5.1 Priority Encoder

Encoders are available as MSI circuit elements in the form of *priority* encoders, which allow the activation of several inputs simultaneously, or no input at all. An example of an 8-to-3 priority encoder is the 74'148 shown in Fig. 4.14. As illustrated in the functional



(a) Functional block diagram

Inputs								Outputs					
EI	0	1	2	3	4	5	6	7	A ₂	A ₁	A ₀	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	L	H	H	H	H	L	H	H	L	H
L	X	X	L	H	H	H	H	H	H	L	L	L	H
L	X	L	H	H	H	H	H	H	H	L	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

(b) Voltage table

Inputs								Outputs					
EI	0	1	2	3	4	5	6	7	A ₂	A ₁	A ₀	GS	EO
0	X	X	X	X	X	X	X	X	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	X	X	X	X	X	X	X	1	1	1	1	1	0
1	X	X	X	X	X	X	1	0	1	1	0	1	0
1	X	X	X	X	1	0	0	0	1	0	0	1	0
1	X	X	X	1	0	0	0	0	0	1	1	1	0
1	X	X	1	0	0	0	0	0	0	1	0	1	0
1	X	1	0	0	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0

(c) Truth table.

Figure 4.14 74'148 priority encoder.

block diagram of Fig. 4.14(a), the eight inputs (0, 1, . . . , 7) and the three outputs (A_2 , A_1 , A_0) are active-low. Furthermore, there is an active-low enable input (EI) and two active-low outputs, GS and enable output (EO).

The 74'148 functions as an encoder only when the enable input EI is true. Otherwise all outputs are false, as shown in row 1 of the truth table of Fig. 4.14(c). Note that when more than one input is activated, the 74'148 encodes the input with the highest priority, with input 7 having priority over input 6, which has priority over input 5, and so forth. When input 0 is the only input activated (the last row), then the output code generated is $A_2A_1A_0 = 000$. When no input is activated (row 2), the code generated is also $A_2A_1A_0 = 000$. In this case, the GS output is used to make the distinction. GS is true only when at least one of the inputs is activated. The enable output EO and enable input EI can be used to readily cascade 74'148 priority encoders for octal expansion. (See Problem 4.13.)

4.6 MULTIPLEXER

In the design of a digital circuit, another frequently required logic function is the selection function. A *multiplexer* (MUX) is a combinational circuit element that selects data from one of many inputs and directs it to a single output. Conceptually, the function of a multiplexer can be illustrated by the multipositional switch of Fig. 4.15, which has a number of input lines, but a single output line. Depending on the selection control, the switch will connect a specific one of the inputs to the output Z. This electrical circuit connection is the popular way of considering multiplexer action. But, as will be seen, with a digital circuit multiplexer, there is no direct connection between any input and the output. Despite this, though, the state of the output is the same as that of the selected input.

The block diagram of a four-input MUX is shown in Fig. 4.16(a). As specified in the truth table of Fig. 4.16(b), if for the selection signals $S_1S_0 = 00$, then the output Z is electrically connected to I_0 . If $S_1S_0 = 01$, then Z is electrically connected to I_1 , and so forth. Note that the truth table in Fig. 4.16(b) is a condensed version of the actual truth table, which would have six input variables (I_0 , I_1 , I_2 , I_3 , S_1 , and S_0) and 2^6 rows. (See Problem 4.14.) The design and realization of the four-input MUX is shown in Fig. 4.16(c).

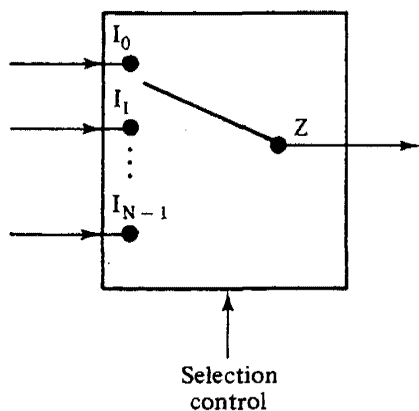
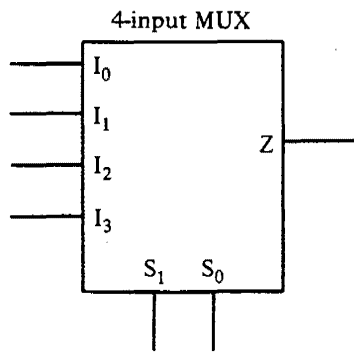


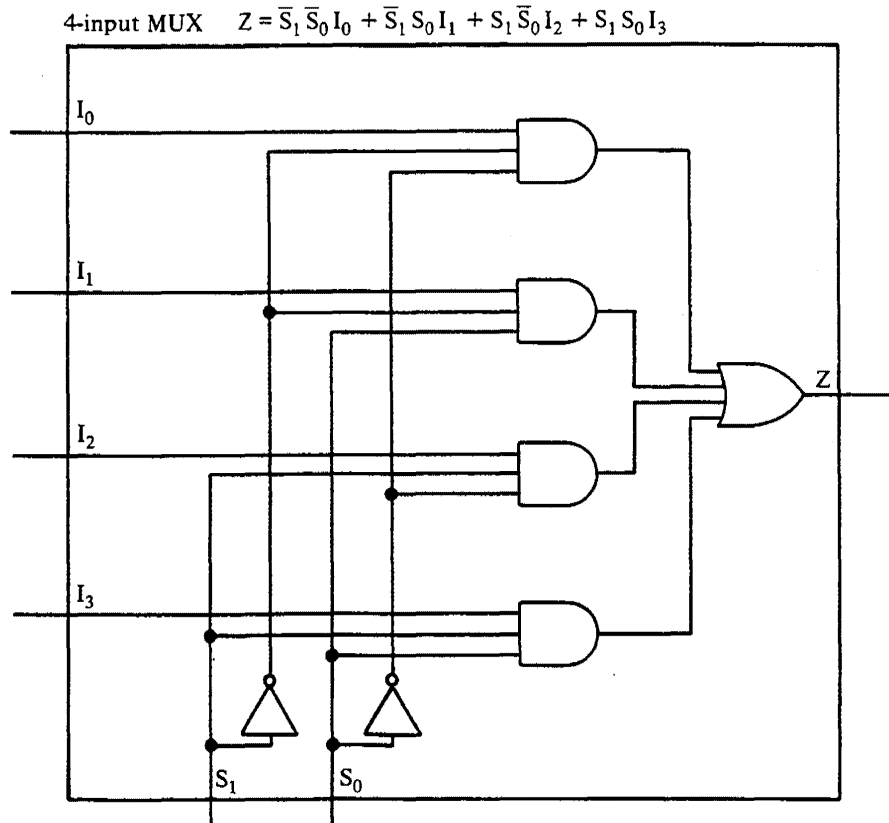
Figure 4.15 N-input switch.



(a) Functional block diagram

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

(b) Condensed truth table

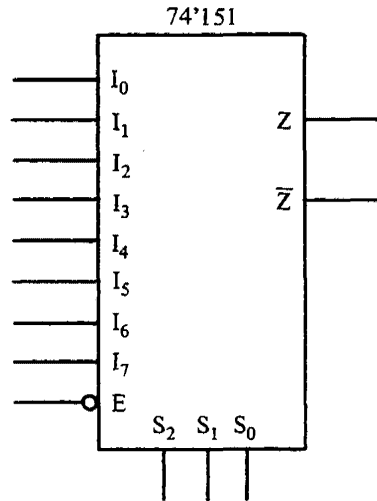


(c) Design and realization

Figure 4.16 Four-input multiplexer.

Multiplexers are commercially available as MSI circuit elements with 2, 4, 8, and 16 inputs and with inverting and/or noninverting outputs (or active-low and active-high outputs). An example of a commercially available eight-input MUX with inverting and noninverting outputs is the 74'151 shown in Fig. 4.17. As specified in the voltage table and the logic equation, the 74'151 functions as a multiplexer only if the enable input E is true (L). Otherwise, the output Z is false. When the enable input E is true, then one of the inputs (selected by the selection inputs) is electrically connected to the output Z.

Multiplexers are useful in the design of a digital circuit where the data for an input of a device is from several sources. Using a multiplexer, we can easily control the source



(a) Functional block diagram

E	S ₂	S ₁	S ₀	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	Z̄	Z
H	X	X	X	X	X	X	X	X	X	X	X	H	L
L	L	L	L	L	X	X	X	X	X	X	X	H	L
L	L	L	L	H	X	X	X	X	X	X	X	L	H
L	L	L	H	X	L	X	X	X	X	X	X	H	L
L	L	L	H	X	H	X	X	X	X	X	X	L	H
L	L	H	L	X	X	L	X	X	X	X	X	H	L
L	L	H	L	X	X	H	X	X	X	X	X	L	H
L	L	H	H	X	X	X	L	X	X	X	X	H	L
L	L	H	H	X	X	X	H	X	X	X	X	L	H
L	H	L	L	X	X	X	X	L	X	X	X	H	L
L	H	L	L	X	X	X	X	H	X	X	X	L	H
L	H	L	H	X	X	X	X	X	L	X	X	H	L
L	H	L	H	X	X	X	X	X	H	X	X	L	H
L	H	H	L	X	X	X	X	X	X	L	X	H	L
L	H	H	L	X	X	X	X	X	X	H	X	L	H
L	H	H	H	X	X	X	X	X	X	X	L	H	L
L	H	H	H	X	X	X	X	X	X	X	H	L	H

(b) Voltage table

$$Z = E(\bar{S}_2\bar{S}_1\bar{S}_0I_0 + \bar{S}_2\bar{S}_1S_0I_1 + \bar{S}_2S_1\bar{S}_0I_2 + \bar{S}_2S_1S_0I_3 + S_2\bar{S}_1\bar{S}_0I_4 + S_2\bar{S}_1S_0I_5 + S_2S_1\bar{S}_0I_6 + S_2S_1S_0I_7)$$

(c) Logic equation

Figure 4.17 74'151 MUX.

of the input. Figure 4.18 shows how four 4-input MUXs can be used to select one of four 4-bit data to be processed by the device. For S₁S₀ inputs of 00, 01, 10, or 11, either the 4-bit data A, B, C, or D is connected to input X of the device.

4.6.1 Three-State Logic Element

Multiplexing can also be realized with a *three-state* (sometimes called tristate) logic element. Ordinarily, the voltage level of an output of a device can only be in one of two

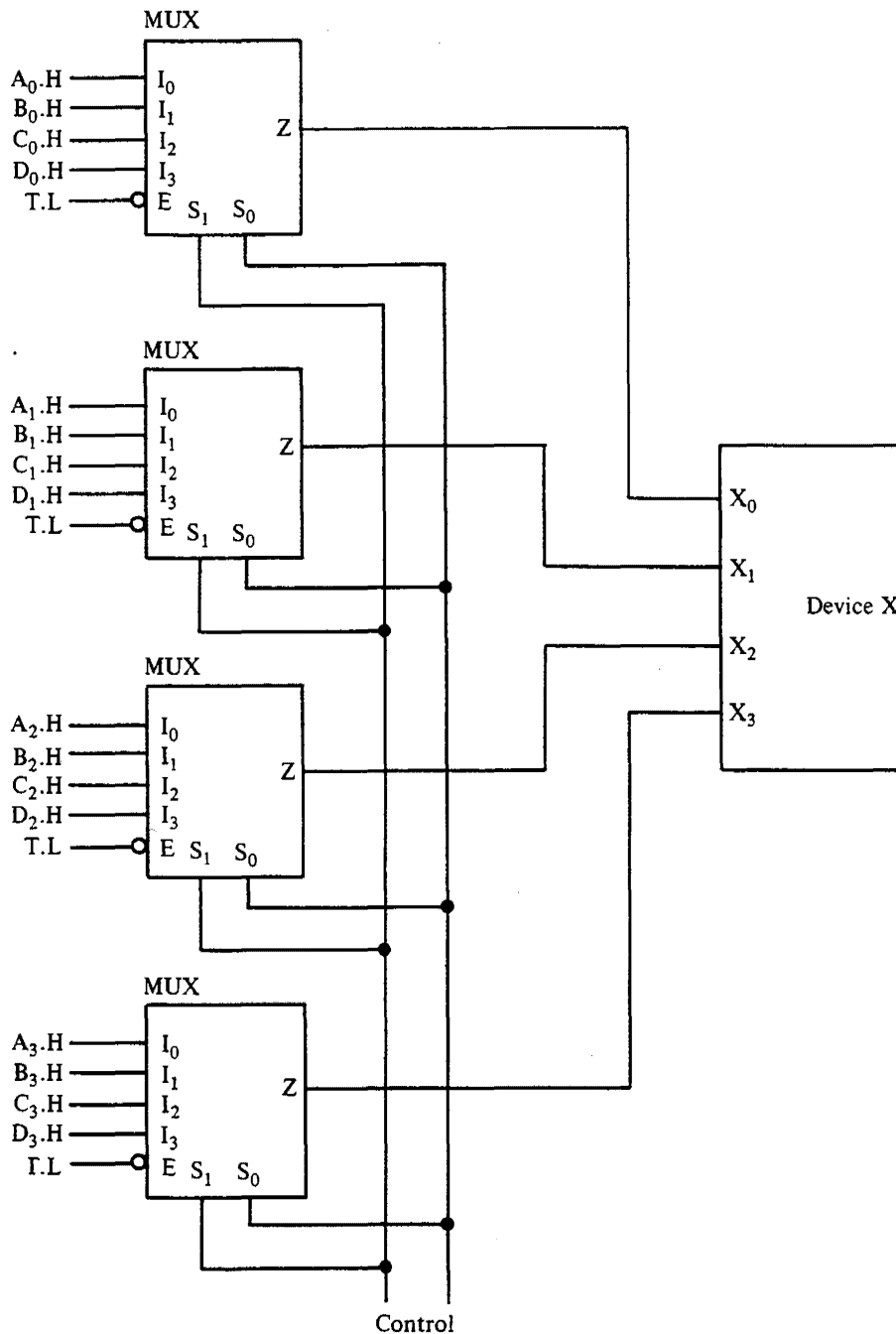
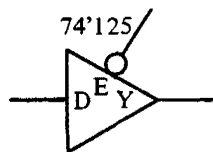


Figure 4.18 Using MUXs for data selection.

states: high or low. However, for a three-state device, such as the 74'125 shown in Fig. 4.19, a third high-impedance state is possible. As shown by the voltage table of Fig. 4.19(b), if the enable input E is true (L), then the device behaves normally with the two states of high and low. If, however, the enable input E is false (H), then the output is in the high-impedance state.

In the high-impedance state, the output does not drive or load any circuit connected to it. In other words, the output behaves as if it were electrically disconnected. As a result, we can use this three-state logic element to realize the multiplexing function. An example of this application is shown to Fig. 4.19(c). Note that the circuit shown here is equivalent to the part of the circuit that is connected to X_0 of the device in Fig. 4.18.

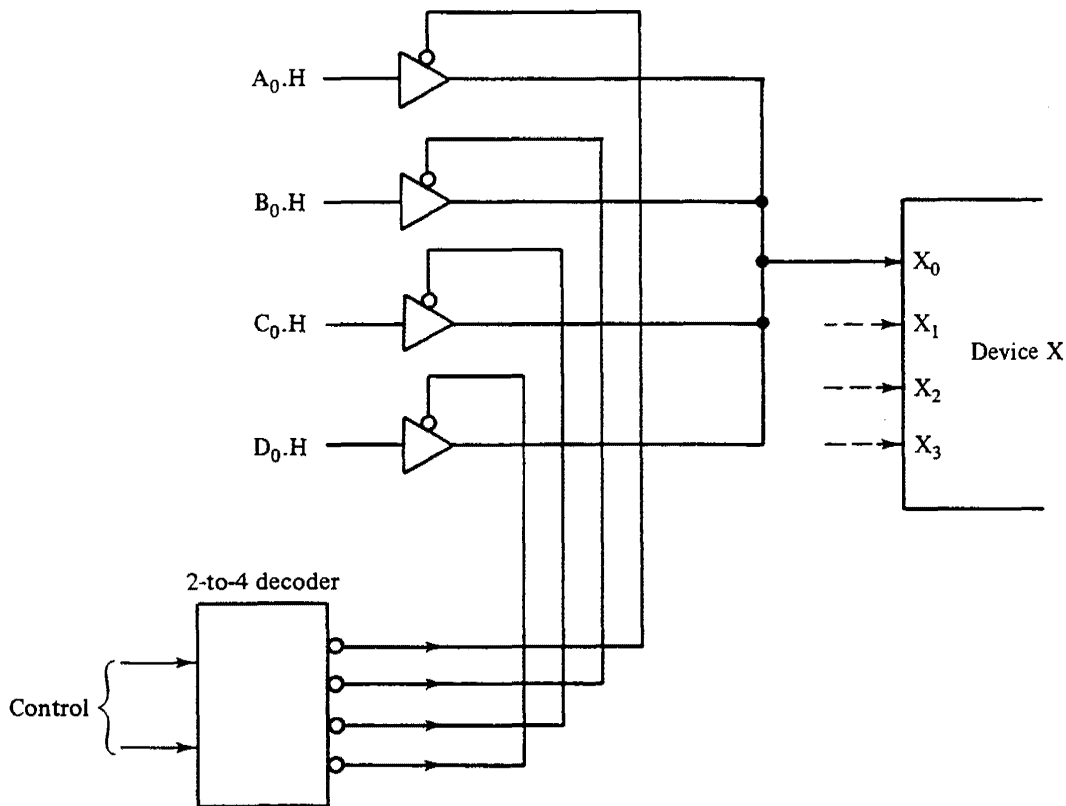


(a) Functional block diagram

E	D	Y
L	L	L
L	H	H
H	X	(Z)

(Z) = high impedance

(b) Voltage table



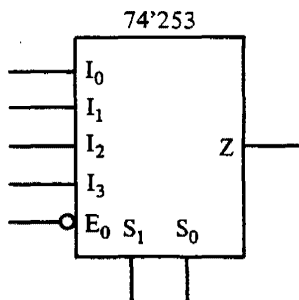
(c) Use of a three-state device for data selection

Figure 4.19 Three-state logic element.

Select inputs		Data inputs				Output enable	Output
S ₁	S ₀	I ₀	I ₁	I ₂	I ₃	E ₀	Z
X	X	X	X	X	X	H	(Z)
L	L	L	X	X	X	L	L
L	L	H	X	X	X	L	H
L	H	X	L	X	X	L	L
L	H	X	H	X	X	L	H
H	L	X	X	L	X	L	L
H	L	X	X	H	X	L	H
H	H	X	X	X	L	L	L
H	H	X	X	X	H	L	H

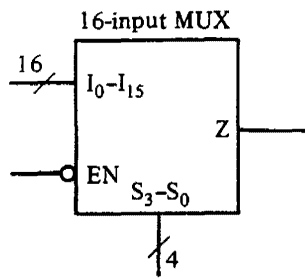
Where (Z) is a high impedance

(b) Voltage table

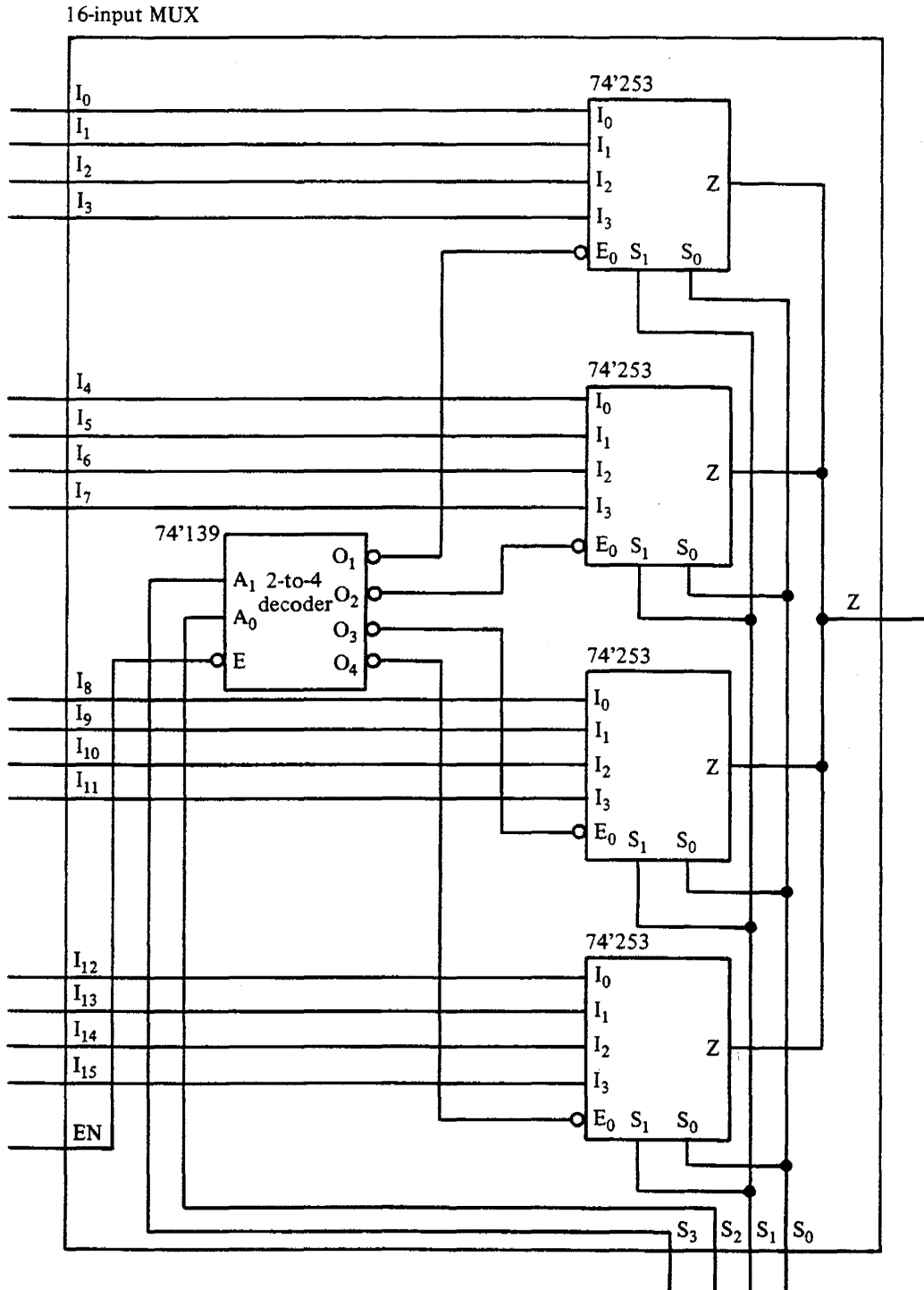


(a) Functional block

Figure 4.20 MUX with three-state outputs.



(c) Functional block diagram of a 16-input MUX



(d) Realization

Figure 4.20 (cont.)

Specifically, when the control signal is 00, then the three-state device for A_0 is enabled while the others are disabled. Consequently, only A_0 is connected to X_0 since B_0 , C_0 , and D_0 are electrically disconnected from X_0 . Similarly, when the control signal is 01, then B_0 is connected to X_0 , and so forth.

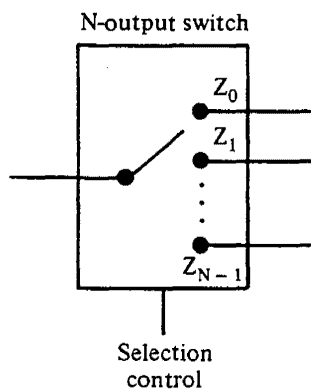
Three-state outputs for bus control are often required in the designs of digital circuits. In other words, it is often necessary to connect together a number of three-state outputs from different devices to form a bus for the transfer of data among devices. Consequently, many of the commercially available circuit elements have built-in three-state outputs. For example, the 74'253 is a four-input multiplexer with a three-state output. The functional block diagram and voltage table for it are shown in Figs. 4.20(a) and (b), respectively. When the output enable E_0 is true (L), then the 74'253 functions as a four-input MUX. But when the output enable E_0 is false (H), then the output Z is in a high-impedance state. Figure 4.20(d) shows how four 74'253 MUXs and a 2-to-4 decoder can be connected to function as a 16-input MUX. When its selection signal is $S_3S_2S_1S_0 = 0000$, then the input I_0 is connected to the output Z . When $S_3S_2S_1S_0 = 0001$, then I_1 is connected to the output Z , and so forth. Note that at any one time, S_3 and S_2 enable one of the four MUXs and disable the other three. Consequently, although the outputs of all four MUXs are connected together, only one of them is electrically connected to the output at any one time.

4.7 DEMULTIPLEXER

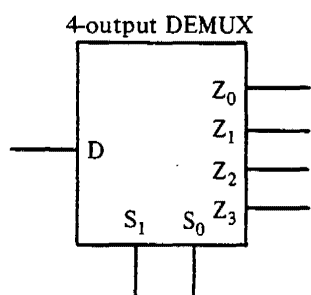
The inverse of multiplexing is demultiplexing. A *demultiplexer* (DEMUX) is a combinational circuit element that selects one from a number of outputs and connects it to a single input. Conceptually, the operation of a demultiplexer can be illustrated by another multipositional switch, as shown in Fig. 4.21(a). Here, there are a number of output lines, but only a single input line. Depending on the selection control, the switch will connect one of the outputs to the input D .

A block diagram of a four-output demultiplexer is shown in Fig. 4.21(b), and its truth table in Fig. 4.21(c). As shown in Fig. 4.21(c), if the selection signal $S_1S_0 = 00$, then input D is electrically connected to Z_0 . If $S_1S_0 = 01$, then D is connected to Z_1 , and so forth. Observe that the truth table can be reduced by combining the first four rows to a single row with don't-care entries for S_1 and S_0 since for D equal to 0, all outputs are 0, regardless of the S_1 and S_0 values.

Demultiplexers are commercially available as MSI circuit elements, an example of which is the 74'138 DEMUX. Its functional block diagram is shown in Fig. 4.22(a). The input E_1 is the data input and Z_7-Z_0 are the eight outputs. E_2 and E_3 are enable inputs. Depending on the selection control signals energizing the $A_2A_1A_0$ inputs, a particular one of the outputs is electrically connected to the input E_1 . At least this is the way the operation is usually explained. Actually, the output is not connected to the input, but the operation is the same as if it were. As shown, what really happens is that if the input is false (H), then all outputs are false (H). But if the input is true (L), then only one output is true (L), the particular output depending on the selection control signal. The voltage table for the 74'138 DEMUX is shown in Fig. 4.22(b), and the corresponding truth table in Fig. 4.22(c).



(a) N-output switch



(b) Functional block diagram

D	S ₁	S ₀	Z ₀	Z ₁	Z ₂	Z ₃
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

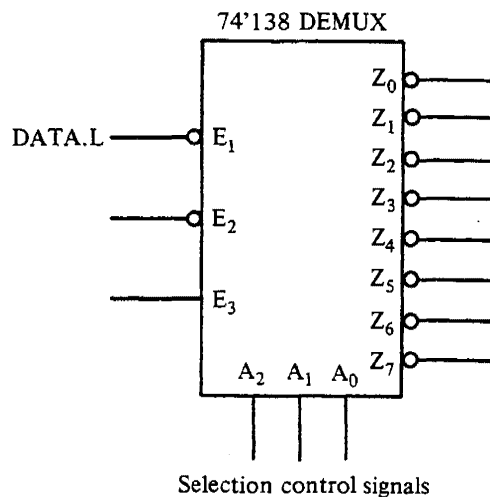
(c) Truth table

Figure 4.21 Demultiplexer.

Recall that this 74'138 DEMUX was used as a decoder in Sec. 4.4 (Fig. 4.10). This double usage is possible because the truth table for a demultiplexer is identical to that of a decoder that has one or more enable inputs. Consequently, any commercially available decoder with one or more enable inputs can also function as a demultiplexer.

4.8 DESIGN CONSIDERATIONS FOR INTEGRATED CIRCUIT (IC) ELEMENTS

In this chapter we have studied some of the commonly used logic functions and the designs and applications of the circuit elements that realize these logic functions. These combinational MSI circuit elements are a part of the building blocks that are required in



(a) Functional block diagram

Inputs						Outputs							
E ₁	E ₂	E ₃	A ₂	A ₁	A ₀	Z ₀	Z ₁	Z ₂	Z ₃	Z ₄	Z ₅	Z ₆	Z ₇
H	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	L	X	X	X	H	H	H	H	H	H	H	H
L	L	H	L	L	L	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

(b) Voltage table

Enable			Data	Selection			Outputs							
E ₂	E ₃	E ₁	A ₂	A ₁	A ₀	Z ₀	Z ₁	Z ₂	Z ₃	Z ₄	Z ₅	Z ₆	Z ₇	
X	X	0	X	X	X	0	0	0	0	0	0	0	0	
0	X	X	X	X	X	0	0	0	0	0	0	0	0	
X	0	X	X	X	X	0	0	0	0	0	0	0	0	
1	1	1	0	0	0	1	0	0	0	0	0	0	0	
1	1	1	0	0	1	0	1	0	0	0	0	0	0	
1	1	1	0	1	0	0	0	1	0	0	0	0	0	
1	1	1	0	1	1	0	0	0	1	0	0	0	0	
1	1	1	1	0	0	0	0	0	0	1	0	0	0	
1	1	1	1	0	1	0	0	0	0	0	1	0	0	
1	1	1	1	1	0	0	0	0	0	0	0	1	0	
1	1	1	1	1	1	0	0	0	0	0	0	0	1	

(c) Truth table

Figure 4.22 74'138 DEMUX.

the designs of digital circuits, as will be discussed in Chapter 7. Before we proceed with the study of the sequential MSI circuit elements in the next chapter, let us consider some of the design considerations for these integrated circuit elements.

4.8.1 TTL Digital Logic Family

TTL (Transistor-Transistor Logic), which was briefly considered in Sec. 3.2, is currently the dominant digital logic family for SSI and MSI ICs. The popularity of the TTL family results from its ease of use, low cost, low power consumption, relatively high speed of operation, low noise susceptibility, and good output drive capability. Since its introduction in the early 1960s, it has been expanded to include hundreds of logic functions, and has evolved into the following series:

1. Standard TTL
2. H-TTL (high-speed TTL)
3. L-TTL (low-power TTL)
4. S-TTL (Schottky TTL)
5. LS-TTL (low-power Schottky TTL)
6. ALS-TTL (advanced LS-TTL)
7. AS-TTL (advanced S-TTL)

As mentioned in Sec. 3.2, a TTL element is identified by a label, the first two digits of which are 74 for commercial-grade products and 54 for military-grade products. If the element is other than standard TTL, then letters follow the 74 to identify the series. No letter corresponds to standard TTL, the letter H to high-speed TTL, L to low-power, S to Schottky, LS to low-power Schottky, ALS to advanced low-power Schottky, and AS to advanced Schottky. The remaining part of the label is a two- or three-digit number that identifies the type of element and perhaps some other features, such as the number of elements per chip. For example, 00 identifies a quadruple two-input positive-logic NAND gate, which means that there are four NAND gates on a chip, and each of the NAND gates has two inputs. Manufacturers identify the elements by their positive-logic functions.

The corresponding circuit elements of each of the TTL series are functionally identical. For example, a 7400, a 74LS00, and a 74ALS00 are all two-input NAND gates. The differences among them are physical characteristics such as speed of operation, power dissipation, input loading, output drive capacity, noise margin, and so forth.

A comparison of the various series in terms of speed and power is given in Table 4.1. Perhaps a couple of the column headings need some explanation. As will be described in more detail in Sec. 4.8.3, propagation delay is approximately the time required for a signal to pass through an element. So, the *smaller* the propagation delay, the greater the maximum speed of operation. The power-delay product (PDP) is the product of the propagation delay and power dissipation. This product is one measure of the desirability of an element. Often, but not always, the smaller this product, the better the element.

As shown in Table 4.1, the standard TTL, which was the first series, is relatively fast, but has a fairly high power dissipation. The H-TTL is a high-speed series. Its power dissipation, however, is enormous. For high-speed applications, the H-TTL ICs have

TABLE 4.1 TYPICAL PERFORMANCE COMPARISON OF THE TTL LOGIC FAMILY

Series	Gate propagation delay (ns)	Power dissipation (mW)	Power-delay product (pJ)
Standard TTL	10	10	100
H-TTL	6	22	132
L-TTL	33	1	33
S-TTL	3	19	57
LS-TTL	9.5	2	19
ALS-TTL	4	1	4
AS-TTL	1.5	10	15

been replaced by those of the newer S-TTL Schottky TTL series. The S-TTL series is faster than the H-TTL series. Also, the power dissipation is less, but still substantial.

The L-TTL is a low-power series. Unfortunately, it also has very slow speed. The L-TTL series has been replaced by the newer LS-TTL low-power Schottky series. The LS-TTL ICs consume slightly more power than those of the L-TTL series, but are significantly faster. In fact, as is evident from Table 4.1, the LS-TTL series has one of the best power-delay products. Additionally, there are an extensive number of logic functions available in this series, and the chips are reasonably priced. Consequently, the LS-TTL series is currently the most popular series in the TTL family. The ALS-TTL (advanced LS-TTL) and the AS-TTL (advanced S-TTL) are high-performance series that are improvements over the LS-TTL and the S-TTL series, respectively. The ALS-TTL series has the best power-delay product, but the AS-TTL series is the fastest. However, product offerings in these two series have yet to match those of the LS-TTL and the S-TTL series.

Ordinarily, circuit elements from the same TTL series are used throughout a digital circuit. Sometimes, however, circuit elements from different series are used together to obtain optimum performance. High-performance AS-TTL circuit elements can be used, for example, in speed-critical portions of a digital circuit. And, for portions of the digital circuit where speed is not crucial, slower circuit elements with lower power dissipation can be used. The actual mix of ICs from the different TTL series is, of course, a function of the overall circuit specifications and requirements. For an optimum selection of a TTL series or a mix of circuit elements from different TTL series, it is important to understand the digital IC physical characteristics presented in the following sections.

4.8.2 Parameters for Static Characteristics

The various series within the TTL logic family are characterized by static (dc) parameters and switching (ac) parameters. Static parameters describe the input and output behavior of the IC elements under stable operating conditions. The major static parameters are illustrated in Fig. 4.23 and defined as follows:

- I_{IH} High-level input current; the current that flows into an input when a high voltage is applied. This represents the current requirement of an input that is in a high state.

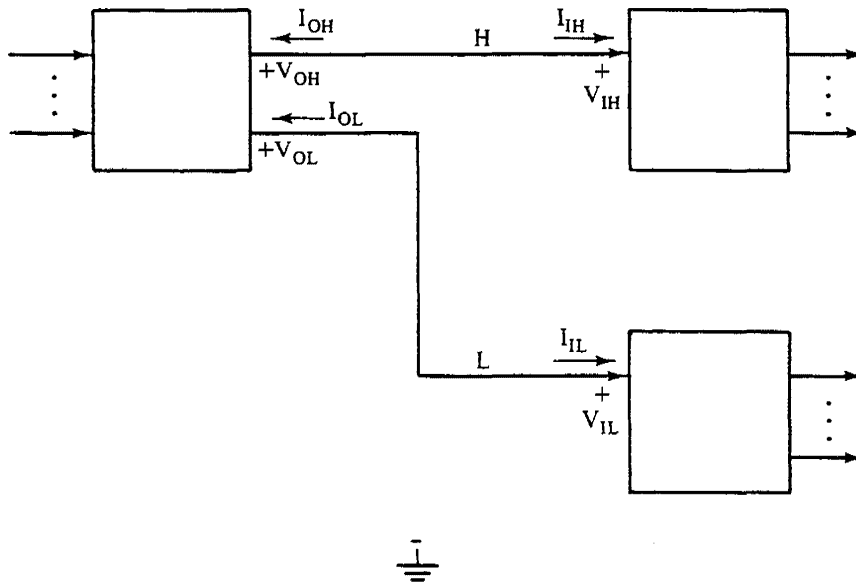


Figure 4.23 Illustration of static parameters.

- I_{IL} Low-level input current; the current that flows into an input when a low voltage is applied. This represents the current requirement of an input that is in a low state.
- I_{OH} High-level output current; the current that flows into an output that is in a high state.
- I_{OL} Low-level output current; the current that flows into an output that is in a low state.
- V_{IH} High-level input voltage; the input voltage that is recognized by the circuit element as a high level.
- V_{IL} Low-level input voltage; the input voltage that is recognized by the circuit element as a low level.
- V_{OH} High-level output voltage; the output voltage that the circuit element will provide when the output is at a high level.
- V_{OL} Low-level output voltage; the output voltage that the circuit element will provide when the output is at a low level.
- I_{CC} Supply current; the current flowing into the V_{CC} supply terminal of an IC. The product of this current and the supply voltage is the power dissipation of the IC.

Note that both input and output current parameters have references into the TTL device. Consequently, if an input or output current is specified as being negative, the actual current flow is out of the TTL device input or output.

The worst-case and typical values for these parameters for three TTL series are given in Table 4.2. Applications for these parameters are presented in the next two sections.

TABLE 4.2 TTL STATIC PARAMETERS

	Standard TTL			LS-TTL			ALS-TTL			Units
	Min.	Typ.	Max.	Min.	Typ.	Max.	Min.	Typ.	Max.	
I_{IH}			40.0			20.0			20.0	μA
I_{IL}			-1.6			-0.4			-0.1	mA
I_{OH}			-0.4			-0.4			-0.4	mA
I_{OL}			16.0			8.0			8.0	mA
V_{IH}	2.0			2.0			2.0			V
V_{IL}			0.8			0.8			0.8	V
V_{OH}	2.4	3.4		2.7	3.5		2.5	3.0		V
V_{OL}		0.2	0.4		0.35	0.5		0.35	0.5	V

Input Loading and Output Drive

In a digital circuit an output of a circuit element is usually connected to inputs of other circuit elements. Each additional input presents an additional load to the output because it requires a certain amount of current. But an output can supply only a limited amount of current. Exceeding the specified maximum amount will cause the corresponding circuit element to function improperly and possibly be damaged. The *output drive* capability of an output (called the fan-out) is a measure of the number of inputs that the output of a circuit element can drive without impairing its operation. The output drive capability of a TTL element can be calculated from the following formulas:

For high-voltage level,

$$\text{Output drive} = \left| \frac{I_{OH}(\text{max})}{I_{IH}(\text{max})} \right|$$

For low-voltage level,

$$\text{Output drive} = \left| \frac{I_{OL}(\text{max})}{I_{IL}(\text{max})} \right|$$

In other words, the output drive is measured by the maximum amount of current that an output can supply compared to the maximum amount of current that an input will require. For example, from Table 4.2 we see that for the LS-TTL series, $I_{OH}(\text{max}) = -0.4$ mA and $I_{IH}(\text{max}) = 20$ μA . Therefore, an LS-TTL series output in the high state can drive up to 20 LS-TTL inputs. For the low-voltage level, $I_{OL}(\text{max}) = 8$ mA and $I_{IL}(\text{max}) = -0.4$ mA. Therefore, an LS-TTL series output can also drive up to 20 LS-TTL inputs in the low state. So, the fan-out is 20. Similarly, we can determine that an LS-TTL series output in the high state can drive up to 20 ALS-TTL inputs, but up to 80 ALS-TTL inputs in the low state. When the two numbers differ, as they do here, the smaller of the two numbers must be used to ensure proper operation. In other words, an LS-TTL output can safely drive up to 20 ALS-TTL inputs.

Noise Margin

In the operation of a digital circuit, “noise” voltages often occur. These are nonsignal voltage pulses or spikes caused by electrical disturbances such as lightning, automobile ignitions, or sudden changes in supply voltage levels. It is, of course, not desirable for noise voltages to affect the circuit operation.

Noise margin is a measure of the ability of a digital IC to withstand these noise voltages. Noise margin is defined as follows:

$$\text{High-level noise margin} = V_{OH(\min)} - V_{IH(\min)}$$

$$\text{Low-level noise margin} = V_{IL(\max)} - V_{OL(\max)}$$

For the LS-TTL series, for example, the high-level noise margin is $2.7 - 2.0 = 0.7$ V. So, in the worst case, a high-level signal can drop 0.7 V in going from an LS-TTL output to an LS-TTL input and still be recognized as a high level. For the LS-TTL series, the low-level noise margin is $0.8 - 0.5 = 0.3$ V.

4.8.3 Parameters for Switching Characteristics

For combinational circuit elements, the most important switching (ac) parameters are the *propagation delays*. A signal takes a finite amount of time in propagating through a circuit element from an input to an output. This amount of time is the propagation delay. There are two types of propagation delay:

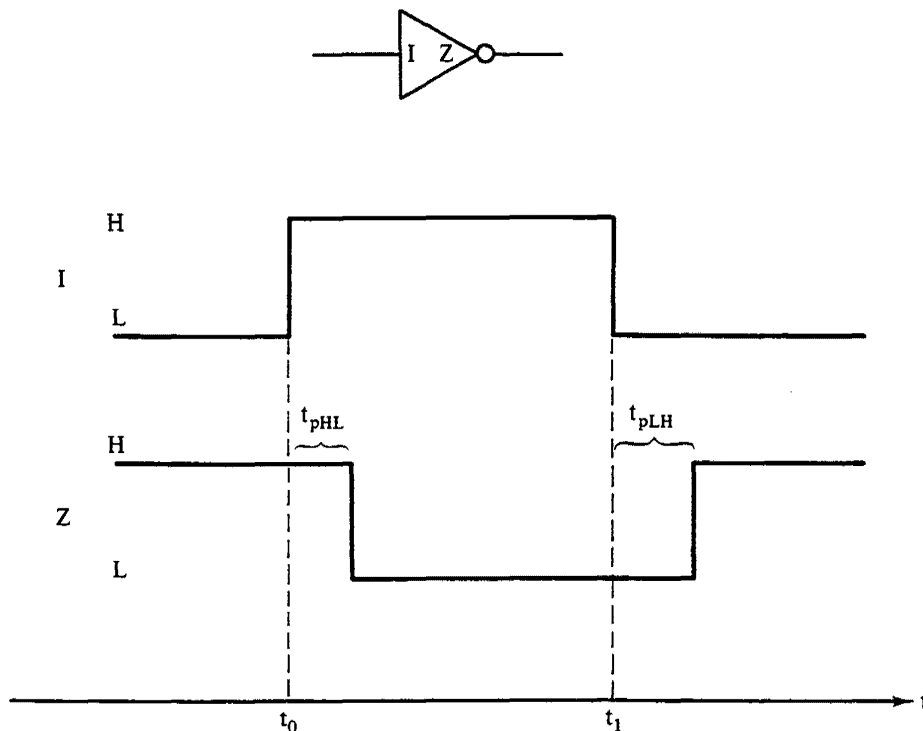


Figure 4.24 Propagation delays.

- t_{pHL} The delay from the time the input changes to the time the output switches from H to L.
- t_{pLH} The delay from the time the input changes to the time the output switches from L to H.

The propagation delays t_{pHL} and t_{pLH} of an inverter are illustrated by the *timing diagrams* of Fig. 4.24, which are plots of the voltage levels versus time. In Fig. 4.24, the voltage level at input I changes from L to H at time t_0 . But, the output Z does not change from H to L until the time $t_0 + t_{pHL}$. Similarly, the voltage level at I changes from H to L at t_1 , but the output Z does not change from L to H until $t_1 + t_{pLH}$. Timing diagrams are very useful in the analysis of digital circuits, and will be encountered frequently throughout the remainder of this text.

SUPPLEMENTARY READING (see Bibliography)

[Bartee 85], [Blakeslee 79], [Fletcher 80], [Hill 81], [Mano 79], [McCluskey 75], [Motorola], [Peatman 80], [Prosser 87], [Texas Instruments]

PROBLEMS

- 4.1. The active-high view and the active-low view for the 74'283 adder are shown in Figs. 4.4(b) and 4.4(c), respectively. Using the voltage table given in Fig. 4.4(a), prove that both of these views perform the binary addition function.
- 4.2. Prove that the 74'283 adder performs the binary subtraction function for the voltage assignment shown in Fig. 4.6(b).
- 4.3. Design a 4-bit subtractor that has an active-high minuend and an active-high subtrahend. Use a 74'283 plus any additional gates that are needed.
- 4.4. If all the inputs are applied simultaneously to the ripple adder shown in Fig. 4.3, how long does it take before the sum and C_4 become valid? Assume that the delay of each gate (within each adder stage) is t_p .
- 4.5. Design the 4-bit BCD adder of Fig. 4.25 using two 74'283s plus any additional gates that are needed. The following examples may be helpful in clarifying the problem specification:

If	A =	B =	$C_{in} =$	then	$C_{out} =$	BCDSUM =
	0101	0011	0		0	1000
	0101	0100	0		0	1001
	0101	0101	0		1	0000
	0101	0110	0		1	0001
	0101	0111	0		1	0010
	0101	1000	0		1	0011
	⋮					
	⋮					
	⋮					
	⋮					
	1001	1001	1		1	1001

Note that A, B, or BCDSUM > 1001 is not allowed in the BCD notation.

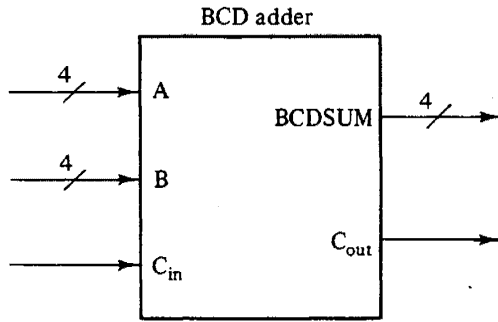


Figure 4.25 BCD adder for Problem 4.5.

- 4.6. Using 75'85 comparators, design a 16-bit magnitude comparator.
- 4.7. Design the 4-bit magnitude comparator of Fig. 4.26, using a 74'85 comparator plus any additional gates that are needed. Note that this comparator has three more than the usual number of outputs. These are \leq , \geq , and $\langle \rangle$, which represent less than or equal to, greater than or equal to, and not equal to, respectively.

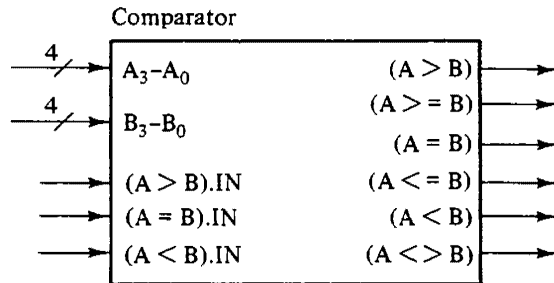


Figure 4.26 Comparator for Problem 4.7.

- 4.8. Design a 5-to-32 decoder using 74'138 decoders and any additional gates that are required.
- 4.9. Given an 8-bit address A_7-A_0 , what are the addresses that will enable the modules M_0 , M_1 , M_2 , and M_3 shown in Fig. 4.27. For convenience, use X for a don't-care address bit.

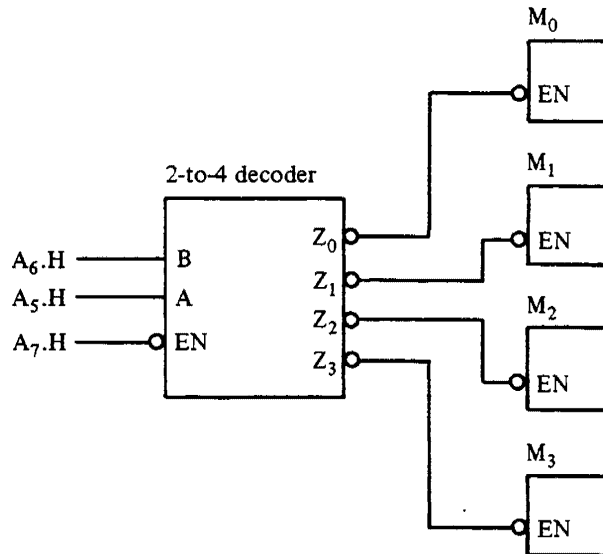


Figure 4.27 Circuit for Problem 4.9.

- 4.10. Using the truth table for a BCD-to-7-segment decoder shown in Fig. 4.11(c), derive the logic equations for the seven outputs a, b, c, d, e, f, and g.

- 4.11. A chain of 74'47 BCD-to-7-segment decoders can be connected together as shown in Fig. 4.12(c) to display leading zeros as blanks. Reconnect the 74'47s in such a way that leading zeros are displayed as zeros.
- 4.12. Design Module M in Fig. 4.28 to obtain a 16-to-4 priority encoder. (*Hint: Module M is a combinational circuit with eight inputs and five outputs. You are to determine the logic equations for the five outputs.*)

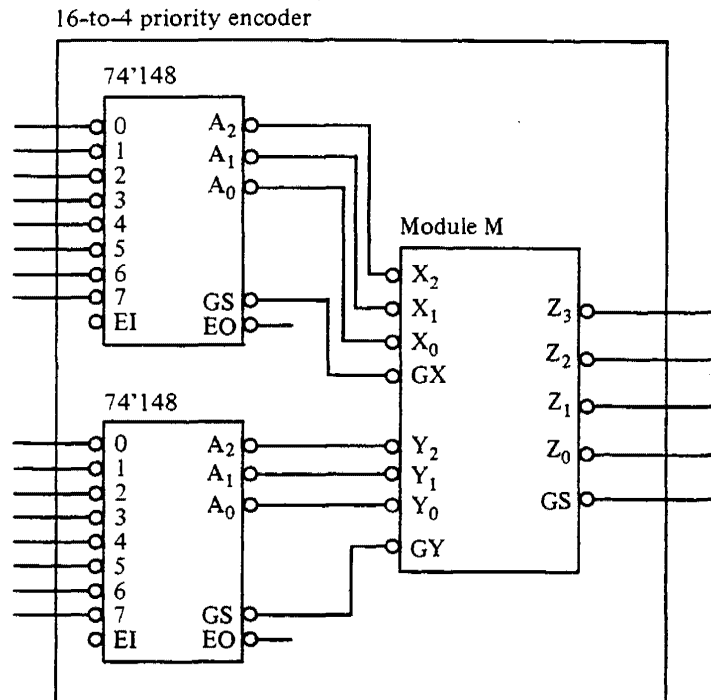
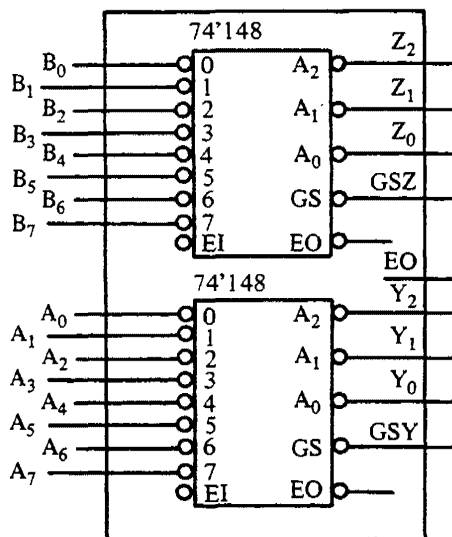


Figure 4.28 Encoder for Problem 4.12.

- 4.13. The enable output EO and enable input EI of a 74'148 can be used to cascade 74'148 priority encoders for easy octal expansion. Such an expansion for the 74'148s of Fig. 4.29(a) is shown in Fig. 4.29(b). Show connections for the 74'148s of Fig. 4.29(a) that will accomplish this expansion. No additional gates are required.



(a)

EI	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅	A ₆	A ₇	B ₀	B ₁	B ₂	B ₃	B ₄	B ₅	B ₆	B ₇	Z ₂	Z ₁	Z ₀	Y ₂	Y ₁	Y ₀	GSZ	GSY	E0		
0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0		
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		
1	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0		
1	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0		
1	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0		
1	X	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0		
1	X	X	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	
1	X	X	X	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	
1	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	
1	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	
1	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
1	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	0	1	0	0	0	1	0	0	
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1	0	0	1	0	0	0	1	0	0	

(b)

Figure 4.29 Table and components for Problem 4.13.

- 4.14. (a) Determine the complete truth table for the four-input MUX shown in Fig. 4.16.
(b) Derive the logic equation for Z.
- 4.15. In Fig. 4.20, a 16-input MUX is realized with 4 four-input MUXs and a decoder. But, suppose no decoder is available. Design a 16-input MUX using any number of 4-input MUXs (74'253), but no decoders.
- 4.16. Using a 74'148 priority encoder and any additional gates that are required, design the circuit of Fig. 4.30 for generating the S₀ and S₁ control inputs for the circuit of Fig. 4.18. Each of the requesting devices (A, B, C, and D) can request a connection to device X through the signals REQA, REQB, REQC, and REQD, respectively. If there are competing requests, then the order of priority is as follows: D, C, B, and A, with D having the highest priority. If no request is made, then device X is connected to A by default.

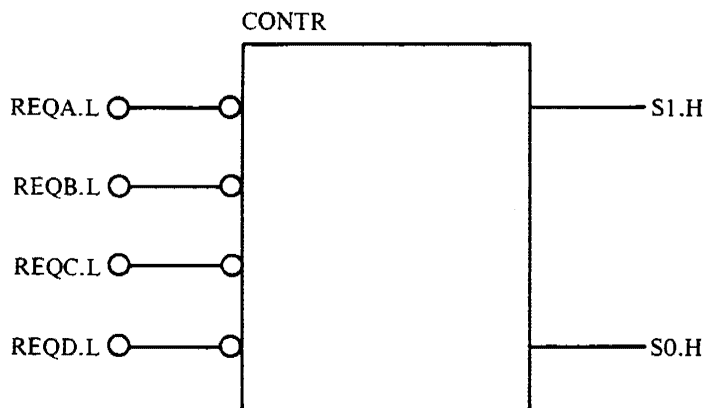


Figure 4.30 Circuit for Problem 4.16.

- 4.17. Design the circuit BIDIR of Fig. 4.31 such that the signal DATA is bidirectional. Specifically, when IOCTR is 1 (H), then DATA is connected to INPUT and the direction of the data flow is "in." But when IOCTR is 0 (L), then DATA is connected to OUTPUT and the direction of the data flow is "out." (Hint: Use 74'125 three-state logic elements.)

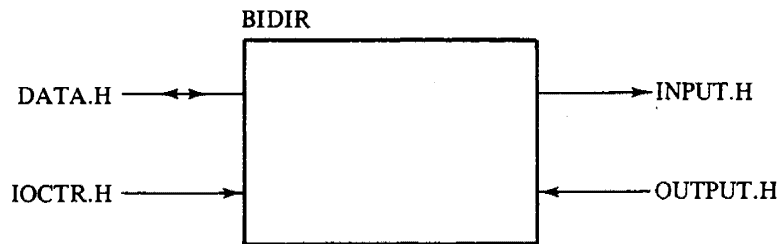


Figure 4.31 Circuit for Problem 4.17.

- 4.18. (a) Discuss the similarities and the differences between a decoder and a demultiplexer.
- (b) An encoder performs the inverse function of a decoder, and a multiplexer performs the inverse function of a demultiplexer. Then, is there any relationship between an encoder and a multiplexer? Explain.
- 4.19. What is the maximum number of standard-TTL inputs that an ALS-TTL output can drive?
- 4.20. If the signal MEMCS shown in Fig. 4.32 activates the CS inputs of a bank of memory chips with the specified characteristics, how many CS inputs can it safely drive?

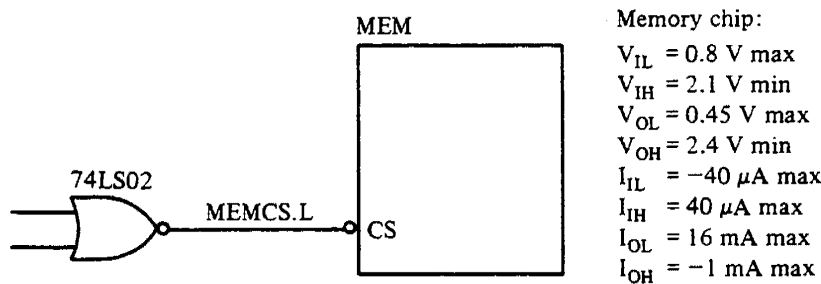


Figure 4.32 Circuit for Problem 4.20.

- 4.21. What is the noise margin for the ALS-TTL series components?
- 4.22. If an ALS-TTL output drives a number of LS-TTL inputs, what is the resultant noise margin?
- 4.23. Given the Exclusive OR gate of Fig. 4.33(a), complete the shown timing diagram for Z in Fig. 4.33(b). Be sure to show and label the propagation delays t_{pHL} and t_{pLH} .

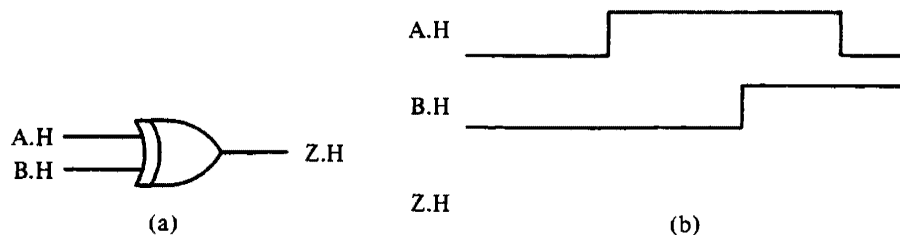


Figure 4.33 Gate and timing diagram for Problem 4.23.

- 4.24. Given the circuit diagram of Fig. 4.34(a), complete the shown timing diagram in Fig. 4.34(b) for the signals \bar{A} and Z. Be sure to show and label the propagation delays t_{pHL} and t_{pLH} .

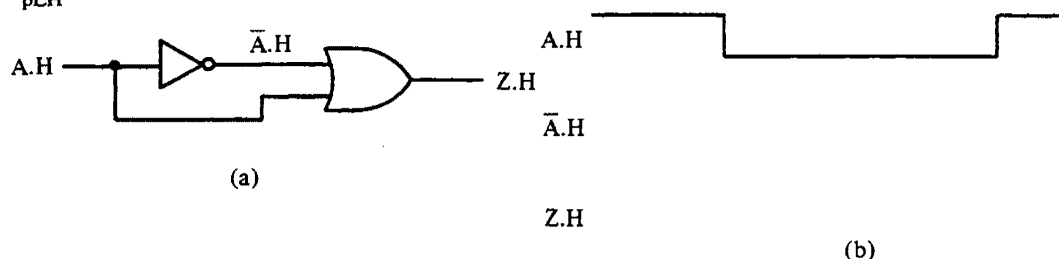


Figure 4.34 Gate and timing diagram for Problem 4.24.

Sequential MSI Circuit Elements

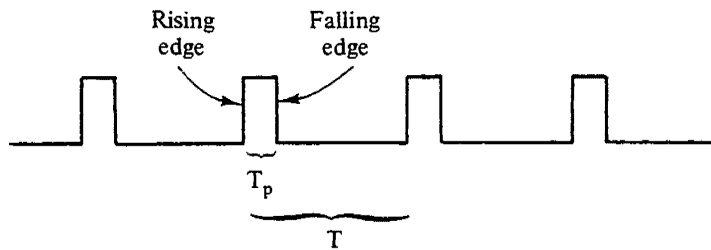
5.1 INTRODUCTION

In Chapter 4 we studied the designs and applications of some combinational MSI circuit elements that realize many of the commonly used logic functions. In the present chapter we will study the designs and applications of *sequential* MSI circuit elements. Unlike combinational circuit element output values, which are functions only of the present input values, the outputs of a sequential circuit element depend on both the present and past input values. In effect, a sequential circuit element has a *memory* in which the effects of past input values can be stored.

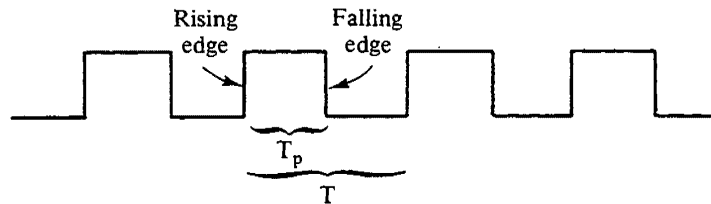
Sequential circuit elements are classified as either *clocked* (synchronous) or *unclocked* (asynchronous). A clocked sequential circuit element responds to a change of input signals only at discrete instants of time, as determined by a *clock* input. Most sequential circuit elements that are available and used in the designs of digital circuits are of the clocked type. On the other hand, an unclocked circuit element is not regulated by a clock input and so can respond to a change in the inputs at any instant of time. The most important examples of unclocked sequential elements are the random access memory (RAM) and the read-only memory (ROM), which are presented in Chapter 6. The circuit elements considered in the present chapter are mainly clocked circuit elements.

5.2 THE CLOCK SIGNAL

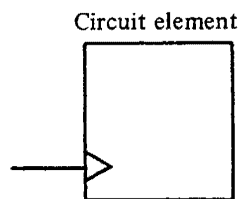
As stated, the *clock* input is the controlling input to a clocked sequential circuit element. A clock input signal is generally a periodic waveform consisting of equally spaced pulses. Two examples of clock signals are shown in the timing diagrams of Figs. 5.1(a) and (b). The *pulse width* of each clock signal is T_p and the *period* is T , as shown. The *duty cycle* of a clock signal is the percentage of the time in which the signal is true, and is equal to $(T_p/T) \times 100$ percent. For example, the clock signal of Fig. 5.1(b) has a 50 percent duty cycle.



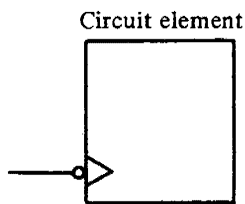
(a) A clock signal with a pulse width of T_p and a period of T



(b) A clock signal with a 50% duty cycle



(c) Symbol for an active-high clock input



(d) Symbol for an active-low clock input

Figure 5.1 Clock signals and clock inputs.

As shown in Figs. 5.1(a) and (b), the *rising edge* of a clock pulse is the positive-going (from low to high) transition, and the *falling edge* of a clock pulse is the negative-going (from high to low) transition. A clocked sequential circuit element responds to the input signals only at the *active edges* of the clock signals. For an *active-high* clock input, the active edge is the rising edge of the clock pulse. In other words, during a clock cycle, a circuit element with an active-high clock input will respond to the values of the inputs only “at the moment” that the clock signal goes from low to high. For the rest of the clock cycle, any change in the input values will have no effect on the state and outputs of the circuit element. For an *active-low* clock input, the active edge is the falling edge of the clock pulse. The symbols for an active-high and active-low clock input are shown in Figs. 5.1(c) and (d), respectively.

Now that we have defined the relevant parameters of a clock signal, we are ready to see how the clock input is used to control the functioning of clocked sequential circuit elements.

5.3 FLIP-FLOPS

Flip-flops are fundamental memory devices that can assume one of two stable states: 0 (false) or 1 (true). Consequently, a flip-flop is capable of storing 1 bit of information. The state of a flip-flop will remain stable (“remembering” a 0 or a 1), as long as there is no change in the inputs. A change in the flip-flop inputs, however, can produce a change in the flip-flop state, causing it to go from the *present state* to the *next state*. The next state of a flip-flop is a function of the flip-flop inputs *and* the present state of the flip-flop. In other words, the same values applied at the flip-flop inputs may produce different next states (and consequently different outputs), depending on whether the present state of the flip-flop is a 0 or a 1. A flip-flop is characterized by the types of inputs and the manner in which the inputs affect the operation. In the following sections, we will study the types of flip-flops that are commonly used in the designs of digital circuits. We will study flip-flops in a top-down manner, considering first the function of each type of flip-flop. Then, after the functions are well understood, we will consider the realization details of the flip-flops. This material is presented in Sec. 5.5.

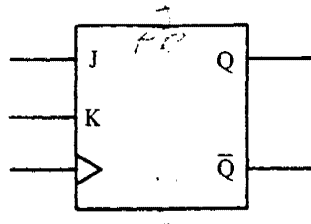
5.3.1 The J-K Flip-Flop

Figure 5.2(a) shows the functional block diagram of a clocked J-K flip-flop. It has two inputs, J and K, in addition to an active-high clock input. It also has two outputs: Q, which represents the state of the flip-flop, and \bar{Q} , which is simply the inverted value of the flip-flop state.

The truth table for a J-K flip-flop, commonly called the *characteristic table*, is shown in Fig. 5.2(b). Observe that Q^+ , which is the next state of the flip-flop, is a function of the flip-flop inputs J and K, and the present state Q of the flip-flop. Note that Q and Q^+ in the truth table represent the value of the *same* flip-flop output Q, but *at different moments in time*. More specifically, let Q represent the value of the output Q at some moment in time, then Q^+ represents the value of the output Q at a time “just after” the *next* active clock edge. For this flip-flop, the active clock edge is the rising edge, as indicated in the truth table by the upward arrow (\uparrow).

The function of the J-K flip-flop can be best understood from a timing diagram, as shown in Fig. 5.2(c). Recall that a timing diagram illustrates the behavior of a device over time. For the timing diagram of Fig. 5.2(c), time is divided into intervals (T 's), each corresponding to a period of the clock signal CLK. Since the J-K flip-flop shown in Fig. 5.2(a) has an active-high clock input, the diagram has, for emphasis, dashed lines at the rising edges of the clock signal.

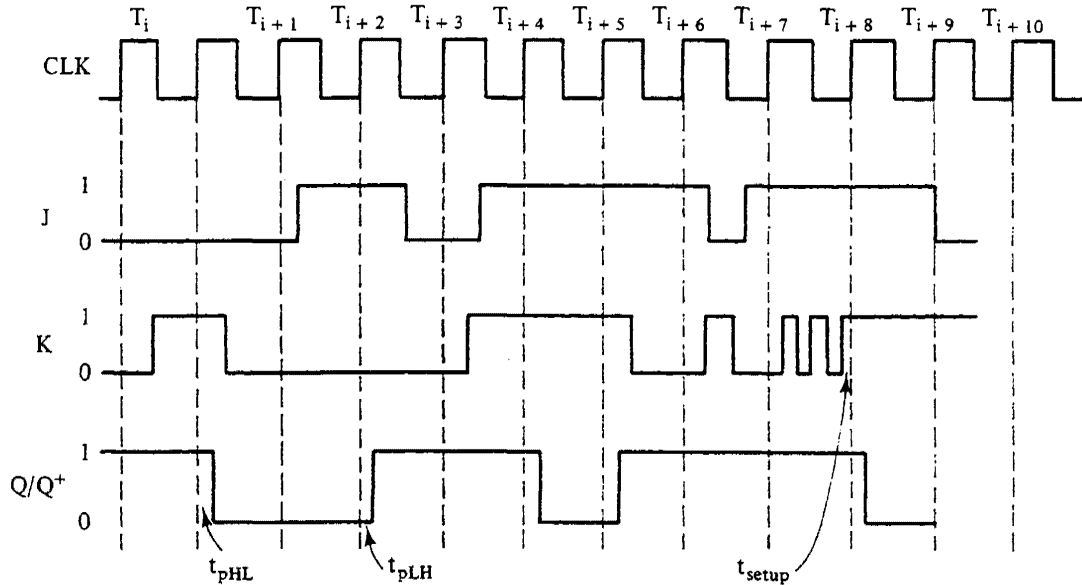
For an explanation of this timing diagram, assume that the present interval is T_i , and that the present state of the J-K flip-flop, represented by Q, is 1, as shown. At the next active clock transition (the dashed line between T_i and T_{i+1}), we see that $J = 0$, $K = 1$, and the present state $Q = 1$. For this condition, the fourth row of the truth table of Fig. 5.2(b) specifies that the next state Q^+ is 0. This is graphically illustrated in the timing diagram by the transition of Q from 1 to 0 “immediately” after the active clock transition between T_i and T_{i+1} . Actually, there is no “immediate” transition of Q, but rather one that occurs later after a time equal to the propagation delay t_{pHL} , as shown.



(a) Functional block diagram

CLK	J	K	Q	Q ⁺
↑	0	0	0	0
↑	0	0	1	1
↑	0	1	0	0
↑	0	1	1	0
↑	1	0	0	1
↑	1	0	1	1
↑	1	1	0	1
↑	1	1	1	0

(b) Characteristic table



(c) Timing for a J-K flip-flop

J	K	Q ⁺	Operation
0	0	Q	hold
0	1	0	clear
1	0	1	set
1	1	\bar{Q}	toggle

(d) Condensed characteristic table

Figure 5.2 The J-K flip-flop.

For the next interval T_{i+1} , the present flip-flop state Q is equal to 0. And, at the next active clock transition, we see that $J = 0$, $K = 0$, and $Q = 0$. For this condition, the first row of the truth table of Fig. 5.2(b) specifies that the next state Q^+ is 0. Again, this is graphically illustrated in the timing diagram by the fact that Q remains 0 after the next active clock transition (the dashed line between T_{i+1} and T_{i+2}).

In this manner, the timing diagram of Fig. 5.2(c) illustrates the effect of the eight possible combinations of inputs and present state of a J-K flip-flop. Note that the flip-flop responds only to the input values at the next active clock transition, and not at any

other time. Consequently, no matter how many times the inputs change during a clock cycle, only the values at the time of the next active clock transition will affect the next state of the flip-flop. This is clearly illustrated in intervals T_{i+7} and T_{i+8} of the timing diagram.

This brings up an interesting point. What happens if the inputs are changed “right at” the active clock transition, as shown in the timing diagram between T_{i+9} and T_{i+10} ? In this case, the input J is ambiguous. It may be “seen” by the flip-flop as either a 0 or a 1. This is, of course, an undesirable situation. Care must be taken to ensure that the desired value is placed at the input at some specified setup time, t_{setup} , before the next active clock edge. This t_{setup} is a switching (ac) parameter that is included in the manufacturer’s data sheet on the sequential circuit element, and is the time prior to an active clock transition that an input must be stable to assure reliable operation. Later in this

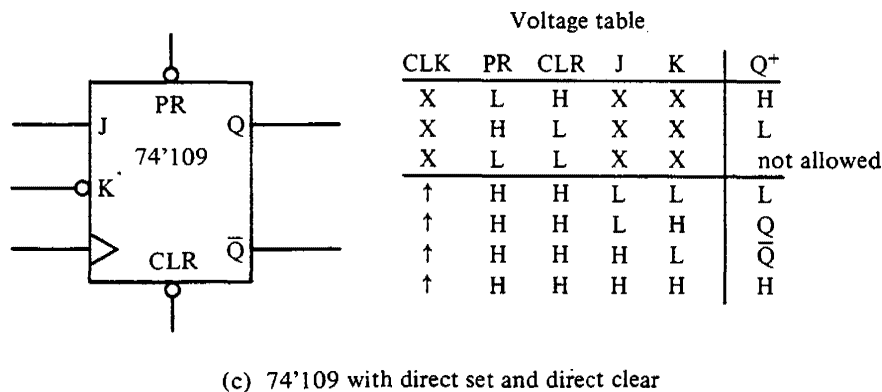
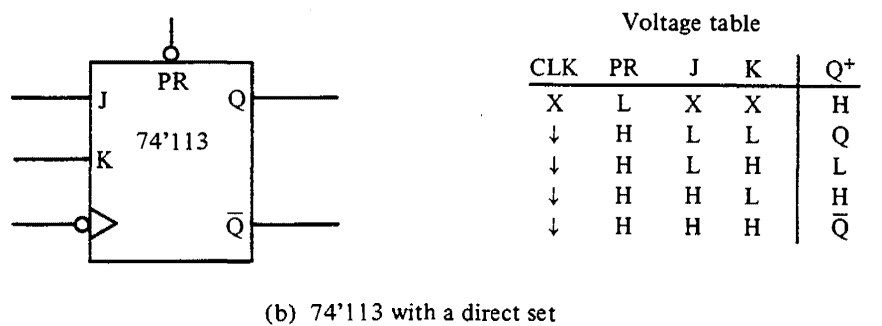
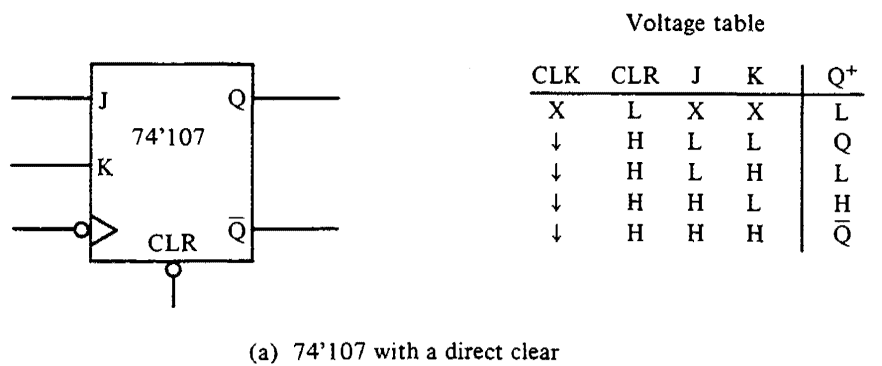


Figure 5.3 Commercially available J-K flip-flops.

book, we will study design methods for ensuring that inputs are synchronized to change and become stable in the early part of the clock cycle so that the setup times of the inputs of the sequential circuit elements are never violated.

The truth table of Fig. 5.2(b) can be condensed into the one shown in Fig. 5.2(d) by specifying the present state as simply Q instead of a 0 or a 1. From this condensed truth table, we can clearly see the four operations that can be performed by a J-K flip-flop. If J and K are both 0 at an active clock transition, then the next state Q^+ is the same as the present state Q . If $J = 0$ and $K = 1$, then the next state is 0, regardless of the value of the present state. On the other hand, if $J = 1$ and $K = 0$, then the next state is 1, regardless of the value of the present state. Finally, if J and K are both 1, then the next state is the complement of the present state, a behavior called *toggling* .

J-K flip-flops are commercially available with a variety of active-high and active-low inputs and outputs. Several common ones are shown in Fig. 5.3, along with their voltage tables. Note that in addition to the normal J and K inputs, these J-K flip-flops have a preset input PR and/or a clear input CLR. Unlike the J and K inputs, these inputs are *asynchronous*, which means that they are not regulated by the clock input. When PR is true (L), then the output Q is “immediately” reset to 1 regardless of the clock input, as indicated by the don’t care in the first row of the voltage table of the 74’113 and 74’109. Similarly, when CLR is true (L), then the output Q is “immediately” cleared to 0. The most interesting J-K flip-flop of Fig. 5.3 is the 74’109, which has an active-high J input, an active-low K input, an active-high clock input, and both an asynchronous preset PR and an asynchronous clear CLR.

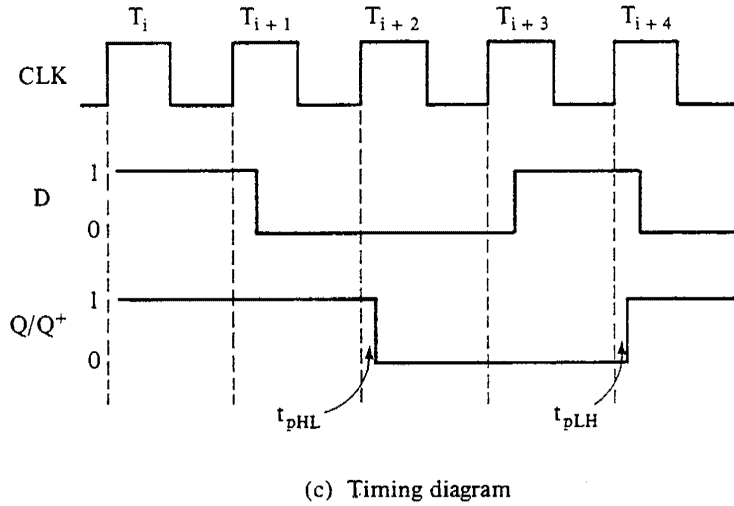
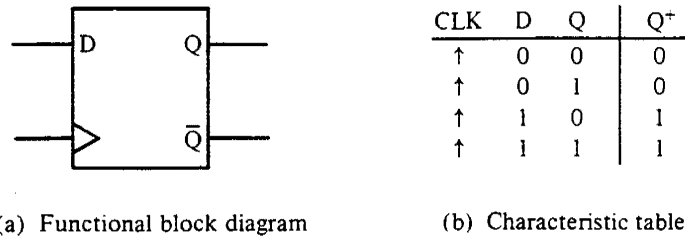
5.3.2 The D Flip-Flop

Another type of commonly used flip-flop is the clocked D flip-flop, also called the delay flip-flop. The functional block diagram for it is shown in Fig. 5.4(a). Observe that it has only one input, D, in addition to the clock input. It also has two outputs, Q and \bar{Q} . The characteristic table for a D flip-flop is shown in Fig. 5.4(b). As illustrated in the timing diagram of Fig. 5.4(c), the operation of the D flip-flop is simpler than that of the J-K flip-flop. In fact, the next state Q^+ is a function of the D input only and, unlike the J-K flip-flop, is independent of the present state Q . If the D input is equal to 1 at an active clock transition, as at T_i/T_{i+1} and T_{i+3}/T_{i+4} , then Q^+ is equal to 1 regardless of the value of the present state Q . Similarly, if the D input is equal to 0 at an active clock transition, as at T_{i+1}/T_{i+2} and T_{i+2}/T_{i+3} , then Q^+ is equal to 0 regardless of the value of the present state Q . Consequently, the characteristic table for a D flip-flop can be condensed to that shown in Fig. 5.4(d).

The D flip-flop is a useful circuit element for storing 1 bit of information. Another common application of the D flip-flop is for delaying the value of a signal for one clock cycle, as is illustrated by the following example.

Example 5.1 Serial Adder

The parallel adder described in Chapter 4 can add two N -bit numbers simultaneously by using N full adders. But, with only one full adder, a serial adder can add two N -bit numbers. The serial adder, however, will require N clock cycles to perform the addition.



CLK	D	Q ⁺
↑	0	0
↑	1	1

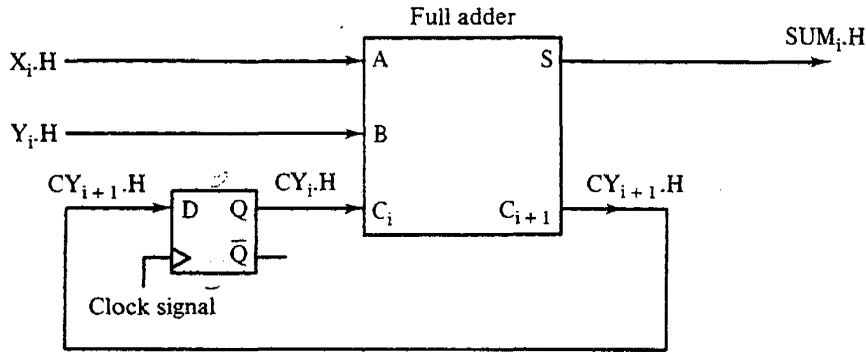
(d) Condensed characteristic table

Figure 5.4 The D flip-flop.

A circuit diagram of a serial adder is shown in Fig. 5.5(a). It consists of a full adder and a D flip-flop. The operation of the serial adder is described below. We will consider this example in some detail since it illustrates some important characteristics of combinational and sequential circuit elements.

In the serial adder of Fig. 5.5(a), the N -bit numbers X and Y are fed in serially, one pair of bits at a time, at inputs A and B , respectively. First, the least significant bits X_0 and Y_0 are fed in, then X_1 and Y_1 , and so forth until finally the most significant bits X_{N-1} and Y_{N-1} are fed in. The carry-in CY_i for each stage of the addition is provided by the output of the D flip-flop. Since the input to the D flip-flop is CY_{i+1} , the carry-in CY_i of the current stage of addition is the CY_{i+1} of the preceding stage of addition. In other words, the D flip-flop delays the value of the carry-out of the preceding stage of addition by one clock cycle so that this carry-out can be used as the carry-in for the current addition stage, as is required for serial addition. Initially, the contents of the D flip-flop should be 0.

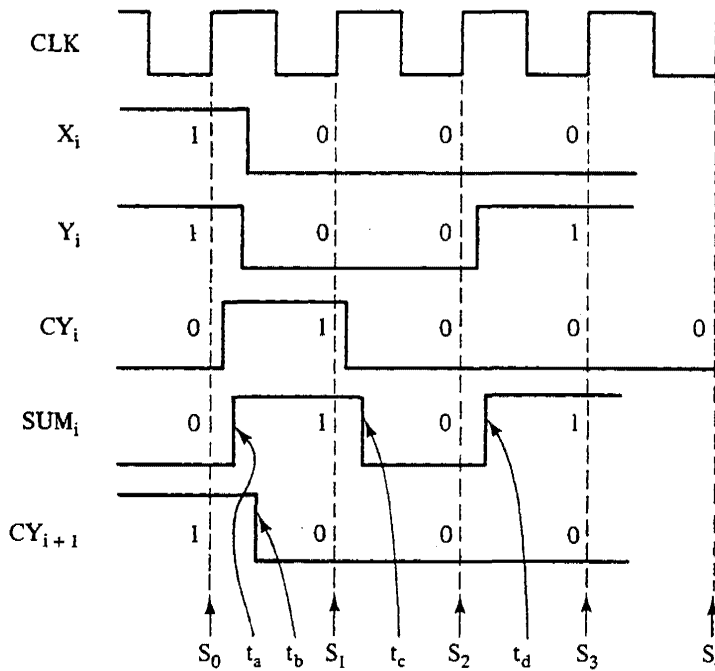
For proper operation, the inputs X and Y must be synchronized with the clock input of the D flip-flop so that a new pair of bits of X and Y is present at each clock cycle. This can be done by storing the bits for X and Y in sequential circuit elements called *shift registers*, and then by shifting them out one at a time to be fed into the adder



(a) Circuit diagram

Stage	S_4	S_3	S_2	S_1	S_0
CY	0	0	0	1	0
X		0	0	0	1
Y	+	1	0	0	1
SUM		1	0	1	0

(b) Sample 4-bit addition



(c) Timing diagram for the 4-bit addition

Figure 5.5 A serial adder.

inputs at each clock cycle. Similarly, the output SUM can be stored by shifting each SUM_i bit into a shift register. The operation of a shift register is explained in Sec. 5.7.2.

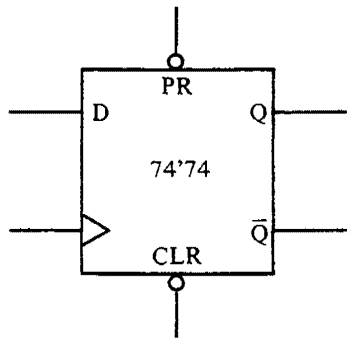
Perhaps the operations of the serial adder can be best understood by tracing through a timing diagram such as the one shown in Fig. 5.5(c). As shown, each stage of addition in the sample 4-bit addition of Fig. 5.5(b) is represented by the values of each of the active clock transitions in Fig. 5.5(c). In other words, the values of the inputs and outputs are valid only at the active clock transitions. In this timing diagram, the active clock transitions are labeled $S_0, S_1, S_2, S_3,$ and S_4 . At S_0 , the least significant bits $X_0, Y_0,$ and

CY_0 are added together to produce SUM_0 and CY_1 . At S_1 , the inputs X_1 , Y_1 , and CY_1 are added together to produce SUM_1 and CY_2 . The process continues until at S_3 , the most significant bits X_3 , Y_3 , and CY_3 are added together to produce SUM_3 and CY_4 .

Note that the D flip-flop is a clocked sequential circuit element. Consequently, its output CY_i changes only at each active clock transition, as can be seen graphically in the timing diagram. The value of CY_{i+1} at each clock transition is loaded into the D flip-flop to be used as CY_i at the next active clock transition. For example, the value of CY_{i+1} is 1 at S_0 , at which time it is loaded into the D flip-flop. It then becomes available at input C_i at time $S_0 + t_{pLH}$, but it will not be used as the new value of CY_i until the next active clock transition S_1 .

The full adder, on the other hand, is a combinational circuit element. Consequently, its outputs respond to the inputs “immediately” after (actually a propagation delay after) the change in the inputs. This can be seen graphically in the timing diagram at t_a where SUM_i responds to the change in CY_i , at t_b where CY_{i+1} responds to the changes in X_i and Y_i , at t_c where SUM_i responds to the change in CY_i , and at t_d where SUM_i responds to the change in Y_i . Due to this behavior, the inputs and outputs of the serial adder are guaranteed to be valid and stable only at the active clock transitions. For example, between S_0 and S_1 , the inputs and outputs are changing at various times. In fact, at time t_a , the values are $X_i = 1$, $Y_i = 1$, $CY_i = 1$, $SUM_i = 1$, and $CY_{i+1} = 1$, which do not correspond to any of the addition stages for this example. However, all the inputs and outputs have become stable by the time t_b , which is well before the required setup time for the next active clock transition at S_1 . At this time, the values of the inputs and outputs correspond to Stage 1 of this addition example. ■ ■

D flip-flops are commercially available in a form such as the 74'74 D flip-flop shown in Fig. 5.6. Like some J-K flip-flops, this D flip-flop also has a preset input PR and a clear input CLR. Again, these inputs are asynchronous, and so are not regulated by the clock input. Consequently, they can be used to “immediately” set or clear the state of the flip-flop during any part of the clock cycle, if either is required.



(a) 74'74 flip-flop

CLK	PR	CLR	D	Q ⁺
X	L	H	X	H
X	H	L	X	L
↑	H	H	L	L
↑	H	H	H	H

(b) Voltage table

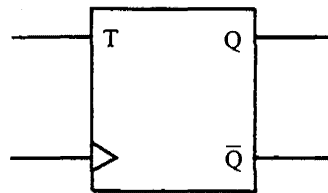
Figure 5.6 Commercially available D flip-flop.

5.3.3 The T Flip-Flop

A less common, but still useful, type of flip-flop is the clocked T flip-flop, also called the toggle flip-flop. The functional block diagram for it is shown in Fig. 5.7(a). This flip-flop has one input T, in addition to the clock input. It also has two outputs Q and \bar{Q} . The characteristic table for the T flip-flop is shown in Fig. 5.7(b). A condensed version of this table is shown in Fig. 5.7(c).

The function of the T flip-flop is quite simple. If the T input is false (0), then at the next active clock transition nothing happens. So the previous state of the flip-flop is retained. But if the T input is true (1), then at the next active clock transition the output of the flip-flop is complemented. Note that the T flip-flop performs the “hold” and “toggle” functions of the J-K flip-flop.

T flip-flops are not commercially available in IC form since they can be easily derived from other commercially available flip-flops. For example, a T flip-flop can be obtained from a J-K flip-flop by simply connecting the J and K inputs together for the T input, as shown in Fig. 5.7(d).



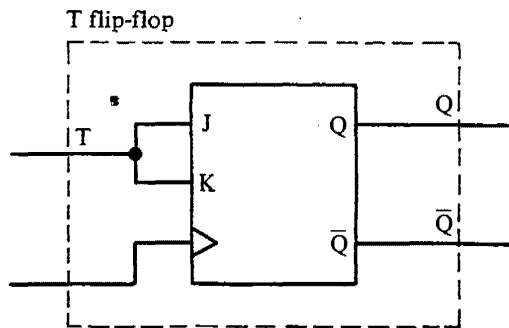
(a) Functional block diagram

CLK	T	Q	Q ⁺
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0

(b) Characteristic table

CLK	T	Q ⁺
↑	0	Q
↑	1	\bar{Q}

(c) Condensed characteristic table



(d) A T flip-flop from a J-K flip-flop

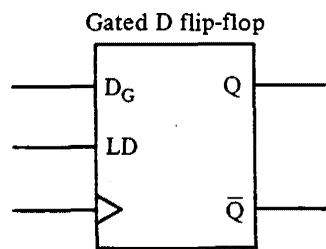
Figure 5.7 The T flip-flop.

5.3.4 Flip-Flop Conversion

As seen from the preceding section, the conversion from a J-K flip-flop to a T flip-flop can be done “intuitively” since the conversion is so simple. When the conversion becomes more complex, however, a systematic procedure is required.

Example 5.2 A Gated D Flip-Flop Using a J-K Flip-Flop

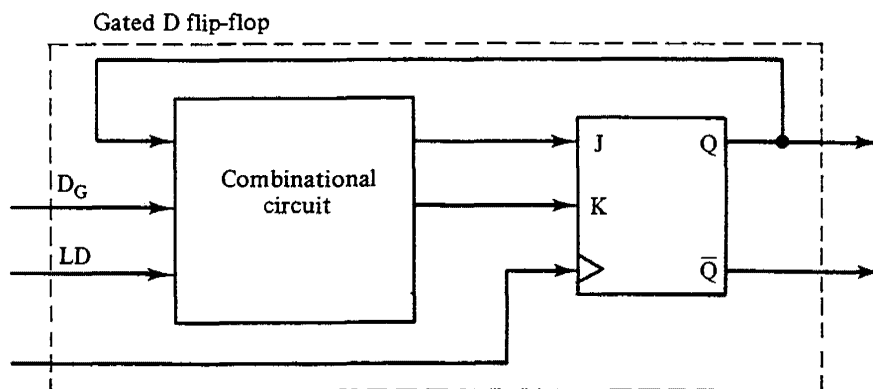
In this example we will design a modified D flip-flop that has a controlling load input LD. The functional block diagram and the truth table for it are shown in Figs. 5.8(a) and (b), respectively. This flip-flop differs from the ordinary D flip-flop which, at each active clock transition, loads in the value at its D input indiscriminantly. With this modified D flip-flop, however, the LD input can be used to load in the value of the D



(a) Functional block diagram

CLK	LD	D_G	Q	Q^+
↑	0	0	0	0
↑	0	0	1	1
↑	0	1	0	0
↑	0	1	1	1
↑	1	0	0	0
↑	1	0	1	0
↑	1	1	0	1
↑	1	1	1	1

(b) Truth table



(c) Block diagram of the flip-flop conversion

Figure 5.8 Gated D flip-flop of Example 5.2.

input only at selected active clock transitions. Specifically, if the LD input is false (0), then at the next active clock transition, nothing happens. The previous state of the flip-flop is retained. But if the LD input is true (1), then at the next active clock transition, the gated D flip-flop loads in the value at the D_G input.

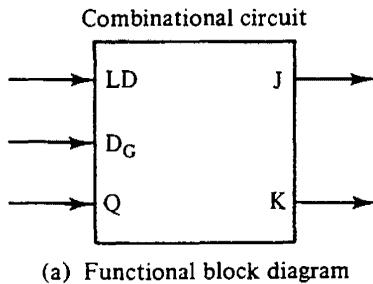
In this design our approach will be to convert a J-K flip-flop into a gated D flip-flop, as shown in the block diagram of Fig. 5.8(c). With the flip-flop selected, the problem reduces to the design of the combinational circuit that will transform the inputs D_G , LD, and Q, the present flip-flop state, into the corresponding J and K inputs for the J-K flip-flop so that the correct next state is outputted at the next active clock transition. A systematic procedure will now be outlined to facilitate this conversion process.

Step 1: Determine the functional block diagram of the combinational circuit. For our example, this functional block diagram is shown in Fig. 5.9(a). The inputs to it are LD, D_G , and the present state Q, which is fed back from the output of the J-K flip-flop. The outputs of the combinational circuit are J and K, which correspond to the J and K inputs of the J-K flip-flop.

Step 2: Determine the truth table for the combinational circuit. In other words, for our circuit we need to complete the truth table shown in Fig. 5.9(b). This step is divided into two substeps.

Step 2(a): Transform the characteristic table of the source flip-flop into its *excitation table*. In this case, the source flip-flop to be converted is a J-K flip-flop, the characteristic table for which is shown in Fig. 5.9(c). This characteristic table specifies the value of the next-state output Q^+ as a function of the inputs J, K, and the present state Q. On the other hand, the excitation table, also shown in Fig. 5.9(c), specifies the values that the inputs J and K must be for the output to change from the specified present state Q to the specified next state Q^+ . For example, the first row of the excitation table specifies that for the J-K flip-flop to change from the present state Q of 0 to the next state Q^+ of 0, the value of the J input must be 0, but the value of the K input can be either 0 or 1 (i.e., a don't care). This information is obtained from rows 1 and 3 of the original characteristic table. Similarly, the second row of the excitation table specifies that for the J-K flip-flop to change from a present state of 0 to a next state of 1, the value of the J input must be 1, but the value of the K input can be either 0 or 1. This information is obtained from rows 5 and 7 of the characteristic table. The remainder of the excitation table can be determined in the same manner. This excitation table for a J-K flip-flop will be used in the next step to obtain the truth table for the combinational circuit.

Step 2(b): Use the excitation table for the source flip-flop to determine the output values for the truth table of the combinational circuit. For our circuit, the truth table of the desired gated D flip-flop is shown on the left of Fig. 5.9(d). The first row of that truth table specifies that for inputs LD = 0, D_G = 0, and a present Q = 0, the next state Q^+ is to be 0. Now consider the source J-K flip-flop. For it to change from Q = 0 to Q^+ = 0, J must be 0 but K can be a don't care at the next active clock transition, as specified by the J-K excitation table. In other words, looking at Fig. 5.8(c), if LD = 0, D_G = 0, and Q = 0, then for Q^+ to be 0, the combinational circuit must be designed such that it generates J = 0 and K = 0 or 1. Similarly, for row 2 of the truth table of Fig. 5.9(d), under the condition of LD = 0, D_G = 0, and Q = 1, the quantity J can be a don't care but K must be a 0 to obtain Q^+ = 1. Continuing in this manner we can determine the rest of the values for J and K. Note, as shown by the arrow in



(a) Functional block diagram

LD	D _G	Q	J	K
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

(b) Truth table to be determined

J	K	Q	Q ⁺
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Q	Q ⁺	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(c) Converting the characteristic table of the J-K flip-flop into the corresponding excitation table

LD	D _G	Q	Q ⁺	J	K
0	0	0	0	0	X
0	0	1	1	X	0
0	1	0	0	0	X
0	1	1	1	X	0
1	0	0	0	0	X
1	0	1	0	X	1
1	1	0	1	1	X
1	1	1	1	X	0

(d) Determination of the J and K values for the combinational circuit

LD	D _G	Q	J	K
0	0	0	0	X
0	0	1	X	0
0	1	0	0	X
0	1	1	X	0
1	0	0	0	X
1	0	1	X	1
1	1	0	1	X
1	1	1	X	0

(e) Final truth table for the combinational circuit

Figure 5.9 Design and realization of the gated D flip-flop using a J-K flip-flop.

Fig. 5.9(d), that the values of J and K for each row are determined by the Q to Q⁺ transition of that row, based on the information specified in the J-K excitation table of Fig. 5.9(c). Since the J and K inputs of the J-K flip-flop are the J and K outputs of the combinational circuit, the truth table for the combinational circuit can be determined, as shown in Fig. 5.9(e).

Step 3: Realize the combinational circuit. Once the truth table for the combinational circuit is determined from step 2, the realization of this circuit is straightforward using the techniques presented in Chapters 2 and 3. The resultant circuit diagram is shown in

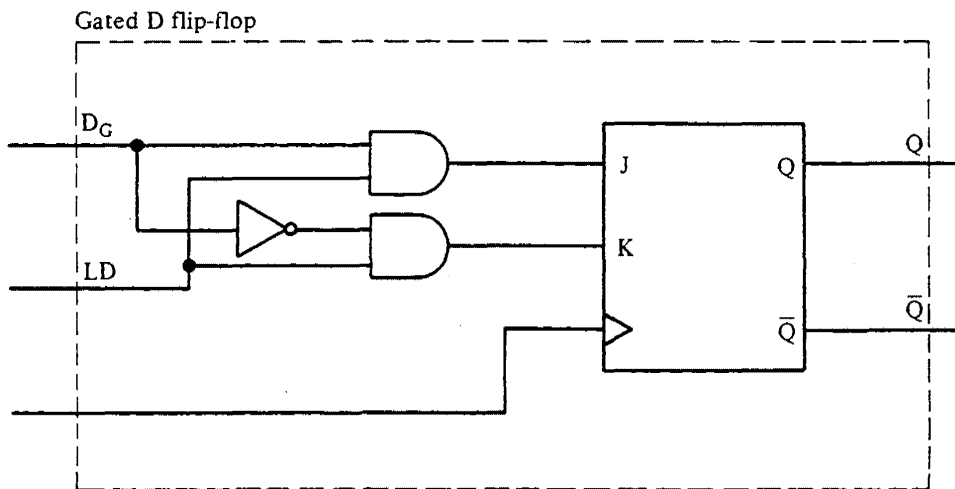
		LD	
		0	1
D _G	Q		
0	0	0	0
0	1	X	X
1	1	X	X
1	0	0	1

$J = LD \cdot D_G$

		LD	
		0	1
D _G	Q		
0	0	X	X
0	1	0	1
1	1	0	0
1	0	X	X

$K = LD \cdot \overline{D_G}$

(f) K-maps for the J and K outputs



(g) Final circuit diagram

Figure 5.9 (cont.)

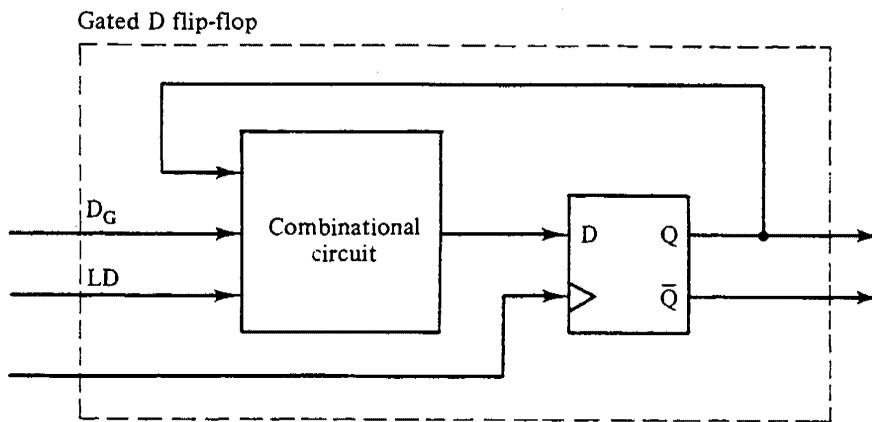
Fig. 5.9(g). Verifying its operation, we see that if LD is 0, then both AND gates are disabled and the J and K inputs are 0. Consequently, the gated D flip-flop will retain its present state. If, however, LD is 1, then at the next active clock transition, the gated D flip-flop will load in the value of D_G . ■ ■

Example 5.3 A Gated D Flip-Flop Using a D Flip-Flop

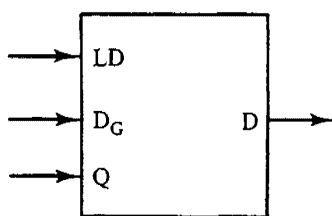
In this example, we will redesign the gated D flip-flop, using an ordinary D flip-flop. The procedure outlined in Example 5.2 still applies. Step 2(a), however, will be different since the source flip-flop is now a D flip-flop.

Step 1: Determine the functional block diagram of the combinational circuit. The functional block diagrams of the flip-flop conversion and of the combinational circuit are shown in Figs. 5.10(a) and (b), respectively. Observe that the inputs to the combinational circuit are LD, D_G , and the present state Q, just as before. The output, however, is now D, corresponding to the D input of the D flip-flop.

Step 2: Determine the truth table for the combinational circuit.



(a) Block diagram of the flip-flop conversion



(b) Functional block diagram

LD	D _G	Q	D
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

(c) Truth table to be determined


Characteristic table			Excitation table		
D	Q	Q ⁺	Q	Q ⁺	D
0	0	0	0	0	0
0	1	0	0	1	1
1	0	1	1	0	0
1	1	1	1	1	1

(d) Determining the excitation table for a D flip-flop

Figure 5.10 Design and realization of the gated D flip-flop using a D flip-flop.

Step 2(a): Transform the characteristic table of the source flip-flop into its excitation table. For our source D flip-flop, the transformation process, which is shown in Fig. 5.10(d), is analogous to that for the J-K flip-flop in Example 5.2. For example, row 2 of the excitation table specifies that for the D flip-flop to change from the present state (Q) of 0 to the next state (Q⁺) of 1, the value of the D input must be 1 at the next active clock transition. This information is obtained from row 3 of the characteristic table.

Step 2(b): Use the excitation table for the source flip-flop to determine the output values for the truth table of the combinational circuit. The process, which is shown in Fig. 5.10(e), is again analogous to that for the J-K flip-flop of Example 5.2. For example, in row 2 of the truth table in Fig. 5.10(e), under the condition of LD = 0, D_G = 0,



LD	D _G	Q	Q ⁺	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

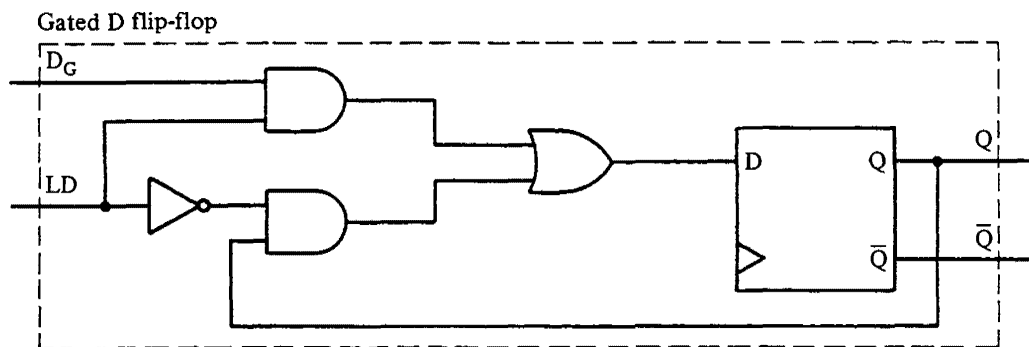
(e) Determination of the D values for the combinational circuit

LD	D _G	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(f) Final truth table for the combinational circuit

		LD		D = $\overline{LD} \cdot Q + LD \cdot D_G$
		0	1	
D _G	Q			
	0	0	0	
	0	1	0	
	1	1	1	1
1	0	0	1	

(g) K-map for the D output



(h) Circuit diagram

Figure 5.10 (cont.)

and $Q = 1$, the D input must be a 1 to obtain $Q^+ = 1$ at the next active clock transition. Again note, as shown by the arrow in Fig. 5.10(e), that the value of D for each row is determined by the Q to Q^+ transition of that row, based on the information given in the D flip-flop excitation table of Fig. 5.10(d). The final truth table for the combinational circuit is shown in Fig. 5.10(f).

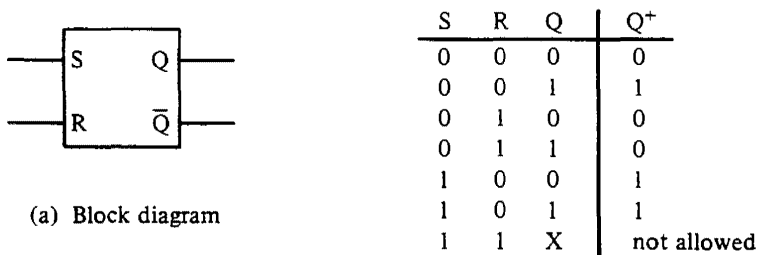
Step 3: Realize the combinational circuit. The realization of the combinational circuit is shown in Figs. 5.10(g) and (h). Verifying the operation of the circuit, we see

that if LD is 0, then the top AND gate is disabled. However, the bottom AND gate is activated, thereby enabling the present state Q to be passed through and loaded back into the flip-flop at the next active clock transition. If LD is 1, though, the bottom AND gate is disabled and the top AND gate is activated, thereby enabling the new value at D to be loaded into the flip-flop at the next active clock transition. ■ ■

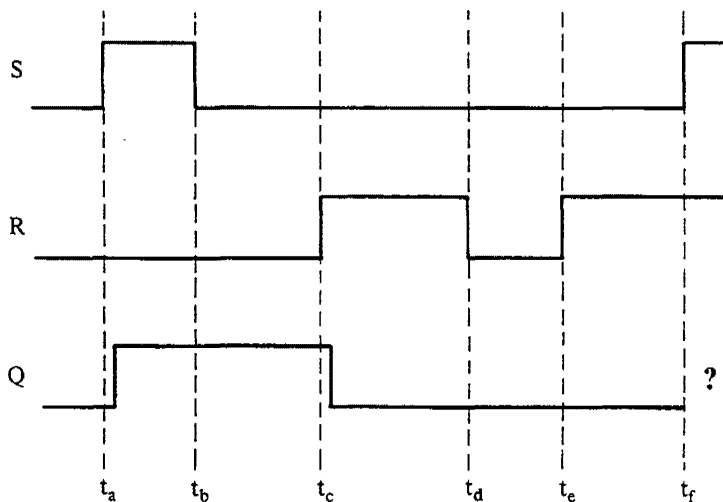
In summary, the procedure outlined in this section can be used to convert any type of flip-flop to any other type of flip-flop. In fact, this procedure can be modified to be used for the design of more complex sequential circuits, as we will see later in this chapter.

5.4 THE UNCLOCKED S-R FLIP-FLOP

The functional block diagram for an *unclocked* S-R flip-flop is shown in Fig. 5.11(a). It has two inputs S (set) and R (reset), but has no clock input. The S-R flip-flop also has two outputs Q and \bar{Q} . As shown in the truth table in Fig. 5.11(b), the operation of an S-R flip-flop is similar to that of a J-K flip-flop, except that both inputs of 1 (S = R =



(b) Truth table



(c) Timing diagram

Figure 5.11 The S-R flip-flop.

1) are not allowed. Furthermore, unlike the clocked flip-flops that have been presented, the unclocked S-R flip-flop is not controlled by a clock input, but rather responds asynchronously to changes in the inputs.

The operation of the S-R flip-flop is illustrated in the timing diagram of Fig. 5.11(c). Note the asynchronous nature of this flip-flop. The transition to the next state Q^+ occurs “immediately” after a change in one of the inputs, and is not synchronized by an active transition of a clock input. For example, at $t = t_a$, the input S change from 0 to 1 causes Q, at $t = t_a + t_{pLH}$, to change from 0 to 1. The t_{pLH} here is the low-to-high propagation delay of the S-R flip-flop. Also note that at $t = t_f$, S and R are both 1, which is not allowed for an S-R flip-flop. We will see the reason for this in Sec. 5.5, when the realization details of flip-flops are discussed.

Example 5.4 Using an S-R Flip-Flop for Switch Debouncing

A common application of the S-R flip-flop is for the debouncing of a mechanical switch. When a mechanical switch is first closed, it does not make contact cleanly, but rather makes and breaks contact several times before remaining closed. Similarly, when a mechanical switch is first opened, it will “bounce” several times before remaining open.

The problem with switch bouncing is illustrated by the waveform of the switch signal SW shown in Fig. 5.12(a). If this signal is used as an input to other circuit

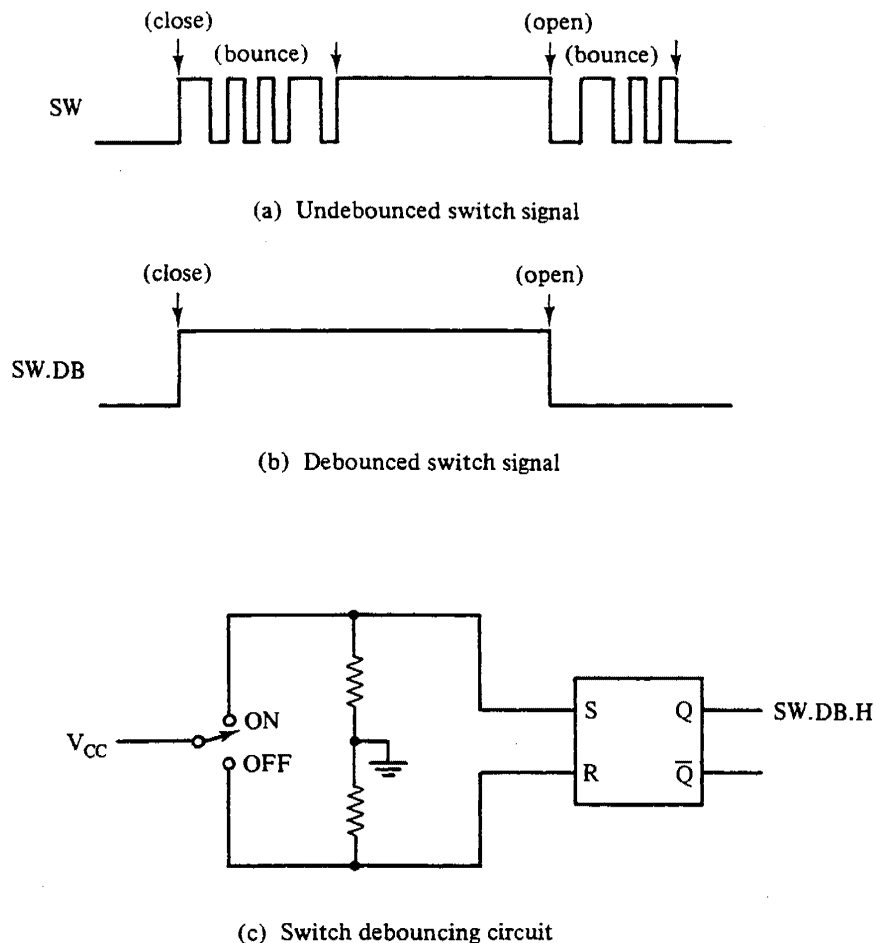


Figure 5.12 Switch debouncing using an S-R flip-flop.

elements, undesirable circuit behavior may result because each bounce may be interpreted as a signal change. What is desired is a debounced switch signal, as shown in Fig. 5.12(b), that has a single low-to-high transition for each switch closure, and a single high-to-low transition for each switch opening.

Figure 5.12(c) shows the use of an S-R flip-flop in a switch debouncing circuit for a single-pole, double-throw switch. For an understanding of the operation let us assume that initially the switch arm is between OFF and ON and that SW.DB is low. Then, when the switch arm first makes contact with the ON terminal, the S input of the S-R flip-flop becomes 1, causing SW.DB to make a low-to-high transition. During the time that the switch bounces after this initial closing, the switch arm makes and breaks contact with the ON terminal. Each time the contact is broken, the values at the S and R inputs of the S-R flip-flop are both 0. And, each time the contact is made, the values are $S = 1$ and $R = 0$. In either case, SW.DB remains 1. Consequently, there is only a single low-to-high transition of SW.DB, as shown.

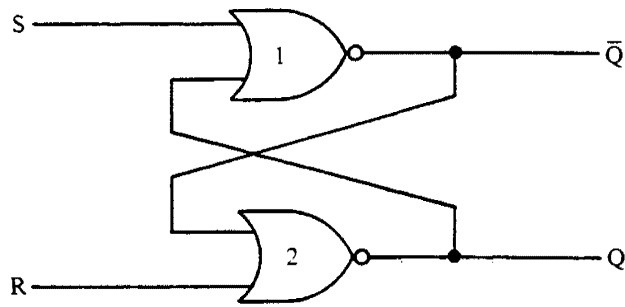
Now if the switch arm is moved from the ON terminal to the OFF terminal, the value of SW.DB remains 1 until the switch arm first makes contact with the OFF terminal. At that moment, the inputs to the S-R flip-flop become $R = 1$ and $S = 0$, with the result that SW.DB makes a high-to-low transition. Again, the switch bounces, thereby causing the value of R to alternate between 0 and 1 while the value of S stays at 0. In either case, SW.DB remains 0. Consequently, there is only one high-to-low transition of SW.DB, as shown. ■ ■

5.5 REALIZATION OF FLIP-FLOPS

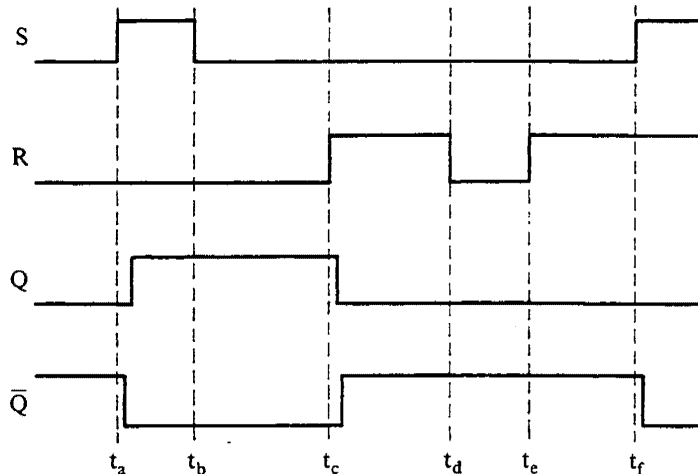
Our primary concern with flip-flops is the *use* of them as circuit elements in the design of digital systems. It is nevertheless important to have some insight into the internal structures of these devices. Consequently, we will now digress to study some realization details of flip-flops. We do this now rather than at the beginning of our consideration of flip-flops because of the top-down manner of our study. In the preceding sections we learned how the common types of flip-flop function. Now that we understand this, we can better appreciate the realization details of flip-flops.

The unlocked S-R flip-flop of the preceding section is a basic flip-flop from which other flip-flops can be derived. So, we will consider it first. A realization of the S-R flip-flop is shown in Fig. 5.13(a). It is a simple logic circuit consisting of two interconnected NOR gates. The feature of this circuit that distinguishes it from the combinational circuits presented in Chapter 4 is the *feedback* connections from Q to NOR gate 1 and from \bar{Q} to NOR gate 2. It is these feedback paths that provide the memory property, allowing the flip-flop to assume one of two stable states: 0 (false) and 1 (true).

The operation of this flip-flop circuit is illustrated by the timing diagram shown in Fig. 5.13(b). As a starting point, let us assume that initially (before $t = t_a$), the flip-flop circuit is in a stable state with the following values: $S = 0$, $R = 0$, $Q = 0$, and $\bar{Q} = 1$. At $t = t_a$, though, S changes from 0 to 1. Consequently, the output of NOR gate 1, \bar{Q} , goes from 1 to 0, as shown in the timing diagram at $t_a + t_{pHL}$. And since \bar{Q} is fed back to the input of NOR gate 2, it causes Q to change from 0 to 1 at $t_a + t_{pHL} + t_{pLH}$. Similarly, Q is fed back to the input of NOR gate 1. Since Q is now 1 and S is 1, then



(a) Circuit diagram



(b) Timing diagram

Figure 5.13 Realization of an S-R flip-flop.

\bar{Q} should be 0. However, \bar{Q} is already 0. Therefore, the circuit has become stable at $t_a + t_{pHL} + t_{pLH}$, with the following values: $S = 1$, $R = 0$, $Q = 1$, and $\bar{Q} = 0$.

The state of the flip-flop circuit remains in that stable state until time t_b , when S changes from 1 to 0. Although the S input to NOR gate 1 goes to 0, there is still a 1 input from Q . Consequently, the output \bar{Q} of NOR gate 1 remains 0, and again the flip-flop circuit is in a stable state with the following values: $S = 0$, $R = 0$, $Q = 1$, and $\bar{Q} = 0$. Note that even though the S input has returned to 0, the flip-flop “remembers” the last set command by remaining in a $Q = 1$ state.

At time $t = t_c$, the input R changes from 0 to 1, causing the output Q of NOR gate 2 to change from 1 to 0 at $t_c + t_{pHL}$. Since Q was providing the only 1 input to NOR gate 1, this change in Q causes \bar{Q} to change from 0 to 1 at $t_c + t_{pHL} + t_{pLH}$. This \bar{Q} is fed back to NOR gate 2. However, the circuit is already stable with the following values: $S = 0$, $R = 1$, $Q = 0$, and $\bar{Q} = 1$.

At $t = t_d$ and $t = t_e$, the changes at the R input have no effect on the state of the flip-flop circuit, and the outputs Q and \bar{Q} remain at 0 and 1, respectively. At $t = t_f$, however, S changes from 0 to 1 to make both the S and R inputs equal to 1. Then, \bar{Q} becomes 0, but Q also remains 0, and the flip-flop circuit is in a stable state with $S = 1$, $R = 1$, $Q = 0$, and $\bar{Q} = 0$. Since it is not meaningful to have both Q and \bar{Q}

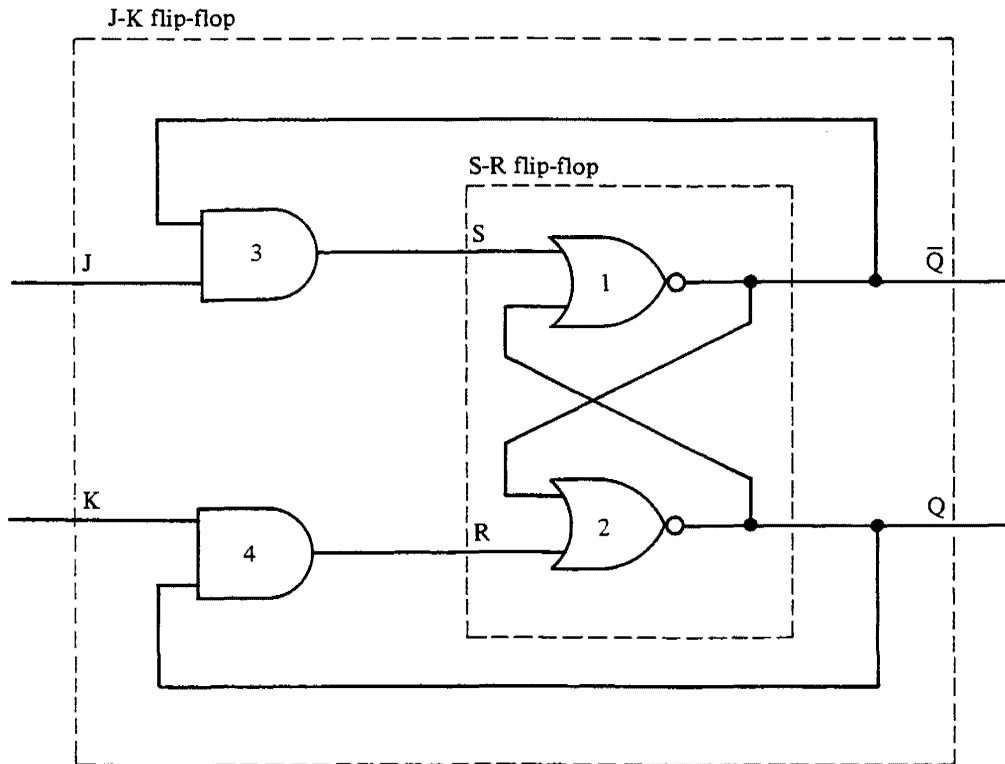


Figure 5.14 Unlocked J-K flip-flop.

equal to 0, this combination of inputs ($S = R = 1$) is not allowed for the unlocked S-R flip-flop.

The circuit for the unlocked S-R flip-flop has been considered in some detail here because it is the basic flip-flop circuit upon which the more complex flip-flops are built.

5.5.1 J-K Flip-Flops

The flip-flop circuit of Fig. 5.13(a) can be modified to allow both flip-flop inputs to be equal to 1. The result is the unlocked J-K flip-flop circuit shown in Fig. 5.14. As shown, two AND gates are added along with feedback connections from \bar{Q} to AND gate 3 and from Q to AND gate 4. As can be verified with an analysis similar to that of the last section, these additions change only the operation for two 1 inputs. Specifically, for $J = 1$ and $K = 1$ applied, the output of either AND gate 3 or AND gate 4 becomes 1, the particular one depending on the values of Q and \bar{Q} . If $Q = 1$ (and $\bar{Q} = 0$), then the output of AND gate 4 becomes 1 and so the next state Q^+ becomes 0. But, if $\bar{Q} = 1$ (and $Q = 0$), then the output of AND gate 3 becomes 1 and so the next state Q^+ becomes 1. In other words, for the inputs $J = K = 1$, the state of the flip-flop is toggled.

This J-K flip-flop has two serious operational problems. First, to toggle the state of the flip-flop, the J and K inputs must be made 1 simultaneously. Otherwise, a *race condition* exists in which the flip-flop may be set to 1 if J is made 1 slightly earlier, or it may be cleared to 0 if K is made 1 slightly earlier, and then the flip-flop will toggle and return to its original state. Second, to toggle the state of the flip-flop just once, the J and K inputs have to be *pulsed* to 1, with the duration of the pulse being less than the propagation delay of the flip-flop. Otherwise, if J and K are 1 too long, the change in

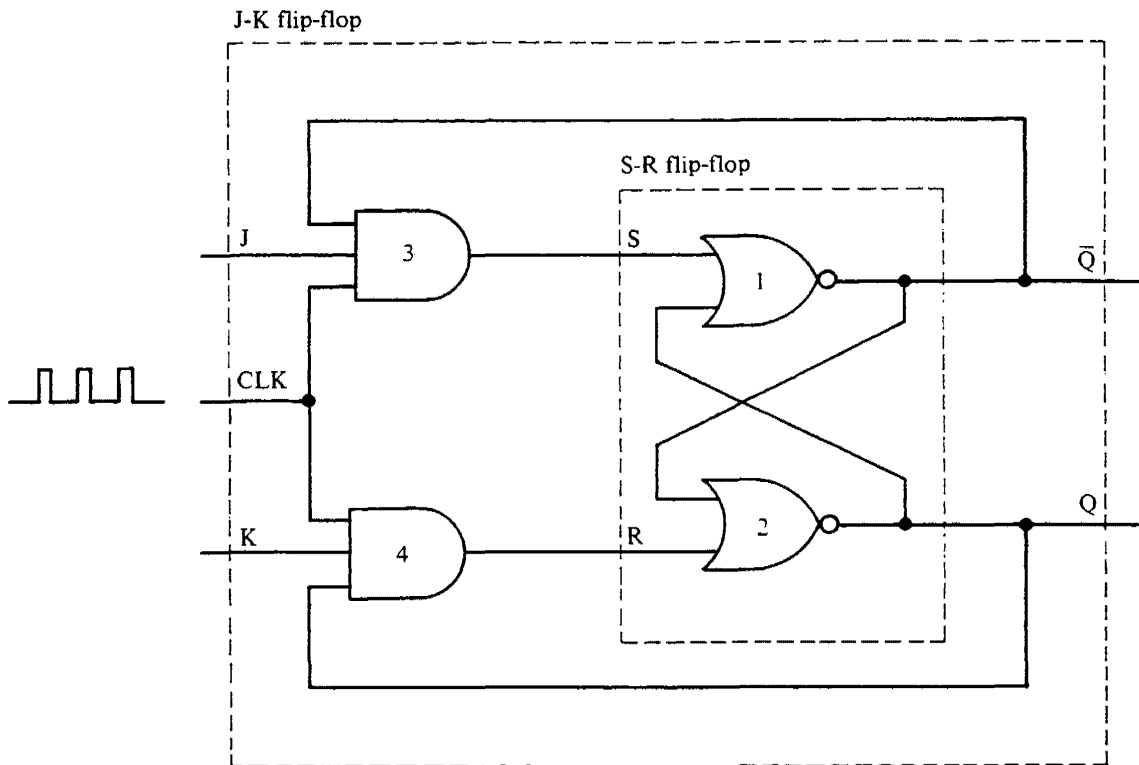


Figure 5.15 Clocked J-K flip-flop.

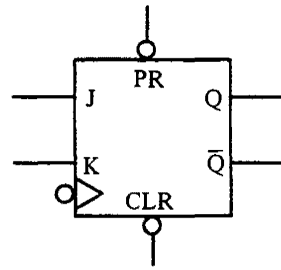
Q will propagate back to the AND gates while J and K are still 1, thereby causing another change of state.

The race condition problem can be eliminated by adding a clock input (CLK), as shown in Fig. 5.15. For this clocked J-K flip-flop, the changes in the J and K inputs will not affect the state of the flip-flop unless the clock input is 1. Consequently, provided that the changes in the J and K inputs are made before the clock pulse becomes 1, the timing of these inputs is not crucial as it was for the unclocked J-K flip-flop of Fig. 5.14. However, unlike the edge-triggered flip-flops that were presented earlier in this chapter, the duration of the clock pulse is still undesirably restricted to being less than the propagation delay of the flip-flop. To obtain an edge-triggered flip-flop, additional modifications to the basic flip-flop circuit are required. An example of such modifications is shown in Fig. 5.16, which is a circuit diagram for a 74'114 negative edge-triggered J-K flip-flop, with asynchronous preset and clear inputs. A detailed analysis of this flip-flop is quite cumbersome and so will not be presented here.

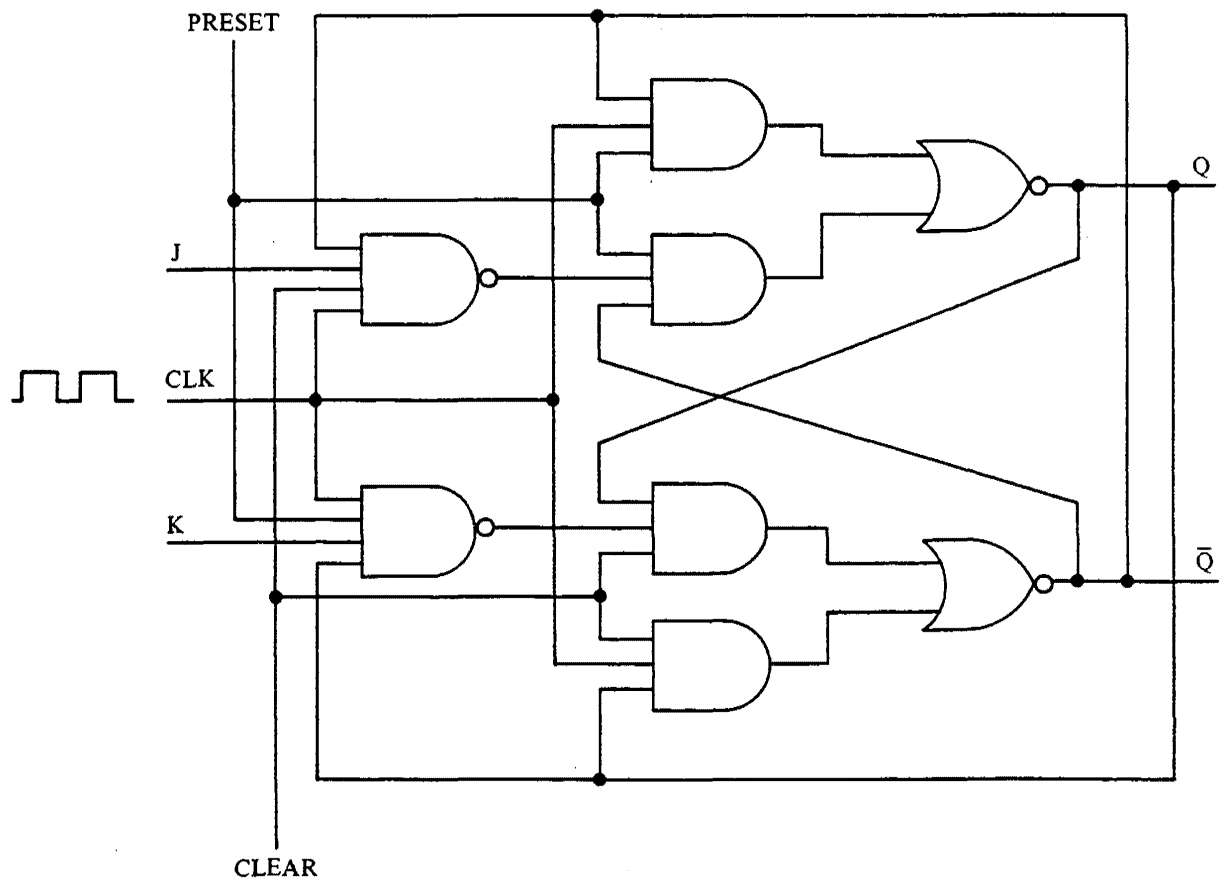
At this point of our study, we have gained enough of an understanding of the internal structure of flip-flops to use them effectively as circuit elements in digital systems. Therefore, we will not belabor the realization details, but instead will proceed to consider the applications of these devices.

5.6 COUNTERS

A digital *counter* is a sequential circuit element that counts through a prescribed sequence of numbers repeatedly. A counter may, for example, count through a 3-bit binary se-



(a) Block diagram



(b) Circuit diagram

Figure 5.16 74'114 negative edge-triggered clocked J-K flip-flop.

quence such as

... , 101, 110, 111, 000, 001, 010, 011, 100, 101, 110, 111, ...

or a 4-bit arbitrary sequence such as

... , 0111, 1010, 1001, 1111, 0000, 0110, 0011, 0111, 1010, 1001, ...

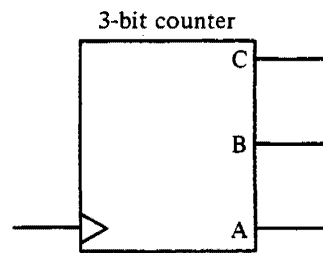
Counters are commonly used building blocks in the designs of digital systems.

This section begins with methods for the design and realization of counters using flip-flops. Then some commercially available realizations of counters, integrated as MSI circuit elements, will be described. Finally, the applications of MSI counters will be considered.

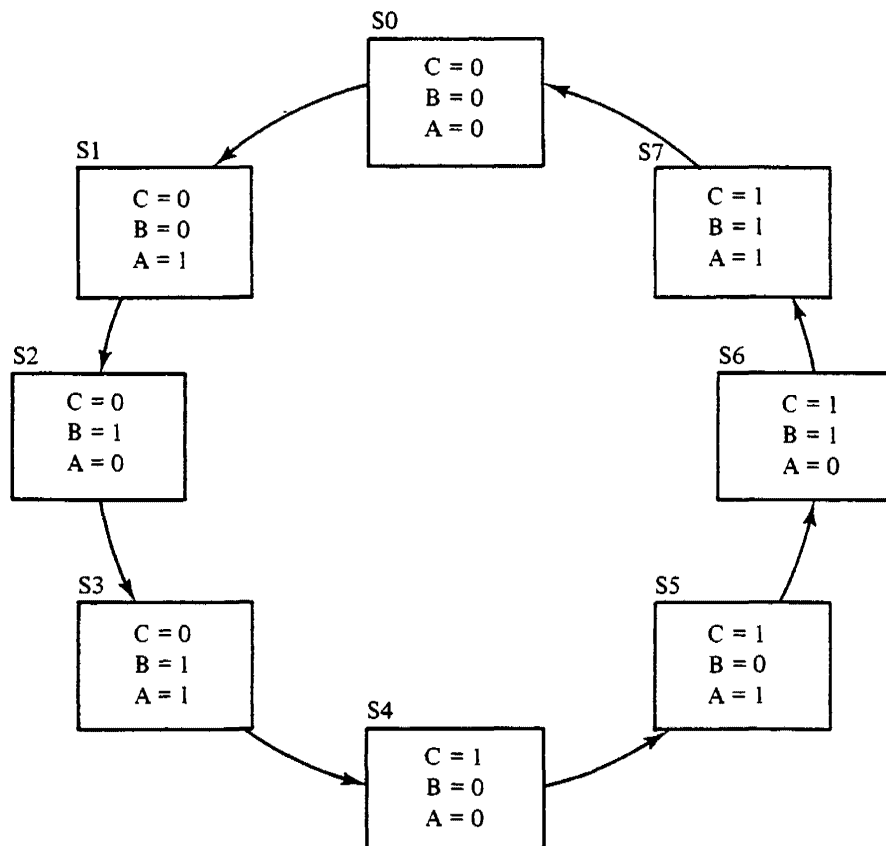
5.6.1 The Design and Realization of Synchronous Counters

In this subsection we will use examples to outline a systematic procedure for the design and realization of synchronous counters using flip-flops. We will design a specific synchronous 3-bit binary counter using D flip-flops first, and then using J-K flip-flops. Last, we will design and realize a synchronous counter with an arbitrary sequence.

Consider the synchronous 3-bit binary counter described in Fig. 5.17. As shown in the functional block diagram of Fig. 5.17(a), it has a single active-high clock input and three active-high outputs: C, B, and A. The operation of this 3-bit counter can be

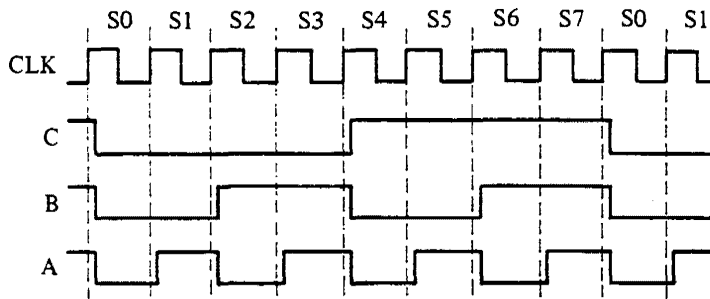


(a) Functional block diagram

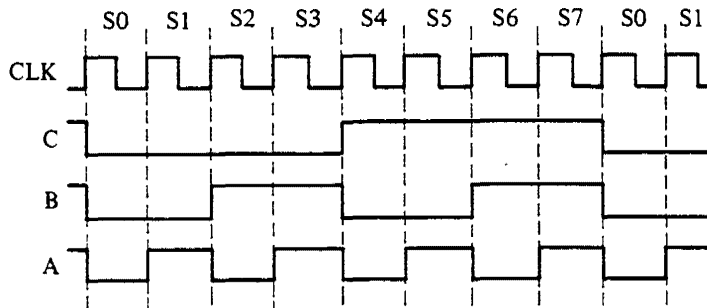


(b) State diagram

Figure 5.17 Definition of a 3-bit binary counter.



(c) Timing diagram with explicit propagation delays



Propagation delay not shown

(d) Timing diagram without explicit propagation delays

Figure 5.17 (cont.)

described through the use of the *state diagram* shown in Fig. 5.17(b). In a state diagram, a *state* is represented by a rectangular box with the name of the state labeled on the outside. For the state diagram of Fig. 5.17(b), there are eight states: S0, S1, S2, S3, S4, S5, S6, and S7. The duration of each state is one clock cycle of the counter clock input. Conceptually, a state diagram describes the outputs of a sequential circuit element over time. Specifically, the state diagram of Fig. 5.17(b) specifies that if the present state outputs of the counter are $C = 0$, $B = 0$, $A = 0$, then the next state outputs should be $C = 0$, $B = 0$, $A = 1$. Continuing, if the present state outputs are $C = 0$, $B = 0$, $A = 1$, then the next state outputs should be $C = 0$, $B = 1$, $A = 0$. And if the present state outputs are $C = 1$, $B = 0$, $A = 1$, then the next state outputs should be $C = 1$, $B = 1$, $A = 0$, and so forth. Essentially, the state diagram of Fig. 5.17(b) specifies that the 3-bit binary counter is a modulo 8 counter that counts the number of active clock transitions. In other words, it counts repeatedly through the following sequence at a rate equal to the frequency of the counter clock input.

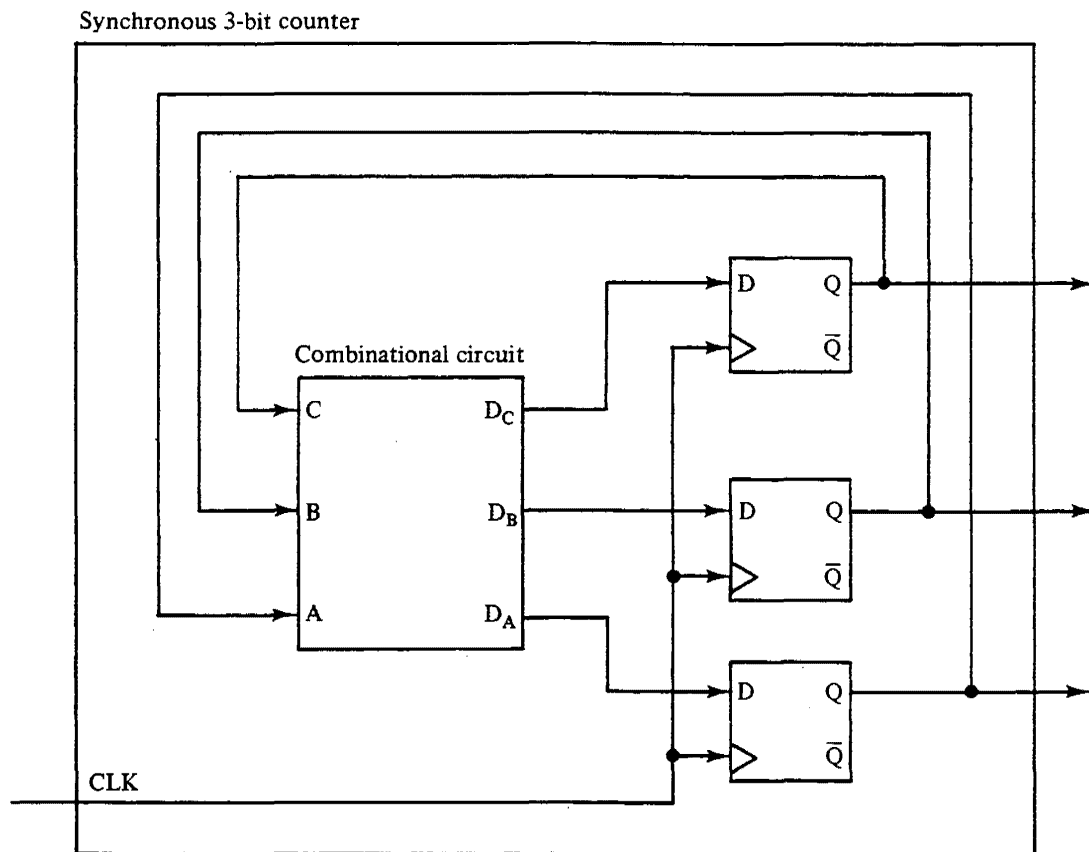
. . . , 101, 110, 111, 000, 001, 010, 011, 100, 101, 110, 111, 000, . . .

The operation of this 3-bit counter can also be described through the use of the timing diagram shown in Fig. 5.17(c). This timing diagram contains the same information as the state diagram of Fig. 5.17(b), but being more familiar, can be used to clarify and confirm the explanation of the state diagram. Note from Fig. 5.17(c) that the counter outputs change synchronously a propagation delay after the next active clock transition.

This is the reason that this type of counter is called a synchronous counter. Note also that the propagation delays are shown explicitly in the timing diagram, as they have been in all the previous timing diagrams. However, in practice, the propagation delays are usually not explicitly shown. Therefore, unless they are necessary to resolve ambiguities, the propagation delays will just be assumed and not explicitly shown in subsequent timing diagrams in this text. As an illustration, the timing diagram of Fig. 5.17(c) is reproduced in Fig. 5.17(d) without an explicit showing of the propagation delays.

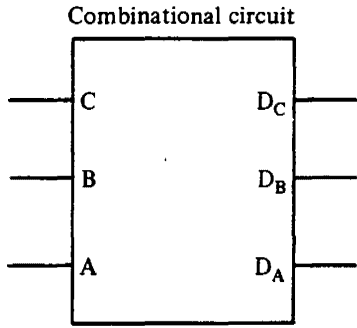
Example 5.5 Design and Realization of a Synchronous 3-Bit Counter Using D Flip-Flops

The design procedure for an N -bit synchronous counter is similar to that for a gated D flip-flop, which was outlined in Sec. 5.3.4. In this design procedure, N flip-flops are required to realize an N -bit counter. For an illustration, the functional block diagram of a 3-bit counter with three D flip-flops is shown in Fig. 5.18(a). Incidentally, note its similarity to the functional block diagram of the gated D flip-flop shown in Fig. 5.10(a). In Fig. 5.18(a) the D flip-flop outputs, which represent the *present-state* outputs of the counter, are fed back as the inputs to the combinational circuit. Also, the combinational circuit outputs, which represent the *next-state* outputs of the counter, are the inputs to the D flip-flops. With inputs corresponding to the current number (i.e., the present-state counter outputs), the combinational circuit is designed such that the next number in the



(a) Functional block diagram

Figure 5.18 Design and realization of a synchronous 3-bit counter.



Q	Q ⁺	D
0	0	0
0	1	1
1	0	0
1	1	1

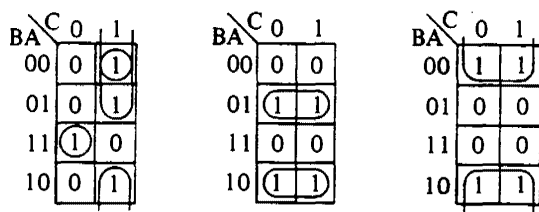
(c) D flip-flop excitation table

(b) Combinational circuit

Present-state outputs			Next-state outputs			Required flip-flop inputs		
C	B	A	C ⁺	B ⁺	A ⁺	D _C	D _B	D _A
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Next-state table

(d) Next-state table with required flip-flop inputs



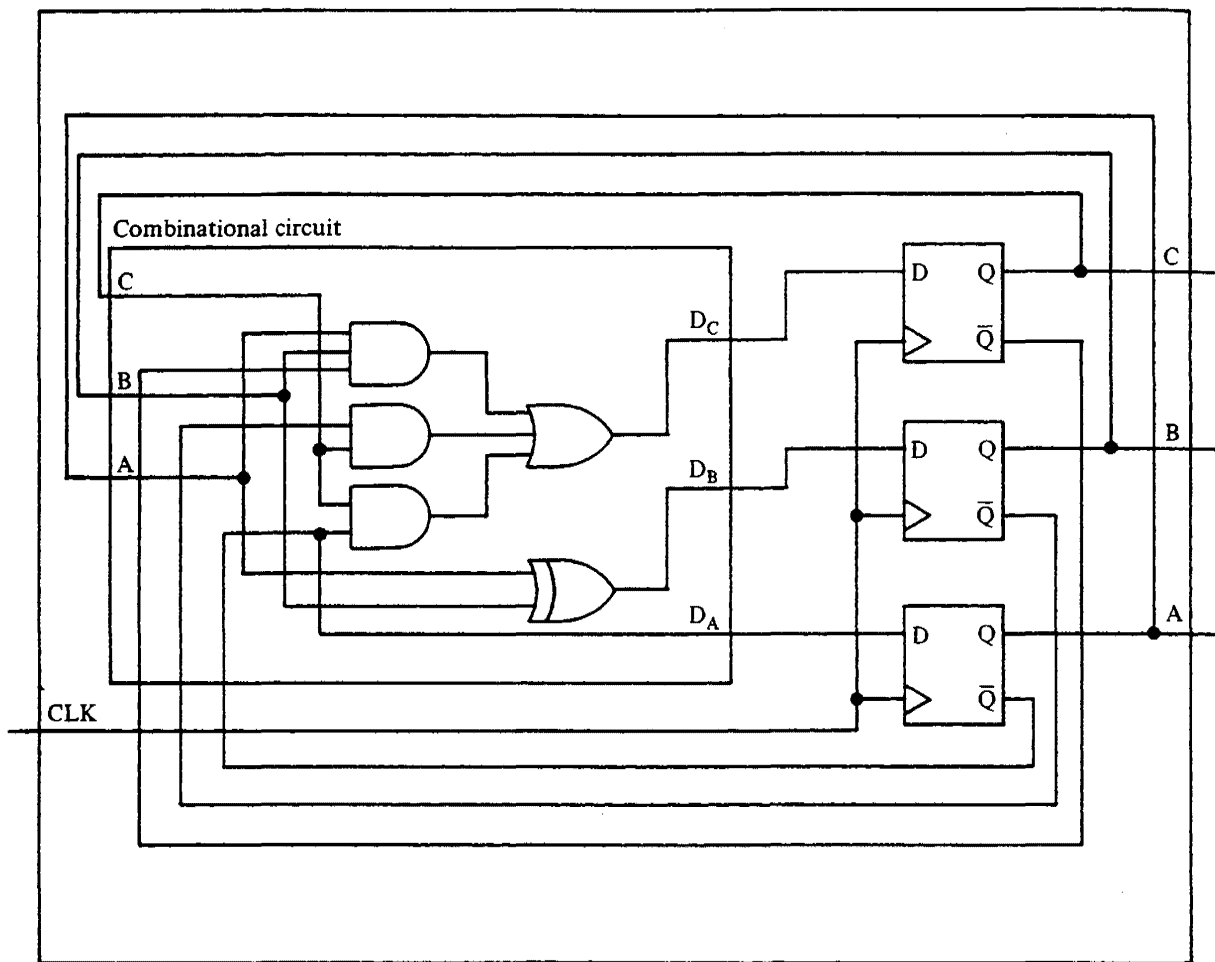
$$D_C = ABC\bar{C} + \bar{B}C + \bar{A}C \quad D_B = A\bar{B} + \bar{A}B = A \oplus B \quad D_A = \bar{A}$$

(e) K-maps for flip-flop inputs

Figure 5.18 (cont.)

sequence (i.e., the next-state counter outputs) is outputted. Consequently, at the next active clock transition, this next number in the sequence is loaded into the D flip-flops and outputted. It is apparent that the problem of counter design is reduced to the design of the combinational circuit that will transform the present-state counter outputs into the appropriate inputs for the flip-flops. A systematic procedure for the design and realization of a synchronous counter will now be outlined.

Synchronous 3-bit counter



(f) Circuit diagram

Figure 5.18 (cont.)

Step 1: Using a state diagram, define the count sequence of the counter to be designed. For this example, the count sequence for the 3-bit counter is defined by the state diagram shown in Fig. 5.17(b).

Step 2: Determine the functional block diagram of the N -bit counter to be designed. It consists of N flip-flops and a combinational circuit for generating the valid inputs for the flip-flops. For this example, the functional block diagram of the 3-bit counter is shown in Fig. 5.18(a).

Step 3: Determine the functional block diagram of the combinational circuit. It can be readily extracted from the functional block diagram of the counter obtained in step 2. For this example, the functional block diagram for the combinational circuit is shown in Fig. 5.18(b). The inputs are the present-state counter outputs C , B , A , which are fed back from the outputs of the flip-flops. The outputs of the combinational circuit are D_C , D_B , D_A , which correspond to the respective D flip-flop inputs.

Step 4: Using the excitation table for the selected flip-flop type, determine the truth table for the combinational circuit. For this example, the D flip-flop excitation table of Fig. 5.18(c) is used.

Step 4(a): Determine the next-state table for the counter. The next-state table specifies, in a tabular form, the next-state outputs of the counter corresponding to the present-state outputs. Since the next-state table contains the same information as the state diagram, it can be easily derived from this diagram. For the 3-bit counter, the next-state table, which is shown in Fig. 5.18(d), is derived from the state diagram of Fig. 5.17(b).

Step 4(b): Determine the required flip-flop inputs. Once the next-state table is determined, we can readily obtain the required flip-flop inputs by using the excitation table of the selected flip-flop type. For this example, the pertinent excitation table is that of Fig. 5.18(c) for the D flip-flop. With it we can determine the D inputs required to produce the desired D flip-flop output transitions. As indicated at the top of the table in Fig. 5.18(d), the values of D_C are derived from those of C and C^+ , the values of D_B are derived from those of B and B^+ , and the values of D_A from those of A and A^+ .

Step 5: Realize the combinational circuit. Once the truth table for the combinational circuit is determined from step 4, then the realization of this circuit is straightforward, using the techniques presented in Chapters 2 and 3. The resultant circuit diagram is shown in Fig. 5.18(f).

The operation of this counter circuit is verifiable from an exhaustive analysis of the circuit. Specifically, for each of the eight possible present-state outputs, the next-state counter output (i.e., the next number in the sequence) should agree with the state diagram of Fig. 5.17(b). ■ ■

Example 5.6 Design and Realization of a Synchronous 3-Bit Counter Using J-K Flip-Flops

In this example, we will redesign the 3-bit counter, using J-K instead of D flip-flops. The procedure outlined in Example 5.5 still applies, but in step 4 the excitation table of a J-K flip-flop must be used.

Step 1: Using a state diagram, define the count sequence of the counter to be designed. Since we are designing the same 3-bit counter, the state diagram of Fig. 5.17(b) still applies.

Step 2: Determine the functional block diagram of the 3-bit counter. As shown in Fig. 5.19(a), it consists of three J-K flip-flops and a combinational circuit for generating the valid inputs for the J-K flip-flops.

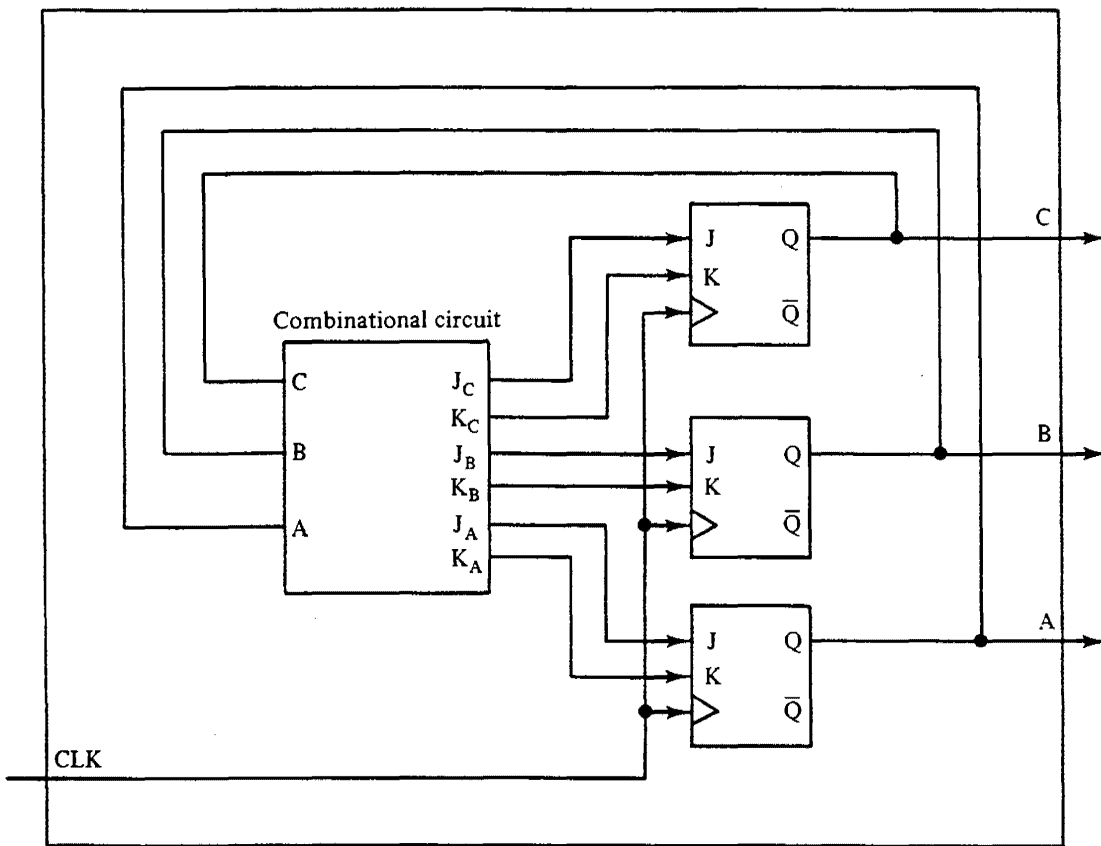
Step 3: Determine the functional block diagram for the combinational circuit. The functional block diagram is shown in Fig. 5.19(b). The inputs are the present-state counter inputs C , B , A , and the outputs are J_C , K_C , J_B , K_B , J_A , K_A corresponding to the J and K inputs of the three flip-flops.

Step 4: Using the excitation table for the selected flip-flop, determine the truth table for the combinational circuit. In this case, we use the excitation table for the J-K flip-flop shown in Fig. 5.19(c).

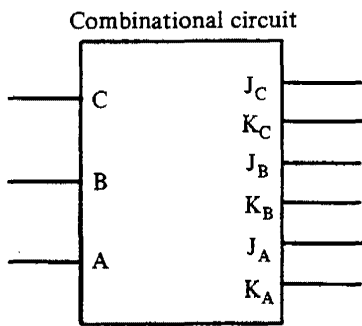
Step 4(a): Determine the next-state table. Since we are designing the same 3-bit counter, the next-state table of Fig. 5.18(d) again applies. It is reproduced in Fig. 5.19(d).

Step 4(b): Determine the required flip-flop inputs. Once the next-state table is defined, we can readily determine the required flip-flop inputs by using the J-K flip-flop excitation table of Fig. 5.19(c). As indicated at the top of the table in Fig. 5.19(d), the

Synchronous 3-bit counter



(a) Functional block diagram



(b) Combinational circuit

Q	Q ⁺	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(c) J-K flip-flop excitation table

Figure 5.19 Design and realization of a synchronous 3-bit counter using J-K flip-flops.

values for J_C and K_C are derived from those of C and C^+ , the values of J_B and K_B are derived from those of B and B^+ , and the values of J_A and K_A from those of A and A^+ .

Step 5: Realize the combinational circuit. Once the truth table for the combinational circuit is determined from step 4, the realization is easy to obtain with the techniques of Chapters 2 and 3. The resultant logic equations for the J-K flip-flops are given in Fig. 5.19(e). We will leave it to the reader to complete the drawing and verify the circuit diagram. ■ ■

Present-state outputs			Next-state outputs			Required flip-flop inputs					
C	B	A	C ⁺	B ⁺	A ⁺	J _C	K _C	J _B	K _B	J _A	K _A
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	0	0	0	X	1	X	1	X	1

Next-state table

(d) Next-state table with required flip-flop inputs

$$\begin{aligned}
 J_C &= K_C = AB \\
 J_B &= K_B = A \\
 J_A &= K_A = 1
 \end{aligned}$$

(e) Logic equations for flip-flop inputs

Figure 5.19 (cont.)

Example 5.7 The Design and Realization of a Synchronous Counter with an Arbitrary Sequence

Counters with an arbitrary counting sequence can also be designed with the above procedure. In this example, a counter is to be designed to count through the following sequence:

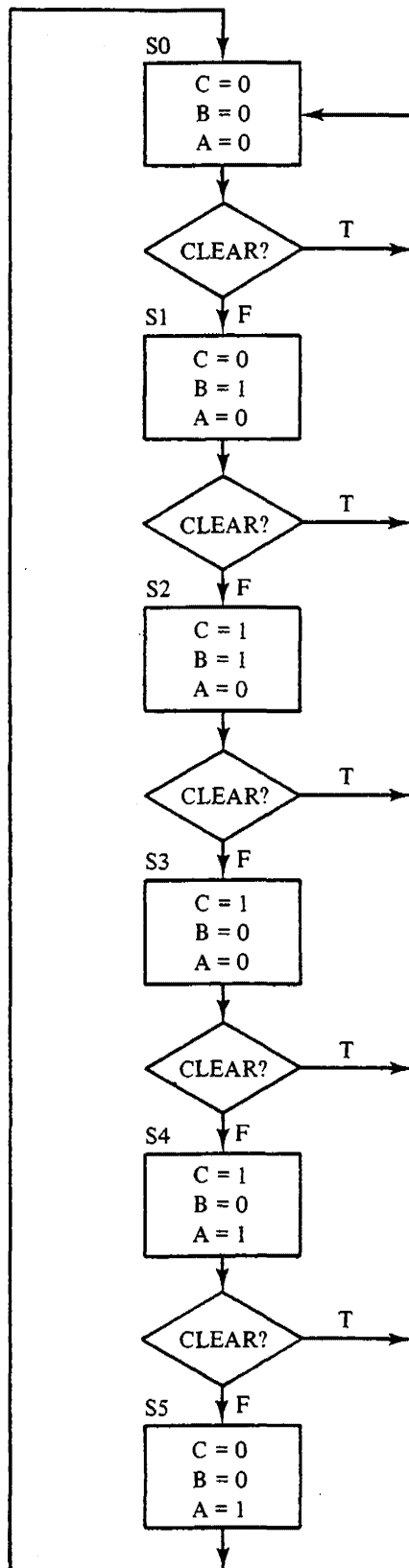
... , 101, 001, 000, 010, 110, 100, 101, 001, 000, 010, ...

Additionally, there is an external input CLEAR that when true will synchronously clear the counter. In other words, when CLEAR is made true, the normal count sequence is interrupted. And, at the next active clock transition, the counter outputs are cleared to 000. Then, normal counting will continue from this 000. An example of this clearing is shown in the following sequence:

... , 101, 001, 000, 010, 110, 100, 000, 010, 110, 100, 101, 001, 000, ...

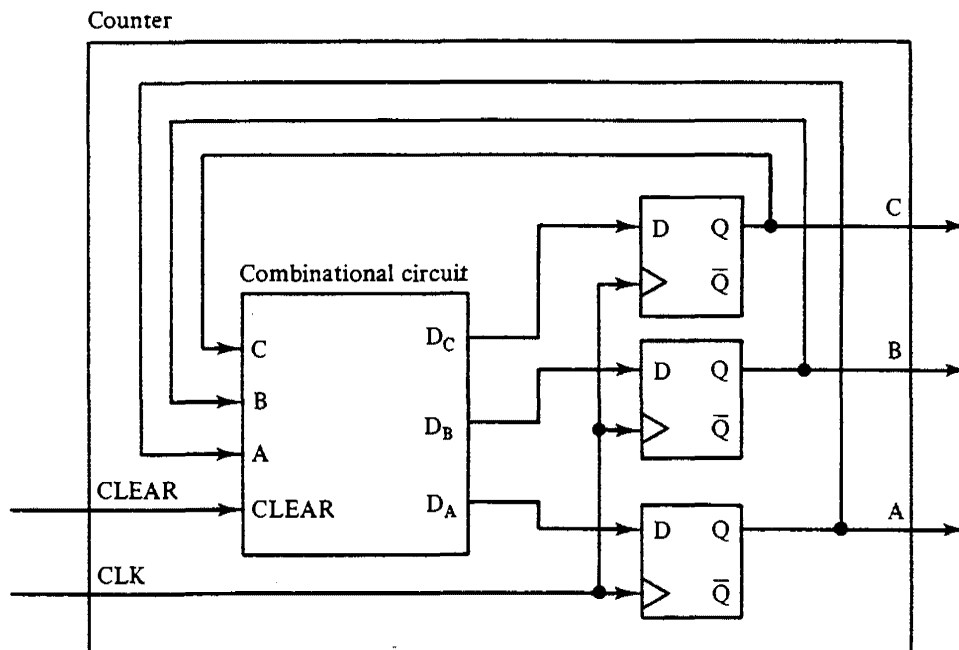
↑
CLEAR becomes true here

Step 1: Using a state diagram, define the count sequence of the counter that is to be designed. For this example, the count sequence for the counter is defined by the state diagram shown in Fig. 5.20(a). Note that when the CLEAR signal is false, the counter counts through the normal sequence. At any time, however, that CLEAR becomes true, the next state is S0.



(a) State diagram

Figure 5.20 Counter for Example 5.7.



(b) Functional block diagram of counter

CLEAR	C	B	A	C ⁺	B ⁺	A ⁺	D _C	D _B	D _A
0	0	0	0	0	1	0	0	1	0
0	0	0	1	0	0	0	0	0	0
0	0	1	0	1	1	0	1	1	0
0	0	1	1	X	X	X	X	X	X
0	1	0	0	1	0	1	1	0	1
0	1	0	1	0	0	1	0	0	1
0	1	1	0	1	0	0	1	0	0
0	1	1	1	X	X	X	X	X	X
1	X	X	X	0	0	0	0	0	0

$$D_C = \overline{\text{CLEAR}} \cdot B + \overline{\text{CLEAR}} \cdot C \cdot \bar{A}$$

$$D_B = \overline{\text{CLEAR}} \cdot \bar{C} \cdot \bar{A}$$

$$D_A = \overline{\text{CLEAR}} \cdot C \cdot \bar{B}$$

(d) Logic equations for flip-flop inputs

(c) Next-state table with required flip-flop inputs

Figure 5.20 (cont.)

Step 2: Determine the functional block diagram of the counter to be designed. In this example, we will arbitrarily decide to use D flip-flops. Since it is a 3-bit counter, three D flip-flops are required, as is shown in Fig. 5.20(b).

Step 3: Determine the functional block diagram for the combinational circuit. This circuit is also shown in Fig. 5.20(b).

Step 4: Using the excitation table for the selected flip-flop, determine the truth table for the combinational circuit. For the selected D flip-flop, the result is shown in Fig. 5.20(c). Note that the count sequence does not include the binary numbers 011 and 111. Consequently, in the next-state table shown in Fig. 5.20(c), the entries corresponding to those two numbers are don't cares.

Step 5: Realize the combinational circuit. The resulting logic equations for the flip-flop inputs are shown in Fig. 5.20(d). ■ ■

5.6.2 MSI Counters

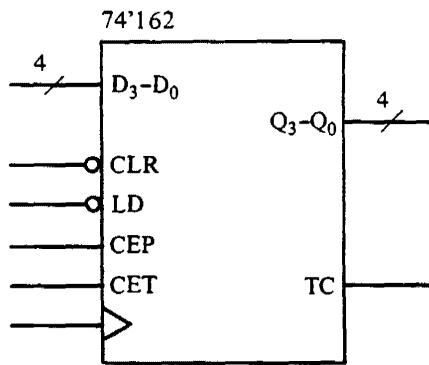
Digital counters are commercially available as MSI circuit elements in the form of multibit counters. Most common of these are the 4-bit binary counters and the 4-bit decade counters. A 4-bit binary counter is a modulo 16 counter that counts through the following sequence:

... , 0000, 0001, 0010, ... , 1111, 0000, ...

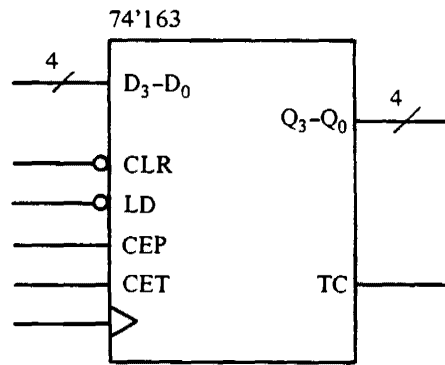
A 4-bit decade counter is a modulo 10 counter that counts through the following sequence:

... , 0000, 0001, 0010, ... , 1001, 0000, ...

Observe that this count does not include the numbers 1010 through 1111 of the count of the modulo 16 counter.



(a) Functional block diagram for the 74'162



(b) Functional block diagram for the 74'163

CLR	LD	CEP	CET	Function
0	0	1	1	Normal count
0	1	X	X	Parallel load
1	X	X	X	Clear

For the 74'162
 $TC = CET \cdot Q_3 \cdot \bar{Q}_2 \cdot \bar{Q}_1 \cdot Q_0$
 For the 74'163
 $TC = CET \cdot Q_3 \cdot Q_2 \cdot Q_1 \cdot Q_0$

(c) Functional descriptions

... , 1111, 0000, 0001, 0010, 1011, 1100, 1101, 1110, 1111, 0000, ...
 ↑
 during this clock cycle
 LD = 1 and D₃-D₀ = 1011

(d) Example of the LD input function

... , 1111, 0000, 0001, 0010, 0000, 0001, 0010, 0011, 0100, ...
 ↑
 during this clock cycle
 CLR = 1

(e) Example of the CLR input function

Figure 5.21 MSI counters.

The 74'163 is an example of a synchronous 4-bit binary counter, and the 74'162 is an example of a synchronous 4-bit decade counter. The functional block diagrams and the functional descriptions of both counters are shown in Fig. 5.21. Except for the count sequence, both counters operate in an identical manner. Specifically, for a count in a normal manner, both the count enable parallel input (CEP) and the count enable trickle input (CET) have to be true (H). Also, the parallel load input (LD) and the clear input (CLR) must be false (H).

There are other modes of operation. If the LD input is true (L) and the CLR input is false (H), then any 4-bit number applied at the D_3 - D_0 inputs is synchronously loaded into the counter at the next active clock transition. So, the normal count sequence is interrupted. Further, when the normal counting resumes, it will continue from the loaded number, as shown in the example in Fig. 5.21(d). Finally, if the CLR input is true (L), then the counter is synchronously cleared to 0000 at the next active clock transition. In other words, the normal count sequence is again interrupted. And when normal counting resumes, it will continue from 0000, as shown in the example in Fig. 5.21(e). Incidentally, since in this example, as well as that of Fig. 5.21(d), the count is shown as exceeding 1001, the specific counts shown must be from the 74'163 binary counter.

Both the 74'162 and the 74'163 have a terminal count (TC) output. This output becomes true (H) at the active clock transition at which the terminal count (1001 for the 74'162 and 1111 for the 74'163) of the counter is reached, resulting in a positive-going pulse that lasts for one clock period. This pulse can be applied as a control signal to other circuit elements. Most often this TC output is used in the cascading of 4-bit counters to obtain a counter of greater length. As an illustration, in Fig. 5.22 an 8-bit binary counter is shown as constructed from two 74'163 4-bit counters without any additional circuitry. Note that the TC output from the low-order 4-bit counter controls the operation of the high-order 4-bit counter so that this high-order counter counts only once for every 16 clock cycles. In general, a number of 4-bit counters can be cascaded in this manner to produce a binary counter of any reasonable length.

5.7 REGISTERS

A register is a group of flip-flops that are used for data storage and perhaps also for performing some function on the stored data. Strictly speaking, the counters of the last section are also registers in which the function performed is counting through a prescribed sequence. In this section, we will consider two common types of registers: storage registers and shift registers.

5.7.1 Storage Registers

A storage register is a group of flip-flops that are used simply for data storage. The simplest storage register consists of several flip-flops with common clock inputs. An example of a 4-bit storage register is shown in Fig. 5.23. It consists of four D flip-flops in which 4 bits of data (D_3 - D_0) are loaded into the storage register at each active clock

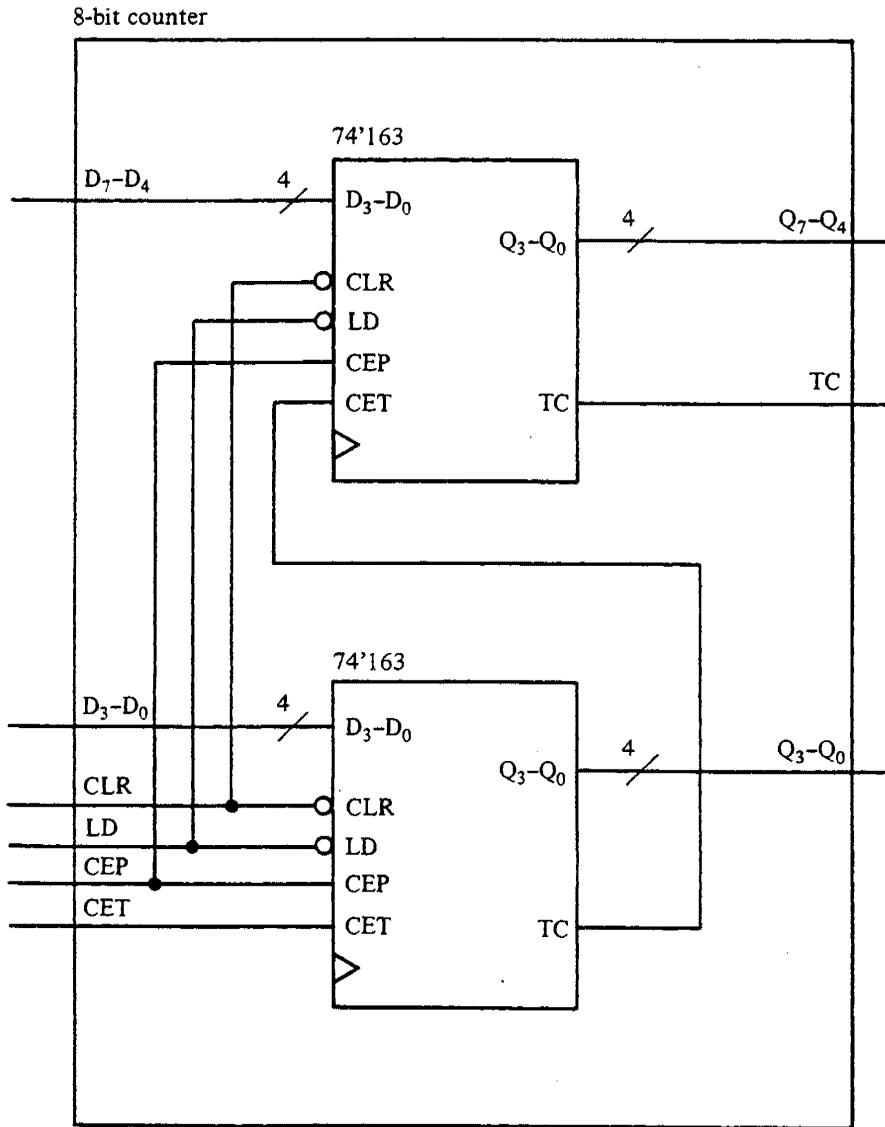
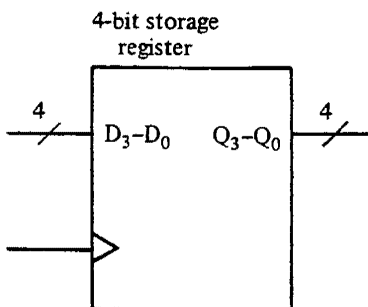
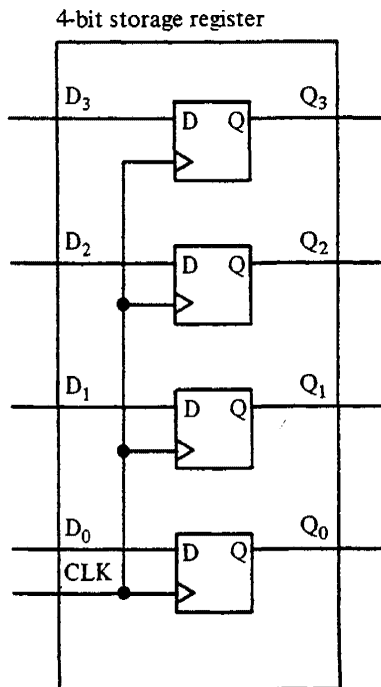


Figure 5.22 8-bit binary counter.



(a) Functional block diagram

Figure 5.23 4-bit storage register.



(b) Realization

Figure 5.23 (cont.)

transition. With this type of storage register the data is stored in the register until the next active clock transition.

Storage registers are commercially available as MSI circuit elements in various forms, with various features. Three examples are shown in Fig. 5.24. The 74'273 storage register of Fig. 5.24(a) has an asynchronous master reset input MR. As shown in the accompanying voltage table, if the MR input is false (H), then the 74'273 functions as a normal storage register. If, however, the MR input is true (L), then the 8-bit register is *asynchronously* cleared to 0.

The difference between an asynchronous operation and a synchronous operation is illustrated in Fig. 5.25. An asynchronous operation is performed "immediately" after the control input is applied, as shown in Figs. 5.25(a) and (b). In contrast, a synchronous operation is synchronized by a clock input, and so the operation is performed only if the control signal is true at the active clock transition, as is illustrated in Figs. 5.25(c) and (d). In Fig. 5.25(d) note that the pulse at the MR input has no effect on the synchronous clear operation.

In Fig. 5.24(b) is illustrated a 74'378 storage register with an enable input E. As shown in the accompanying voltage table, the 6-bit storage register is loaded with new data (D_5 – D_0) only if the enable input is true (L). Otherwise, the existing data will be stored in the register indefinitely. A storage register is most useful if it has an enable input, because then we can connect the system clock signal directly to the clock input of the register and a control signal to the enable input.

Shown in Fig. 5.24(c) is a 74'163 counter, arranged to be a 4-bit storage register with a *synchronous* master reset input (CLR) and an enable input (LD). This storage feature is obtained by disabling the count enable ($CEP = CET = \text{false}$) and not using the terminal count (TC) output.

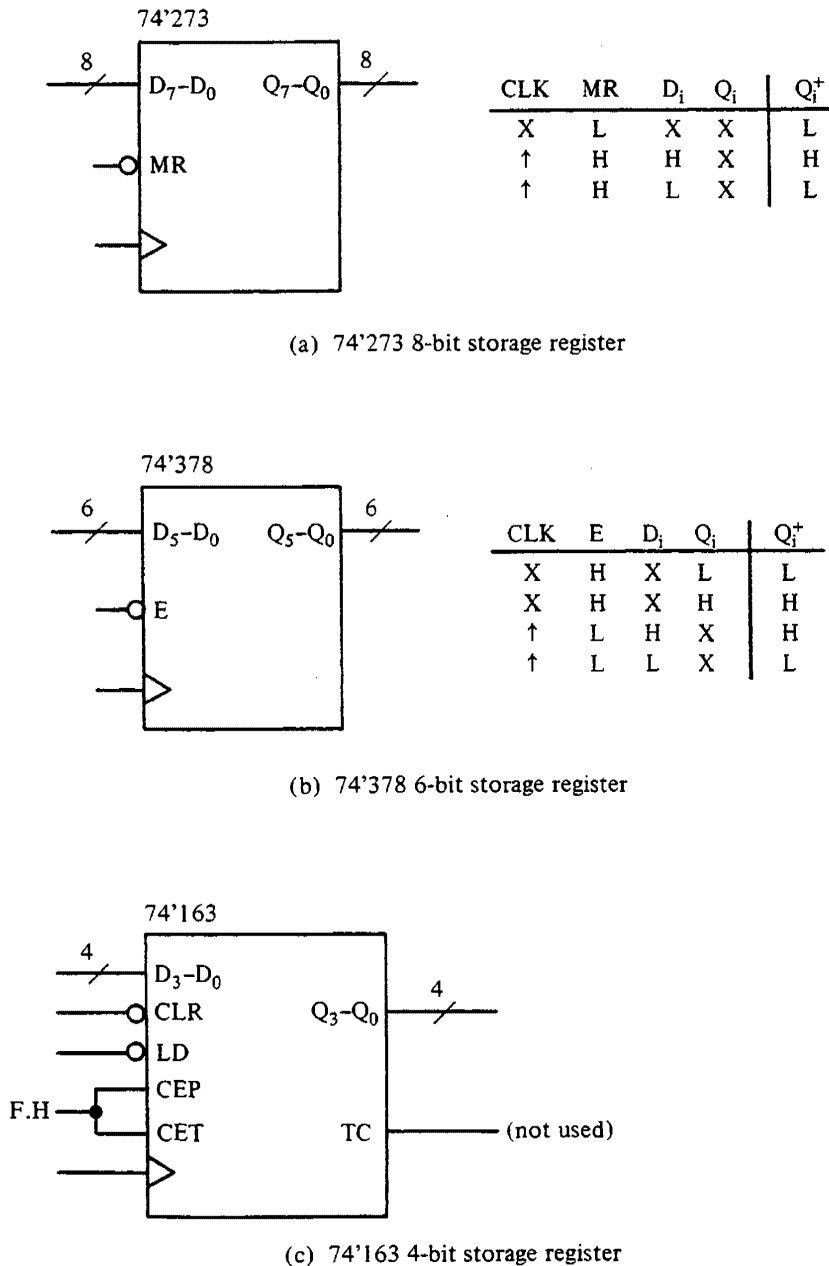


Figure 5.24 Commercially available storage registers.

5.7.2 Shift Registers

A shift register is a register in which the stored data can be shifted to the left or to the right. The simplest shift register consists of flip-flops connected as shown in Fig. 5.26(b). For this shift register, the contents of each flip-flop are loaded into the adjacent one on the right at each active clock transition. In other words, the stored contents are shifted to the right by one flip-flop. In this process, the old contents of the rightmost flip-flop are lost, and the new contents of the leftmost flip-flop are the data applied at DIN.

Shift registers are commercially available as MSI circuit elements in various forms and with various features. An example of one with many features is the 74'194 4-bit

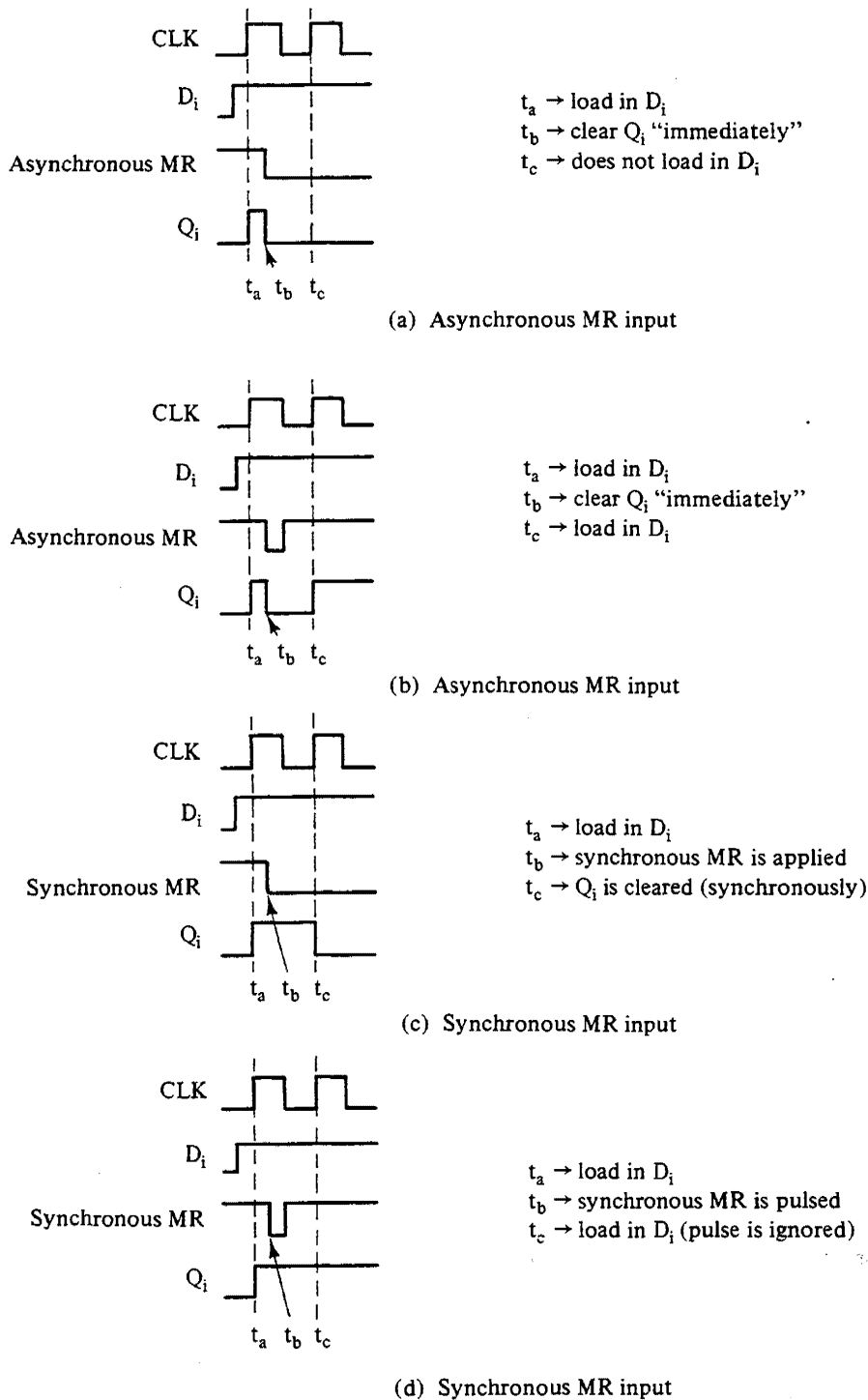
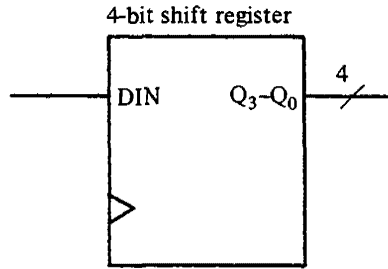
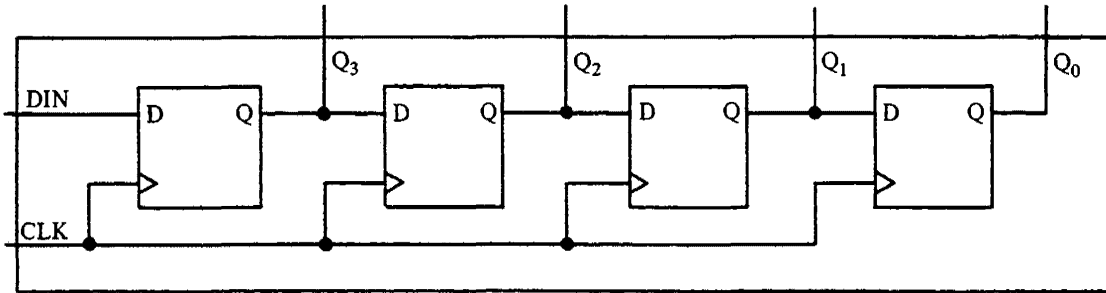


Figure 5.25 Asynchronous and synchronous control inputs.

bidirectional universal shift register, which is illustrated in Fig. 5.27. As shown in the voltage table, it performs five major operations, all synchronously. If the CLR input is true (L), then regardless of the other inputs, the shift register is synchronously cleared. But if the CLR input is false (H), the operation of the shift register depends on the mode control inputs S_1 and S_0 . If $S_1S_0 = LL$, then the shift register retains the existing data and functions essentially as a storage register. If $S_1S_0 = LH$, then the contents of the shift register are shifted to the right by 1 bit at the next active clock transition, with the

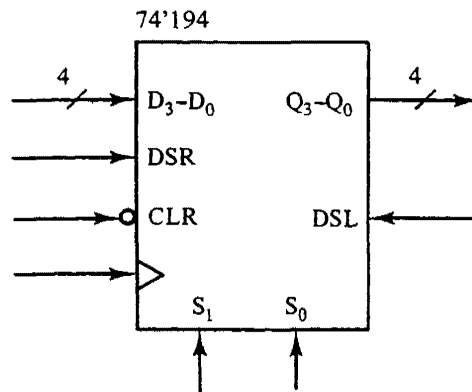


(a) Functional block diagram



(b) Realization

Figure 5.26 4-bit shift register.



(a) Functional block diagram

Operation	CLK	CLR	S ₁	S ₀	DSR	DSL	Q ₃ ⁺	Q ₂ ⁺	Q ₁ ⁺	Q ₀ ⁺
Clear	↑	L	X	X	X	X	L	L	L	L
Hold	↑	H	L	L	X	X	Q ₃	Q ₂	Q ₁	Q ₀
Shift right	↑	H	L	H	L	X	L	Q ₃	Q ₂	Q ₁
	↑	H	L	H	H	X	H	Q ₃	Q ₂	Q ₁
Shift left	↑	H	H	L	X	L	Q ₂	Q ₁	Q ₀	L
	↑	H	H	L	X	H	Q ₂	Q ₁	Q ₀	H
Parallel load	↑	H	H	H	X	X	D ₃	D ₂	D ₁	D ₀

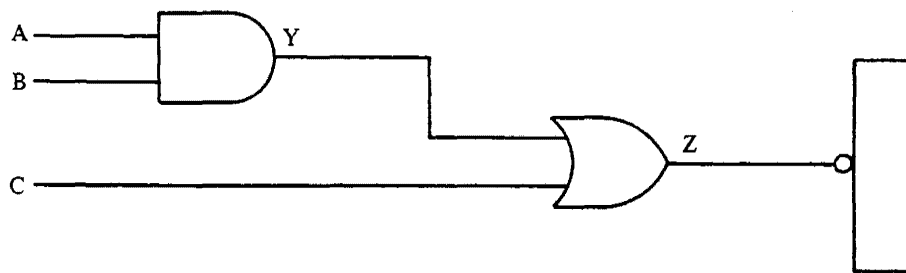
(b) Voltage table

Figure 5.27 74'194 bidirectional universal shift register.

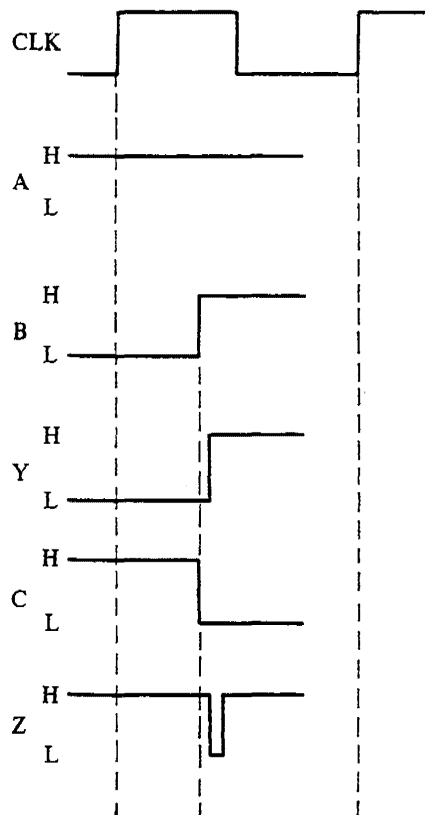
leftmost flip-flop (Q_3) receiving the data applied at DSR. If $S_1S_0 = HL$, then the contents are shifted to the left by 1 bit, with the rightmost flip-flop (Q_0) receiving the data applied at DSL. Finally, if $S_1S_0 = HH$, then the data applied at D_3-D_0 is parallel loaded into the shift register, and there is no shifting.

5.8 SYNCHRONOUS VERSUS ASYNCHRONOUS DESIGNS

The difference between asynchronous and synchronous operations was illustrated by the examples in Fig. 5.25. Generally, in the design of digital circuits, synchronous operations using synchronous sequential circuit elements are preferred. To understand why, consider the simple circuit shown in Fig. 5.28(a), and assume that the initial values for the inputs



(a) Circuit diagram



(b) Timing diagram

Figure 5.28 Glitch resulting from unequal path delays.

are $A = H$, $B = L$, and $C = H$. For these inputs, the output Z , of course, should be H , as shown in Fig. 5.28(b). Now if B is changed from L to H and at the same time C is changed from H to L , then logically the output Z should remain H . Unfortunately, as shown in the timing diagram of Fig. 5.28(b), these changes of inputs cause a momentarily unwanted glitch to occur at the output Z . This is a result of the unequal path delays through the combinational circuit between B to Z and C to Z . In other words, a *race condition* exists between the two signals. The change in C arrives at Z first, after one gate delay, causing Z to become L momentarily. Finally, after two gate delays, the change in B arrives and the Z output settles down to its correct H value. The circuit becomes stable at two gate delays after the changes. But if the Z output is used as an input to an *asynchronous* control input such as a clear input, then the circuit element would be cleared erroneously by the glitch.

In general, asynchronous operations and asynchronous circuit elements are to be avoided in the design of a digital circuit. In addition to erroneous operations such as the one just illustrated, there can be other serious operational problems and restrictions. Not being synchronized by a clock signal, the outputs of an asynchronous digital circuit element depend on the *order* of the changes in the asynchronous inputs. As a result, the element outputs can be transiently unstable and unpredictable. Because of these and other problems related to timing, the design of asynchronous digital circuits is much more difficult than that of synchronous digital circuits.

In Chapter 7, a procedure for designing synchronous digital circuits is described and formalized. The general idea is to avoid the use of asynchronous operations and to have the transient inputs and outputs isolated in the early part of the clock cycle. In this manner, all inputs to the sequential circuit elements are assured of being stable at the next active clock transition. Before considering this design procedure, however, we will complete the study of digital building blocks by considering LSI and other circuit elements in the next chapter.

SUPPLEMENTARY READING (see Bibliography)

[Bartee 85], [Blakeslee 79], [Hill 81], [Mano 79], [McCluskey 75], [Motorola], [Peatman 80], [Prosser 87], [Roth 85], [Texas Instruments]

PROBLEMS

- 5.1. How does a sequential circuit element differ from a combinational circuit element?
- 5.2. A clock signal has a frequency of 5.0 MHz and a 30 percent duty cycle. Draw and label the waveform for the clock signal.
- 5.3. Given the circuit diagram of Fig. 5.29(a), complete the timing diagram of Fig. 5.29(b) for Q_A , Q_B , and Q_C . Show the propagation delays t_{pHL} and t_{pLH} .

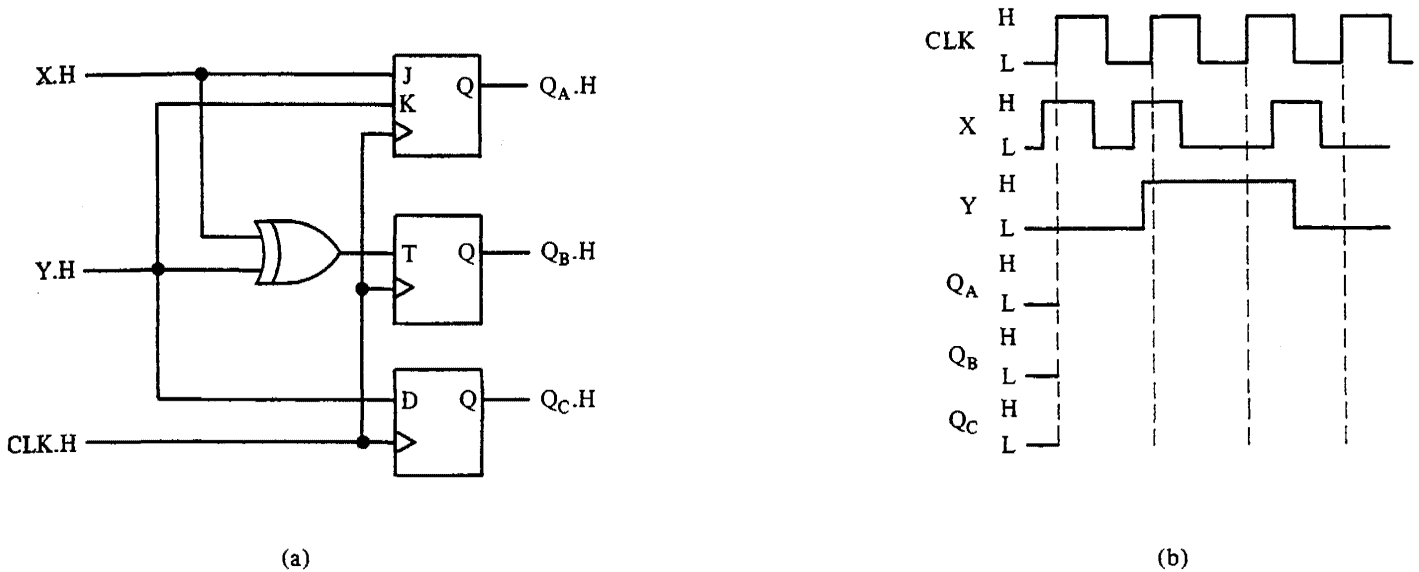


Figure 5.29 Circuit diagram and timing diagram for Problem 5.3.

5.4. Figure 5.30(a) shows a J-K flip-flop with an active-low clock input.

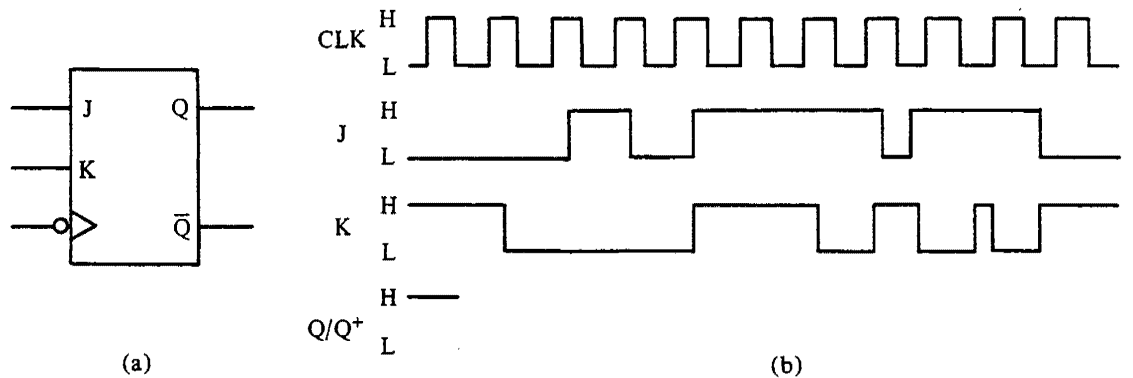


Figure 5.30 J-K flip-flop and timing diagram for Problem 5.4.

(a) Complete the timing diagram of Fig. 5.30(b) and compare it with the one shown in Fig. 5.2(c). Show the propagation delays t_{pHL} and t_{pLH} .

(b) Redraw the timing diagram without showing the propagation delays.

5.5. Recall that binary subtraction can be performed by adding the 2s-complement form of the subtrahend to the minuend. In other words,

$$A - B = A + (-B)$$

With this in mind, convert the serial adder shown in Fig. 5.5(a) into a serial subtractor. What must be the initial value of the D flip-flop output? Assume that a 74'74 D flip-flop, as shown in Fig. 5.6(a), is to be used. Be sure to initialize the contents of the D flip-flop with an active-low signal, INIT_PULSE ().

5.6. Convert the serial adder of Fig. 5.5(a) into a serial adder/subtractor, the block diagram of which is shown in Fig. 5.31. The operation is as follows: When \bar{A}/S is false (L), the addition operation is performed. But when \bar{A}/S is true (H), the subtraction operation is performed. The active-low signal INIT_PULSE is used to initialize the D flip-flop contents. Assume that a 74'74 D flip-flop is to be used. *(Hint: Also use an XOR gate and an inverter.)*

(Hint: An XOR gate would be useful.)

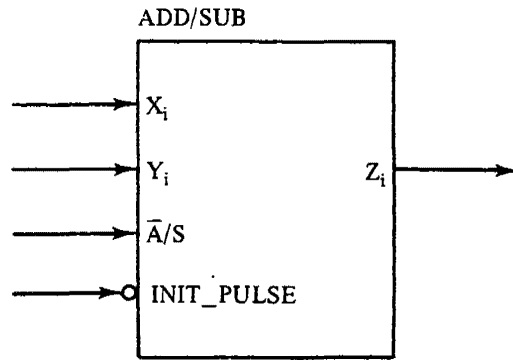


Figure 5.31 Serial adder/subtractor for Problem 5.6.

- 5.7. (a) Analyze the circuit diagram of Fig. 5.32(a) and complete the timing diagram of Fig. 5.32(b). Do not show the propagation delays.
 (b) Assuming that the circuit of Fig. 5.32(a) represents a type of flip-flop, derive its characteristic table and its condensed characteristic table.
 (c) Derive its excitation table.

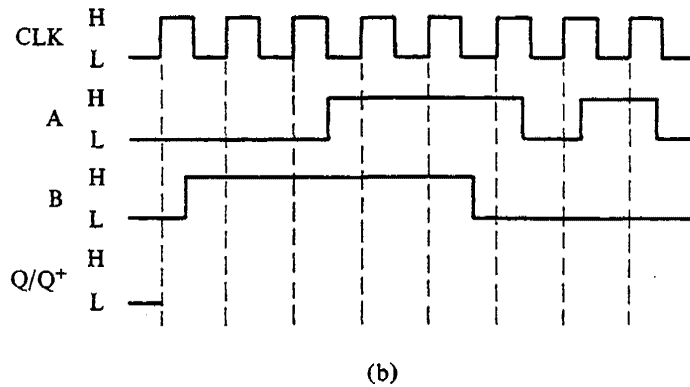
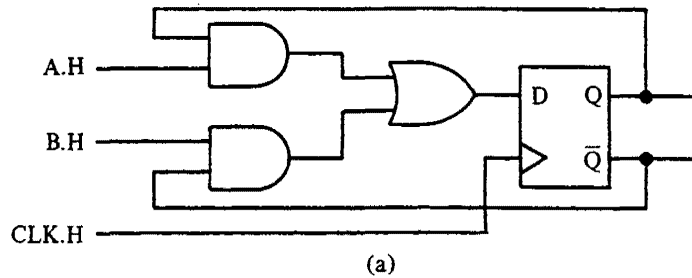
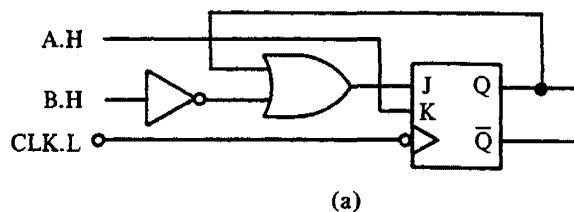
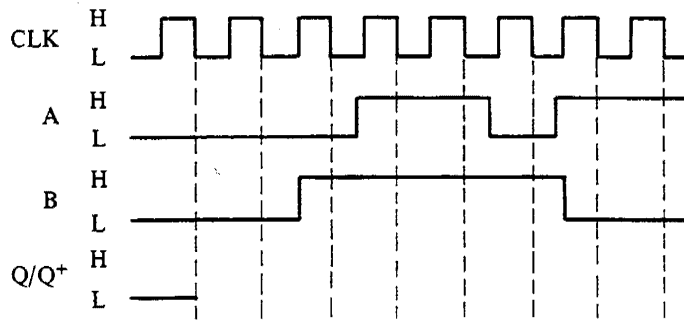


Figure 5.32 Circuit and timing diagrams for Problem 5.7.

- 5.8. Repeat Problem 5.7 for the circuit and timing diagrams of Fig. 5.33. Note that the clock input is active-low.





(b)

Figure 5.33 Circuit and timing diagrams for Problem 5.8.

- 5.9. Implement a J-K flip-flop using a 74'74 D flip-flop and any gates that are needed. Label all gates.
- 5.10. Implement a T flip-flop using a 74'74 D flip-flop and any gates that are needed. Label all gates.
- 5.11. Implement the unclocked S*-R* flip-flop of Fig. 5.34 with a normal unclocked S-R flip-flop and any gates that are needed. An unclocked S*-R* flip-flop functions exactly like a normal unclocked S-R flip-flop except that S* = R* = 1 is allowed. For this input the S*-R* flip-flop retains its previous Q value.

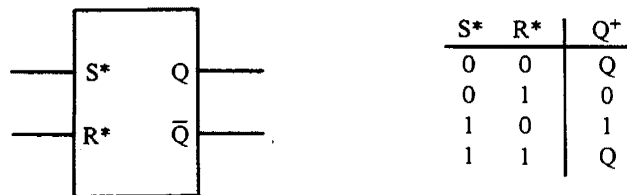


Figure 5.34 Flip-flop for Problem 5.11.

- 5.12. The A-B flip-flop of Fig. 5.35 is to be implemented. Note that this flip-flop functions as a J-K flip-flop except for the inputs A = B = 1, for which the "set" input A dominates.

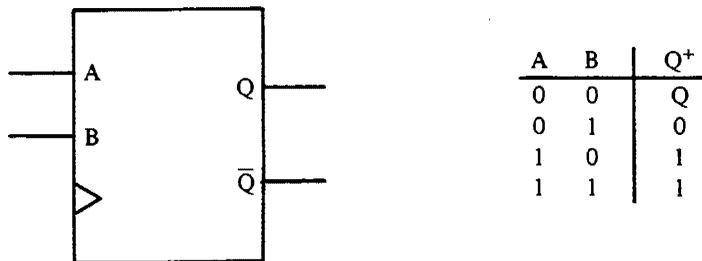


Figure 5.35 Flip-flop for Problem 5.12.

- (a) Implement it using a 74'109 J-K flip-flop plus any gates that are needed.
- (b) Implement it using a 74'74 D flip-flop plus any gates that are needed.
- (c) Implement it using the T flip-flop of Fig. 5.7 plus any gates that are needed.

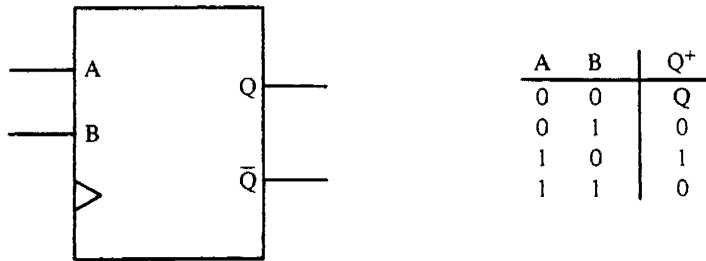


Figure 5.36 Flip-flop for Problem 5.13.

- 5.13.** Repeat Problem 5.12 for the A-B flip-flop of Fig. 5.36. Note that this flip-flop functions as a J-K flip-flop except for the inputs $A = B = 1$, for which the “clear” input B dominates.
- 5.14.** (a) Given the truth table of Fig. 5.11(b) for an unlocked S-R flip-flop, determine the logic equation for Q^+ as a function of S, R, and Q.
 (b) The circuit diagram of Fig. 5.13(a) is the most popular gate implementation for an S-R flip-flop. Algebraically show that this implementation is consistent with your answer to part (a).
 (c) Draw a mixed-logic circuit diagram corresponding to your answer to part (a).
- 5.15.** Using commercially available gates, implement the unlocked S-R flip-flop of Fig. 5.37. Note the active-low inputs. Specify and label all components and signals.

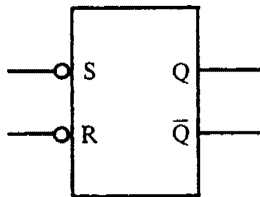


Figure 5.37 S-R flip-flop for Problem 5.15.

- 5.16.** The operation of the switch debouncing circuit of Fig. 5.12(c) depends upon the voltage levels at the S and R inputs being both L when the switch arm is not in contact with either the ON or OFF terminal. For this L voltage level to occur, the resistor resistances must be small enough that the voltages at the S and R inputs are less than or equal to V_{IL} , which is 0.8 V for LS-TTL. With this in mind, determine the maximum resistance, R_{max} , allowable for the resistors. Use LS-TTL values.
- 5.17.** The unlocked S-R flip-flop with active-low inputs that is specified in Problem 5.15 can also be used to debounce a mechanical switch. The circuit will be similar to that shown in Fig. 5.12(c). In this case, however, a *high* voltage is required at the S and R inputs to present a false value when the switch arm is between the ON and OFF terminals. With this in mind,
 (a) Design a switch debouncing circuit using this “active-low” S-R flip-flop.
 (b) Determine the minimum resistance, R_{min} , of the resistors that will give proper operation when the switch arm is between the ON and OFF terminals. Use LS-TTL values.
- 5.18.** Complete the timing diagram of Fig. 5.38, which is for the unlocked J-K flip-flop shown in Fig. 5.14. Show the propagation delays t_{pHL} and t_{pLH} . Also, what happens after t_f ?

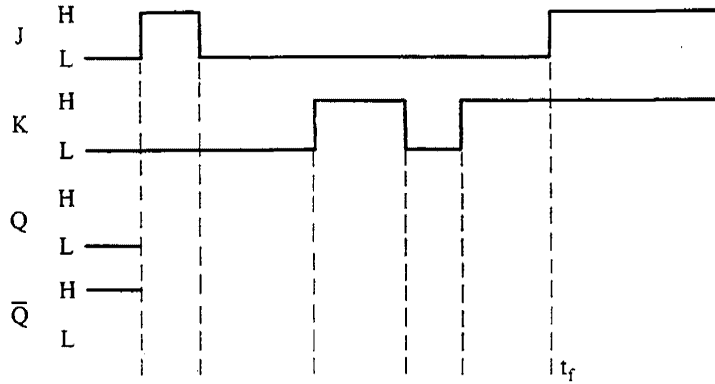


Figure 5.38 Timing diagram for Problem 5.18.

- 5.19. The circuit diagram of Fig. 5.15 is for a clocked J-K flip-flop.
- (a) How does this flip-flop differ from the clocked J-K flip-flop discussed in Sec. 5.3.1?
 - (b) What is the restriction on the clock signal for the flip-flop shown in Fig. 5.15? Why?
- 5.20. Complete the timing diagram of Fig. 5.39 for the 74'107 J-K flip-flop of Fig. 5.3(a) to demonstrate the difference between a synchronous and an asynchronous clear operation. Do not show the propagation delays.

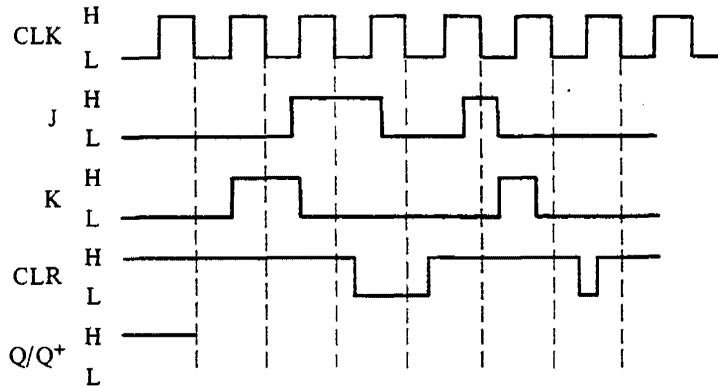
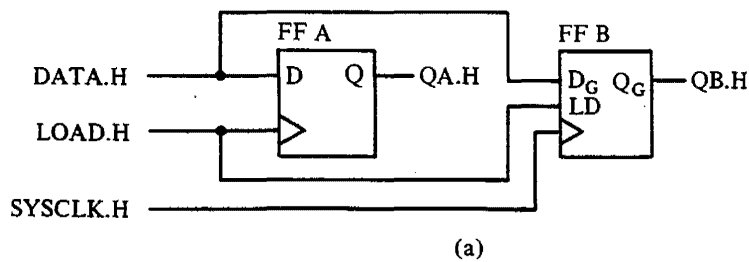


Figure 5.39 Timing diagram for Problem 5.20.

- 5.21. Given the circuit diagram of Fig. 5.40(a) consisting of a normal D flip-flop and a gated D flip-flop (see Example 5.2 and Fig. 5.8 for details), complete the timing diagram of Fig. 5.40(b). Do not show the propagation delays.



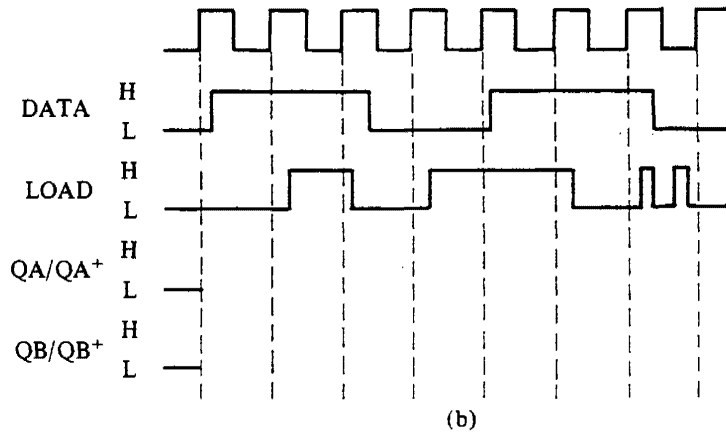


Figure 5.40 Circuit diagram and timing diagram for Problem 5.21.

- 5.22.** A 4-bit binary counter is to be designed and realized using D flip-flops. The count is to be as follows:

. . . , 0000, 0001, 0010, 0011, 0100, . . . , 1110, 1111, 0000, 0001, . . .

- (a) Draw the state diagram for the count sequence.
- (b) Draw the functional block diagram for the counter, including the D flip-flops and the corresponding combinational circuit, in the manner shown in Fig. 5.18(a).
- (c) Determine the required logic equations and draw the circuit diagram for the counter.

- 5.23.** Repeat Problem 5.22 using J-K flip-flops.

- 5.24.** Repeat Problem 5.22 using T flip-flops.

- 5.25.** Design and realize a 3-bit counter that counts in the following sequence:

. . . , 111, 010, 001, 110, 100, 000, 111, 010, 001, . . .

- (a) Use D flip-flops.
- (b) Use J-K flip-flops.
- (c) Use T flip-flops.

- 5.26.** Design and realize a 4-bit decade counter with a synchronous CLEAR input. Use D flip-flops.

- 5.27.** Design and realize a 3-bit Gray-code counter with an enable (EN) input. Use J-K flip-flops. The counter is to count in the prescribed Gray-code sequence if EN is true at the next active clock transition. But if EN is false at this transition, then the counter does not count and, instead, retains its current value. The Gray-code sequence is as follows:

. . . , 000, 001, 011, 010, 110, 111, 101, 100, 000, 001, . . .

Observe for the Gray code that only one bit value changes from one number to the next in the sequence. This is an important feature in some applications.

- 5.28.** A counter is to be designed for counting in four different sequences under the control of two inputs X_1 and X_2 as follows:

For inputs		The sequence is
X_1	X_2	
0	0	..., 00, 01, 10, 11, 00, ...
0	1	..., 11, 10, 01, 00, 11, ...
1	0	..., 10, 11, 01, 00, 10, ...
1	1	..., 01, 11, 10, 00, 01, ...

The inputs X_1 and X_2 can affect the count sequence at any point during the sequence.

- (a) Draw a state diagram for this counter.
- (b) Design and implement this counter. Use J-K flip-flops.

5.29. Determine the count sequence for the counter of Fig. 5.41. Also, draw a state diagram for it.

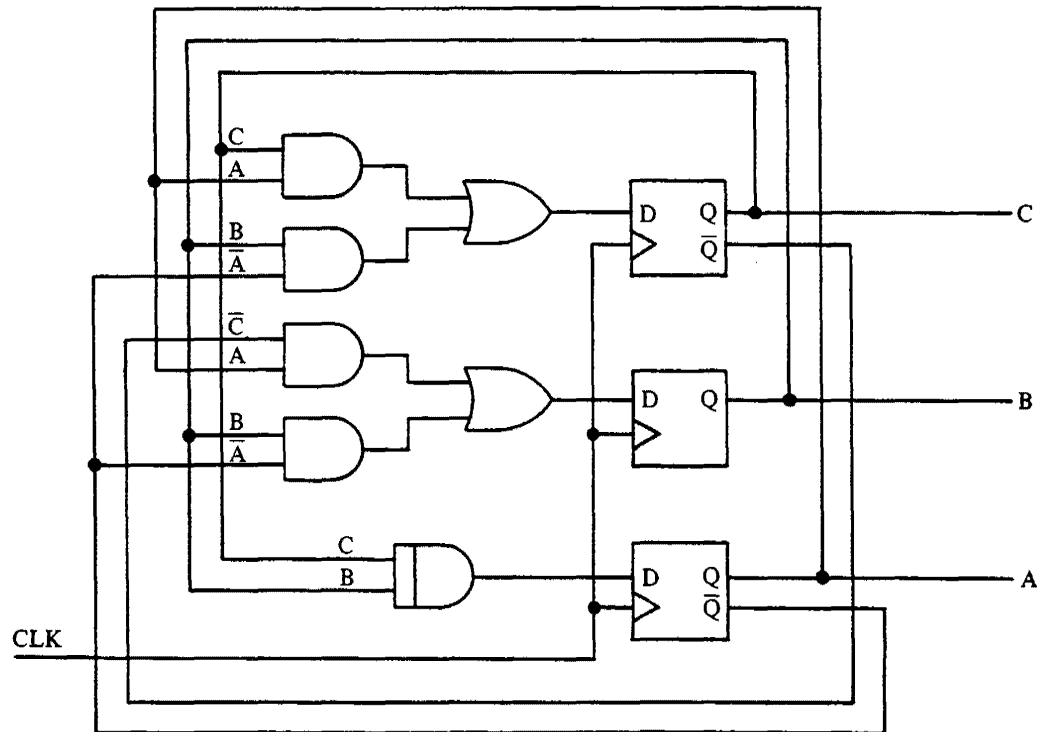


Figure 5.41 Counter circuit for Problem 5.29.

5.30. Design the 4-bit binary down-counter shown in Fig. 5.42 using a 74'163 plus any gates that are necessary. This counter is to have the same features as the 74'163 except for the count sequence, which is as follows:

..., 0000, 1111, 1110, 1101, 1100, 1011, ..., 0010, 0001, 0000, 1111, ...

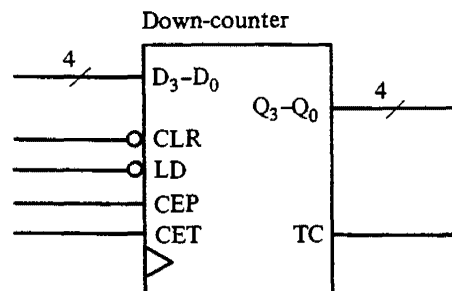


Figure 5.42 Down-counter for Problem 5.30.

- 5.31. Show that the 4-bit binary down-counter of Fig. 5.43 can be realized with only a 74'163. No additional components are needed.

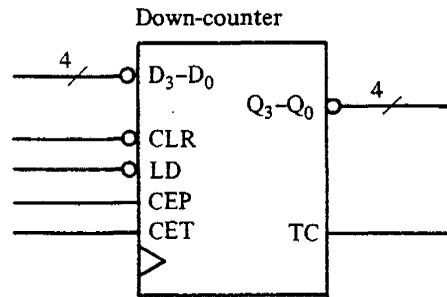


Figure 5.43 Down-counter for Problem 5.31.

- 5.32. Using a 74'163 and any gates that are needed, realize a decade counter similar to the 74'162.
- 5.33. Design and implement the counter circuit of Fig. 5.44 using a 74'163 and any gates that are needed. This circuit is to be used to determine if an event has occurred ten or more times. It is synchronously cleared when the CLR input is equal to true or when the count has reached 1111. This counter circuit will count only if it detects a true value at the EVENT input at an active clock transition. The (COUNT \geq 10) output is true only when the count is greater than or equal to 1001.

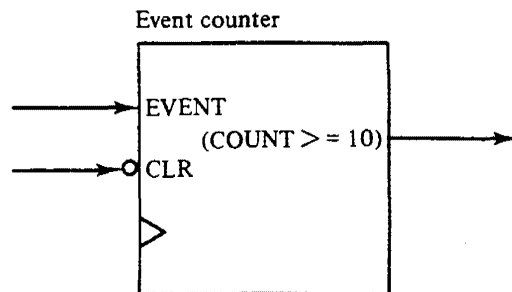
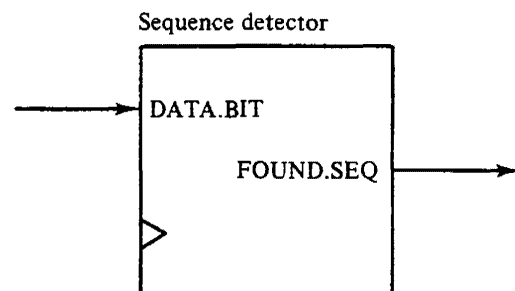
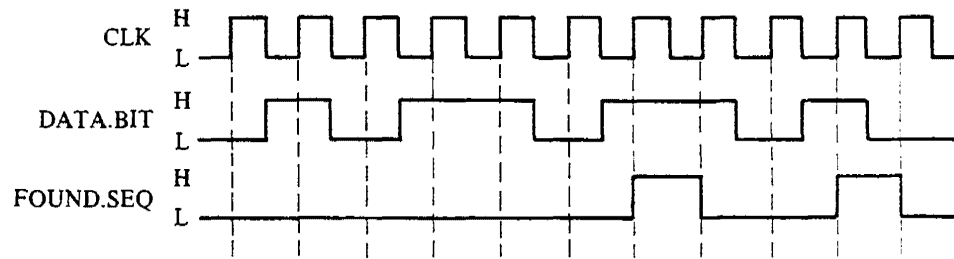


Figure 5.44 Counter circuit for Problem 5.33.

- 5.34. Show what must be done to the 74'194 to transform it into a 4-bit storage register.
- 5.35. Using a 74'194 and any gates that are needed, design and implement the sequence detector circuit shown in Fig. 5.45(a). The circuit input is a sequence of 1-bit values (either a 0 or a 1) detected at each active transition of the clock signal. When the circuit detects the sequence 1101, the output FOUND.SEQ becomes true for one clock cycle, as shown in the timing diagram of Fig. 5.45(b).



(a)



(b)

Figure 5.45 Sequence detector and timing diagram for Problem 5.35.

- 5.36.** By cascading two 74'194 shift registers, construct an 8-bit universal bidirectional shift register.

LSI Circuit Elements

6.1 INTRODUCTION

Recent rapid advances in large-scale integration technology have resulted in a larger and larger number of standard logic functions being integrated on a single chip. A single LSI (large-scale integrated) chip can now perform a number of logic functions that formerly required an entire circuit board of MSI (medium-scale integrated) and SSI (small-scale integrated) circuit elements.

There are, of course, significant advantages in the use of the LSI circuit elements instead of SSI or MSI. For a given number of logic functions, the use of the LSI circuit elements results in a reduction in the IC package count. This translates into a reduction of power consumption as well as of the PC (printed circuit) board space that is required. A reduction in the number of IC packages also results in an increase in reliability since the number of interconnections, a major source of failure, is reduced. The end result is a reduction in the total cost of the digital product. Less than half the cost of a digital product is in the actual purchase price of the ICs. Most of the cost relates to PC board area, assembly, and the testing associated with the product, all of which are reduced with the use of LSI circuit elements.

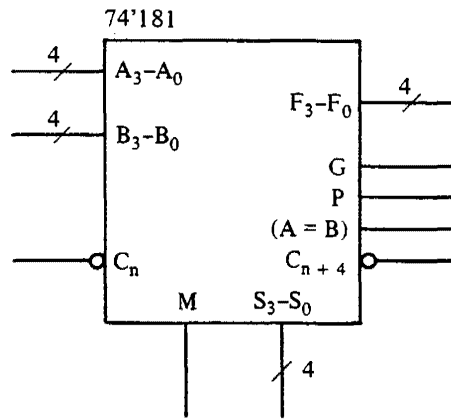
A less obvious but more important advantage of using higher-density circuit elements is the decrease in design complexity. In the top-down design approach to be presented in Chapter 7, the design process naturally leads into the use of high-level logic functions that are realized as LSI and MSI circuit elements. This design, being systematic and less complex, results in an increase in the ease of realization, testability, and maintainability of the digital circuits.

In the preceding two chapters common MSI circuit elements were presented. In this chapter we will conclude the presentation of the digital building blocks with the study of the commonly used LSI circuit elements, including the arithmetic logic unit (ALU), the look-ahead carry adder, the programmable logic array (PLA), and the programmable array logic (PAL). Also presented in this chapter are the various types of

random access memories, including the read-write memories (static and dynamic RAMs) and the read-only memories (ROM, PROM, and EPROM). The currently most important LSI circuit element, the microprocessor, is omitted from this chapter, but it is the major topic, along with microprocessor-based design, of the second half of this book.

6.2 ARITHMETIC LOGIC UNIT

An arithmetic logic unit (ALU) is a combinational circuit element that performs a set of commonly used arithmetic and logic operations. A representative example of a commercially available ALU is the 74'181, a block diagram of which is shown in Fig. 6.1(a). The corresponding functional description is given by the table of Fig. 6.1(b).

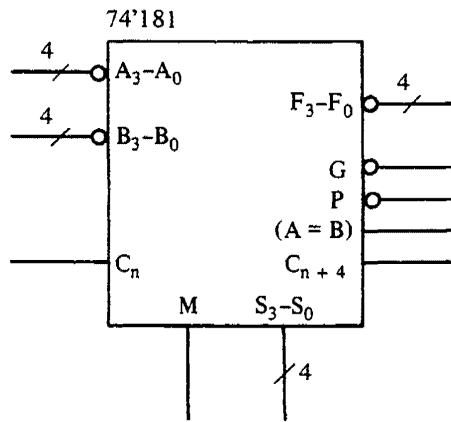


(a) Active-high view

Selection				Active-high data		
				M = H logic functions	M = L; Arithmetic operations	
S3	S2	S1	S0		C _n = H (no carry)	C _n = L (with carry)
L	L	L	L	$F = \bar{A}$	$F = A$	$F = A \text{ PLUS } 1$
L	L	L	H	$F = \overline{A + B}$	$F = A + B$	$F = (A + B) \text{ PLUS } 1$
L	L	H	L	$F = \bar{A}B$	$F = A + \bar{B}$	$F = (A + \bar{B}) \text{ PLUS } 1$
L	L	H	H	$F = 0$	$F = \text{MINUS } 1 \text{ (2's COMP)}$	$F = \text{ZERO}$
L	H	L	L	$F = \overline{AB}$	$F = A \text{ PLUS } \bar{A}\bar{B}$	$F = A \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$
L	H	L	H	$F = \bar{B}$	$F = (A + B) \text{ PLUS } \bar{A}\bar{B}$	$F = (A + B) \text{ PLUS } \bar{A}\bar{B} \text{ PLUS } 1$
L	H	H	L	$F = A \oplus B$	$F = A \text{ MINUS } B \text{ MINUS } 1$	$F = A \text{ MINUS } B$
L	H	H	H	$F = \overline{AB}$	$F = \bar{A}\bar{B} \text{ MINUS } 1$	$F = \bar{A}\bar{B}$
H	L	L	L	$F = \overline{A + B}$	$F = A \text{ PLUS } AB$	$F = A \text{ PLUS } AB \text{ PLUS } 1$
H	L	L	H	$F = A \oplus \bar{B}$	$F = A \text{ PLUS } B$	$F = A \text{ PLUS } B \text{ PLUS } 1$
H	L	H	L	$F = B$	$F = (A + \bar{B}) \text{ PLUS } AB$	$F = (A + \bar{B}) \text{ PLUS } AB \text{ PLUS } 1$
H	L	H	H	$F = AB$	$F = AB \text{ MINUS } 1$	$F = AB$
H	H	L	L	$F = 1$	$F = A \text{ PLUS } A$	$F = A \text{ PLUS } A \text{ PLUS } 1$
H	H	L	H	$F = A + \bar{B}$	$F = (A + B) \text{ PLUS } A$	$A = (A + B) \text{ PLUS } A \text{ PLUS } 1$
H	H	H	L	$F = A + B$	$F = (A + \bar{B}) \text{ PLUS } A$	$F = (A + \bar{B}) \text{ PLUS } A \text{ PLUS } 1$
H	H	H	H	$F = A$	$F = A \text{ MINUS } 1$	$F = A$

(b) Functional description of the active-high view

Figure 6.1 The 74'181 ALU.



(c) Active-low view

Selection					Active-low data		
					M = H logic functions	M = L; Arithmetic operations	
S3	S2	S1	S0	C _n = L (no carry)		C _n = H (with carry)	
L	L	L	L	$F = \bar{A}$	F = A MINUS 1	F = A	
L	L	L	H	$F = \bar{A}\bar{B}$	F = AB MINUS 1	F = AB	
L	L	H	L	$F = \bar{A} + B$	F = $\bar{A}\bar{B}$ MINUS 1	F = $\bar{A}\bar{B}$	
L	L	H	H	F = 1	F = MINUS 1 (2's COMP)	F = ZERO	
L	H	L	L	$F = \overline{A + B}$	F = A PLUS (A + \bar{B})	F = A PLUS (A + \bar{B}) PLUS 1	
L	H	L	H	F = B	F = AB PLUS (A + \bar{B})	F = AB PLUS (A + \bar{B}) PLUS 1	
L	H	H	L	$F = \overline{A \oplus B}$	F = A MINUS B MINUS 1	F = A MINUS B	
L	H	H	H	$F = A + \bar{B}$	F = A + \bar{B}	F = (A + \bar{B}) PLUS 1	
H	L	L	L	F = $\bar{A}\bar{B}$	F = A PLUS (A + B)	F = A PLUS (A + B) PLUS 1	
H	L	L	H	$F = A \oplus B$	F = A PLUS B	F = A PLUS B PLUS 1	
H	L	H	L	F = B	F = $\bar{A}\bar{B}$ PLUS (A + B)	F = $\bar{A}\bar{B}$ PLUS (A + B) PLUS 1	
H	L	H	H	F = A + B	F = (A + B)	F = (A + B) PLUS 1	
H	H	L	L	F = 0	F = A PLUS A	F = A PLUS A PLUS 1	
H	H	L	H	F = $\bar{A}\bar{B}$	F = $\bar{A}\bar{B}$ PLUS A	F = $\bar{A}\bar{B}$ PLUS A PLUS 1	
H	H	H	L	F = AB	F = $\bar{A}\bar{B}$ PLUS A	F = $\bar{A}\bar{B}$ PLUS A PLUS 1	
H	H	H	H	F = A	F = A	F = A PLUS 1	

(d) Functional description of the active-low view

Figure 6.1 (cont.)

The primary inputs to this ALU are two 4-bit operands: A_3-A_0 and B_3-B_0 . The ALU performs some operation on these operands to produce a 4-bit output F_3-F_0 . The specific operation that is performed depends on the function selection inputs, S_3-S_0 , as well as on the mode control input M. As shown in the table of Fig. 6.1(b), if $M = 1$ (H), then one of the 16 logic operations is performed, the particular one depending on the values of S_3-S_0 . If $M = 0$ (L), however, then one of the 32 predominantly arithmetic operations is performed, the particular one depending on the values of S_3-S_0 and C_n .

The input C_n is the carry-in input for arithmetic operations. And C_{n+4} is the carry-out output. The output (A = B) is for magnitude comparison operations. It is true (H)

when A_3-A_0 is equal to B_3-B_0 , and is false (L) otherwise. The group-generate output G and the group-propagate output P will be discussed in following sections.

With mixed logic, a second view of the 74'181 ALU, with active-low operands, is possible. The block diagram of it is shown in Fig. 6.1(c). As specified by the corresponding functional description of Fig. 6.1(d), this view gives rise to a new set of arithmetic and logic operations.

From Figs. 6.1(b) and (d), it is evident that the 74'181 performs all the commonly used arithmetic and logic operations, along with some not so common ones. The 74'181 provides a powerful building block and an economic means for the design of digital circuits that require a variety of these operations.

6.3 LOOK-AHEAD CARRY CIRCUITS FOR ADDERS AND ALUs

As was explained in Sec. 4.2.2 of Chapter 4, parallel adders may require excessive amounts of time for adding operations. Consider the 4-bit parallel adder of Fig. 4.3, which is reproduced in Fig. 6.2 for convenience. Since the carry-out of each full-adder stage is connected to the carry-in of the next stage, the carry-in for each stage is not stable until the preceding stage produces a stable carry-out output. For example, the carry-in for Stage 1 is not stable until Stage 0 produces a stable output at C_1 . Similarly, the carry-in to Stage 2 is not stable until Stage 1 produces a stable output at C_2 , and so forth. In this manner, the carry "ripples" down the chain of full adders. Consequently, after the inputs are applied to an N -bit ripple adder, the outputs do not become stable until a time equal to $N \times t_p(\text{FA})$, in which $t_p(\text{FA})$ is the propagation delay of a full-adder stage. If N is large, say 64 bits, then the time for the carry to propagate to the last stage can be substantial.

Since the addition operation is a fundamental arithmetic operation upon which other arithmetic operations are frequently based, it is extremely desirable to optimize its performance and reduce the time for the carry-in of each adder stage to become stable. The most common technique to accomplish this is with *look-ahead carry circuits*. Recall from Sec. 4.2.1 that the equations for the outputs for the full adder at each adder stage are

$$S_i = A_i \oplus B_i \oplus C_i \quad \text{and} \quad C_{i+1} = A_i B_i + (A_i \oplus B_i) C_i$$

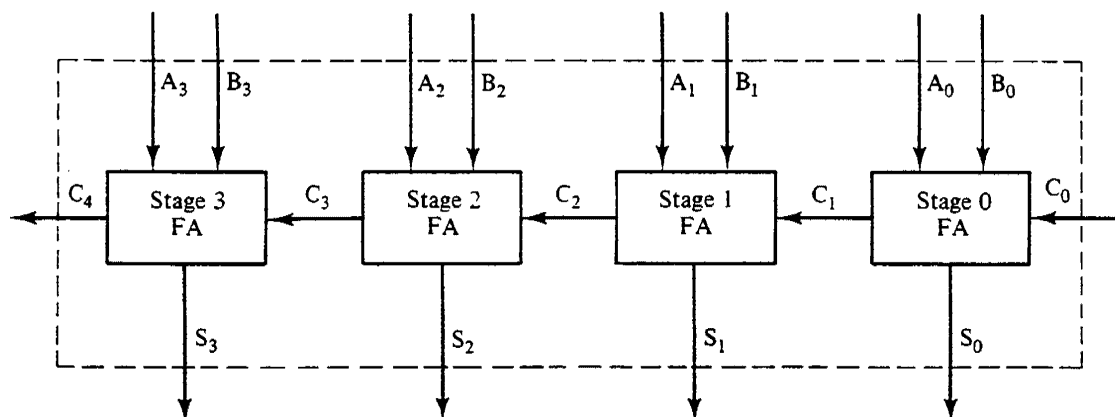


Figure 6.2 Parallel adder circuit diagram.

Let us now define the following variables:

$$G_i = A_i B_i \quad \text{and} \quad P_i = A_i \oplus B_i$$

With the substitution of these variables, the C_{i+1} equation becomes

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

The variable G_i is called the *carry-generate* for the i th adder stage. It is the logic AND of the two input bits to that stage, A_i and B_i . Its significance is that a carry-out is *generated* by Stage i if $G_i = 1$ (i.e., $A_i = 1$ and $B_i = 1$), regardless of what transpires in the adder stages preceding Stage i . In other words, if $G_i = 1$, then $C_{i+1} = 1$ regardless of the value of C_i . For example,

Stage	··· (i + 1)	(i)	(i - 1)	··· (0)
C	1	X	X	··· X
A		1	X	··· X
B		1	X	··· X
Sum		X	X	··· X

where X = don't care

The variable P_i is called the *carry-propagate* for the i th adder stage. It is the Exclusive OR of the two input bits A_i and B_i to that stage. Consequently, if $P_i = 1$ (i.e., $A_i = 1$ and $B_i = 0$, or $A_i = 0$ and $B_i = 1$), then the carry-in C_i for this stage will be *propagated* to the carry-out C_{i+1} . In other words, if $P_i = 1$, then $C_{i+1} = C_i$, as is illustrated by the following examples:

Stage	··· (i + 1)	(i)	(i - 1)	··· (0)
C	0	0	X	··· X
A		1	X	··· X
B		0	X	··· X
Sum		X	X	··· X

Stage	··· (i + 1)	(i)	(i - 1)	··· (0)
C	1	1	X	··· X
A		1	X	··· X
B		0	X	··· X
Sum		X	X	··· X

Using this modified C_{i+1} equation, we can calculate the values of the carries at each adder stage:

$$C_1 = G_0 + P_0 C_0$$

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

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

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

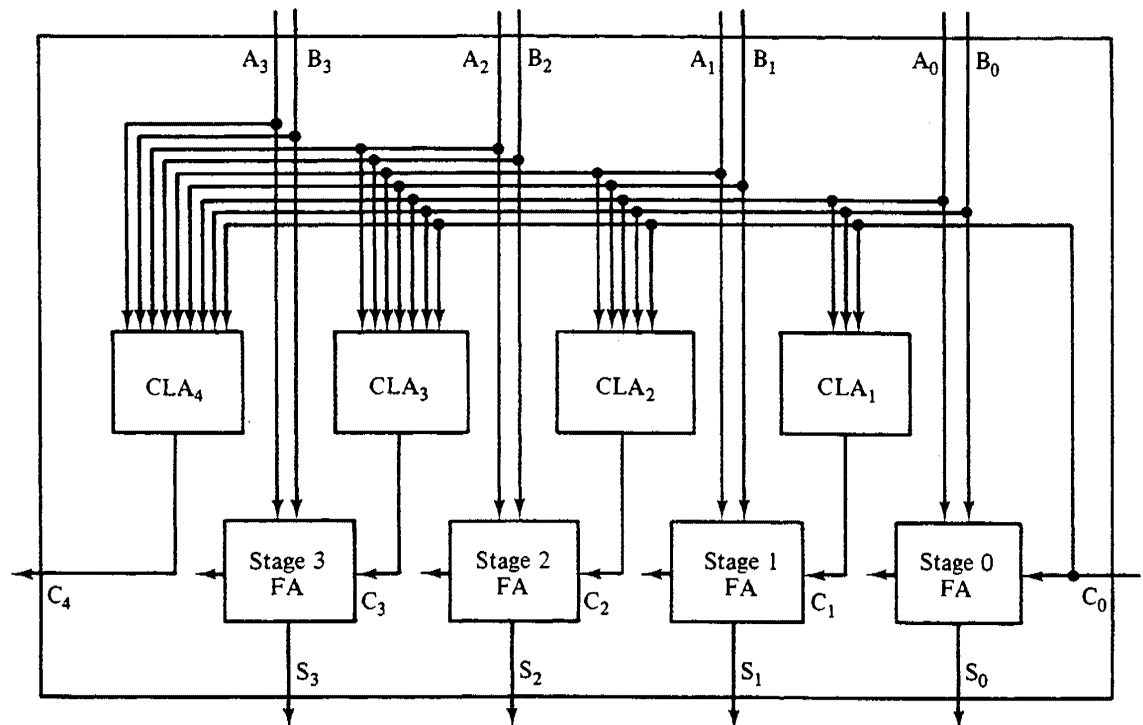
In general,

$$\begin{aligned} C_n &= G_{n-1} + P_{n-1}C_{n-1} \\ &= G_{n-1} + P_{n-1}G_{n-2} + P_{n-1}P_{n-2}G_{n-3} + \cdots + P_{n-1}P_{n-2}P_{n-3} \cdots P_0C_0 \end{aligned}$$

Note that C_0 is the only carry that appears in this equation. Consequently, the carry-in of any adder stage in an N -bit parallel adder can be determined from only the A_i and B_i inputs and C_0 . Thus, to calculate the value of any C_n we do not need the value of C_{n-1} , C_{n-2} , or of any of the preceding carries except C_0 . Clearly, then, it is possible to design a parallel adder in which the carry-in of any adder stage does not have to wait for the carry to be propagated down the adder chain.

In terms of hardware, each C_i can be realized as a combinational circuit consisting of two levels of AND-OR gates, as shown in Fig. 6.3. This circuit is called a *look-ahead carry circuit*. After the inputs have been applied, the outputs of an N -bit adder with look-ahead carry circuitry will be stable at a time equal to $t_p(\text{CLA}) + t_p(\text{FA})$, *independently* of the value of N . Here, $t_p(\text{FA})$ is again the propagation delay of a full adder, and $t_p(\text{CLA})$ is the propagation delay of the two-level AND-OR look-ahead carry circuit.

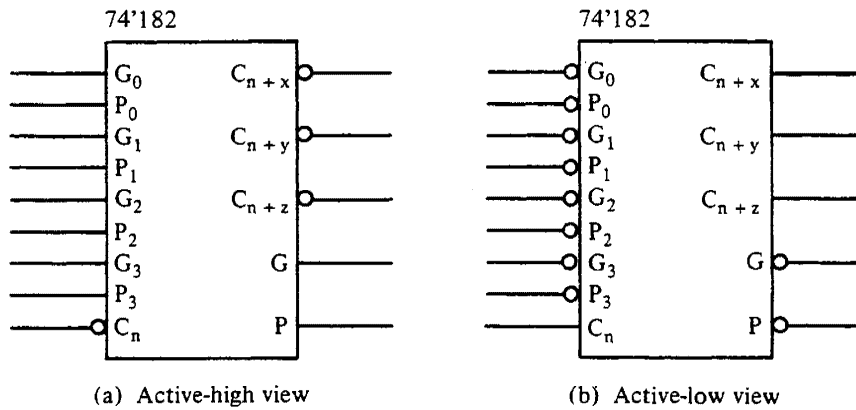
Look-ahead carry circuits based on the above discussion are frequently used in commercially available ICs. For example, both the 74'83 and 74'283 MSI 4-bit adders, described in Sec. 4.2.2, have look-ahead carry circuitry for faster addition. The 74'181 ALU, described in the preceding section, also has the look-ahead carry feature. Fur-



$$\begin{aligned} C_1 &= G_0 + P_0C_0 \\ C_2 &= G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \\ C_4 &= G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

in which $G_i = A_iB_i$ and $P_i = A_i \oplus B_i$

Figure 6.3 4-bit adder with carry look-ahead circuitry.



$$\begin{aligned}
 C_{n+x} &= G_0 + P_0 C_n \\
 C_{n+y} &= G_1 + P_1 G_0 + P_1 P_0 C_n \\
 C_{n+z} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_n \\
 G &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \\
 P &= P_3 P_2 P_1 P_0
 \end{aligned}$$

(c) Functional description

Figure 6.4 The 74'182 look-ahead carry generator.

thermore, look-ahead carry circuits themselves are commercially available as ICs. An example is the 74'182 look-ahead carry generator. Two functional block diagrams for it are shown in Figs. 6.4(a) and (b), and the corresponding functional description is given in Fig. 6.4(c). The inputs to the look-ahead carry generator are the various carry-generates and carry-propagates (G_i 's and P_i 's) and the carry-in (C_n) for the group of adders. The outputs from it are the carries (C_{n+x} , C_{n+y} , and C_{n+z}), the group-generate (G), and the group-propagate (P). The use of the 74'182 look-ahead carry generator will be illustrated in the next section.

6.3.1 Modified Look-Ahead Carry Approaches

In the look-ahead carry scheme, the price that is paid for the performance improvement is the additional hardware, which consists of the look-ahead carry circuit for each adder stage. As is evident from Fig. 6.3, for each successive adder stage, the look-ahead carry circuit becomes more complex and can quickly become unmanageable. As an illustration, the look-ahead carry circuit for Stage 15 requires 33 inputs. Clearly, to handle look-ahead carry for larger multibit adders, the look-ahead carry scheme needs to be modified.

One approach is to have a scheme that is a combination of look-ahead carry and ripple carry. This approach is illustrated by the 16-bit adder shown in Fig. 6.5. In this approach, look-ahead circuitry is used *within* each of the 4-bit adder groups, but carries are rippled *between* each group. The propagation delay for this 16-bit adder is equal to $4 \times t_p(\text{adder group})$, in which $t_p(\text{adder group})$ is the time required for an adder group to produce the sum and the carry-out for that group. Although this delay time is greater than that of a 16-bit adder that employs full look-ahead carry across all 16 bits, the hardware requirement is far less. Also, this delay time is still substantially less than that of a 16-bit ripple adder without any look-ahead carry circuitry.

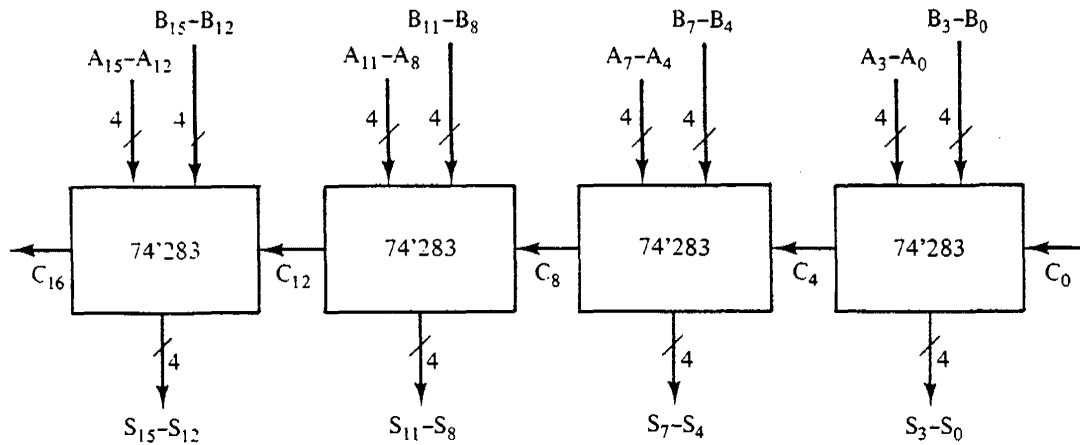


Figure 6.5 16-bit adder with look-ahead carry and ripple carry.

Another approach is to have a multilevel look-ahead carry scheme. Let us define the following variables for a multibit adder group (here they are defined only for a 4-bit adder group for simplicity of explanation):

$$G(\text{group}) = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

$$P(\text{group}) = P_3P_2P_1P_0$$

The variable $G(\text{group})$ is the *group-generate* for a multibit adder group. In general, it is some logic function of the individual carry-generates (G_i 's) and carry-propagates (P_i 's) within that group. This definition is such that a carry-out, $C_{\text{out}}(\text{group})$, is *generated* out of this multibit adder group if $G(\text{group}) = 1$, regardless of what transpires in the adder groups preceding this group. So if $G(\text{group}) = 1$, then $C_{\text{out}}(\text{group}) = 1$. The variable $P(\text{group})$ is the *group-propagate* for this multibit adder group. It is the logic AND of the individual carry propagates within that group. If $P(\text{group})$ is equal to 1, then the carry-in for this adder group is propagated to the carry-out of the adder group. In other words, if $P(\text{group}) = 1$, then $C_{\text{out}}(\text{group}) = C_{\text{in}}(\text{group})$. From what has been stated, it is evident that the $G(\text{group})$ and $P(\text{group})$ variables are defined such that

$$C_{\text{out}}(\text{group}) = G(\text{group}) + P(\text{group})C_{\text{in}}(\text{group})$$

For the 74'181 ALU of Fig. 6.1, the group-generate G and the group-propagate P outputs are produced in this manner. Using these group-generate and group-propagate outputs with look-ahead carry generators, such as the 74'182 shown in Fig. 6.4, we can connect a number of 74'181 ALUs in a multilevel look-ahead scheme. Shown in Fig. 6.6 is a 16-bit ALU constructed in this manner. Note that in performing the addition operation each of the 74'181 ALUs cannot produce stable outputs until its respective C_n input is stable. This 16-bit ALU functions as follows: Given that the two 16-bit operands and the C_{in} are applied, look-ahead circuitry within each 74'181 produces the group-generate and group-propagate outputs after a delay of $t_p(181PG)$. Then, it takes the 74'182 look-ahead carry generator a delay of $t_p(182)$ to produce C_{n+x} , C_{n+y} , and C_{n+z} . At this point in time, the C_n for each of the 74'181s is stable. Therefore, the outputs for the 16-bit ALU become stable after another delay of $t_p(181ADD)$, which is the time required for the 74'181 to produce the sum. So, the total propagation delay for the 16-bit ALU for the addition operation is $t_p(181PG) + t_p(182) + t_p(ADD)$.

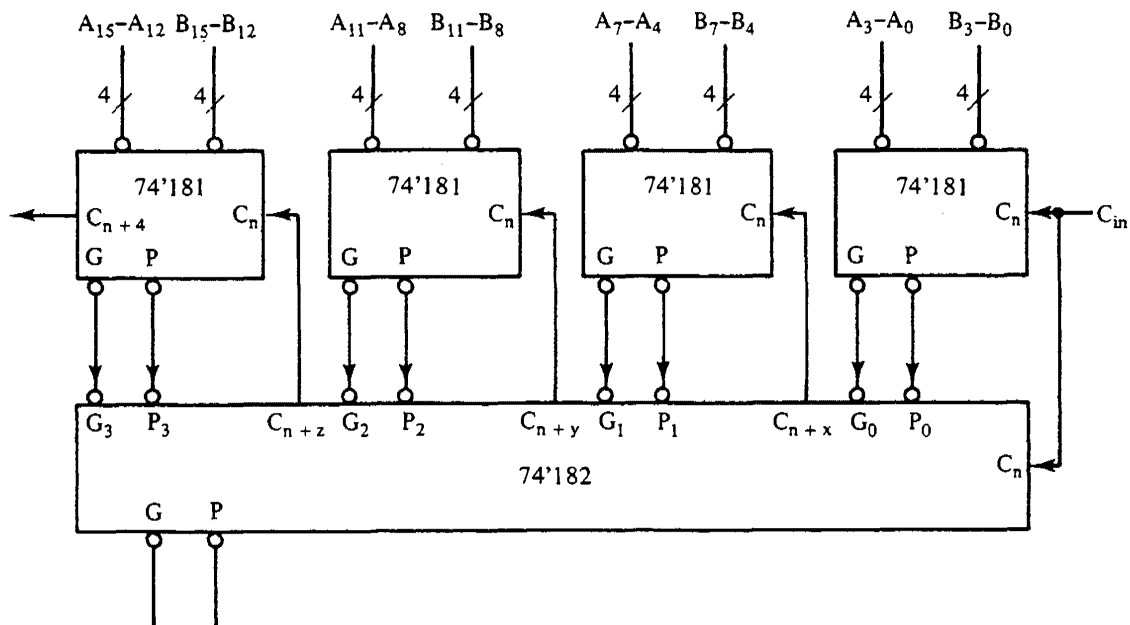


Figure 6.6 16-bit ALU with multilevel look-ahead carry structures.

As shown in Fig. 6.7, additional levels of look-ahead carry generators can be used to realize larger multibit ALUs. What is the propagation delay for this 64-bit ALU for the addition operation? (See Problem 6.8.) In general, with this multilevel look-ahead carry scheme, the addition operation delay is determined by the propagation delays through the levels of look-ahead circuitry. Certainly this delay is less than for the combinational approach of Fig. 6.5, where the carry is rippled down a chain of ALUs. Also, although the delay is somewhat greater, the hardware required is far less than that of an N -bit adder that employs full look-ahead carry across all N bits.

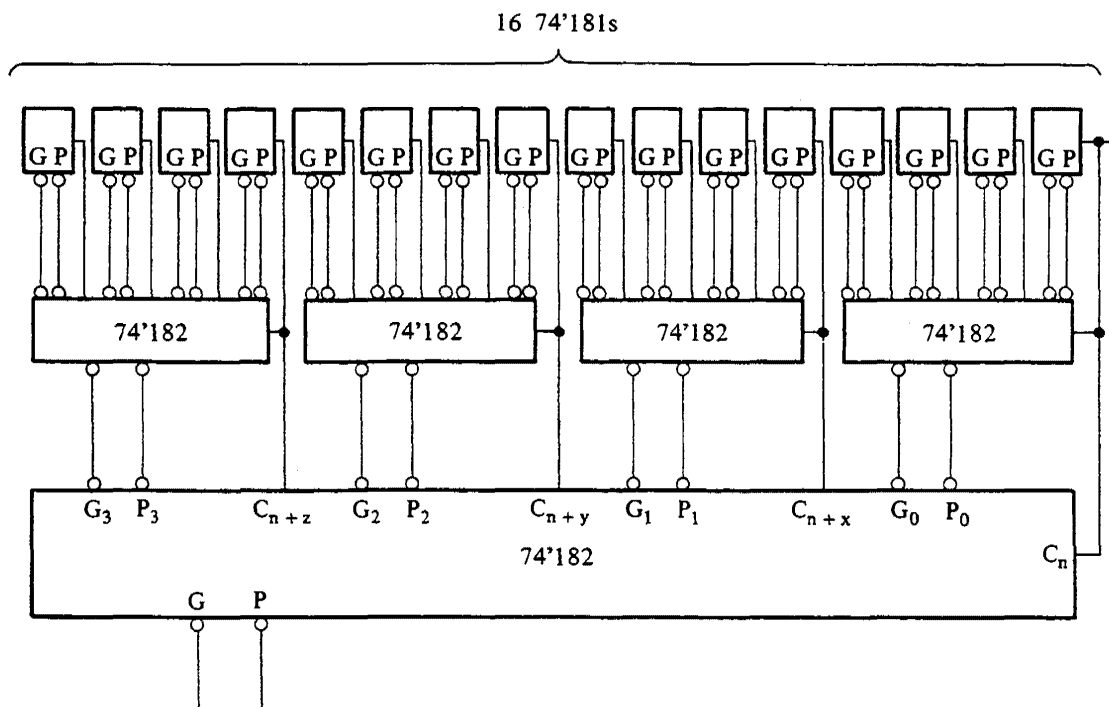


Figure 6.7 A 64-bit ALU with multilevel look-ahead carry structures.

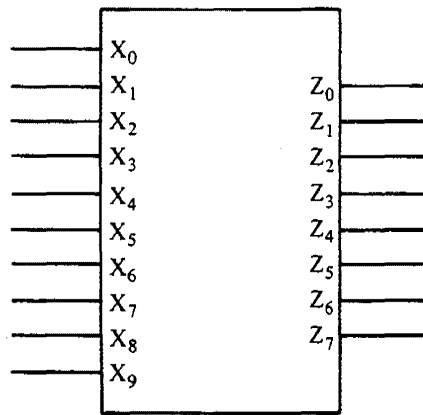
This discussion of the various approaches to look-ahead carry provides a good illustration of the typical design trade-off between performance and hardware complexity. Which approach to be taken is, of course, a function of the performance requirements of the particular digital circuit along with other constraints such as cost, and also the quantities of the parts that are to be manufactured.

6.4 PROGRAMMABLE LOGIC ARRAY (PLA) AND PROGRAMMABLE ARRAY LOGIC (PAL)

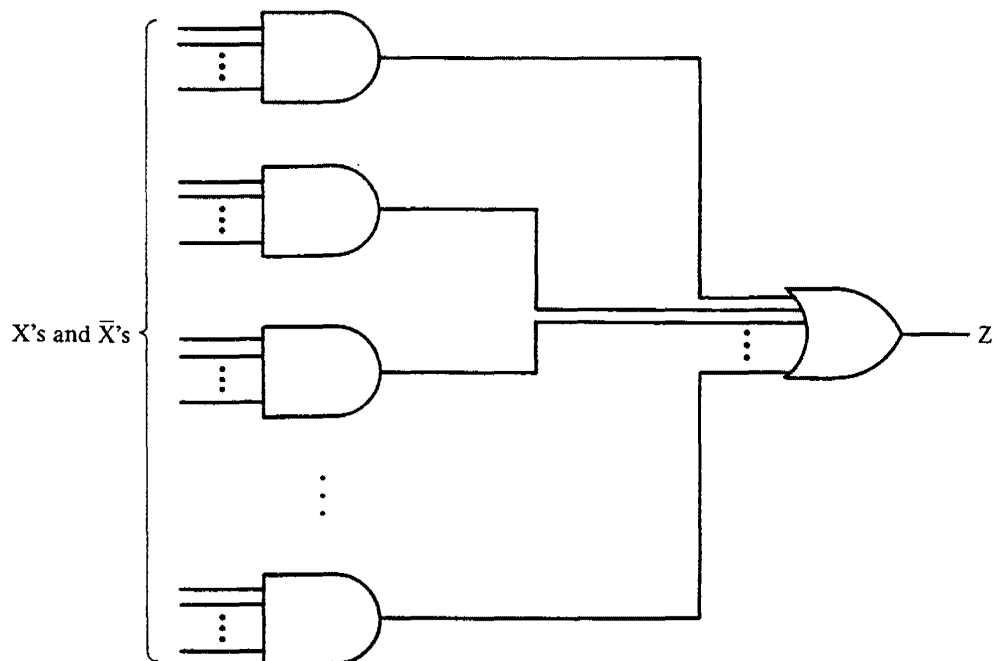
Consider a digital circuit with ten inputs and eight outputs, as shown in Fig. 6.8(a). Suppose that each of the outputs is some combinational function of the ten inputs:

$$Z_i = f(X_0, X_1, X_2, \dots, X_9)$$

A typical output, Z_i , can be realized with a two-level, sum-of-products, AND-OR gate structure, as shown in Fig. 6.8(b). Consequently, the entire circuit of Fig. 6.8(a) can be



(a) Functional block diagram



(b) Two-level AND-OR structure

Figure 6.8 Conventional realization of a multiinput and multioutput circuit.

realized with eight of these structures. If this circuit is realized with discrete SSI circuit elements, such as the AND and OR gates of Chapter 3, then the package count for the circuit would be substantial. Fortunately, there are attractive alternatives.

In this section we will study two circuit elements, programmable logic arrays (PLAs) and programmable array logic devices (PALs), that can be used as alternatives to discrete logic circuit elements for the realizations of multi-input, multioutput combinational circuits. Conceptually, PLAs and PALs are straightforward circuit elements that simply realize sum-of-products gate structures in a systematic manner. The power of the PLAs and PALs is that a large number of these sum-of-products structures can be integrated on a *single* IC.

6.4.1 Programmable Logic Array

As is illustrated by the one shown in Fig. 6.8(b), logic diagrams are convenient for representing small logic functions. They can, however, become cumbersome for the large logic functions that are typically used with PLAs. Consequently, it is desirable to devise a shorthand notation to simplify logic diagrams for use with PLAs. The notation that will be used here has been adopted by IC manufacturers.

Shown in Fig. 6.9 are three common logic diagram representations and their equivalent PLA representations. In the PLA logic diagram of Fig. 6.9(a), the AND gate for

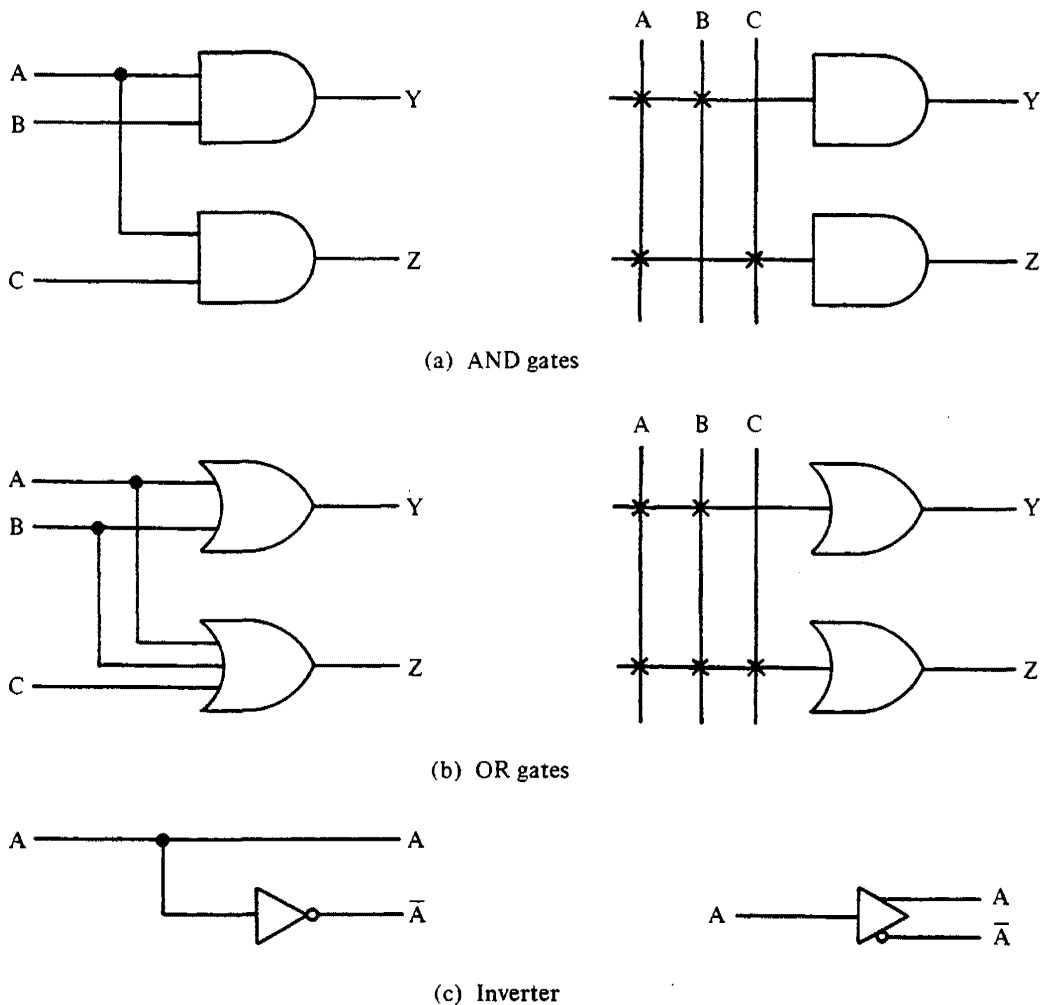
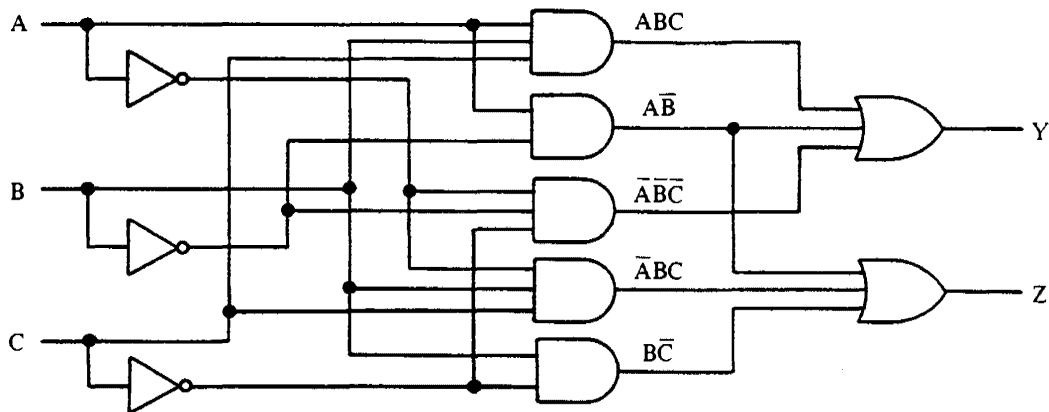


Figure 6.9 Equivalent PLA logic diagrams.

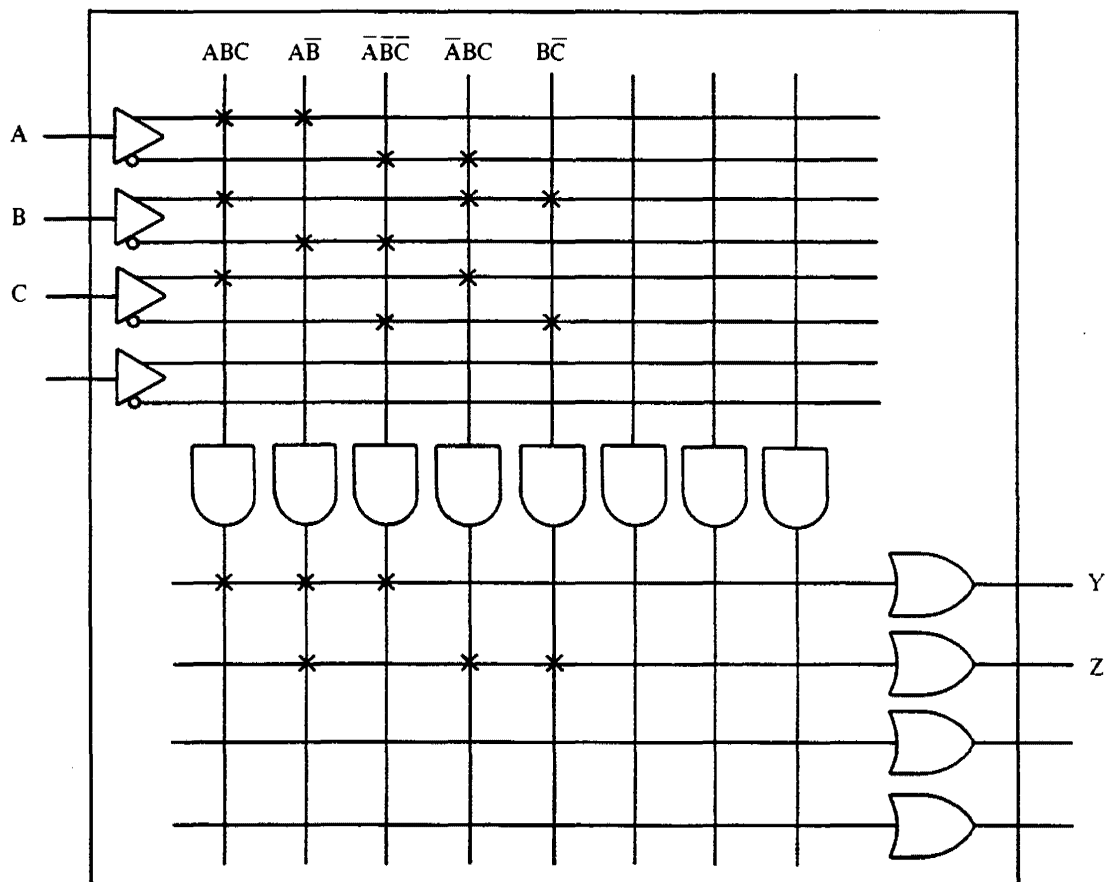
output Y has three inputs. But only two of the inputs are connected to A and B , as indicated by the two 'x's,' and one input is left unused (absence of an 'x'). Similarly, the OR gates in the PLA logic diagram in Fig. 6.9(b) have three inputs, and an x indicates a connection to an input. The PLA diagram of Fig. 6.9(c) for an inverter is self-explanatory.

Example 6.1 PLA Equivalent of a Two-Level AND-OR Circuit

Using the notation just described for the circuit diagram of Fig. 6.10(a), we can obtain the equivalent PLA circuit diagram of Fig. 6.10(b). Note that the full capacity of the



(a) Common circuit diagram



(b) PLA circuit diagram

Figure 6.10 PLA circuit diagram for Example 6.1.

PLA structure is not utilized. This PLA has four inputs, four outputs, and eight product terms (AND terms). Thus it can realize up to four logic functions (outputs of the OR gates), but only two of them, Y and Z, are used. Also, each of the OR gates can have up to eight inputs, and so OR up to eight product terms each. And, each of the AND gates can have up to eight inputs (actually four inputs or their complements). ■ ■

Example 6.2 PLA Realization of a Truth Table Specification

In this example, the PLA of Fig. 6.10(b) is used to realize a combinational logic circuit of four inputs and two outputs, the functional block diagram of which is shown in Fig. 6.11(a). The functional description of the circuit is given in Fig. 6.11(b) in the form of a truth table, and the corresponding PLA circuit diagram is in Fig. 6.11(c).

We can program the PLA by finding the minterms corresponding to 1s in the output columns of the truth table, and by making suitable connections (the x's), as should be apparent. And, we can associate these minterms with the AND gates which are numbered 0 through 7. Note that minterm 0 is $\bar{X}_0\bar{X}_1\bar{X}_2X_3$, and is used by OR gate 0, the output of which is Z_0 . Minterm 1 is $\bar{X}_0\bar{X}_1X_2\bar{X}_3$, and is used by both OR gate 0 (Z_0) and OR gate 1 (Z_1), and so forth with the other connections. With this approach we can program the PLA connections in a straightforward manner. Note from the truth table that the total number of minterms for the two functions Z_0 and Z_1 is ten. The number of *distinct* minterms, however, is only seven. Three of them (1, 2, and 4) are shared. Since the required number of minterms is fewer than that provided by the PLA, we can program the PLA connections directly from the truth table without the need of any reduction process. ■ ■

Example 6.3 PLA Realization That Requires Reduction

In this example, the same PLA is used to realize another combinational logic circuit. The functional block diagram of this circuit is shown in Fig. 6.12(a), and its truth table is given in Fig. 6.12(b). As is evident from the truth table, and unlike the circuit of Example 6.2, the total number of distinct minterms (12) exceeds the capacity of eight that is supported by this PLA. Consequently, we cannot program the PLA connections directly from the truth table, but first must reduce the number of product terms.

Using conventional minimization techniques, as shown in Fig. 6.12(c), we can reduce the number of distinct product terms to nine, but this number still exceeds the capacity of the PLA. As shown in Fig. 6.12(d), however, some of the K-map implicants [(a), (b), (c), and (d)] can be grouped such that they can be shared by both functions. With this grouping, the total number of distinct product terms is reduced to seven, and so within the capacity of the PLA. The resultant PLA circuit diagram is shown in Fig. 6.12(e). An important conclusion from this example is that it is more important to minimize the number of distinct product terms than to minimize the number of gates in the conventional sense. ■ ■

In general, a PLA has N inputs, M outputs, and supports K distinct product terms, as shown in Fig. 6.13. For a PLA realization of a combinational digital circuit, the number of circuit inputs must be less than or equal to N . Also, the number of circuit outputs must be less than or equal to M . Furthermore, if after reduction the number of distinct product terms still exceeds K , then a PLA with a larger capacity is required.

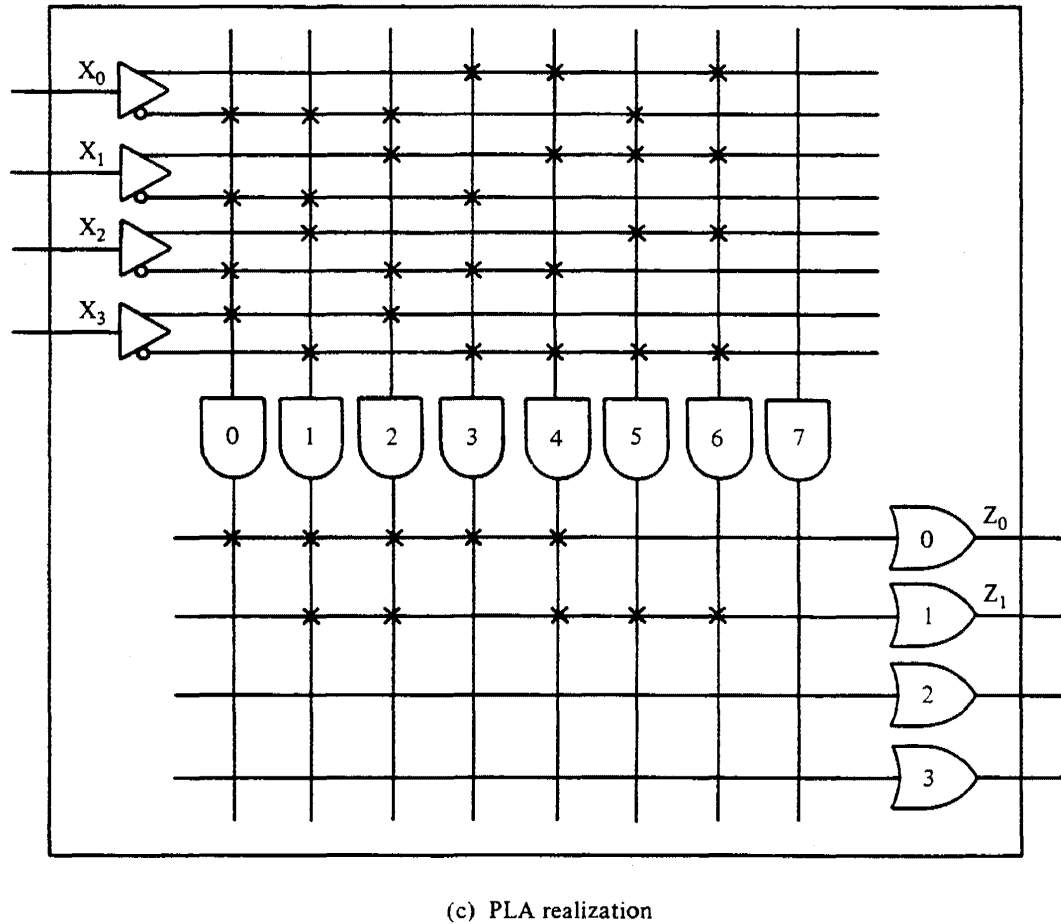
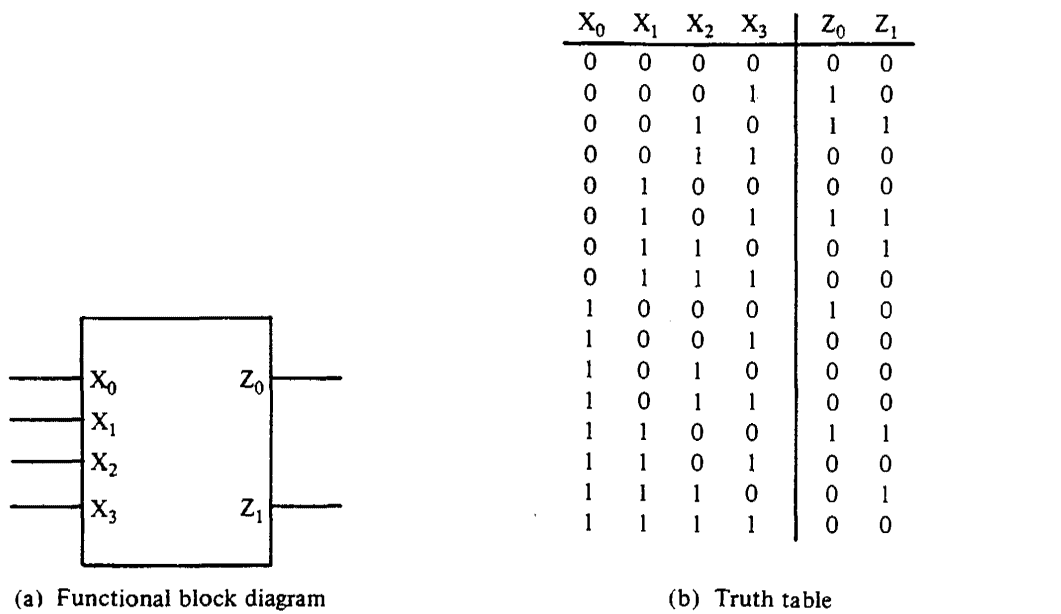
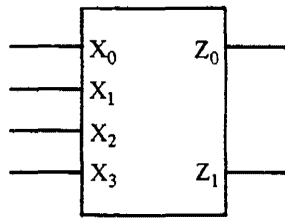


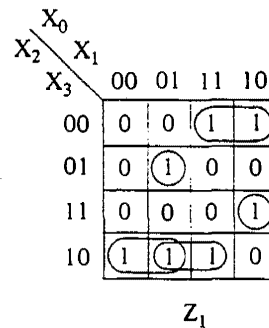
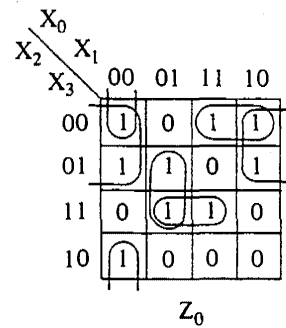
Figure 6.11 Illustration for Example 6.2.



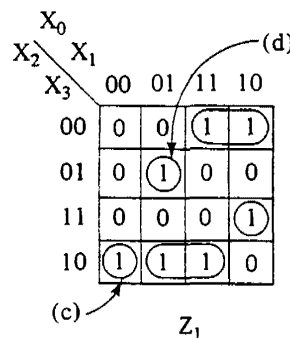
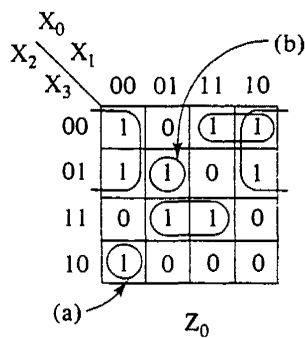
(a) Functional block diagram

X_0	X_1	X_2	X_3	Z_0	Z_1
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	1
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	0	1
1	1	1	1	1	0

(b) Truth table



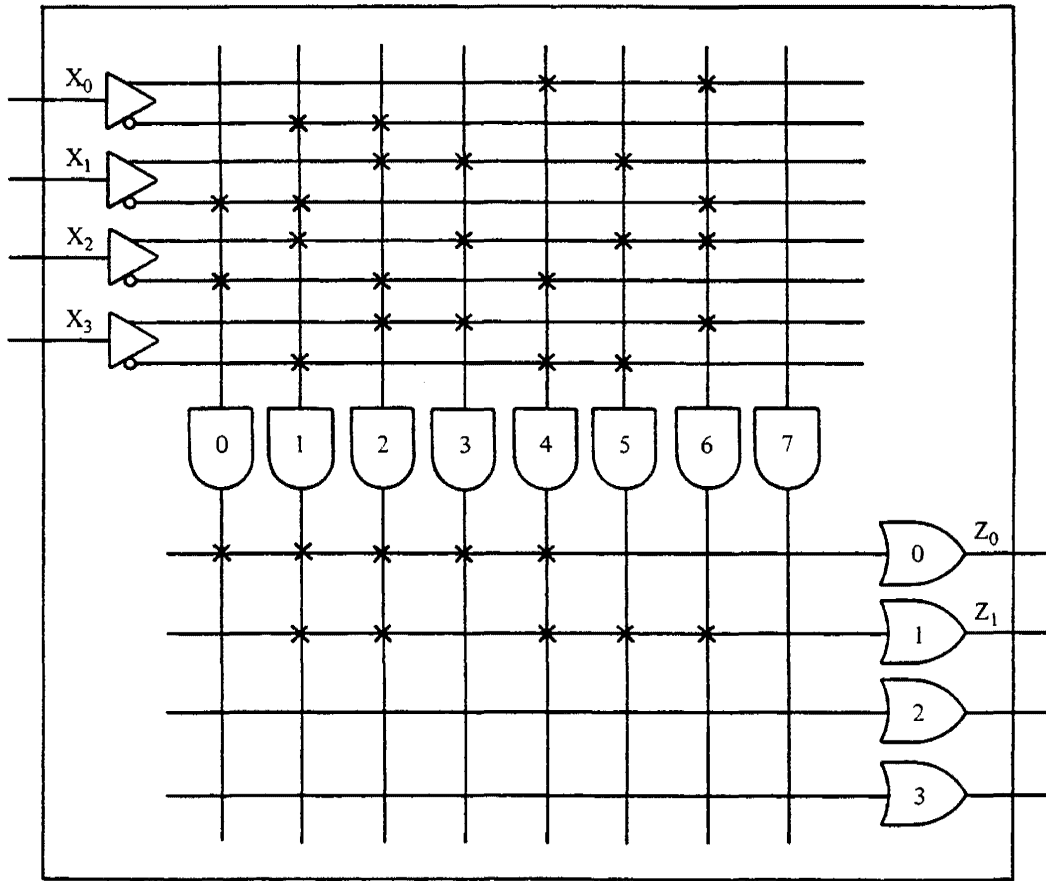
(c) Conventional minimization results in a total of nine distinct AND terms



$$\begin{aligned}
 Z_0 &= \bar{X}_1 \bar{X}_2 + \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 + \bar{X}_0 X_1 \bar{X}_2 X_3 + X_1 X_2 X_3 + X_0 \bar{X}_2 \bar{X}_3 \\
 Z_1 &= \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 + \bar{X}_0 X_1 \bar{X}_2 X_3 + X_0 \bar{X}_2 \bar{X}_3 + X_1 X_2 \bar{X}_3 + X_0 \bar{X}_1 X_2 X_3
 \end{aligned}$$

(d) Minimization for PLA realization – a total of seven distinct AND terms

Figure 6.12 Illustration for Example 6.3.



(e) PLA realization

Figure 6.12 (cont.)

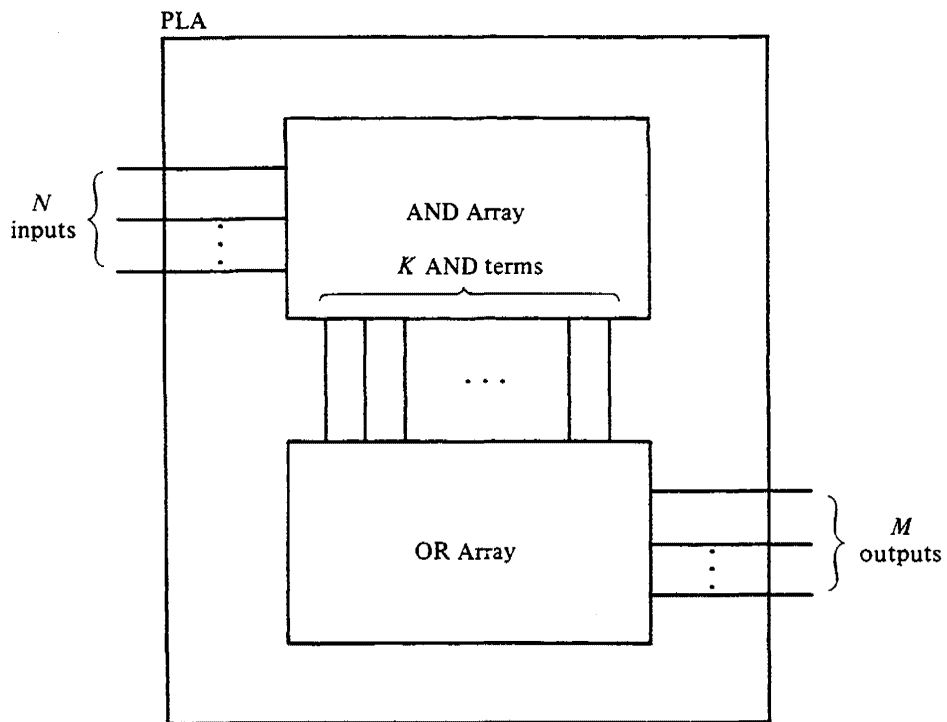
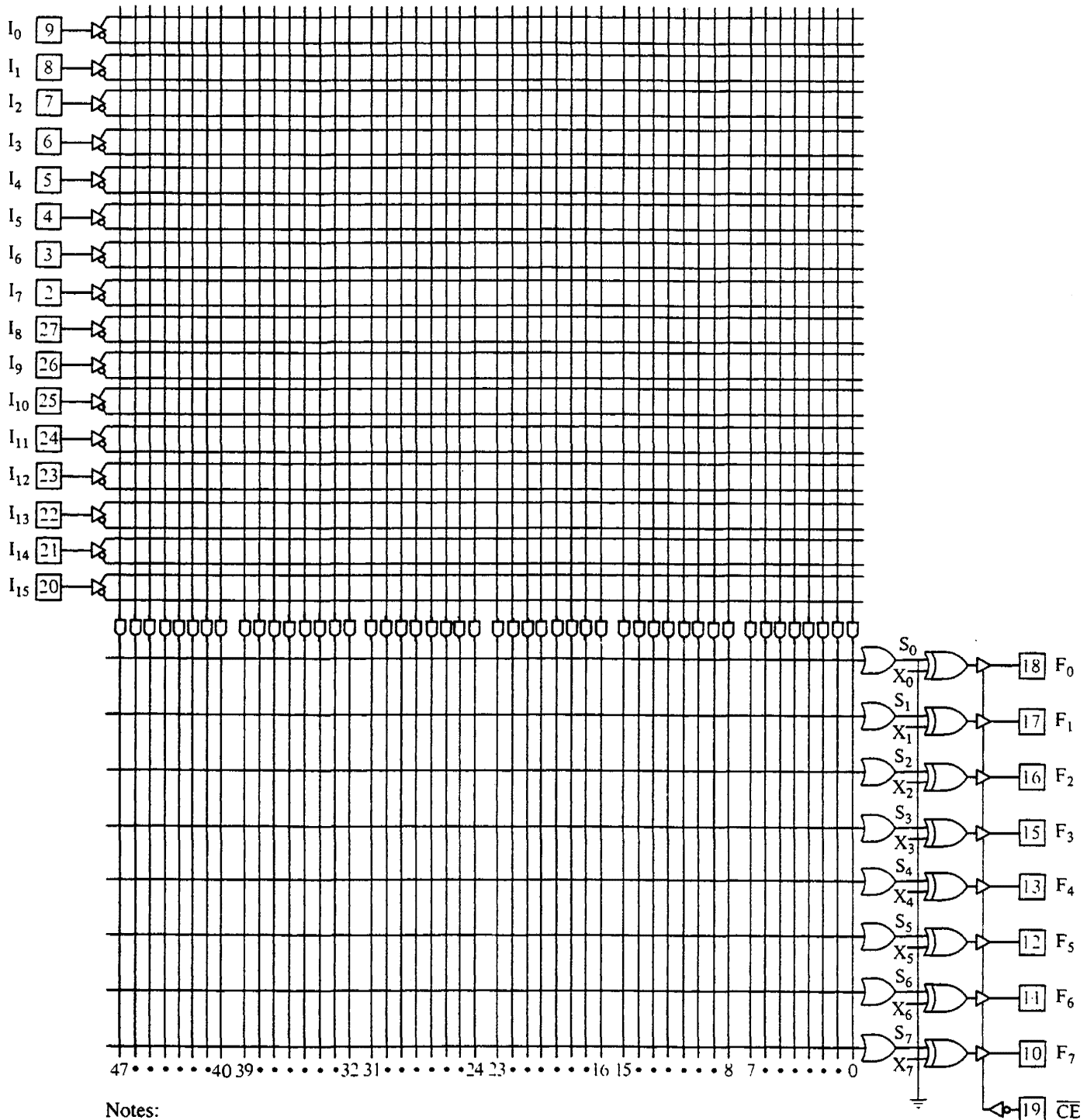


Figure 6.13 PLA with N inputs, M outputs, and K AND terms.

Commercially Available PLAs and FPLAs

Commercially available PLAs come in two forms: PLAs that are programmed by the IC manufacturer, and *field-programmable PLAs (FPLAs)* that can be programmed by users with FPLA programmers. An FPLA comes from the manufacturer with all the connections intact as integrated fuses. Using an FPLA programmer, a user can program an FPLA by leaving intact the desired connections (the x's in a PLA logic diagram) and blowing the fuses of the other unused connections.



Notes:

1. All AND/XOR gate inputs with a blown link float to a logic 1
2. All OR gate inputs with a blown link float to a logic 0

Figure 6.14 82S100 FPLA. (Courtesy of Signetics Corporation.)

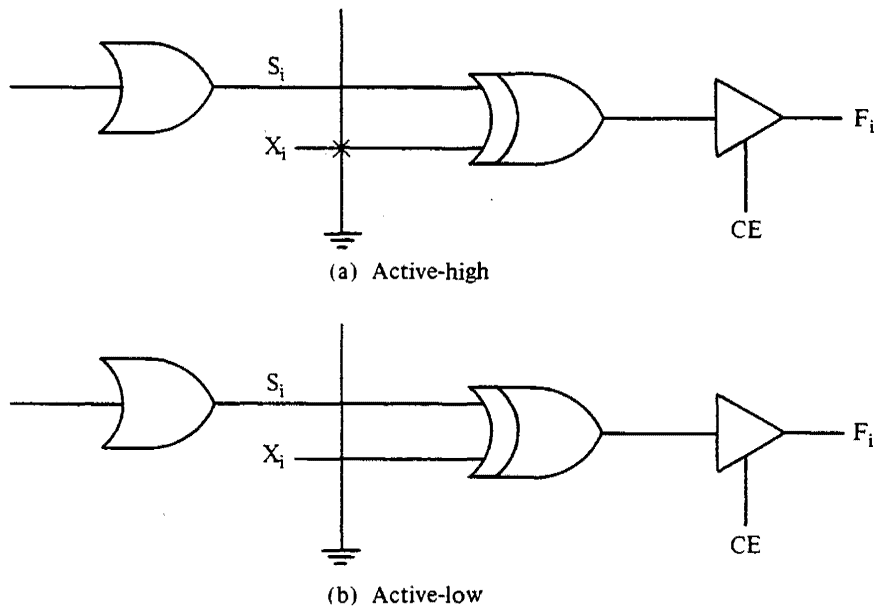


Figure 6.15 Programming the polarity of an FPLA output terminal.

A typical example of a commercially available FPLA is the 82S100 shown in Fig. 6.14. With its 16 inputs, 8 outputs, and support for up to 48 product terms, it is capable of replacing quite a few SSI circuit packages. In addition to the normal functions of a PLA, commercially available PLAs typically provide other functions as well. The 82S100, for example, provides a chip enable CE input (pin 19) that allows the outputs to be three-stated. Additionally, each output can be programmed to be active-high or active-low.

As shown in Fig. 6.15 for an 82S100, the programming of an output F_i to be active-high is obtained by leaving the fuse for X_i connected to ground as is graphically shown by the x . This ground provides a logic 0 to an input of the XOR gate, which also has an S_i input, thereby giving $S_i \oplus 0 = S_i$. On the other hand, for the programming of an output F_i to be active-low, the fuse for X_i is blown. This causes the input X_i to be left floating high, as shown in Fig. 6.15(b), thereby giving $S_i \oplus 1 = \overline{S_i}$.

Other features are also available in other commercially available PLAs. Included is the ability to program a terminal to be an input or an output terminal. Also, some PLAs have on-chip flip-flops for the realization of a single-chip “state machine,” such as those to be presented in Chapter 7.

6.4.2 Programmable Array Logic

The programmable array logic (PAL) also realizes sum-of-products gate structures in a systematic manner. It is a special case of the PLA, having a fixed-OR array instead of the PLA programmable OR array. Consequently, it is sometimes called a fixed-OR array. PALs are commercially available in various sizes for providing various functions. Shown in Fig. 6.16 is a relatively small PAL, the PAL14H4, which has 14 inputs, with internal inverters to provide the respective complements. It also has four active-high outputs, each from an OR gate that can accommodate only four product terms.

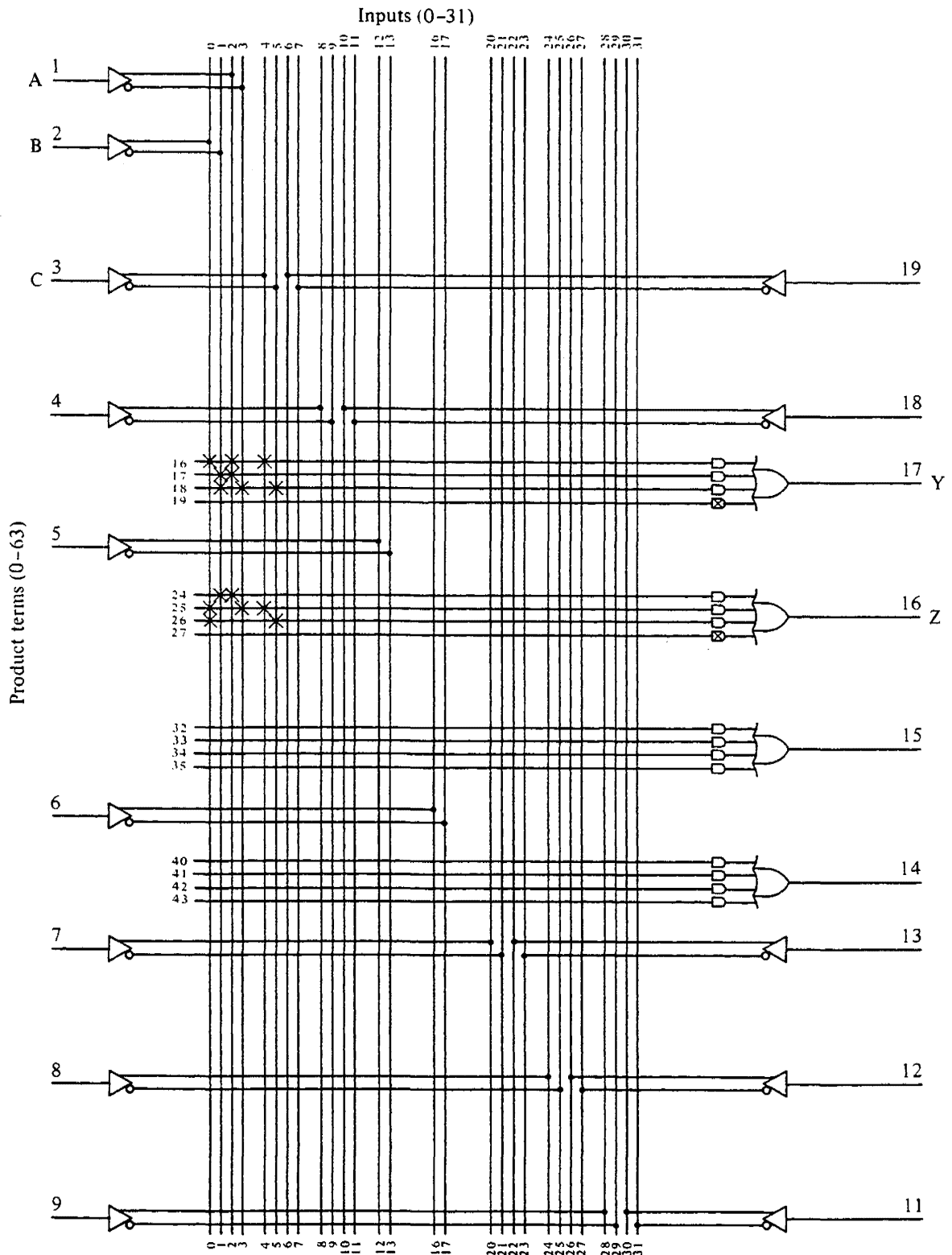
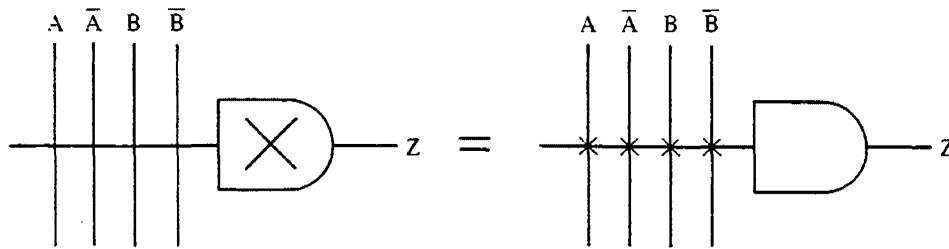


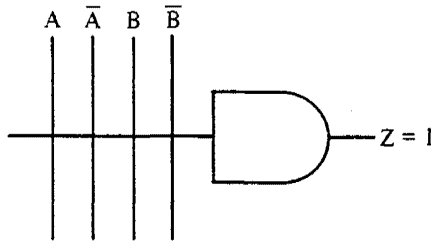
Figure 6.16 PAL1-H4 realization of $Y = ABC + \overline{A}\overline{B}C$ and $Z = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C}$ from Fig. 6.10(a) of Example 6.1.

Figure 6.16 also illustrates the use of a PAL. The shown x's provide the realization of the combinational circuit in Fig. 6.10(a) of Example 6.1. Note the x's in two of the AND gate symbols. When an input of an OR gate is *not* used, an x is graphically placed in the corresponding AND gate symbol. As shown in Fig. 6.17(a), the x is simply a shorthand notation to designate that all the inputs (including the complement values) are



$$Z = A \cdot \bar{A} \cdot B \cdot \bar{B} = 0$$

(a) All inputs are connected



$$Z = 1$$

(b) All inputs are disconnected and left floating high

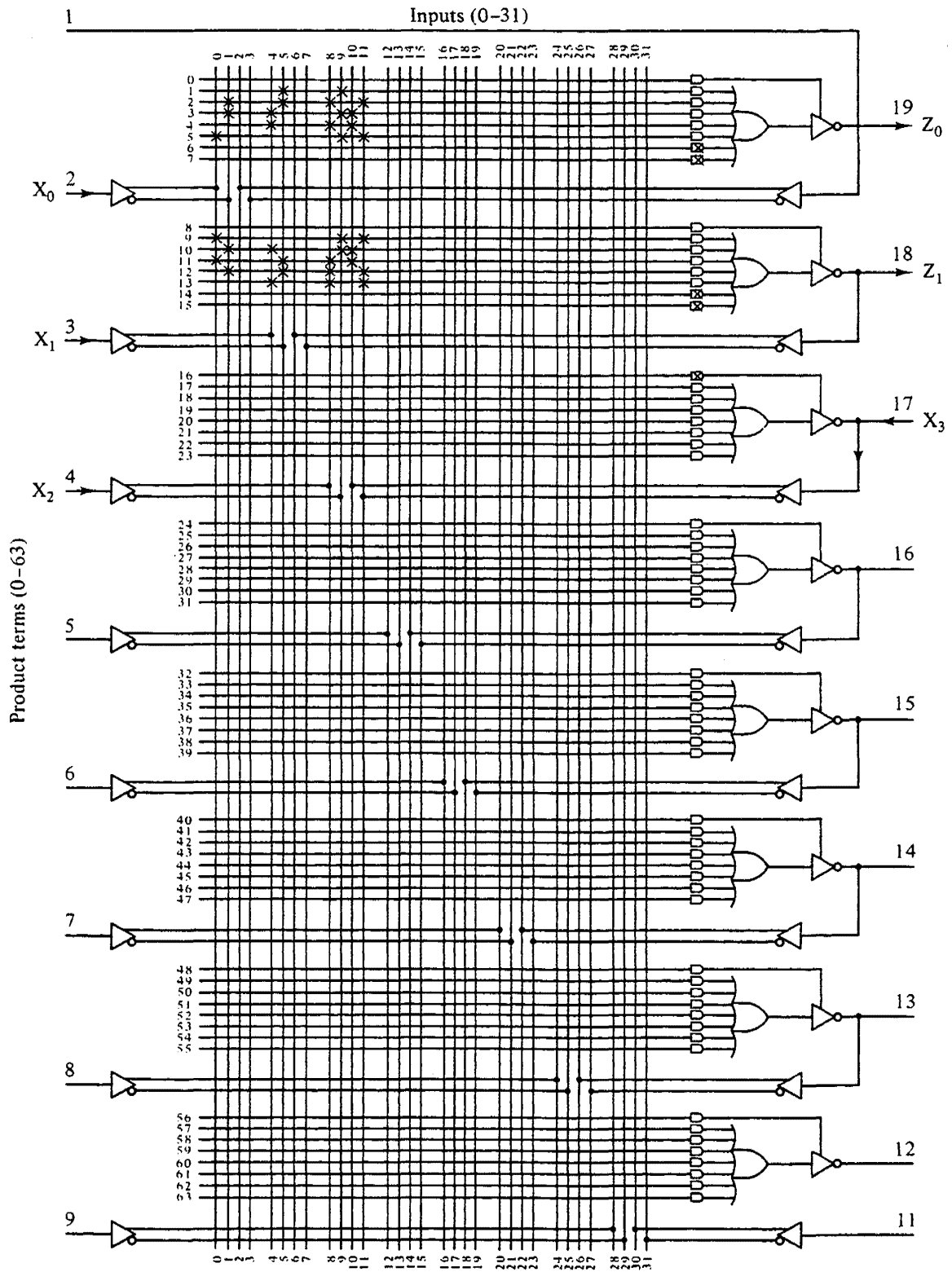
Figure 6.17 PAL shorthand notation.

left connected. As a result, the output of the AND gate is false (L) and so will not affect the function of the following OR gate. Note also that since the OR array is fixed (four AND gates are permanently assigned), the $A\bar{B}$ product term cannot be shared by the outputs, as it was in the PLA realization of Fig. 6.10(b). Instead, the product term $A\bar{B}$ has to be generated for both Y and Z.

In addition to providing the normal functions of a PAL, commercially available PALs typically provide other functions as well. Shown in Fig. 6.18 is the PAL16L8, which illustrates some of these additional functions. It has ten dedicated inputs (pins 1, 2, 3, 4, 5, 6, 7, 8, 9, and 11), and two dedicated active-low outputs (pins 12 and 19). Additionally, there are six I/O pins (pins 13–18), each of which can be programmed to be either an input or an output pin, as controlled by a three-state inverting buffer. Each output can accommodate up to seven product terms. With the programmable pins, the PAL16L8 can have up to 16 inputs and 2 outputs, or 10 inputs and 8 outputs, or any combination in between.

For an I/O pin to be programmed as an input, the output of the AND gate that controls the three-state buffer must be false (L). As is shown in Fig. 6.19(a), this is obtained by leaving connected all the inputs to that AND gate. For the programming of an I/O pin to be an output, the output of the controlling AND gate must be true (H). As is shown in Figs. 6.17(b) and 6.19(b), this is obtained by disconnecting all the inputs to that AND gate.

For an illustration of the use of a PAL, connections are shown in Fig. 6.18 for realizing the functions specified in the truth table in Fig. 6.12(b) of Example 6.3. The inputs X_0 , X_1 , and X_2 are assigned to dedicated input pins, but input X_3 is assigned to an I/O pin that is programmed as an input pin. The output Z_0 is assigned to a dedicated output pin, and output Z_1 is assigned to an I/O pin that is programmed to be an output pin.

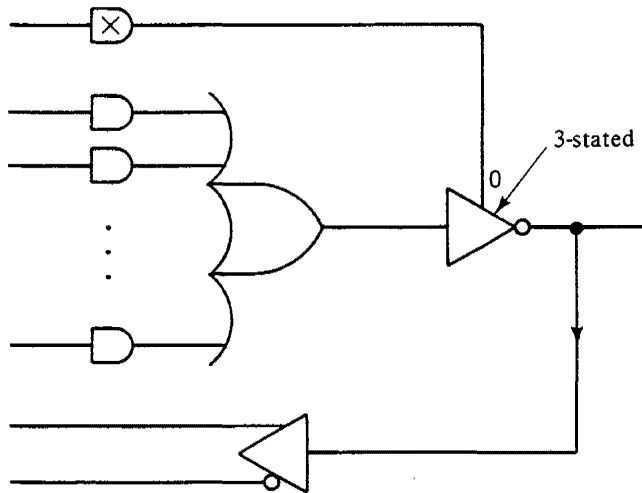


From Fig. 6.12(c):

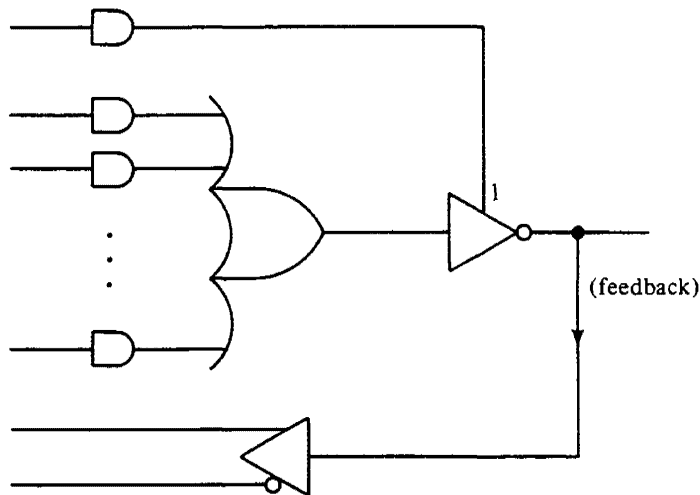
$$Z_0 = \bar{X}_1 \bar{X}_2 + \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 + \bar{X}_0 X_1 \bar{X}_2 X_3 + X_1 X_2 X_3 + X_0 \bar{X}_2 \bar{X}_3$$

$$Z_1 = X_0 \bar{X}_2 \bar{X}_3 + \bar{X}_0 X_1 \bar{X}_2 X_3 + X_0 \bar{X}_1 X_2 X_3 + \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 + X_1 X_2 \bar{X}_3$$

Figure 6.18 PAL16L8 realization of Example 6.3.



(a) PAL I/O terminal programmed as an input



(b) PAL I/O terminal programmed as an output

Figure 6.19 Programming of an I/O terminal of a PAL.

Observe that Z_0 and Z_1 are active-low outputs. If active-high outputs are required, there are three ways of obtaining them. External inverters can be used to change the polarity of the outputs. Alternatively, DeMorgan's laws can be used to convert the logic functions to obtain $\overline{Z_0}$ and $\overline{Z_1}$, and then these can be realized with the PAL. [Recall that $\overline{Z.L}$ is equal to $Z.H$. (See Problem 6.14.)] Finally, of course, another comparable PAL with active-high outputs can be used.

In summary, a PAL can be used to realize sum-of-products expressions in a systematic manner, and it is a special case of the PLA. Since the number of product terms for a PAL is fixed to a limited number for each output, a PAL is more restrictive in use than a PLA. However, when applicable, a PAL is less expensive and is generally easier to program than a PLA. These attributes make the PAL an attractive alternative to discrete SSI gates as the basic components of a digital system. According to PAL manufacturers, a single PAL package can realize the equivalent logic of 4 to 12 SSI and MSI packages.

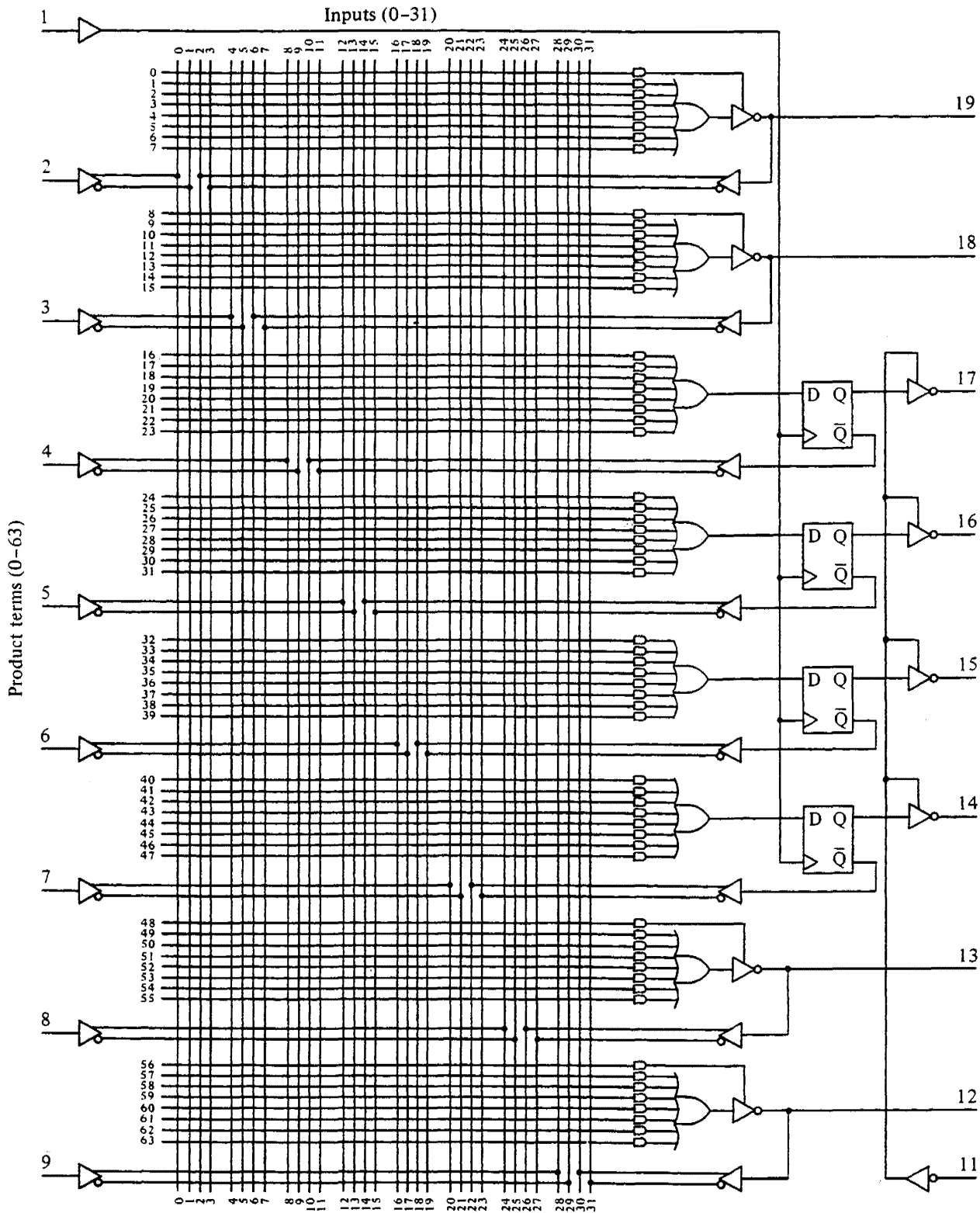


Figure 6.20 Logic diagram of the PAL16R4.

Additionally, other PALs, such as the PAL16R8, the PAL16R6, and the PAL16R4, have on-chip flip-flops along with the PAL arrays, as is shown in Fig. 6.20 for the PAL16R4. With one of these it is possible with a single IC to realize a state machine, such as one of those that are discussed in Chapter 7.

6.5 MEMORIES

Memories are circuit elements that are used in digital circuits for storing large amounts of information. A general model of a memory circuit element is shown in Fig. 6.21(a). Conceptually, it is a collection of 2^n addressable storage registers, each of which contains m bits. Associated with each storage register, which is called a *memory location*, is a unique *memory address*. As shown in Fig. 6.21(a), the address of the first memory location is 0, that of the second memory location is 1, and so forth up to the last memory location which has an address of $2^n - 1$. With the specification of an n -bit address at the ADDRESS inputs, the contents of any of the 2^n memory locations can be accessed directly (i.e., randomly), without the need to sequentially traverse the preceding locations to get to the specified location. For this reason, this type of memory is commonly called a *random access memory*. The m -bit data is transferred to and from a memory location through m bidirectional (input/output) DATA lines of the memory unit.

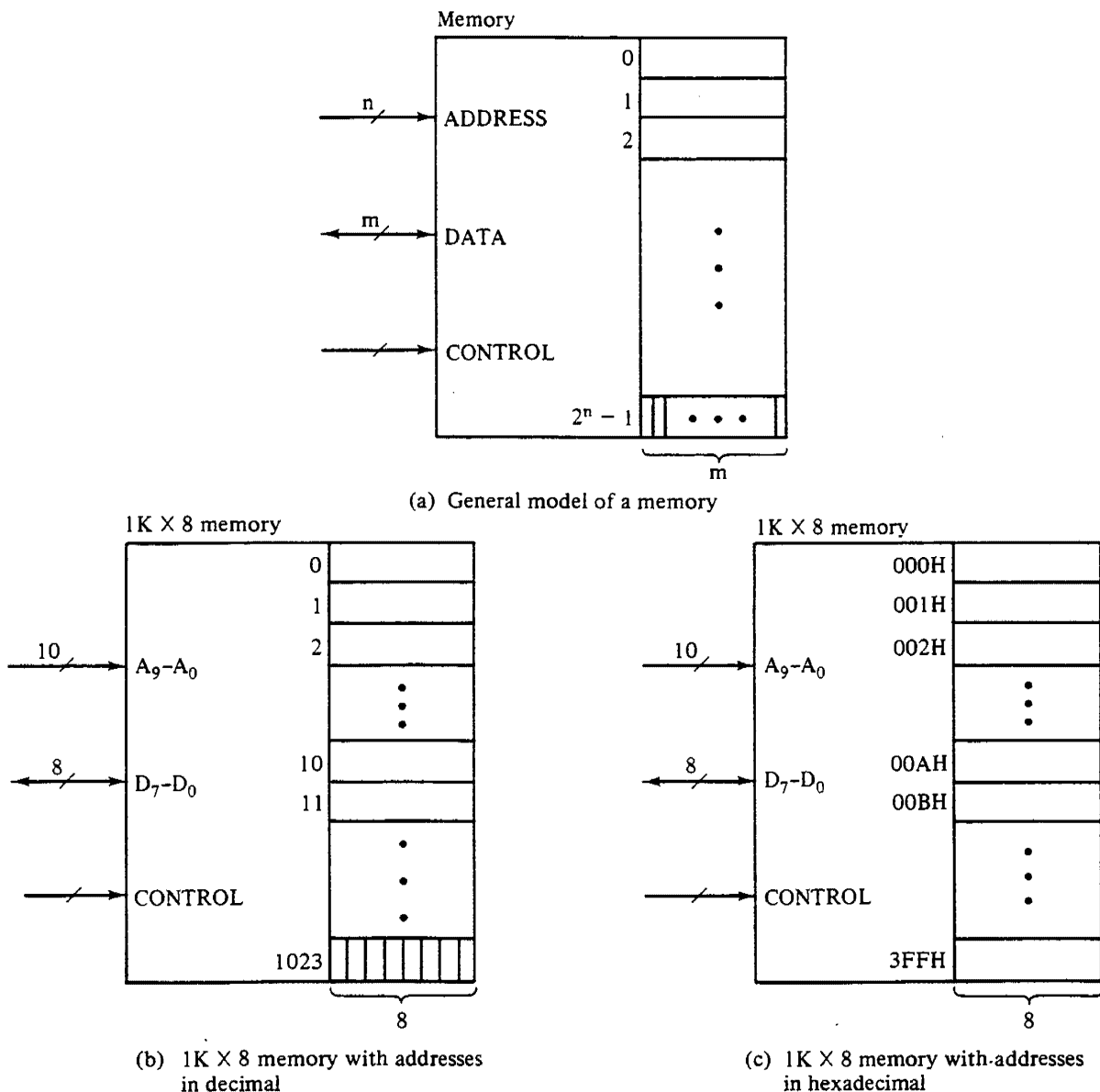


Figure 6.21 Models of memories.

In general, two types of memory operations can be performed: memory *read* and memory *write*. For a read operation, data is retrieved from one of the memory locations by specifying an n -bit address at the ADDRESS inputs, and applying appropriate control signals at the CONTROL inputs to cause the contents to be read from the specified location. After a time equal to the access time of the memory, the m -bit contents of that memory location are available on the DATA outputs. For a write operation, data is stored into one of the memory locations by specifying an n -bit address at the ADDRESS inputs and applying the m -bit data to be stored at the DATA inputs. At the same time, appropriate control signals are applied at the CONTROL inputs to cause the m -bit data to be stored at the specified memory location.

The capacity of a memory unit is characterized by the number of memory locations that it contains and the number of bits per memory location. The capacity of a memory unit can be determined from the number of its ADDRESS input lines and the number of its DATA lines. For an illustration, consider the memory unit of Fig. 6.21(b) which has ten ADDRESS lines and eight DATA lines. With a 10-bit address, we can generate unique addresses for up to $2^{10} = 1024$ different memory locations, which means that this memory unit has this number of memory locations. Also, since the memory has eight DATA lines, each memory location contains 8 bits. Consequently, the memory unit of Fig. 6.21(b) has a capacity of 1024×8 bits. In the terminology of random access memory, it is a $1K \times 8$ memory unit, in which K represents 1024. In general, a memory unit with n ADDRESS lines and m DATA lines has a capacity of $2^n \times m$ bits. Note that in Fig. 6.21(b), the memory addresses are specified in decimal, from 0 to 1023. In digital design, though, it is frequently more useful to specify the memory addresses in hexadecimal, as shown in Fig. 6.21(c). Hex notation for memory addresses will be generally used in this book.

Any random access memory can be classified as either a *read-write* memory (RWM or RAM) or a *read-only* memory (ROM). Read-write memory is commonly referred to as RAM (random access memory), which is a misnomer since a read-only memory is also a random access memory. Although a misnomer, the term RAM is universally accepted, and so it will be used throughout this book to refer to read-write memory.

Read-write memory can be further classified as *static* RAM or *dynamic* RAM. A static RAM is a read-write memory in which data is stored in flip-flop storage elements. With such storage, the data bits retain their values as long as the memory is supplied with power. In contrast, a dynamic RAM is a read-write memory in which data is stored as charges on capacitors. Left unattended, any capacitor will eventually lose its charge. Consequently, periodic refresh operations are required in a dynamic RAM to retain the data bit values. Static RAM is discussed in Sec. 6.5.1, and dynamic RAM in Sec. 6.5.3.

A read-only memory is a random access memory in which, under normal operation, the data stored in each memory location can be read by a read operation, but cannot be altered by a write operation. An advantage of the read-only memory over RAM is that the data storage is nonvolatile. In other words, data stored in a read-only memory is retained even if there is a temporary loss of power. Different versions of ROM are available, including masked programmed read-only memory (ROM), field-programmable ROM (PROM), and erasable program ROM (EPROM). Read-only memories are discussed in Sec. 6.5.2.

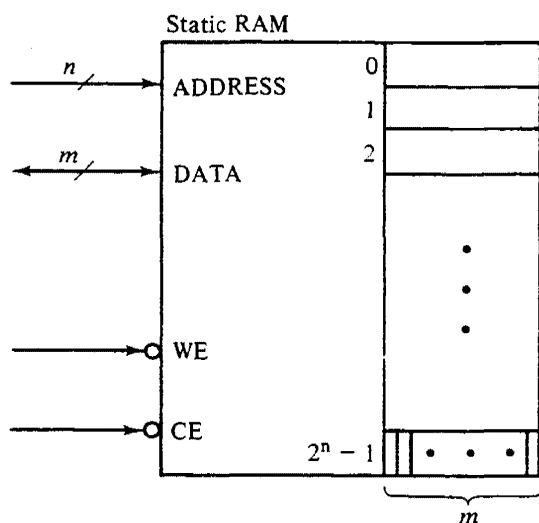
6.5.1 Static RAM

A general model of the static RAM is shown in Fig. 6.22(a). It is essentially the same block diagram as that of Fig. 6.21(a) except for having a detailed specification of the control inputs, which are the (normally) active-low inputs WE and CE. The functions of these inputs are given in the table of Fig. 6.22(b).

For a read operation, the n -bit address is specified at the ADDRESS inputs, and the chip-enable (CE) input is made true (L) and the write-enable (WE) input is made false (H). After a time equal to the access time of the memory, the m -bit contents of the specified location become available at the DATA outputs. For a write operation, the n -bit address is specified at the ADDRESS inputs. Additionally, the m -bit data to be stored is applied at the DATA inputs. Also, the CE and WE inputs are made true (L). Then, after a time equal to the access time of the memory, the data is stored in the specified memory location.

Static RAMs are commercially available in various sizes, such as, for example, 256×4 , $1K \times 1$, $1K \times 4$, $4K \times 1$, $2K \times 4$, $2K \times 8$, and up. Shown in Fig. 6.23 is a typical example of a commercially available $1K \times 4$ -bit static RAM with three-state outputs. As shown in the block diagram of Fig. 6.23(a), it has ten address lines (A_9-A_0) and four data lines (D_3-D_0). There are also two active-low control inputs: WE (write enable) and CS (chip select). The controls for the operations of the RAM are summarized in the table of Fig. 6.23(b). The WE input specifies the operation, and is false (H) for a memory read and true (L) for a memory write. The chip is functional only if the chip-select input (CS) is true (L). Otherwise, the data lines, D_3-D_0 , are put into a high-impedance (high-Z) state that electrically disconnects them from the data outputs.

Often in the design of a digital system, the desired memory module requires a capacity greater than that provided by any commercially available memory chip. Then it is necessary to construct this memory module from several memory modules of smaller sizes. Shown in Fig. 6.24 is a technique for realizing a memory module with an additional number of *bits* per memory location. For the realization of the $1K \times 8$ memory of Fig.



(a) A model of static RAM

Operation	CE	WE
Disable RAM	F(H)	X
Read	T(L)	F(H)
Write	T(L)	T(L)

Where F: false
 T: true
 X: don't care
 L: low
 H: high

(b) Operations

Figure 6.22 Static RAM.

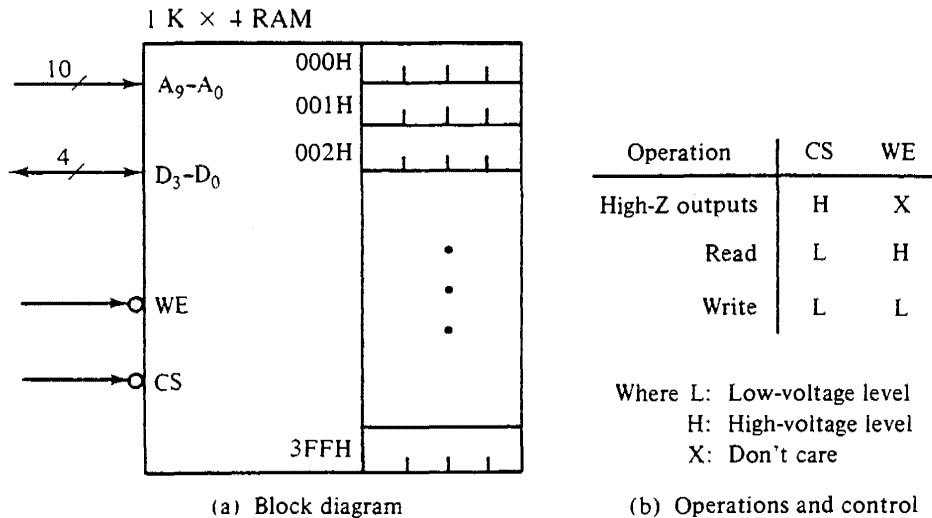
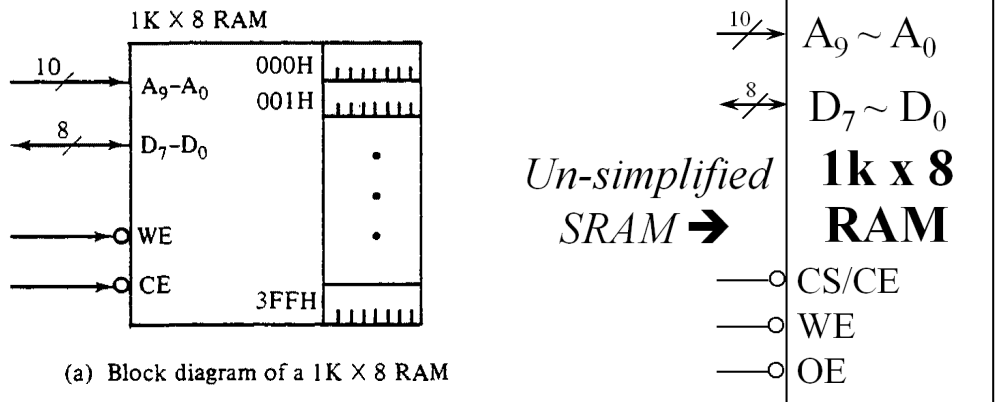
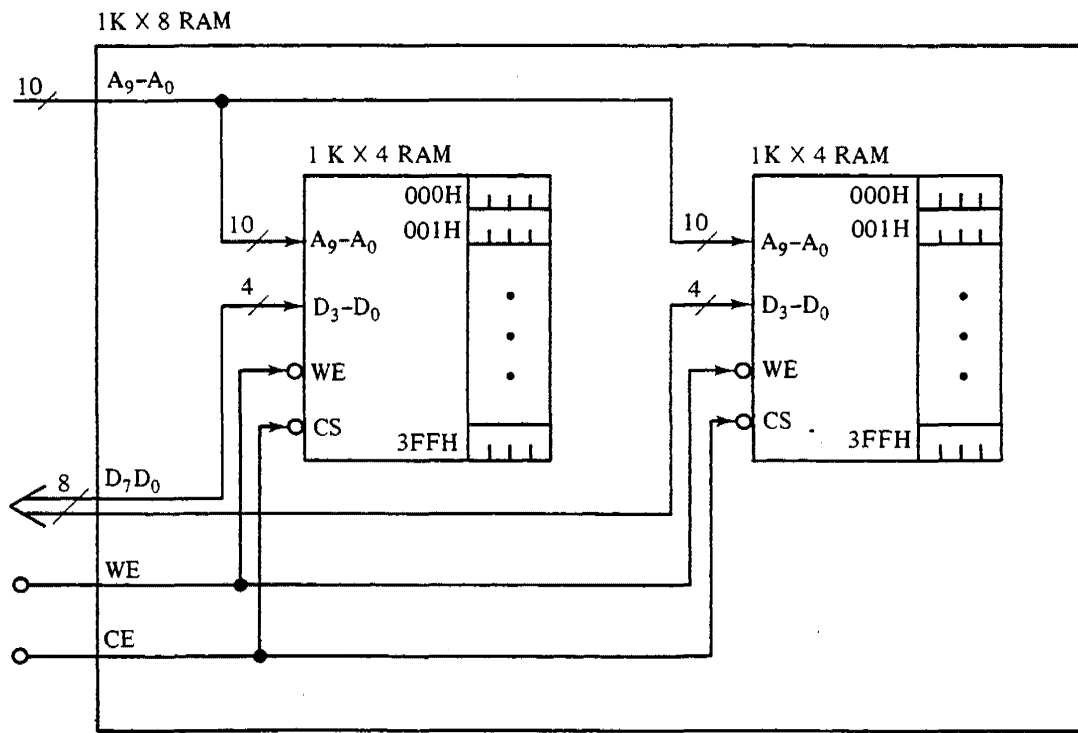


Figure 6.23 Typical commercially available 1K x 4 RAM.

- Sometimes block diagrams for SRAM are simplified, removing OE(L).
 - > Since WE(L) has higher priority than OE(L), OE(L) can be always true, i.e., GND.
 - > In this case, if WE is true, writing to SRAM; otherwise reading.



(a) Block diagram of a 1K x 8 RAM

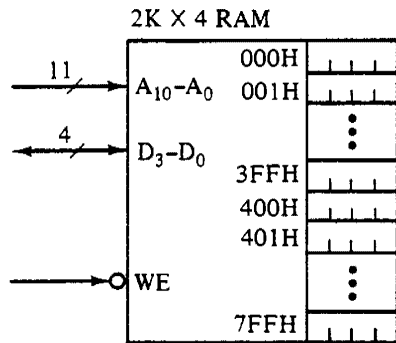


(b) Realization with two 1K x 4 RAMs

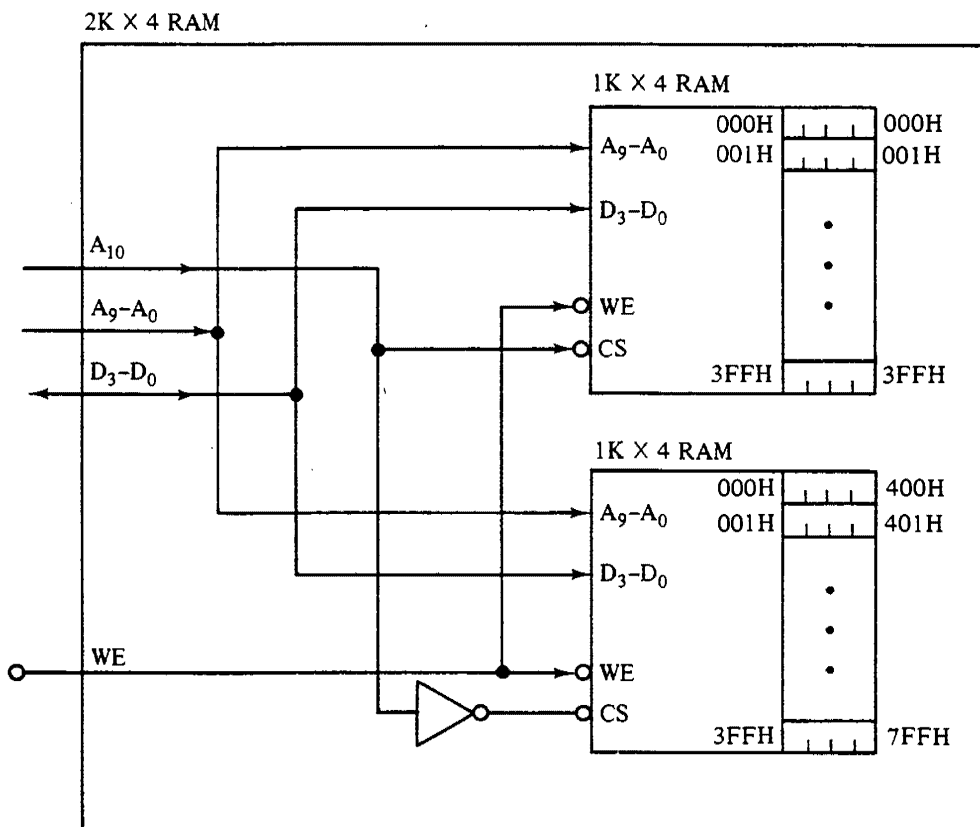
Figure 6.24 Realization of a 1K x 8 RAM.

6.24(a) with $1K \times 4$ RAMs, two of these RAMs are required, as shown in Fig. 6.24(b). Note that the inputs to the two sets of address lines are identical. Consequently, when a 10-bit address (A_9-A_0) is applied, the same relative memory locations for both memory chips are accessed, with the high nibble being found in one memory chip and the low nibble in the other. Together they form the 8-bit data for that address.

In Fig. 6.25 is shown a technique for realizing a memory module having an additional number of *memory locations*. For the realization of the $2K \times 4$ RAM of Fig. 6.25(a) with $1K \times 4$ RAMs, two of these RAMs are again required, as shown in Fig. 6.25(b). In this figure note the convention used for labeling the addresses of the memory



(a) Block diagram of a $2K \times 4$ RAM



(b) Realization with two $1K \times 4$ RAMs

Figure 6.25 Realization of a $2K \times 4$ RAM.

locations. The 000H to 3FFH addresses labeled on the inside of each $1\text{K} \times 4$ RAM are the addresses of the locations with respect to that particular $1\text{K} \times 4$ RAM. But the 000H to 7FFH addresses labeled on the outside of the $1\text{K} \times 4$ RAMs are the addresses of the locations with respect to the entire $2\text{K} \times 4$ RAM.

Observe that the chip-select (CS) inputs of the two $1\text{K} \times 4$ RAMs are controlled by the high-order address bit A_{10} . A value of $A_{10} = 0$ enables the top 1K RAM, representing the first 1K block of memory (00000000000B to 01111111111B). At the same time, this value disables the bottom 1K RAM, representing the second 1K block of memory (10000000000B to 11111111111B), and three-states it from the external data lines. Conversely, a value of $A_{10} = 1$ enables the bottom 1K RAM and disables the top 1K RAM. As a result, each location of the 2K module has a unique 11-bit address even though the corresponding locations of the two 1K RAMs have the same 10-bit address. For example, the first location of the top 1K RAM has an 11-bit address of 00000000000B and the first location of the bottom 1K RAM has an 11-bit address of 10000000000B.

With a combination of these techniques, a memory module of any reasonable size can be realized with smaller memory modules, along with some external circuitry. For example, a realization of a $2\text{K} \times 8$ RAM requires four $1\text{K} \times 4$ RAMs and an inverter (see Problem 6.19). A $4\text{K} \times 4$ RAM requires four $1\text{K} \times 4$ RAMs and a 4-to-2 decoder (see Problem 6.20). And a $2\text{K} \times 4$ RAM with a chip-enable input requires two $1\text{K} \times 4$ RAMs along with inverters and AND gates (see Problem 6.21).

RAM Timings

RAMs, which are LSI devices, have more complex timing requirements than SSI and MSI devices such as gates and flip-flops. RAM operation requires strict adherence to the proper sequencing of the address, data, and control signals, and also to the required durations of these signals. In this section we will consider the most important and commonly used RAM timing parameters. The block diagram of a static RAM shown in Fig. 6.26(a) will be used as the basis of our discussion.

The timing diagrams illustrating the timing requirements for a *memory read cycle* and for a *memory write cycle* of a RAM are shown in Figs. 6.26(b) and (c), respectively. In the memory read cycle, the *read-cycle time*, t_{RC} , is the total time required for the memory read operation, and is the minimum amount of time that the n -bit address must be stable on the ADDRESS inputs. As shown in Fig. 6.26(b), the read cycle begins when the address becomes stable (graphically indicated by the crossing lines) and ends when the address is changed (again indicated by the crossing lines).

The read-cycle time is a function of other timing parameters, the most important of which are the *access-time-from-address*, $t_{\text{A}}(\text{AD})$, and the *access-time-from-chip-enable*, $t_{\text{A}}(\text{CE})$. The parameter $t_{\text{A}}(\text{AD})$ is the time delay from the beginning of the read cycle, when the address is changed, until the time the data becomes valid on the DATA lines. Consequently, for a certainty of data validity, data should not be accessed and used by another system component until after a time equal to this parameter $t_{\text{A}}(\text{AD})$, which is published on the memory device sheet provided by the manufacturer. The other parameter, $t_{\text{A}}(\text{CE})$, is the delay from the time that a CE input of true (L) is applied, until the time that the data becomes valid on the DATA lines. Consequently, enabling the CE input earlier will result in the data being valid earlier, thereby reducing the read-cycle

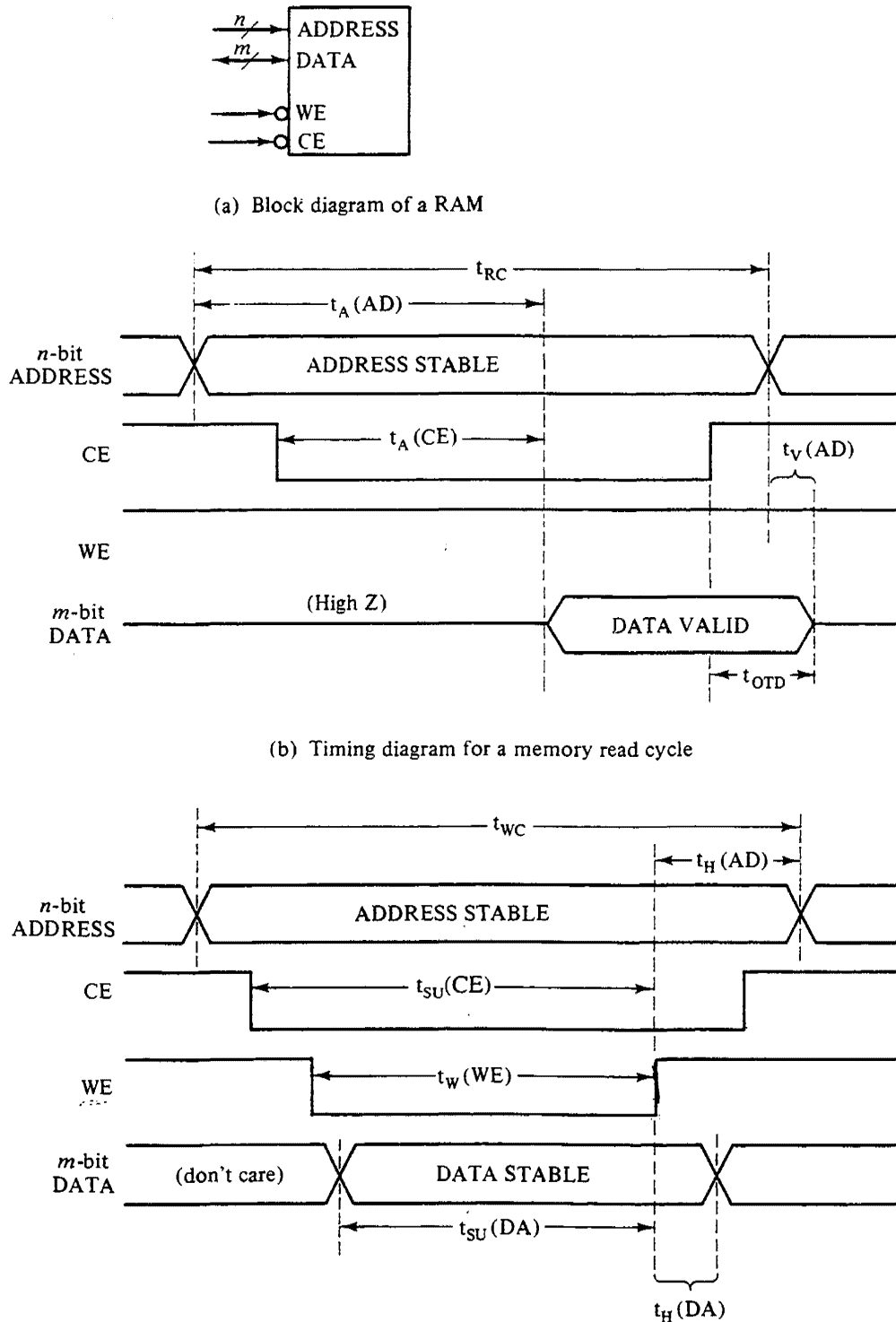


Figure 6.26 Memory read and memory write cycles.

time t_{RC} . As a rule of thumb, for the minimization of t_{RC} , it is best to apply the CE input at the same time as the address and to keep it stable for the entire duration of the read cycle. Of course, the write enable WE should be false (H) for the duration of the read cycle. Minimizing the read-cycle time improves the performance of the digital system since this time t_{RC} determines the maximum rate at which the memory can be read.

The two timing parameters t_{OTD} and $t_{\text{V}}(\text{AD})$, which are of less importance, are also shown in Fig. 6.26(b). The parameter t_{OTD} is the time required for the data *outputs to three-state from deselection*. More specifically, if the CE input is changed to false before the address is removed, then the chip is no longer enabled. The data, however, still remains available on the DATA lines for a period of time equal to t_{OTD} until the data outputs become three-stated. The other parameter $t_{\text{V}}(\text{AD})$ is the *output-valid-after-address-change* time. More specifically, when the address is changed, then the current read cycle is terminated. The current data, however, will still remain valid on the DATA lines for a time equal to $t_{\text{V}}(\text{AD})$ after the address change.

For the memory write cycle shown in Fig. 6.26(c), the *write-cycle time*, t_{WC} , is the total time required to complete a memory write operation. The n -bit address must be stable on the ADDRESS lines for the entire duration of t_{WC} . As shown in Fig. 6.26(c), the write cycle begins when the address becomes stable, and it ends when the address is changed.

The write-cycle time is a function of other timing parameters, the most important of which are the *write-pulse width*, $t_{\text{W}}(\text{WE})$, the *chip-enable-setup time*, $t_{\text{SU}}(\text{CE})$, and the *data-setup time*, $t_{\text{SU}}(\text{DA})$. The write operation is performed during the time that the WE input is true (L)—that is, during $t_{\text{W}}(\text{WE})$. So the write operation must be completed and the data stored by the time the WE input is changed from true (L) to false (H), which means that $t_{\text{W}}(\text{WE})$ is the minimum time that the write signal must be true (L). The most important timing consideration for the write cycle is to make certain that the required signals are applied for the required durations (setup times) before the write operation is completed. Specifically, by the time of the end of the write operation, when the WE input is changed from true to false, the chip-enable input (CE) must have remained true (L) for at least a time equal to $t_{\text{SU}}(\text{CE})$. Also, by that time, the data must have been stable on the DATA lines for a time at least equal to $t_{\text{SU}}(\text{DA})$. As a rule of thumb, to ensure that all the timing requirements for a memory write are going to be satisfied, it is best to apply the CE, WE, and data inputs all at the same time as the address and have them all remain stable for the duration of the write cycle.

Two other timing parameters $t_{\text{H}}(\text{DA})$ and $t_{\text{H}}(\text{AD})$, which are of much less importance, are also shown in Fig. 6.26(c). They are required to obtain proper write operations for certain types of memory chips. The parameter $t_{\text{H}}(\text{DA})$ is the *data hold time*. It is the time that the data must be maintained on the DATA lines after the WE input has become false. The parameter $t_{\text{H}}(\text{AD})$, the *address hold time*, is somewhat similar except that it applies to the address instead of to the data. It is the time that the address must be maintained on the ADDRESS lines after the WE input has become false. For most of the memories that are currently available, both $t_{\text{H}}(\text{DA})$ and $t_{\text{H}}(\text{AD})$ are usually zero.

6.5.2 Read-Only Memory

Functionally, a read-only memory, such as is shown in Fig. 6.27, is a special case of a read-write memory. For the normal operation of a read-only memory, the data that is stored in each memory location can be read with a read operation but cannot be altered by a write operation. As already stated, the principal advantage of the read-only memory over RAM is that the data storage is nonvolatile. More specifically, unlike for a RAM, the data stored in a read-only memory will be retained even after a loss of power.

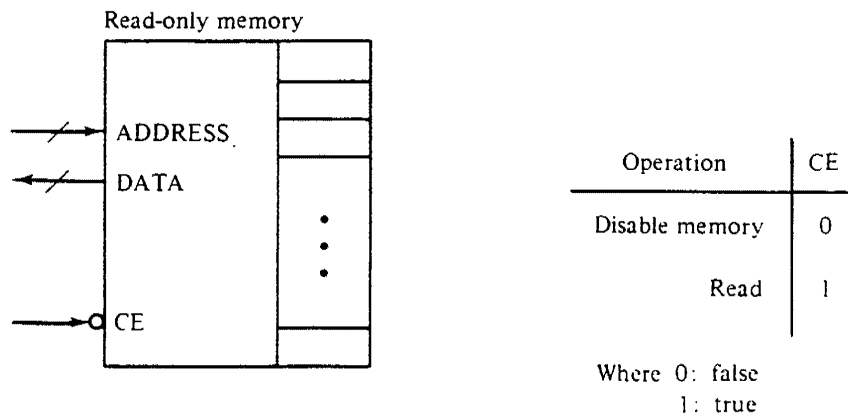


Figure 6.27 A model of a read-only memory under normal operation.

Therefore, a temporary loss of power will not cause a loss of data—a feature that is important in many applications.

The different types of read-only memories can be distinguished by the manner in which the data is originally stored. For a *masked programmed read-only memory* (ROM), the data is permanently stored by the manufacturer during the fabrication of the chip. Once stored, the contents of a ROM are fixed and cannot be altered. Since a custom mask has to be produced for each design of a ROM, mask programmed ROMs are economical only if manufactured in large quantities.

A *field-programmable read-only memory* (PROM) is the functional equivalent of a ROM under normal operating conditions. However, the one-time programming of a PROM is performed by the *user*, using a PROM programmer, rather than by the manufacturer. Physically, the PROM construction is based on the same integrated fuse technology used for field-programmable PLAs, as discussed in Sec. 6.4.1. A PROM comes from the manufacturer with all the connections intact as fuses. The user, using a PROM programmer, programs the PROM by leaving intact the fuses representing the 1-value bits and blowing the fuses representing the 0-value bits. The flexibility and relatively low cost of PROMs make them attractive for the small quantity production of parts that require read-only memories.

The *erasable programmable read-only memory* (EPROM) is another type of read-only memory. Under normal operation, an EPROM is the functional equivalent of a ROM or a PROM. Like a PROM, an EPROM can be programmed by the user, by means of an EPROM programmer. Unlike the ROM and PROM, however, the programming of an EPROM is not irreversible. The stored data, although nonvolatile, can be erased by placing the EPROM in an EPROM eraser. During the erasing process, ultraviolet radiation slowly releases the charge that has been stored as data and thereby restores the EPROM to its original state. The flexibility of the EPROM makes it ideal for developmental prototype implementation, and also for low-volume production in applications where the contents of a read-only memory need to be changed.

As mentioned, the functions of the various types of read-only memories are equivalent under normal operating conditions. For convenience, therefore, we will subsequently use the generic term read-only memory (ROM) to refer to all types of read-only memories (ROM, PROM, and EPROM), unless otherwise specified.

Read-Only Memory Applications

Read-only memories are required for applications in which a large amount of information needs to be stored in a nonvolatile manner. By far the most important application of the ROM is for the permanent storage of microprocessor programs and fixed tables of data for a microprocessor system. Microprocessors and microprocessor-based design are the subjects of the second half of this book. We will, therefore, defer this application till then.

Another common application of the ROM is for the systematic realization of complex combinational circuits. Perhaps this application of the ROM is best understood from a simple example.

Example 6.4 BCD-to-7-Segment Conversion

Consider the design and realization of a combinational circuit for converting a 4-bit BCD number to a 7-bit number that corresponds to the seven segments of a 7-segment display. The problem statement is summarized in Fig. 6.28(a), and the resultant truth table is shown in Fig. 6.28(b). Since this is strictly a combinational circuit, it can be realized in a straightforward manner with the techniques presented in Chapters 2 and 3, and so it will not be shown. Obviously, though, the realization would have seven sum-of-product AND-OR structures, with a substantial package count.

Alternatively, this combinational circuit can be realized with a single ROM that has a capacity greater than or equal to 16×7 bits. The result is shown in Fig. 6.28(c). Note that the right-hand side of the truth table of Fig. 6.28(b) is simply stored as the contents of the 16×7 ROM. Then when a particular set of inputs is applied, this set of incoming 4 bits is used as the address for looking up the contents of the corresponding ROM location, whose contents correspond to the valid outputs for that set of inputs.

From an input-output point of view, the operation of this ROM is indistinguishable from that of a traditional combinational circuit realization. For example, for an input of $B_3B_2B_1B_0 = 0101$, a traditional realization with discrete SSI components would output

$$Z_6Z_5Z_4Z_3Z_2Z_1Z_0 = 1011011$$

For the ROM realization of Fig. 6.28(c) the resultant output would also be

$$D_6D_5D_4D_3D_2D_1D_0 = Z_6Z_5Z_4Z_3Z_2Z_1Z_0 = 1011011$$

And for any other of the 16 possible sets of inputs, the same outputs would be obtained for either realization. Consequently, if the circuits resulting from the two different realization techniques were put into two separate "black boxes," they would be *functionally* indistinguishable. ■ ■

It should be obvious that a ROM can be used for the realization of any combinational logic function. One important application is the realization of a microprogrammed controller, which is one of the subjects of the next chapter. This realization consists of a relatively complex combinational circuit, along with a number of flip-flops. For multi-input and output combinational realizations, the ROM method is generally superior to the traditional method, as is apparent from the power of, for example, the commercially available $8K \times 8$ EPROM, which can be used for the realization of a combinational

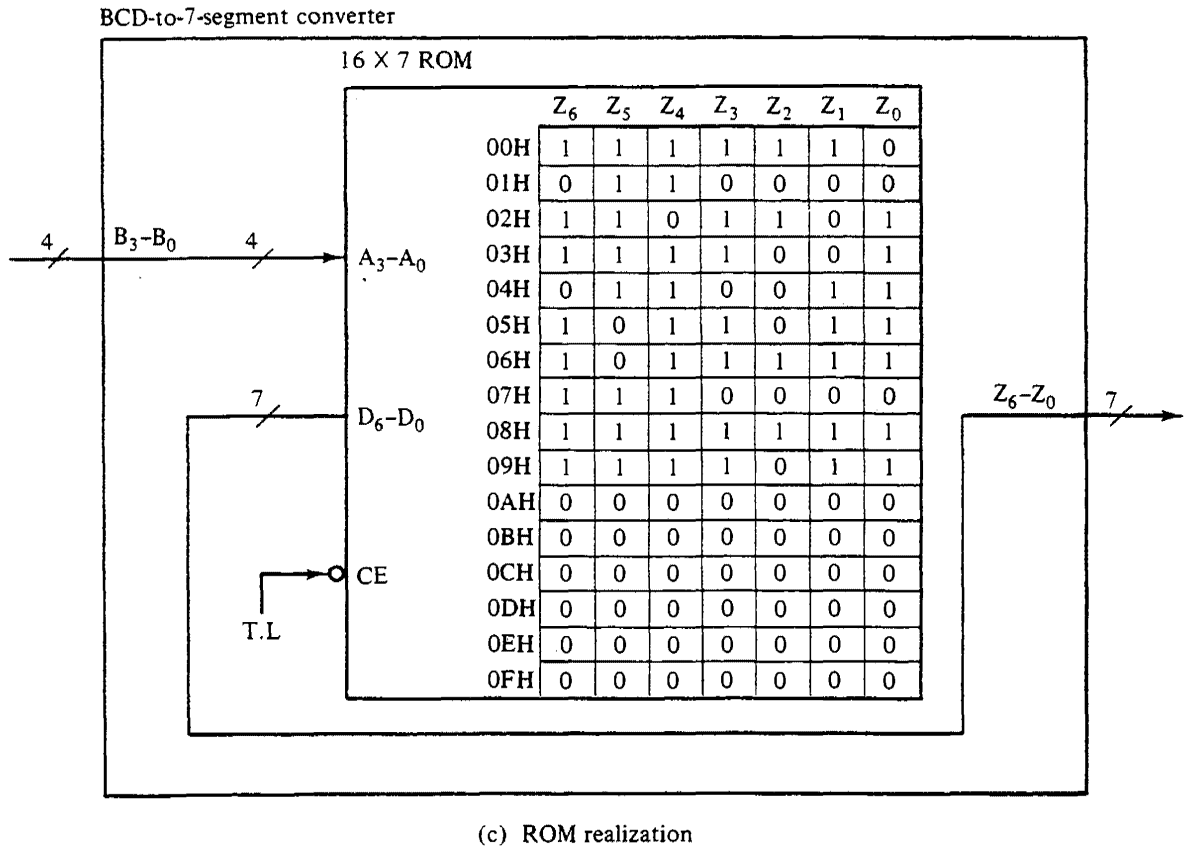
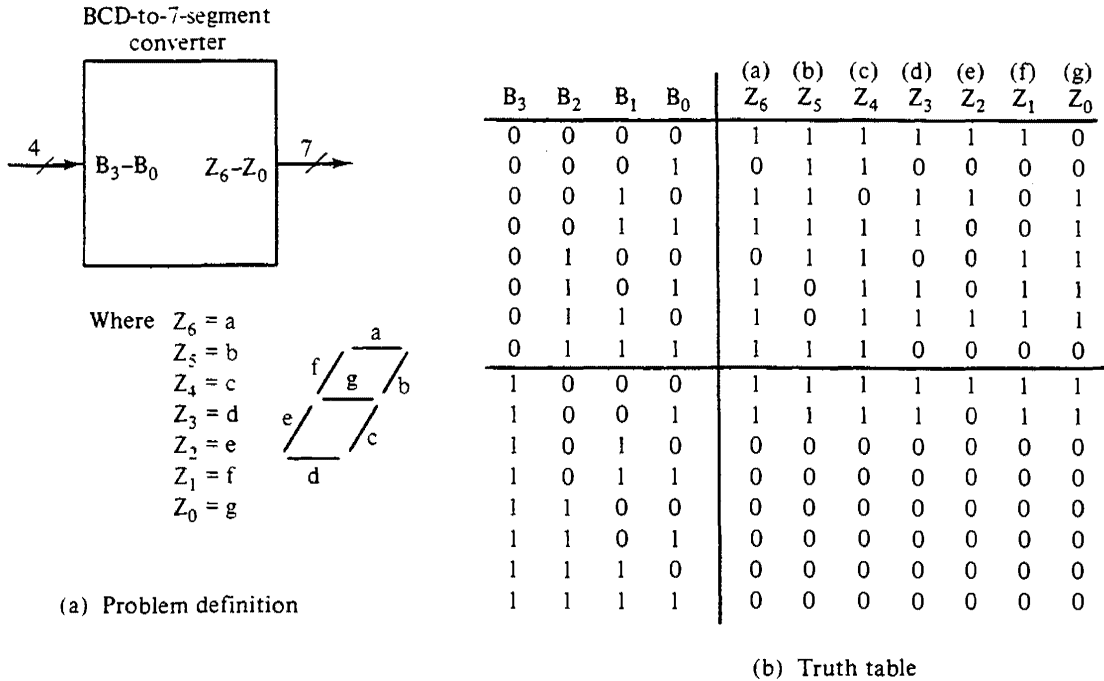


Figure 6.28 ROM realization of a BCD-to-7-segment converter.

circuit having as many as 13 inputs and 8 outputs. The disadvantage of the ROM method is speed. The access time for a ROM, as for any memory, is slow compared to the speed of operation of a discrete logic realization.

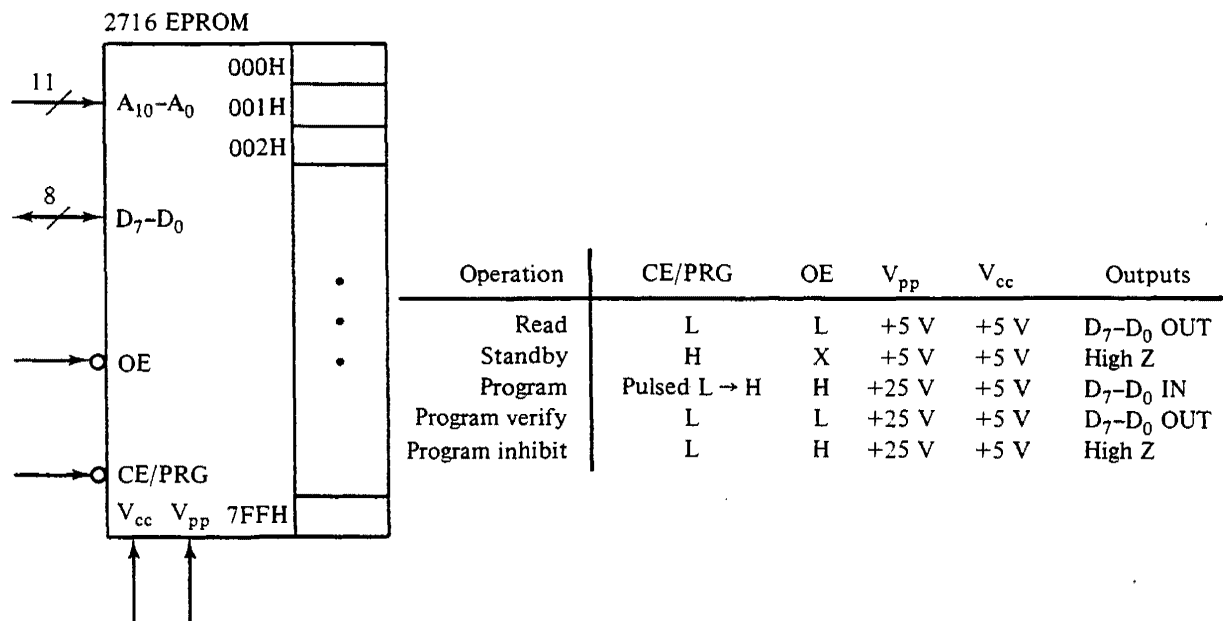
The ROM method has the same advantage of package count reduction as does the PLA/PAL method of combinational circuits presented in Sec. 6.4. A ROM realization,

however, is slower. When speed is not a constraint, then the choice of a PLA/PAL realization versus a ROM realization is dependent on the nature of the design. As is evident from Sec. 6.4, the PLA/PAL realization is ideal for a design with a truth table that is sparsely populated by 1s, corresponding to relatively few distinct product terms. For example, the 82S100 PLA described in Sec. 6.4.1 can realize a combinational circuit with as many as 16 inputs and 8 outputs, but only if the corresponding truth table has 48 or fewer distinct product terms. For this number of inputs and outputs, a ROM realization would require an absurdly huge (2^{16}) $64K \times 8$ ROM, even if the number of distinct product terms were fewer than 48. In summary, for a complex combinational circuit with a truth table that has relatively few distinct product terms, a PLA/PAL should be used. For a circuit with a truth table that has many product terms, then a ROM is the appropriate choice.

Commercially Available EPROMs

EPROMs are commercially available in various sizes, usually in the form of $nK \times 8$ bits, where $n = 2, 4, 8, 16, 32, 64, \dots$. As an example, we will describe the original EPROM, the 2716, which is from the Intel Corporation. It illustrates the most important features of a commercially available EPROM. The 2716 EPROM is a $2K \times 8$ EPROM with three-state outputs. As is shown in Fig. 6.29(a), it has 11 address lines ($A_{10}-A_0$) and 8 data lines (D_7-D_0). Also, it has an output-enable input (OE), a chip-enable/EPROM program input (CE/PRG), and inputs V_{cc} and V_{pp} for two voltage supplies.

As is summarized in the table of Fig. 6.29(b), the 2716 EPROM has five main operations. Under normal operating conditions, if the chip-enable (CE/PRG) and the output-enable (OE) inputs are true (L), then the operation is a memory read. But if the chip-enable input is false, then the EPROM outputs (D_7-D_0) are three-stated, and the EPROM is in the standby mode.



(a) Block diagram of the 2716 EPROM

(b) Table of operations

Figure 6.29 The 2716 EPROM.

The other three operations are for initially programming or for reprogramming the contents of the EPROM. An EPROM is usually programmed through the use of an EPROM programmer, a design tool that is widely available. The sequencing and timing of the various programming operations are the responsibility of the EPROM programmer. The steps for programming a 2716 EPROM are as follows: The 2716 EPROM is first put into the program-inhibit mode by setting CE/PRG to L and OE to H, and by raising the V_{pp} voltage supply to 25 volts. Also, the address of and the contents for the specified memory location are applied at the address and data lines, respectively. Next, the 2716 EPROM is put into the program mode by applying at the CE/PRG input an L-to-H pulse, which causes the data to be stored into the specified location. At the end of the L-to-H pulse, the return of the CE/PRG and OE inputs to L puts the 2716 into the program-verify mode, during which the contents of the just-programmed location are available on the data lines where they can be verified by the EPROM programmer, if desired. Finally, the OE input is changed to false (H) and so the program-inhibit mode is entered again. This programming cycle must be repeated for each location of the EPROM.

Again, the sequencing and timing of the various programming operations are the responsibility of the EPROM programmer. Consequently, EPROM programming is relatively easy for a digital designer. Because of its flexibility and ease of use, the EPROM is a very important tool for developmental and prototype implementations.

6.5.3 Dynamic RAM

As has been stated, read-write memory can be classified as either *static* RAM or *dynamic* RAM. A static RAM is a read-write memory in which data bits are stored in flip-flop storage elements. With this type of storage, the data bits retain their values as long as the memory is supplied with power. On the other hand, a dynamic RAM is a read-write memory in which data bits are stored as charges on capacitors. Unfortunately, left unattended, any capacitor will eventually lose its charge. Consequently, a dynamic RAM requires periodic refresh operations to retain the values of the stored data bits.

Dynamic RAMs offer several advantages over static RAMs. First, dynamic RAMs are approximately four times as dense as static RAMs. The result is a fourfold reduction of board space required for a given amount of memory. Second, and also as a result of the density, the cost per bit of dynamic RAMs is approximately one-fourth that of static RAMs. Finally, the power consumption of dynamic RAMs is significantly less than that of static RAMs.

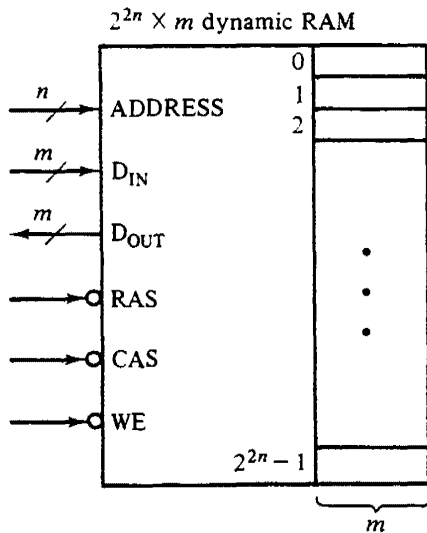
On the other hand, one disadvantage of dynamic RAMs is that they are generally slower in operation than static RAMs. Another disadvantage of dynamic RAMs is the complexity of the functions required to support their operations. The additional support circuitry is justified only if a large memory module is required. Static RAMs are more cost-effective for smaller memory modules.

A general model of a dynamic RAM (DRAM) is shown in Fig. 6.30(a). It has n address lines and separate m -bit data-in (D_{IN}) and data-out (D_{OUT}) lines. For a reduction in the number of address pins, the n address lines are used to specify a $2n$ -bit address. This is accomplished by time-multiplexing the required $2n$ -bit address into two halves over the same n -bit address lines. The first half is commonly called the *row* address, and the second half is commonly called the *column* address. The row address is applied first

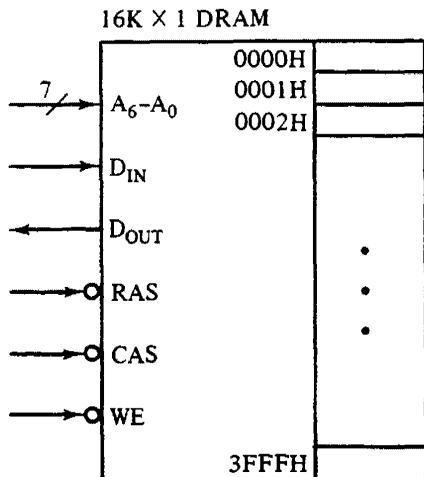
at the address lines, and is latched internally by the dynamic RAM at the trailing edge of the RAS (row address strobe) input. Then the column address is applied at the address lines, and latched internally at the trailing edge of the CAS (column address strobe) input. Together, they form a $2n$ -bit address that is capable of addressing up to 2^{2n} memory locations.

Shown in Fig. 6.30(b) is a typical example of a commercially available dynamic RAM. It is a relatively small $16K \times 1$ dynamic RAM; for it, $n = 7$ and $m = 1$.

In general, a dynamic RAM has three basic operations: *memory read*, *memory write*, and *memory refresh*. For a read or a write operation, the row and column addresses must be applied sequentially, and then latched, respectively, by the negative going edges of the RAS and CAS inputs. For a read operation, the WE (write enable) input must be made false (H), usually before the CAS input is made true (L), as shown in Fig. 6.31(a). After a time equal to the memory access time (measured from the trailing edge of CAS), the m -bit contents of the specified location are available on the D_{OUT} outputs. For a write operation, the row and column addresses are also latched. Additionally, the m -bit data

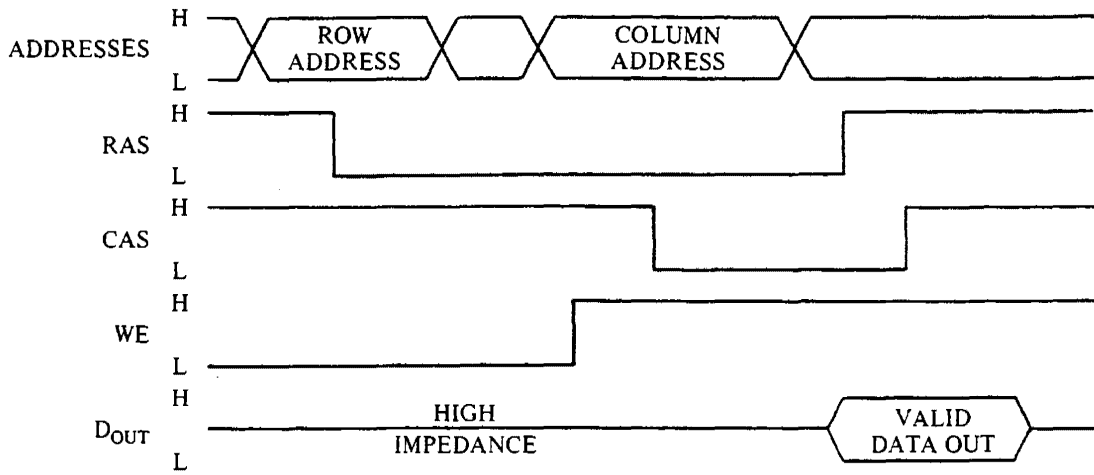


(a) General model of a dynamic RAM

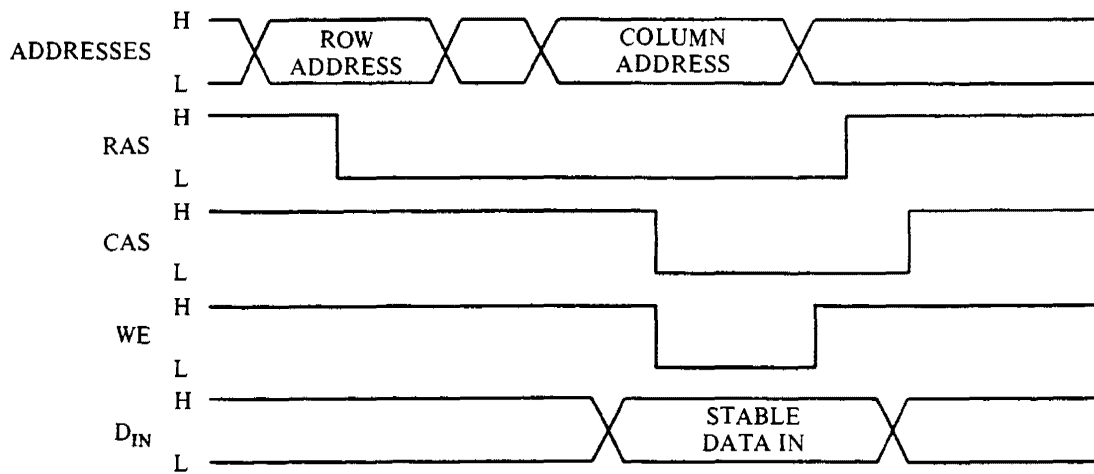


(b) $16K \times 1$ dynamic RAM

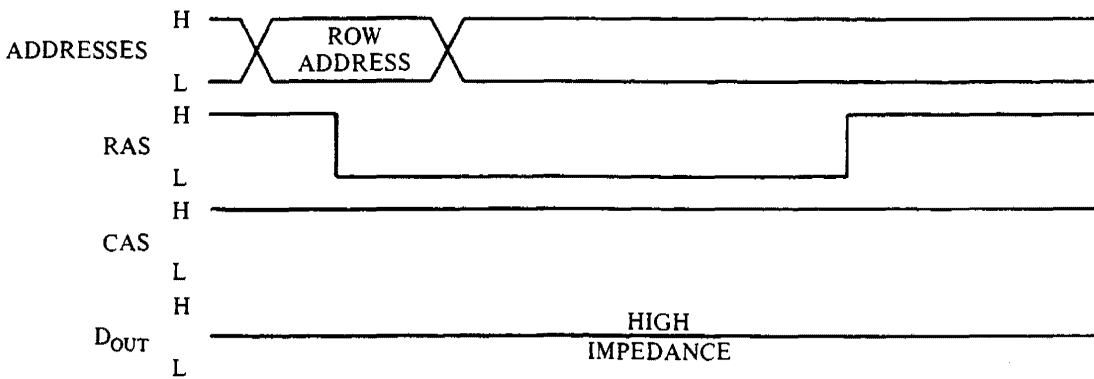
Figure 6.30 Dynamic RAM.



(a) Read cycle



(b) Write cycle



(c) RAS-only refresh cycle

Figure 6.31 Basic operations for a dynamic RAM.

to be stored is applied at the D_{IN} inputs, and the WE input is made true (L). Then the data is latched by the dynamic RAM at the trailing edge of the CAS or WE signal, whichever occurs last, and is stored in the specified location.

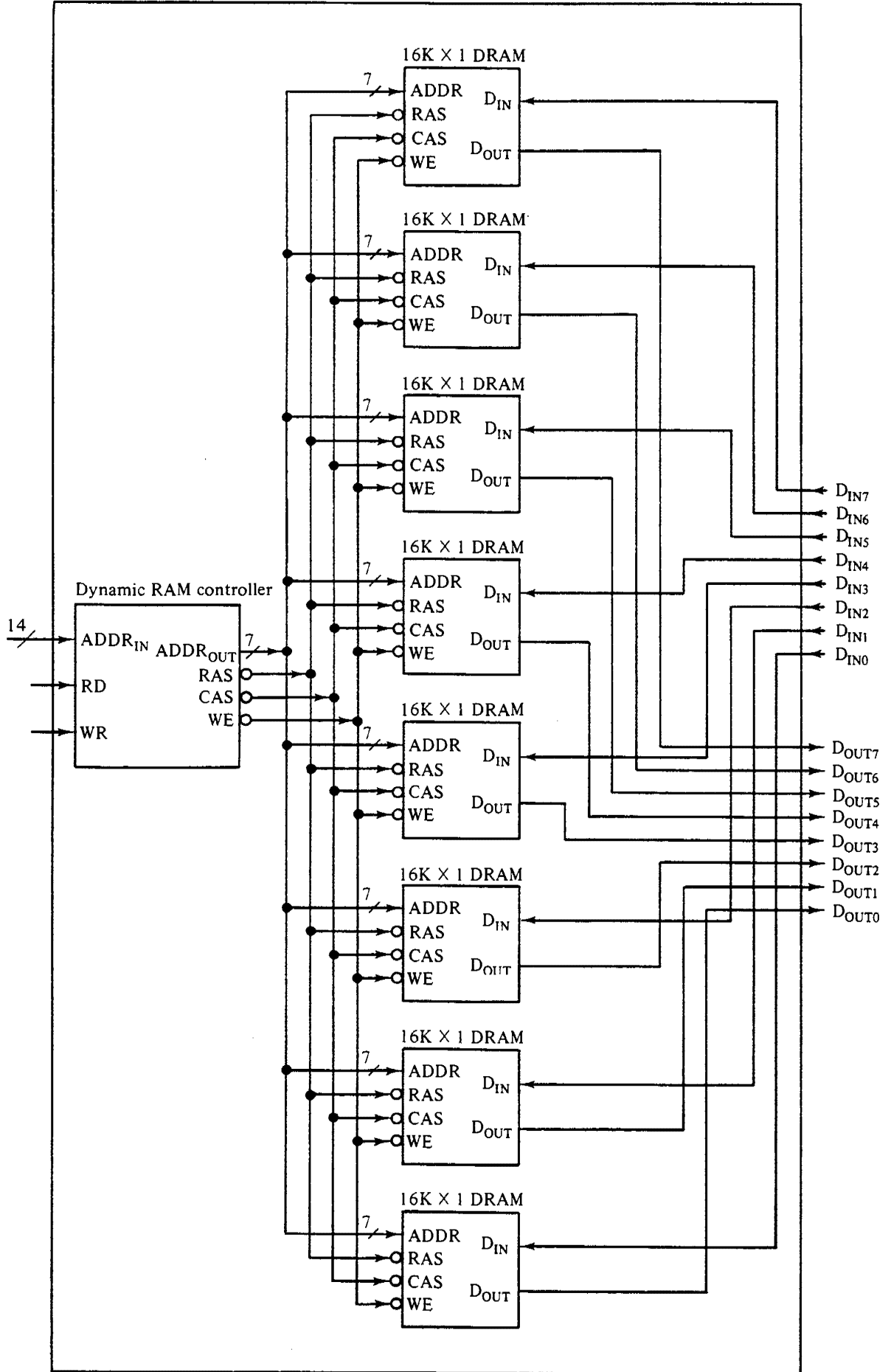


Figure 6.32 A 16K × 8 memory module constructed from dynamic RAMs.

As mentioned earlier, the data bits in a dynamic RAM are stored as charges on capacitors, and these charges must be periodically refreshed. For current commercially available dynamic RAMs, each bit-cell must be refreshed approximately every two milliseconds or less. Fortunately, dynamic RAMs are designed such that an entire row of bit-cells can be refreshed at once during the latching of the row address by the RAS input. Consequently, either the memory read operation or the memory write operation can be used to refresh a row of bit-cells. If we can be certain that every row of the memory will be accessed every 2 milliseconds or less by either a read or a write operation, then we can be certain that the memory will be properly refreshed. This is, of course, not generally the case. Therefore, a special memory refresh operation is required. The refresh cycle for a dynamic RAM is shown in the timing diagram of Fig. 6.31(c). It consists simply of latching the row addresses by the RAS input. Each refresh cycle refreshes one row of the dynamic RAM. Consequently, in order to refresh the entire dynamic RAM, a dynamic RAM controller needs to step through every row address within 2 milliseconds.

Shown in Fig. 6.32 is an example of a $16K \times 8$ -bit memory module constructed from eight $16K \times 1$ dynamic RAMs and a dynamic RAM controller. Note that the inputs to the address lines of all eight $16K \times 1$ DRAMs are identical. Consequently, when an address is specified, the same relative memory location for all eight memory chips are accessed, with bit i being found in memory chip i . Together, the 8 bits from the eight chips form the 8-bit data for that address.

The inputs to the $16K \times 8$ memory and the dynamic RAM controller comprise a 14-bit address along with the RD and WR signals. The controller has the responsibility of time-multiplexing the 14-bit address into a 7-bit row address and a 7-bit column address. The controller must also generate and properly sequence the RAS, CAS, and WE signals as is required for the read and write operations. Furthermore, the controller also has the responsibility for generating and sequencing the signals that are required for refreshing the dynamic RAM bit-cells. Because of their complexity and common use, dynamic RAM controllers are commercially available in IC form.

A dynamic RAM controller is a *state machine* similar to those that will be studied in the next chapter. Therefore, the design methods of the next chapter will provide the reader with some insight into the design and realization of such a digital circuit.

SUPPLEMENTARY READING (see Bibliography)

[Blakeslee 79], [Intel-A], [Kline 83], [Mano 84], [Monolithic], [Motorola], [Short 81], [Signetics], [Texas Instruments]

PROBLEMS

- 6.1. What are the advantages of using LSI circuit elements in a digital circuit as compared to using MSI and SSI circuit elements?
- 6.2. A 16-bit ALU is to be realized by interconnecting four 74'181 ALUs.
 - (a) Draw the circuit diagram.
 - (b) Given that the propagation delay for a 74'181 to perform an add operation is

$t_p(181ADD)$, how long does it take your ALU to perform a 16-bit add operation? Explain.

(c) Given that the propagation delay for a 74'181 to perform a logic operation is $t_p(181LOG)$, how long does it take your ALU to perform a 16-bit logic operation? Explain.

6.3. Using a 74'181 and any additional logic that is required, design and realize the simplified ALU shown in block diagram form in Fig. 6.33. This ALU produces an output F that is the result of some operation on the inputs A and B. The particular operation depends on the control word SEL, as follows:

SEL	Operation	Definition
0	Add	$F \leftarrow A \text{ plus } B$
1	Subtract	$F \leftarrow A \text{ minus } B$
2	Increment	$F \leftarrow A \text{ plus } 1$
3	Decrement	$F \leftarrow A \text{ minus } 1$
4	Complement	$F \leftarrow \text{NOT } A$
5	OR	$F \leftarrow A + B$
6	XOR	$F \leftarrow A \oplus B$

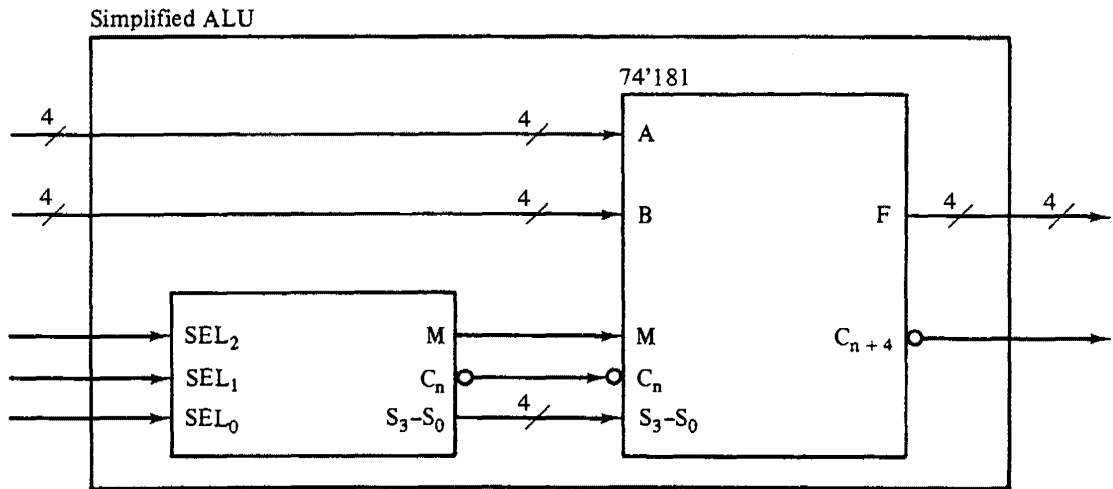


Figure 6.33 Simplified ALU for Problem 6.3.

6.4. Repeat Problem 6.3 using the block diagram of the simplified ALU shown in Fig. 6.34 and also the active-low view of the 74'181.

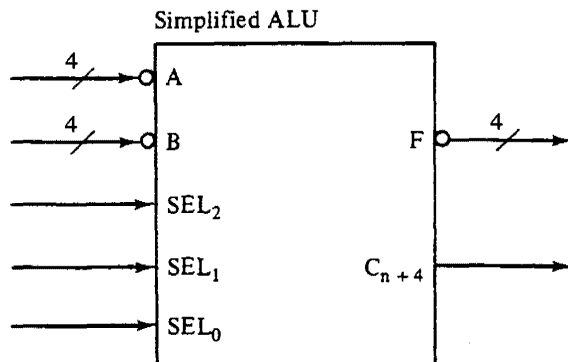


Figure 6.34 Simplified ALU for Problem 6.4.

6.5. For the chain of full adders shown in Fig. 6.2, what is the logic equation for the look-ahead carry circuit for C_2 ?

6.6. The following 2-bit numbers A and B are to be added:

Stage	N	$N - 1$	$N - 2$	$N - 3$	$N - 4$	$N - 5$	$N - 6$...
A	0	1	0	0	1	0	1	...
B	1	0	0	1	1	1	0	...

Find the values of the carry-outs produced by the following stages: (a) $N - 4$, (b) $N - 2$, (c) $N - 3$, (d) $N - 1$, and (e) $N - 5$. Explain your answers.

6.7. Convert the ripple adder circuit shown in Fig. 6.2 into a 4-bit adder with look-ahead carry circuitry, using a 74'182 and any additional logic that is required. (*Hint*: The carry-in of each adder stage will be generated by the 74'182.)

6.8. Determine the propagation delay required by an add operation for the 64-bit ALU with multilevel look-ahead carry structure shown in Fig. 6.7. Assume the following delay values:

- $t_p(181PG) = 33$ ns to produce G and P
- $t_p(181ADD) = 27$ ns to perform an add operation.
- $t_p(182PG) = 25$ ns to produce the P_i and G_i
- $t_p(182C_{xyz}) = 26$ ns to produce C_{n+x} , C_{n+y} , and C_{n+z}

6.9. Transform the logic diagram of Fig. 6.35 into a PLA circuit diagram similar to the one shown in Fig. 6.10(b).

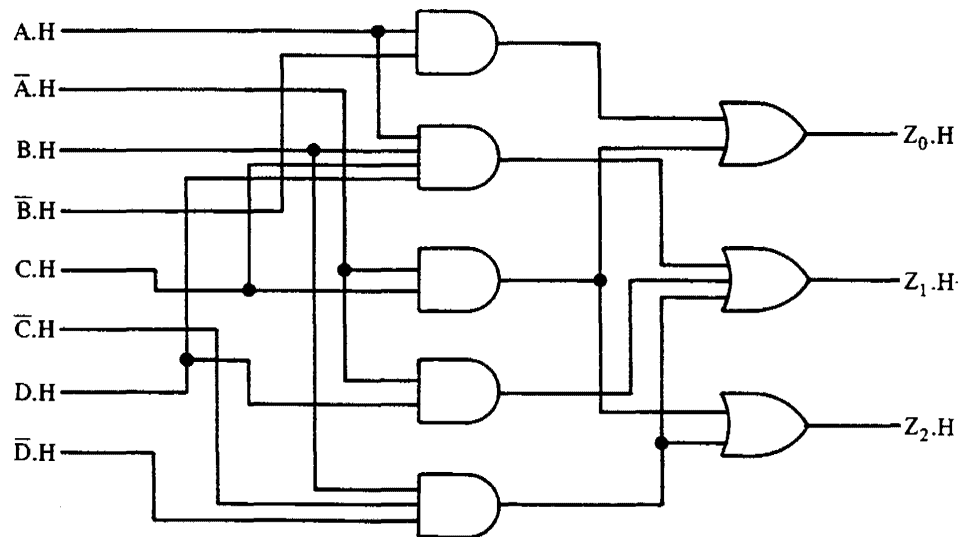
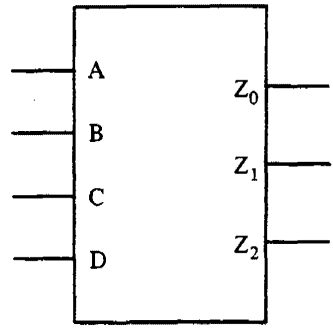


Figure 6.35 Logic diagram for Problem 6.9.

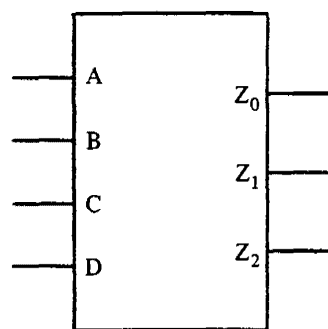
6.10. Given the truth table of Fig. 6.36 for the combinational circuit shown in block diagram form, realize the combinational circuit with a PLA that is similar to the one shown in Fig. 6.10(b) (i.e., one with four inputs, four outputs, and supporting eight product terms).



A	B	C	D	Z ₀	Z ₁	Z ₂
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	1	1	0
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	1	1
1	1	1	0	0	0	0
1	1	1	1	1	0	1

Figure 6.36 Block diagram and truth table for Problem 6.10.

6.11. Repeat Problem 6.10 for the block diagram and truth table shown in Fig. 6.37.



A	B	C	D	Z ₀	Z ₁	Z ₂
0	0	0	0	1	1	0
0	0	0	1	1	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	1	1	0
0	1	1	1	1	0	1
1	0	0	0	0	1	1
1	0	0	1	0	0	1
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	1	1
1	1	1	0	0	0	0
1	1	1	1	1	0	1

Figure 6.37 Block diagram and truth table for Problem 6.11.

- 6.12. Given the programmed PLA of Fig. 6.38 with functions similar to those of the 82S100 FPLA,
- What are the logic/voltage assignments (i.e., active-high or active-low) for the inputs A, B, C, D, and CE and for the outputs Z₀, Z₁, and Z₂?
 - Draw the mixed-logic block diagram for the corresponding combinational circuit.
 - Determine the logic equations for Z₀, Z₁, and Z₂.

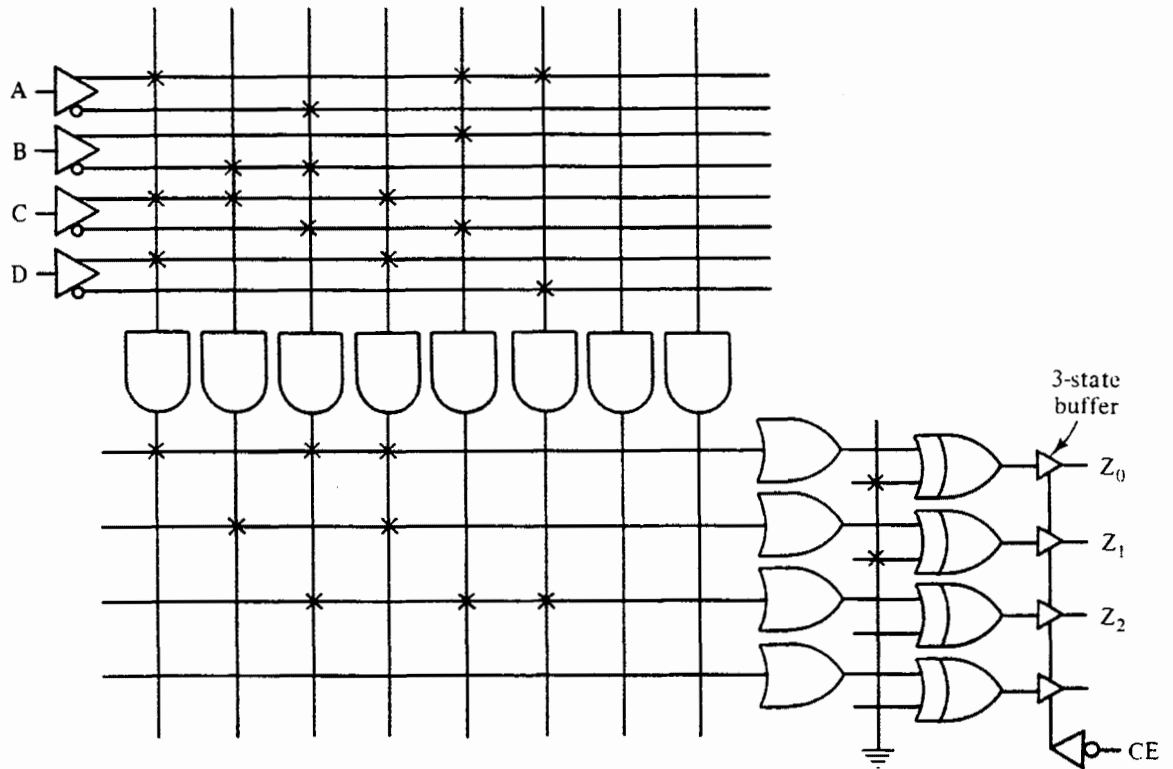


Figure 6.38 Programmed PLA for Problem 6.12.

- 6.13. What is the main difference between a PLA and a PAL?
- 6.14. Using the same PAL16L8, realize the following logic equations and have the realizations based on the following assignments:

Inputs: $X_1.H$ is assigned to pin 2, $X_2.H$ is assigned to pin 3, $S_1.H$ is assigned to pin 17, and $S_2.H$ is assigned to pin 16.

Outputs: $Z_1.L$ is assigned to pin 19, $Z_2.L$ is assigned to pin 18, $Z_3.L$ is assigned to pin 12, and $Z_4.H$ is assigned to pin 13.

All the other pins are not to be used unless specified otherwise.

(a) $Z_1 = S_2 \cdot X_1$ (Hint: Pin 16 needs to be programmed as an input.)

(b) $Z_2 = S_1 + S_1 \cdot \overline{X_2}$

(c) $Z_3 = X_2 \cdot (S_2 + S_1 \cdot \overline{X_1})$ (Hint: You can use pin 11 also if necessary.)

(d) $Z_4 = X_1 + X_2$ (Hint: Since Z_4 is active-high, you may need to use DeMorgan's laws.)

- 6.15. Using a PAL16L8, realize a BCD-to-7-segment decoder similar to the one shown in Fig. 6.28. However, the outputs a, b, c, d, e, f, and g are to be active-low.
- 6.16. Using a PAL16R4, realize a 4-bit decade counter with a synchronous CLEAR input. Compare your realization with the one obtained in Problem 5.26.
- 6.17. Draw block diagrams corresponding to the following static RAM module specifications. Specify the number of address lines and data lines.
- (a) 64×4 bits (b) 4096×8 bits (c) $64K \times 8$ bits

- 6.18. What is the capacity of a static RAM module that has
 (a) Seven address lines and eight data lines?
 (b) Fourteen address lines and four data lines?
 (c) Ten address lines and sixteen data lines?
- 6.19. Realize the $2K \times 8$ RAM module of Fig. 6.39 by using four $1K \times 4$ RAMs, as shown in Fig. 6.23, and an inverter.

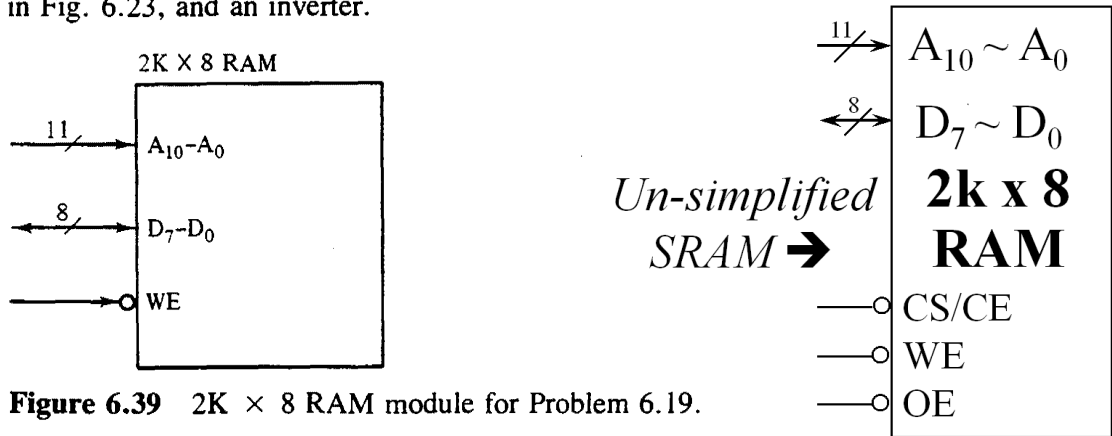


Figure 6.39 $2K \times 8$ RAM module for Problem 6.19.

- 6.20. Realize the $4K \times 4$ RAM module of Fig. 6.40 by using four $1K \times 4$ RAMs, as shown in Fig. 6.23, and a 4-to-2 decoder.

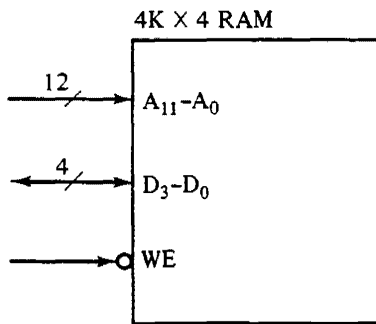


Figure 6.40 $4K \times 4$ RAM module for Problem 6.20.

- 6.21. Realize the $2K \times 4$ RAM module with chip-select input of Fig. 6.41 by using $1K \times 4$ RAMs, as shown in Fig. 6.23, and any additional logic that is necessary.

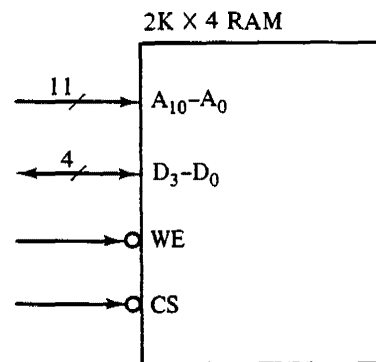


Figure 6.41 $2K \times 4$ RAM module for Problem 6.21.

- 6.22. Realize the memory module of Fig. 6.42 by using a $1K \times 4$ RAM, as shown in Fig. 6.23, and any additional logic that is necessary. Note that the bidirectional data lines of the $1K \times 4$ RAM become two sets of data lines, DIN and DOUT. (*Hint: Use three-state buffers.*)

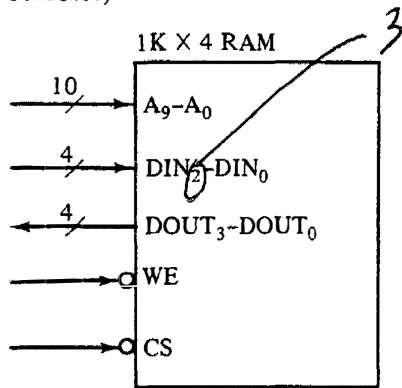


Figure 6.42 Memory module for Problem 6.22.

- 6.23. The static RAM chip shown in Fig. 6.26(a) has the following timing parameter values:

$$t_{RC} = 100 \text{ ns minimum}$$

$$t_A(AD) = 100 \text{ ns minimum}$$

$$t_A(CE) = 75 \text{ ns minimum}$$

At $t = 0$ s, a valid address is applied and the WE signal is set to false (H).

- (a) If the chip-enable signal (CE) is applied at $t = 10$ ns, then when is the time t at which the data first becomes valid?
- (b) If the chip-enable signal (CE) is applied at $t = 50$ ns, then when is the time t at which the data first becomes valid?
- 6.24. The static RAM chip shown in Fig. 6.26(a) has the following timing parameters:

$$t_{WC} = 100 \text{ ns minimum}$$

$$t_{SU}(CE) = 70 \text{ ns minimum}$$

$$t_W(WE) = 100 \text{ ns minimum}$$

$$t_{SU}(DA) = 70 \text{ ns minimum}$$

At $t = 0$ s, a valid address is applied and the WE signal is set to true (L).

- (a) If CE and the data are applied at $t = 0$ s, then the WE signal must remain true (L) until a time t_x to ensure a valid write operation. What is this time t_x ?
- (b) If CE is applied at $t = 50$ ns and the data is applied at $t = 0$ s, then what is this time t_x ?
- (c) If CE is applied at $t = 0$ s and the data is applied at $t = 50$ ns, then what is this time t_x ?
- 6.25. Discuss the similarities and differences among ROMs, PROMs, and EPROMs.
- 6.26. The hardware multiplier of Fig. 6.43 can multiply two 4-bit numbers (MCAND and MPLIER) and produce an 8-bit product (PRODUCT).
- (a) Derive the truth table for this circuit. Use don't cares when convenient.
- (b) If a ROM is used to realize this circuit, what must be the ROM capacity?
- (c) Draw a block diagram of the ROM realization, specifying all connections to the address and data lines.
- (d) What are the contents of the ROM? Explain in words.

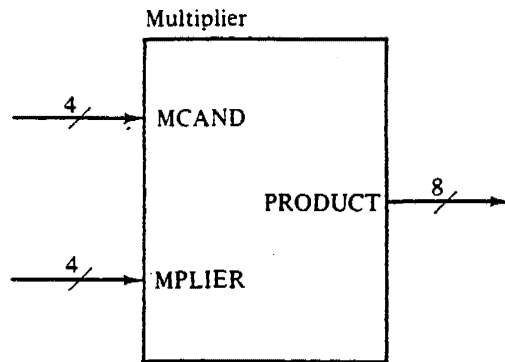


Figure 6.43 Multiplier for Problem 6.26.

- 6.27. Use a ROM to realize the four logic functions $Z_{1.L}$, $Z_{2.L}$, $Z_{3.L}$, and $Z_{4.H}$ specified in Problem 6.14 as follows:
- Draw a block diagram design of the ROM realization, specifying all connections to the address and data lines.
 - Specify in hexadecimal the contents of the ROM.
 - Explain what an active-low output does to the corresponding contents of the ROM.
- 6.28. Consider a PLA with 12 inputs (actually 12 inputs and 12 complements), 8 outputs, and 64 AND gates. Can it be used to realize the following combinational circuits?
- A circuit with eight inputs and six outputs.
 - A circuit with six inputs and eight outputs.
- In each case answer yes, no, or maybe, and explain your answer.
- 6.29. Can you implement the logic equations of the following combinational circuits with a 128×8 ROM?
- A circuit with eight inputs and six outputs.
 - A circuit with six inputs and eight outputs.
- In each case answer yes, no, or maybe, and explain your answer.
- 6.30. Construct the memory module of Fig. 6.44 that provides $6K \times 8$ bits of EPROM and $2K \times 8$ bits of RAM. [Hint: Use three 2716 EPROMs and two $1K \times 8$ RAM modules (see Fig. 6.24), a 2-to-4 decoder, and any additional logic that is necessary.]

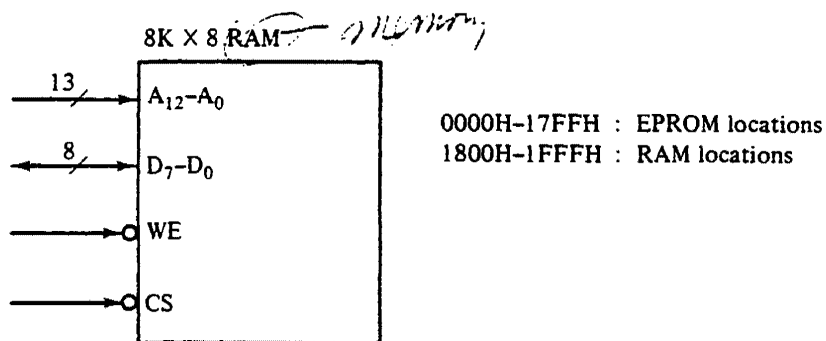


Figure 6.44 Memory module for Problem 6.30.

- 6.31. Discuss the advantages and disadvantages of using static RAMs versus dynamic RAMs in a digital circuit.
- 6.32. Consider the dynamic RAM of Fig. 6.45 that functions similarly to the one shown in Fig. 6.30(b).
- What is the capacity of this DRAM?

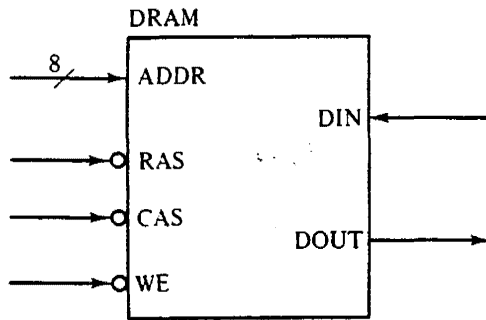


Figure 6.45 Dynamic RAM for Problem 6.32.

- (b) Explain in words the sequence of steps (in terms of signals and order of events) that are required to perform a memory read operation.
 - (c) Explain in words the sequence of steps that are required to perform a memory write operation.
 - (d) Explain in words the sequence of steps that are required to perform a memory refresh operation.
- 6.33. Construct the $16K \times 4$ memory module of Fig. 6.46 by using four $16K \times 1$ dynamic RAMs [as shown in Fig. 6.30(b)], a DRAM controller (similar to the one shown in Fig. 6.32), and any additional logic that is needed. Note that the data lines of the memory module are bidirectional, whereas the data lines of the $16K \times 1$ dynamic RAMs are divided into DIN and DOUT. (*Hint:* Use three-state buffers.)

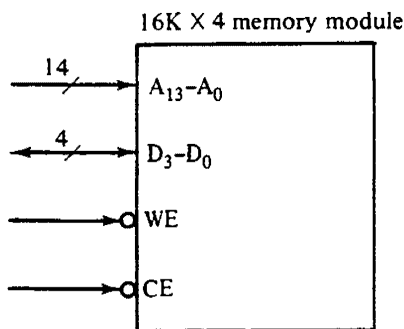


Figure 6.46 Memory module for Problem 6.33.

Digital Circuit Design

7.1 INTRODUCTION

The digital *circuit* design process, as opposed to that of digital *system* design (to be discussed in Chapter 8), begins with a clear and unambiguous requirement specification of the digital circuit. The final product is a detailed design of the circuit. In between is the design process. Like any design process, the digital design process requires a combination of creativity, experience, and understanding of the general design principles. In particular, one does not become a good designer from simply reading a textbook. On the other hand, mindlessly designing digital circuits without any awareness of the general design principles often produces mindless results.

Our purpose in this chapter is to present the general principles of digital circuit design and to discuss various techniques that are useful in the design process, thereby providing a solid foundation on which to build a knowledge of design. But providing the necessary creativity or experience is beyond the scope of this book.

In this chapter we will study the design of *sequential* circuits using circuit elements that were introduced in the preceding chapters. Sequential circuits are classified into two main types: *synchronous* and *asynchronous*. In a synchronous sequential circuit, the circuit elements respond to input signals only at discrete instants of time—at the active transitions of the clock signal. A sequential circuit having this feature is called a *clocked sequential circuit*, as was stated in Chapter 5. All the digital circuits considered in this chapter are clocked sequential circuits.

In an asynchronous sequential circuit, each circuit element operates at its own rate, and there are no clock signals to synchronize operation. Consequently, asynchronous sequential circuits can operate at faster rates than can synchronous sequential circuits. However, there can be serious operational problems because the outputs of the circuit elements depend on the *order* of the change in the input signals. As a result, the element outputs can be transiently unstable and unpredictable. For these and other problems relating to timing, the design of asynchronous sequential circuits is much more difficult

than that of synchronous sequential circuits and is not considered in this introductory text.

We begin with a discussion of the **digital circuit design fundamentals**: the concepts of a controller and the controlled circuit elements, and the various phases of the design process. The next topic is Algorithmic State Machine (ASM) fundamentals, along with the various techniques for directly translating ASM charts into hardware controller circuits. (ASM charts are useful tools in the design and implementation of the controller of a digital circuit.) Finally, a series of detailed design examples are given to illustrate the digital circuit design concepts presented.

7.2 A MODEL FOR DIGITAL CIRCUIT DESIGN

As mentioned, the digital *circuit* design process begins with a clear and unambiguous requirement specification of the digital circuit. The final product is a detailed design of the circuit. In between is the design process. In its most unrefined sense, the digital design process can be viewed as being merely the selecting of the appropriate circuit elements and the interconnecting of them such that they function as specified.

We can formalize this idea into the concept of a *controller* and the *controlled circuit elements*. In other words, a digital circuit design is conceptually divided into two parts, as shown in the general circuit design of Fig. 7.1. The controlled circuit elements comprise a set of circuit elements, like those presented in the preceding chapters, that are selected to implement the functions that are specified for the digital circuit. The controller provides these circuit elements with the appropriate input control signals at every moment in time so that the circuit elements properly implement the specified functions and produce the required external output signals. In this manner, the controller functions as the “brain” of the digital circuit.

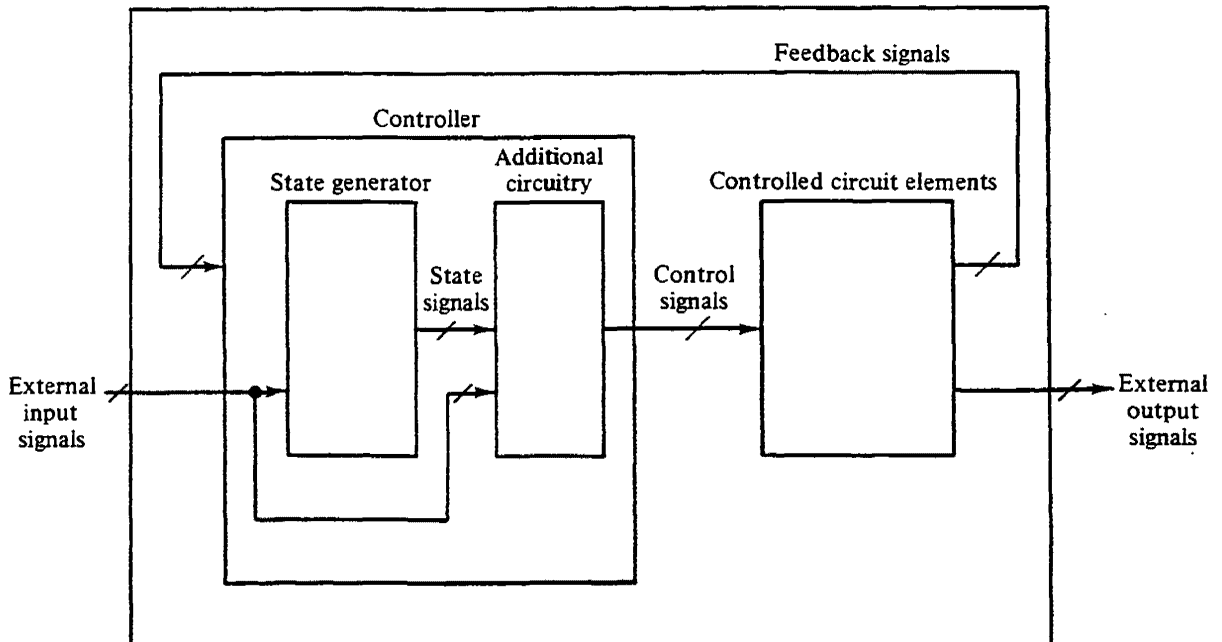


Figure 7.1 General model for digital circuit design.

The controller, itself a digital circuit, consists of a *state generator* and additional circuitry for producing the signals required for controlling the circuit elements. The inputs to the state generator are the external input signals and also feedback signals from the controlled circuit elements. The outputs of the state generator are the *state signals*, each representing a state of the controller. The function of the state generator is to place the controller in the appropriate state at the appropriate time so that it generates the appropriate control signals.

The concepts introduced above are among the most important and fundamental in digital design. Unfortunately, they are also among the most difficult ones to explain and comprehend. The remainder of this chapter will be devoted to making these concepts clearer. More details will be provided. Also, design techniques will be introduced, and design examples given.

7.3 DIGITAL CIRCUIT DESIGN PROCESS

The digital circuit design process can be divided into three major phases:

1. Preliminary design phase
2. Refinement phase
3. Realization phase

These phases occur in the indicated order. Before starting the preliminary design phase of a digital circuit, a designer must be given a well-defined requirement specification of the digital circuit that is to be designed.

In the *preliminary* design phase, the designer makes certain of obtaining a good understanding of the given requirement specification. The designer should develop a block diagram of the overall digital circuit, with the input and output signals well defined, and may use timing diagrams of relevant signals for further clarification. Given that the designer has conceptually formulated a solution for the design problem, the final product of the preliminary design phase is the preliminary design, consisting of the following.

1. A set of major circuit elements with the major data paths defined. The designer should, of course, have some idea of how the major circuit elements eventually will be realized.
2. A preliminary plan (algorithm) for the control of the circuit elements. This control algorithm can be stated in words, or, if the algorithm is sufficiently concrete, in a more formal representation such as a flowchart.

In the *refinement* phase, the designer iteratively refines the circuit design for both the controller and the controlled circuit element parts. For the circuit element part, circuit elements are added or eliminated as the solution becomes more in focus. During this iterative process, the signals of the circuit elements become more defined, and the set of controlled circuit elements converges to a set of actual ICs. Correspondingly, for each iteration of the refinement of the circuit elements, the control algorithm itself becomes more refined. The number of states becomes more stable and the control signals become more defined. Also, the timing among the signals increases in importance. During all

this, the flowchart of the earlier design steps is converging to an ASM chart. Of course, the number of iterations required in the refinement phase depends on the complexity of the design. The end product of this phase is the following:

1. A set of detailed circuit elements with completely defined functions and completely defined signals. At this point of the design the circuit elements are sufficiently defined that they can be realized with available ICs in a straightforward manner.
2. The control algorithm in the form of an ASM chart in which the timing is unambiguously represented.

The final phase of the design process is the *realization* phase. At this point of the design, the hard work is over, and the realization of the detailed circuit elements with available ICs is straightforward. As we will see shortly, the hardware realization of an ASM chart is also straightforward, employing the techniques to be discussed in Sec. 7.5.

Examples are given at the end of this chapter to illustrate these design concepts and the entire design process. Before considering those examples, however, we will study some design tools and techniques that are necessary for digital design.

7.4 ALGORITHMIC STATE MACHINE (ASM)

The design of the controller, and the state generator section in particular, is the most difficult part of the digital circuit design process. If the complexity of the controller is nontrivial, then trying to realize the controller through trial and error is not desirable, even if possible. A systematic design procedure is necessary. Central to such a procedure is an unambiguous notation for representing the control algorithm. This notation enables the designer to bridge the gap between the conceptual control algorithm and the actual hardware realization of that algorithm.

Two characteristics are essential for this notation:

1. For the designer to use it effectively, the notation must provide a clear description of the algorithm, and in terms to which the designer can relate.
2. The notation must support a direct translation into a hardware realization of the control algorithm.

Traditional state diagram methods, such as the Mealy and Moore state machines discussed in Sec. 7.7, satisfy the second condition. Translation from a traditional state diagram to a hardware realization is straightforward. Unfortunately, though, with such diagrams it is difficult to represent complex control algorithms clearly. Moreover, representing a control algorithm with more than a limited number of input and output signals can be unwieldy.

A notation that has both essential characteristics is the Algorithmic State Machine (ASM) chart. Translation from an ASM chart to a hardware realization is practically identical to that for the traditional state diagram. And since the syntax of an ASM chart is very similar to that of a software flowchart, the control algorithm can be expressed in terms that are familiar to the designer.

The most important concept in the algorithm of a controller is the concept of a *state*. The term “state” refers to a stable condition of the controller over a fixed period of time. In terms of a sequential digital circuit, a state is represented by the binary information stored in the memory elements during that period of time. The notation of state will become clearer with the use of the term.

On an ASM chart, a state is represented by a *state box*, which is a rectangle with the name of the state encircled and placed at the upper left corner or at the side of the rectangle. In the ASM chart example of Fig. 7.2(a), there are four states: A, B, C, and D. This chart is for a controller that has two input signals: IN.BIT and BUF.FUL, and three output control signals: COUNT.EN, REG.LD, and OUT.FLAG.

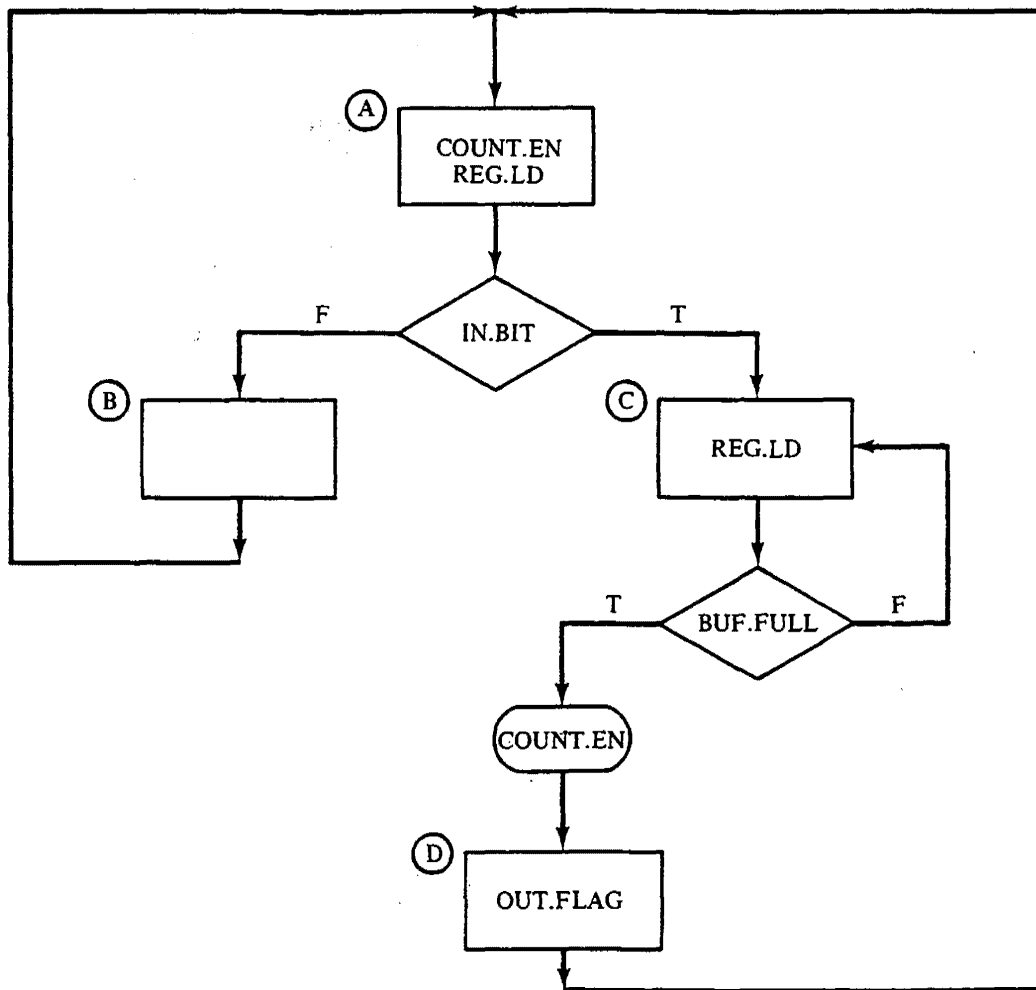
Over time the controller, guided by the control algorithm, moves through a sequence of states. The state transition from the *present* state of the algorithm to the *next* state of the algorithm occurs at the active edge (leading edge in this case) of the system clock signal. In between state transitions the controller is stable.

There are two types of state transitions: *unconditional* and *conditional*. For an unconditional state transition the next state depends only on the present state of the controller and not on any input signals. As an illustration, in Fig. 7.2(a) the transition from state B to state A is unconditional. In other words, if the present state of the controller is state B, then the next state is state A regardless of the input signals IN.BIT and BUF.FULL. Similarly, the transition from state D to state A is unconditional.

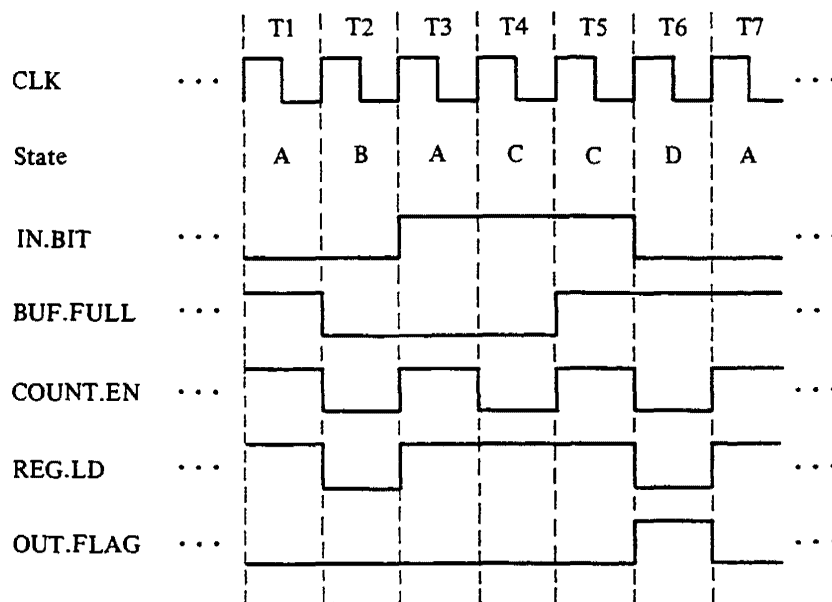
In a conditional state transition the next state depends not only on the present state but also on the present values of the input signals. In an ASM chart a conditional state transition is represented by a *decision diamond*. Note from Fig. 7.2(a) that if the controller present state is state A, then the next state is either state B or state C, depending on the value of the input signal IN.BIT. As you can see from Fig. 7.2(b), the decision is made at the *end* of the present clock cycle, at the next active edge (leading edge) of the system clock signal. For example, for state time T1 in Fig. 7.2(b) the value of IN.BIT is false at the end of T1, and so the next state is state B. On the other hand, at the end of T3 the value of IN.BIT is true, and therefore the next state is state C.

ASM charts also specify controller output values. There are two types of controller outputs: *unconditional* and *conditional*. They differ in that the value of an unconditional output depends only on the present state, but the value of a conditional output depends not only on the present state but also on the present values of the input signals. Unconditional outputs are specified within the state boxes as shown for state boxes A, C, and D in Fig. 7.2(a). As will be described in Sec. 7.7, unconditional outputs are essentially Moore state machine outputs. Conditional outputs are specified in an oval associated with the state and its decision diamond. For the example of Fig. 7.2(a), the only conditional output is the COUNT.EN associated with state C. Conditional outputs are essentially Mealy state machine outputs.

Note from Fig. 7.2(a) and (b) that the unconditional outputs COUNT.EN and REG.LD are true every time the controller is in state A (T1, T3, and T7), and that unconditional output REG.LD is also true every time the controller is in state C (T4 and T5). Further, unconditional output OUT.FLAG is true when the controller is in state D (T6). Finally, the conditional output COUNT.EN is true when the controller is in state C only if the input signal BUF.FULL is true (T5 and not T4).



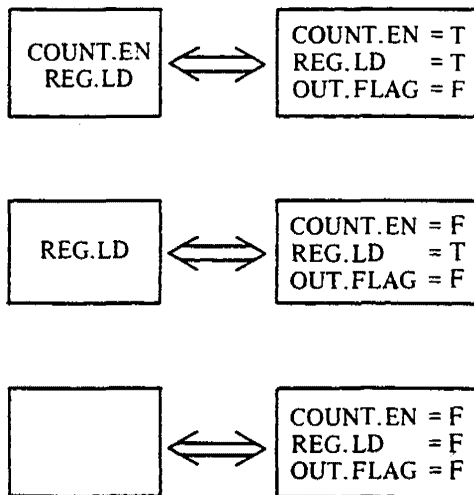
(a)



(b)

Figure 7.2 Example ASM chart and timing diagram.

Conceptually, we must specify the values for every output signal for every state. So, for the example of Fig. 7.2(a), we should specify the values of COUNT.EN, REG.LD, and OUT.FLAG for every state. But to do so would unnecessarily clutter the ASM chart. Therefore, we will adopt the following convention. For each state we will specify only those control signals having a true value; we will not show those output signals having false values in that state. Following are some examples of equivalent notations:



7.5 TRANSLATION FROM ASM CHART TO HARDWARE REALIZATION

In this section we will study some systematic methods for translating an ASM chart representation of the controller into a hardware realization. In most of the examples that illustrate these methods, D flip-flops will be used. Other types of flip-flops, however, could be used just as well.

7.5.1 Code Assignment

We can use a binary code to represent the states of a controller. For the ASM chart of Fig. 7.2(a), a 2-bit code suffices for representing the four states A, B, C, and D. For them we will arbitrarily make the following assignments:

	C1	C0
State A:	0	0
State B:	0	1
State C:	1	0
State D:	1	1

In general, an N -bit code can represent 2^N states.

The 2 bits of the code can correspond to the outputs of two D flip-flops, as in the state generator circuit of Fig. 7.3. The output signal C1 of one flip-flop corresponds to the second bit of the code, and the output signal C0 of the other flip-flop corresponds to

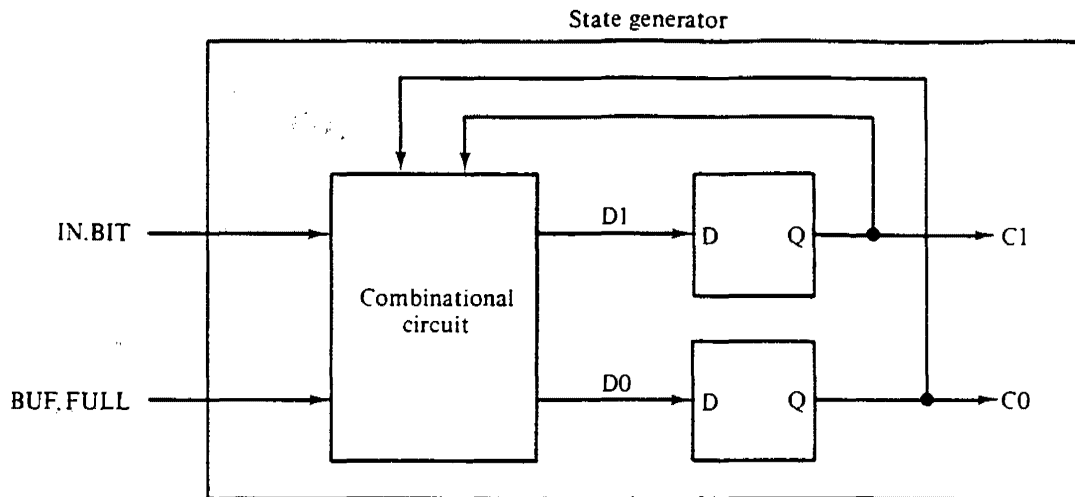


Figure 7.3 State generator circuit.

the first bit. From the ASM chart of Fig. 7.2(a) we can determine how this circuit must function. If the present state of the controller is state A and the input IN.BIT is true at the end of the present state time, then from Fig. 7.2(a) the next state of the controller is state C. Thus, for the state generator circuit of Fig. 7.3, if the present state of the controller is state A ($C1 = 0, C0 = 0$), and the input IN.BIT is 1 (true) at the end of the present clock cycle, then the combinational circuit should be designed such that $D1 = 1$ and $D0 = 0$, so that at the next active edge of the clock signal, $C1 = 1$ and $C0 = 0$. Referring back to Fig. 7.2(a), if the present controller state is C and the input BUF.FULL is true at the end of the present state time, then the next controller state is state D. Consequently, for the state generator circuit of Fig. 7.3, if the present state of the controller is state C ($C1 = 1, C0 = 0$), and the input signal BUF.FULL is 1 (true), then the combinational circuit should produce $D1 = 1$ and $D0 = 1$, so that at the next active edge of the clock cycle, $C1 = 1$ and $C0 = 1$, and so forth. In the following sections, we will consider three systematic methods for designing and realizing such a combinational circuit.

7.5.2 Traditional Method with D Flip-Flops

In the traditional method of ASM realization we follow the same procedure we used in Chapter 5 for the design of counters and shift registers. Basically, we just derive a next-state table for the state generator circuit, and then determine the input equations for the flip-flops. We will use the ASM chart of Fig. 7.2(a) to illustrate this method. This chart is shown in Fig. 7.4 along with the arbitrary code assignments for each state specified at the upper right-hand corner of each state box: state A is 00, state B is 01, state C is 10, and state D is 11.

Using information obtained from this ASM chart, we can form the next-state table of Table 7.1 for the state generator circuit. Each row of Table 7.1 contains the present values of the flip-flop output signals $C1$ and $C0$, the present values of the input signals IN.BIT and BUF.FULL, and the desired values for the flip-flop outputs ($C1^+$ and $C0^+$) immediately after the next active clock edge. For example, the first row shows that when

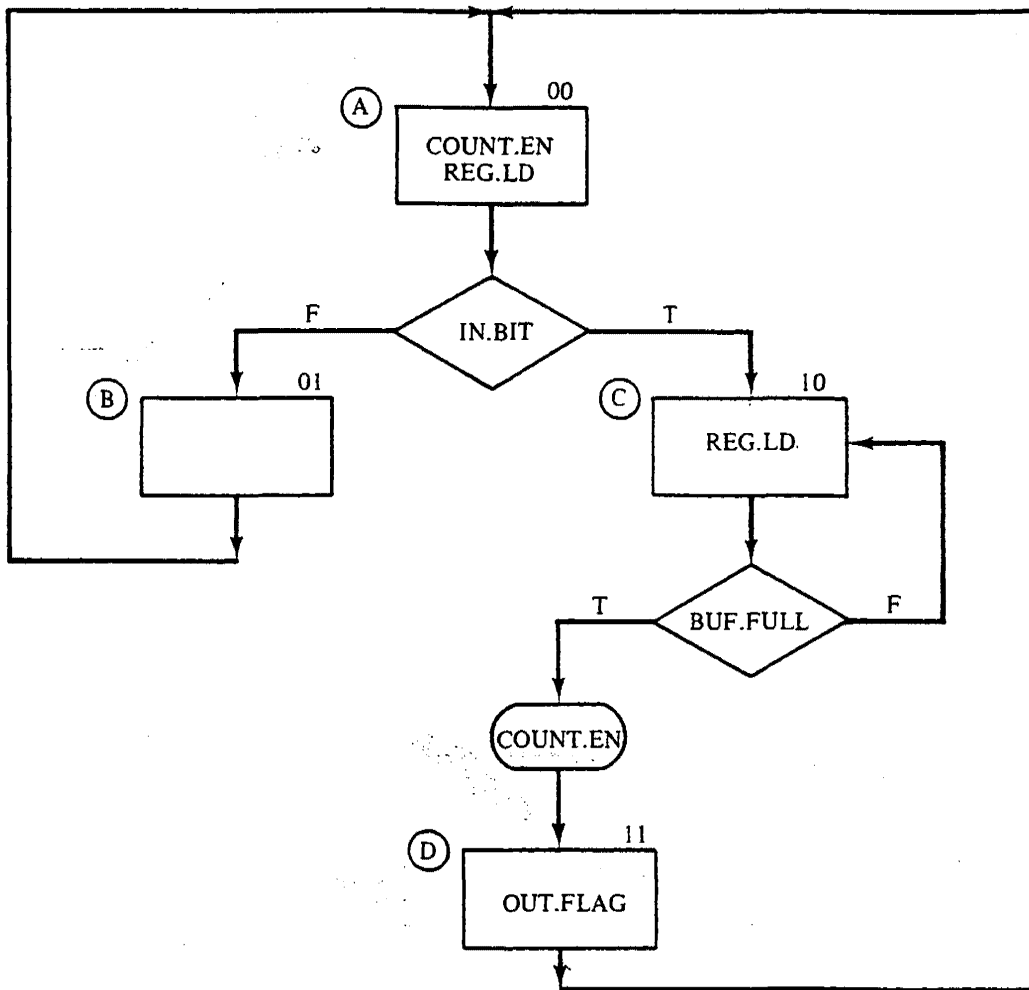


Figure 7.4 Example ASM chart with state assignments.

the present state of the controller is A ($C1 = 0$ and $C0 = 0$), and the inputs are $IN.BIT = 0$ and $BUF.FULL = 0$, then the desired next state of the controller is B ($C1 = 0$ and $C0 = 1$). Similarly, the third row shows that if the present state of the controller is A ($C1 = 0$ and $C0 = 0$) and the inputs are $IN.BIT = 1$ and $BUF.FULL = 0$, then the desired next state is C ($C1 = 1$ and $C0 = 0$). Finally, the last four rows of the table show that if the present state is D ($C1 = 1$ and $C0 = 1$), then the next state is A ($C1 = 0$ and $C0 = 0$), regardless of the values of $IN.BIT$ and $BUF.FULL$.

To design the combinational circuit of Fig. 7.3, we need the logic equations for the inputs $D1$ and $D0$ of the D flip-flops. As in the designs of the counters in Chapter 5, we can get these equations by making a new table from Table 7.1, using the excitation table for the D flip-flop:

D	Q	Q ⁺
0	0	0
0	1	0
1	0	1
1	1	1

characteristic table

rearrange
→

Q	Q ⁺	D
0	0	0
0	1	1
1	0	0
1	1	1

excitation table

The result is shown in Table 7.2.

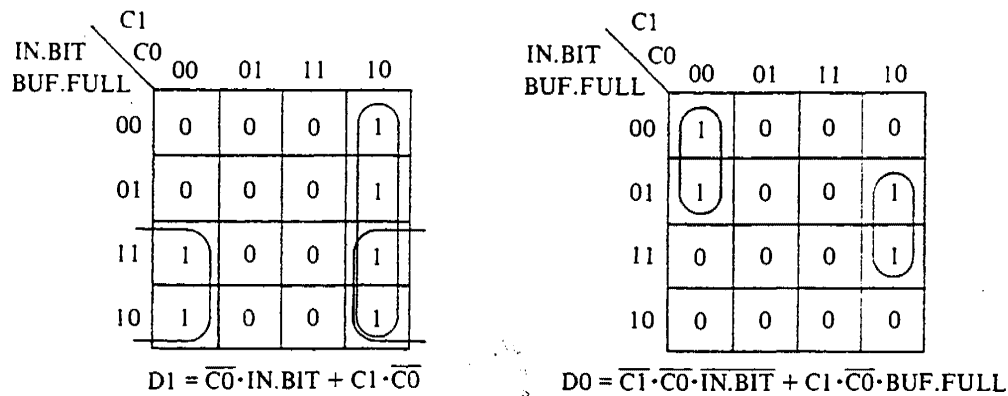
TABLE 7.1 NEXT-STATE TABLE FOR THE STATE GENERATOR CIRCUIT

Present-state code		Inputs		Next-state code	
C1	C0	IN.BIT	BUF.FULL	C1 ⁺	C0 ⁺
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

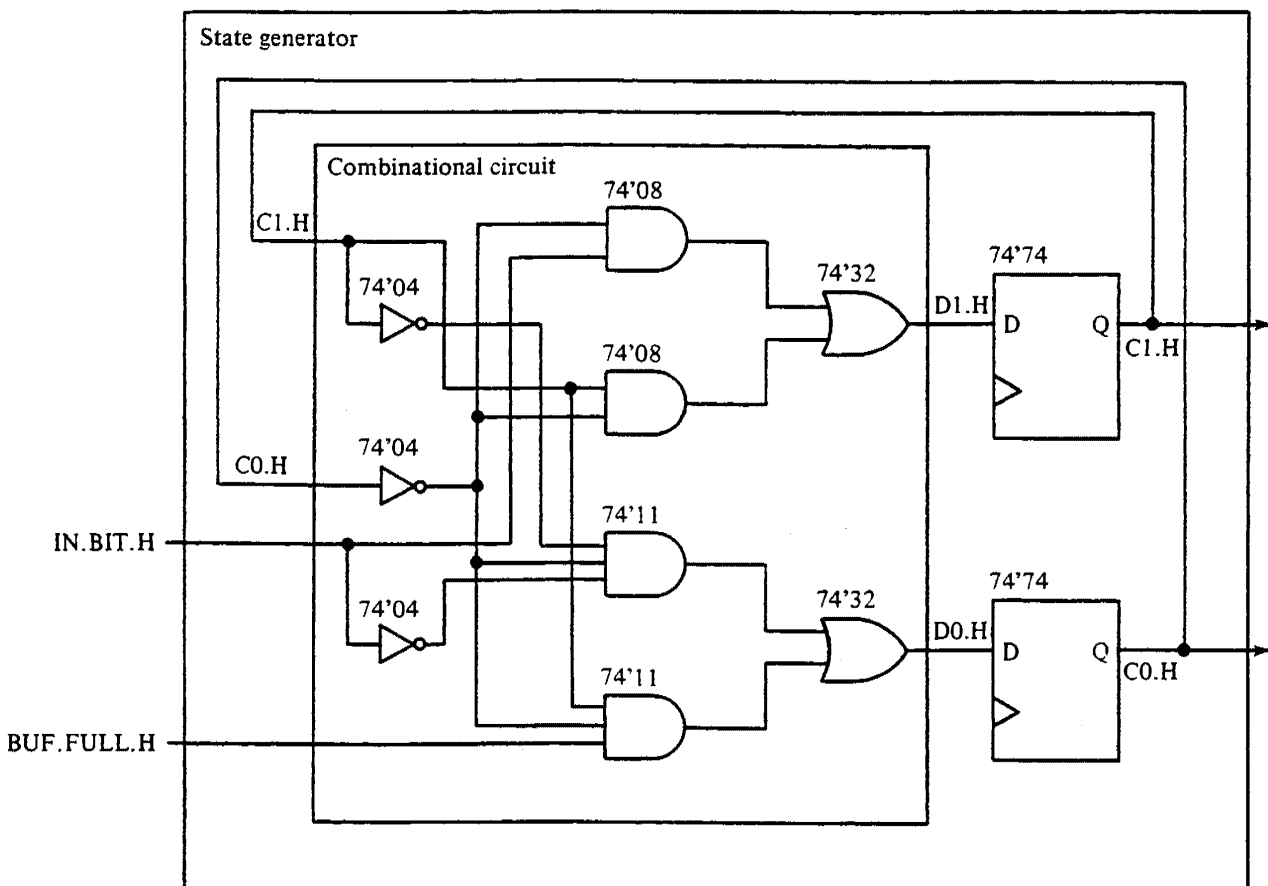
TABLE 7.2 NEXT-STATE TABLE WITH CORRESPONDING D FLIP-FLOP INPUTS

C1	C0	IN.BIT	BUF.FULL	C1 ⁺	C0 ⁺	D1	D0
0	0	0	0	0	1	0	1
0	0	0	1	0	1	0	1
0	0	1	0	1	0	1	0
0	0	1	1	1	0	1	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	1	1	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	1	1	1
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0

For Table 7.2, we can readily determine the D1 and D0 entries by recalling from Chapter 5 that each row of the D flip-flop excitation table indicates the value required for the D input for the desired D flip-flop output transition. As indicated at the top of Table 7.2, the values of D1 are derived from those of C1 and C1⁺ in agreement with the D flip-flop excitation table. For example, in row 3, for the transition from C1 = 0 to C1⁺ = 1, the value of D1 must be 1. In row 13, for the transition from C1 = 1 to C1⁺ = 0, the value of D1 must be 0, and so forth. Similarly, the values of D0 are derived from the values of C0 and C0⁺.



(a)



(b)

Figure 7.5 D flip-flop state generator realization.

Using Table 7.2 we can derive the logic equations for D1 and D0 as functions of C1, C0, IN.BIT, and BUF.FULL. The K-maps for D1 and D0 are given in Fig. 7.5(a) and the resulting circuit diagram in Fig. 7.5(b).

To complete the controller design, we must have a circuit for decoding the state code and producing the control signals REG.LD, OUT.FLAG, and COUNT.EN, which are the ASM outputs. From the ASM chart of Fig. 7.4 we see that REG.LD = 1 when the controller is in state A or state C. Consequently, REG.LD = A + C. Similarly, OUT.FLAG = D. The third output, COUNT.EN, is 1 if the controller is in state A or if it is in state C and if BUF.FULL = 1. Therefore, COUNT.EN = A + C·BUF.FULL. Using a 74'139 decoder, we can decode the state code. The complete controller circuit is shown in Fig. 7.6.

7.5.3 PLA/PAL Method of ASM Realization

In the traditional method of ASM realization that we have just considered, we first derive the next-state table of the state generator circuit (e.g., Table 7.2), then determine the input equations for the flip-flops [e.g., Fig. 7.5(a)], and finally realize the input equations with AND, OR, and NOT gates [e.g., Fig. 7.5(b)]. Note that the combinational circuit of Fig. 7.5(b) is a two-level AND-OR realization. Recall from Sec. 6.4 that a programmable logic array (PLA) or a programmable array logic (PAL) is essentially a two-level AND-OR circuit element. Consequently, we can replace the combinational circuit in Fig. 7.5(b) with a *single* PLA or PAL circuit element. We will now do this with a PLA.

The design procedure for the PLA/PAL method of ASM realization is identical to that of the traditional method up to the realization step. In other words, the PLA/PAL method also involves the derivation of the next-state table, which is Table 7.2 here. From this table we see that the minterm expansions for D1 and D0 are

$$\begin{aligned} D1 &= \overline{C1} \cdot \overline{C0} \cdot \overline{\text{IN.BIT}} \cdot \overline{\text{BUF.FULL}} + \overline{C1} \cdot \overline{C0} \cdot \text{IN.BIT} \cdot \text{BUF.FULL} \\ &\quad + C1 \cdot \overline{C0} \cdot \overline{\text{IN.BIT}} \cdot \overline{\text{BUF.FULL}} + C1 \cdot \overline{C0} \cdot \text{IN.BIT} \cdot \text{BUF.FULL} \\ &\quad + C1 \cdot \overline{C0} \cdot \overline{\text{IN.BIT}} \cdot \text{BUF.FULL} + C1 \cdot \overline{C0} \cdot \text{IN.BIT} \cdot \overline{\text{BUF.FULL}} \\ D0 &= \overline{C1} \cdot \overline{C0} \cdot \overline{\text{IN.BIT}} \cdot \overline{\text{BUF.FULL}} + \overline{C1} \cdot \overline{C0} \cdot \text{IN.BIT} \cdot \text{BUF.FULL} \\ &\quad + C1 \cdot \overline{C0} \cdot \overline{\text{IN.BIT}} \cdot \text{BUF.FULL} + C1 \cdot \overline{C0} \cdot \text{IN.BIT} \cdot \overline{\text{BUF.FULL}} \end{aligned}$$

We can directly realize these logic equations with a single PLA, as shown in Fig. 7.7.

The obvious advantage of this method is the replacement of several IC packages with a single IC. This advantage is more dramatic when the controller circuit is complex and many IC packages are required for the traditional method. This PLA/PAL method is even more attractive with PLAs and PALs that have built-in flip-flops on-board the chips. With one of these PLAs or PALs, we can realize an entire controller with a single chip.

7.5.4 ROM Method of ASM Realization

Recall from Sec. 6.5.2 that a read-only memory (ROM) can also be used to realize random logic. Obviously, we can replace the combinational circuit in Fig. 7.5(b) with a ROM just as we did with a PLA in the last section. In addition, we can use the same ROM to generate the controller outputs. We will now show how to do this.

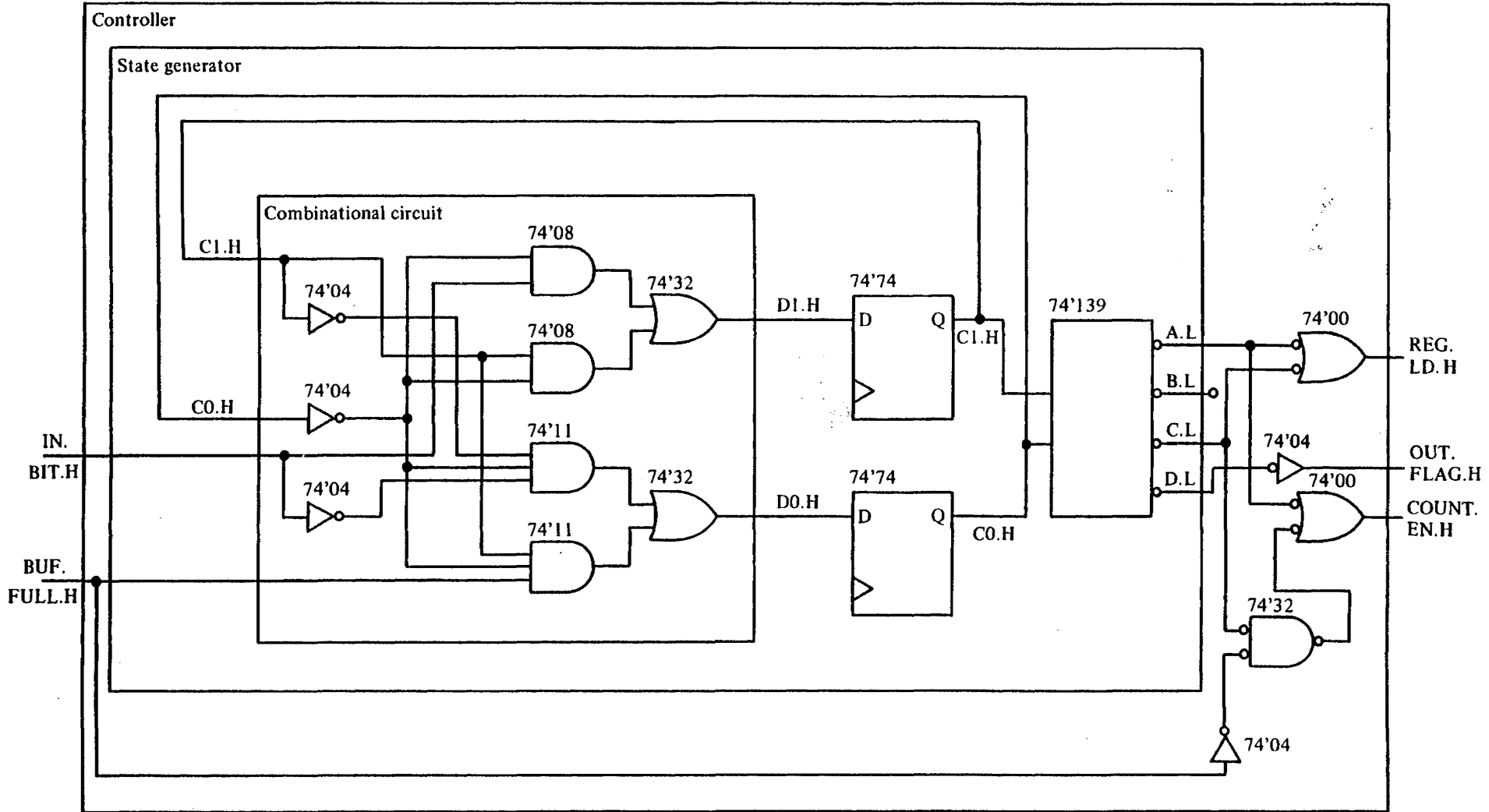


Figure 7.6 Controller realization with D flip-flops for the ASM chart of Fig. 7.4.

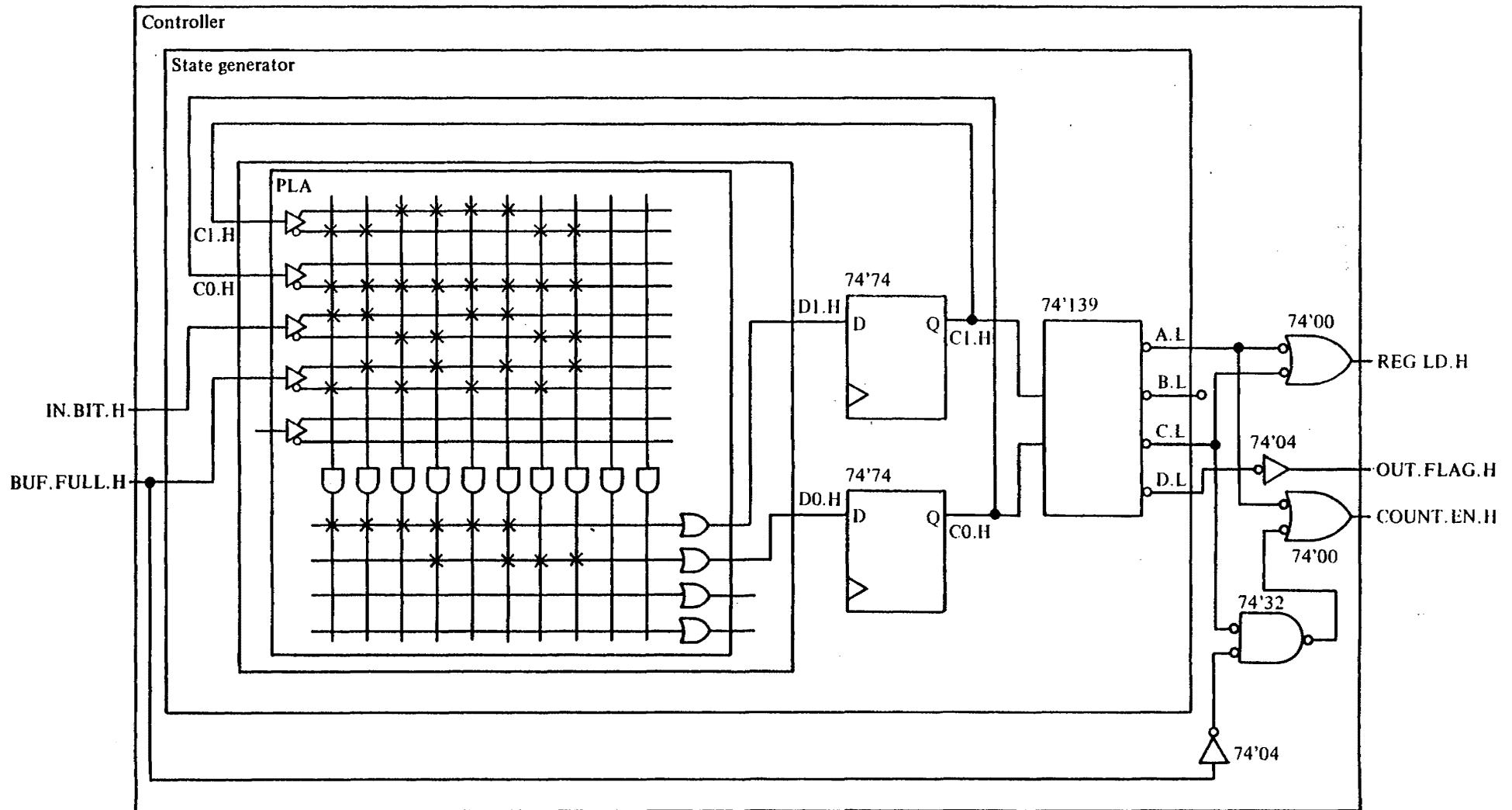


Figure 7.7 Controller with PLA state generator realization for the ASM chart of Fig. 7.4.

In the ROM method of ASM realization, we follow the same initial steps as for the traditional method. Doing this for the ASM chart of Fig. 7.4, we derive, as before, Table 7.2, which is reproduced in Table 7.3 with the addition of the controller output columns. The first row of Table 7.3 shows that if the present-state code is 00 and the inputs are IN.BIT = 0 and BUF.FULL = 0, then the outputs should be REG.LD = 1, OUT.FLAG = 0, and COUNT.EN = 1, as is evident from the ASM chart of Fig. 7.4. Furthermore, the desired next-state code is 01, which implies that the inputs to the D flip-flops are D1 = 0 and D0 = 1. Likewise, all the other entries of Table 7.3 can be verified from the ASM chart of Fig. 7.4.

Using a fictitious 16×5 ROM and two D flip-flops, we can realize the controller circuit represented by Table 7.3. The circuit diagram for the controller is given in Fig. 7.8. As shown, the address bits $A_3, A_2, A_1,$ and A_0 correspond, respectively, to the inputs C1, C0, IN.BIT, and BUF.FULL. Also, the stored bits $Z_4, Z_3, Z_2, Z_1,$ and Z_0 correspond, respectively, to the outputs REG.LD, OUT.FLAG, and COUNT.EN and the D flip-flop inputs D1 and D0. The shown stored values are easy to determine from Table 7.3. For example, for the present-state code of 00 ($C1 = 0, C0 = 0$) and the inputs IN.BIT = 0 and BUF.FULL = 0, the *address* of the memory location to be referenced in the ROM is 0000 ($A_3 = 0, A_2 = 0, A_1 = 0, A_0 = 0$). In this case, the outputs of the ROM are $Z_4 = 1, Z_3 = 0, Z_2 = 1, Z_1 = 0,$ and $Z_0 = 1$. Consequently, the outputs of the controller are REG.LD = 1, OUT.FLAG = 0, and COUNT.EN =

TABLE 7.3 INPUT AND OUTPUT VALUES FOR ROM REALIZATION

Present-state code		Input values for the present state		Output values for the present state			Next-state code		D flip-flop input values	
C1	C0	IN.BIT	BUF.FULL	REG.LD	OUT.FLAG	COUNT.EN	C1 ⁺	C0 ⁺	D1	D0
0	0	0	0	1	0	1	0	1	0	1
0	0	0	1	1	0	1	0	1	0	1
0	0	1	0	1	0	1	1	0	1	0
0	0	1	1	1	0	1	1	0	1	0
0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1	0	1	0
1	0	0	1	1	0	1	1	1	1	1
1	0	1	0	1	0	0	1	0	1	0
1	0	1	1	1	0	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0
1	1	1	1	0	1	0	0	0	0	0

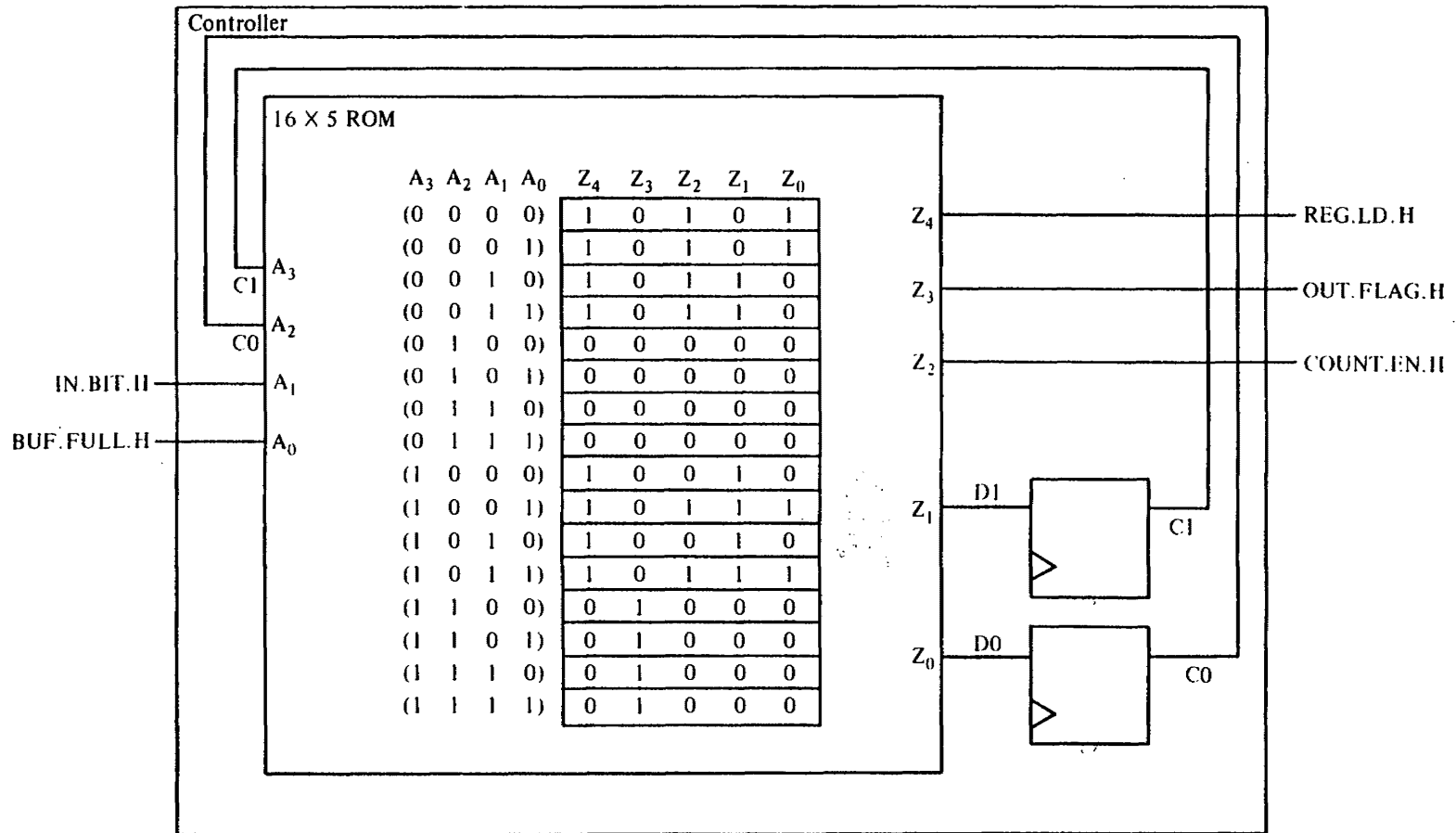


Figure 7.8 ROM realization of the controller for the ASM chart of Fig. 7.4.

1, the same as for the circuit of Fig. 7.6. Other combinations of inputs and present-state codes produce equivalent results. In fact, the circuit of Fig. 7.8 is functionally equivalent to that of Fig. 7.6. In other words, if both circuits were in black boxes, we could not functionally distinguish them—we could not tell one from the other. In effect, instead of designing a circuit by using the traditional method to *realize* Table 7.3, we have *stored* the required values in a table in a ROM. Then when a certain combination of inputs is given, we simply *look up* the desired output values and next-state code.

7.6 AN ADDITIONAL CONTROLLER DESIGN

For this design, the ASM chart shown in Fig. 7.9 will be realized using J-K flip-flops. This ASM chart represents a controller that has one input (IN.BIT) and three outputs (COUNT.EN, FLAG.SET, and COUNT.LD). For the five states, we need a 3-bit code (C2, C1, C0) and a code assignment, which we will arbitrarily make as follows:

State	C2	C1	C0
S0	0	0	0
S1	0	0	1
S2	0	1	0
S3	0	1	1
S4	1	0	0

This assignment is shown in Fig. 7.9. Now we are ready to realize this chart.

Just as we used different types of flip-flops to realize counters in Chapter 5, we can use different types for an ASM realization. The procedure with J-K flip-flops is basically the same as for the D flip-flop procedure, which we have just considered. But, of course, we must determine the logic equations for J and K inputs instead of D inputs.

For an illustration of the J-K procedure we will use J-K flip-flops in a realization of the ASM chart of Fig. 7.9. Since this chart has a 3-bit code, the state generator for the controller must have three J-K flip-flops, as is shown in Fig. 7.10. Of course, the three flip-flop outputs C2, C1, and C0 represent the 3 bits of the code.

To design the combinational circuit of the state generator of Fig. 7.10, we need the logic equations for the flip-flop inputs C2(J), C2(K), C1(J), C1(K), C0(J), and C0(K). As in Chapter 5, these equations are obtained by using the excitation table for the J-K flip-flop, which basically is just a rearrangement of the characteristic table.

J	K	Q	Q ⁺		Q	Q ⁺	J	K
0	0	0	0	rearrange →	0	0	0	X
0	0	1	1		0	1	1	X
0	1	0	0		1	0	X	1
0	1	1	0		1	1	X	0
1	0	0	1					
1	0	1	1					
1	1	0	1					
1	1	1	0					
characteristic table					excitation table			

As is explained in Chapter 5, each row of the excitation table specifies the J and K inputs for the desired flip-flop output transitions.

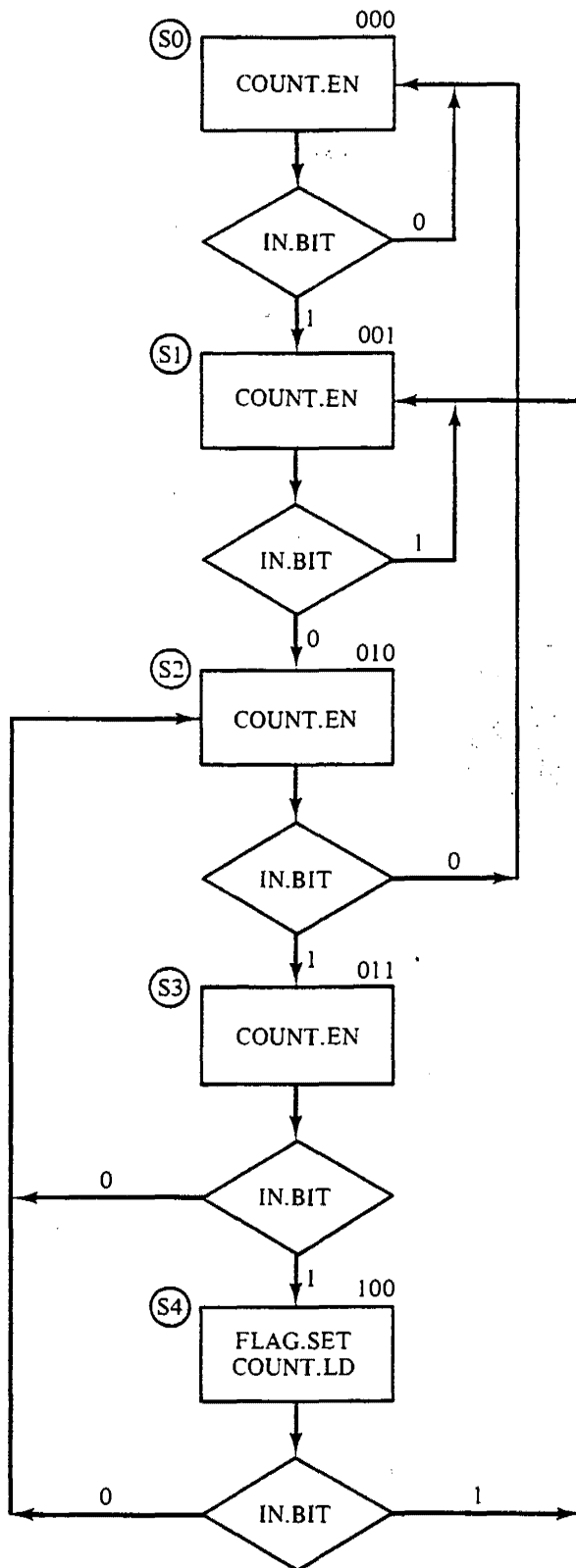


Figure 7.9 ASM chart for a controller that has one input and three outputs.

Using information from the ASM chart of Fig. 7.9, we can readily determine the values of $C2^+$, $C1^+$, and $C0^+$ as a function of IN.BIT and the values of $C2$, $C1$, and $C0$, as shown in Table 7.4. Then, we can use the J-K excitation table to derive the values of $C2(J)$ and $C2(K)$ from the values of $C2$ and $C2^+$. Similarly, we can derive the values of $C1(J)$ and $C1(K)$ from the values of $C1$ and $C1^+$, and the values of $C0(J)$ and $C0(K)$

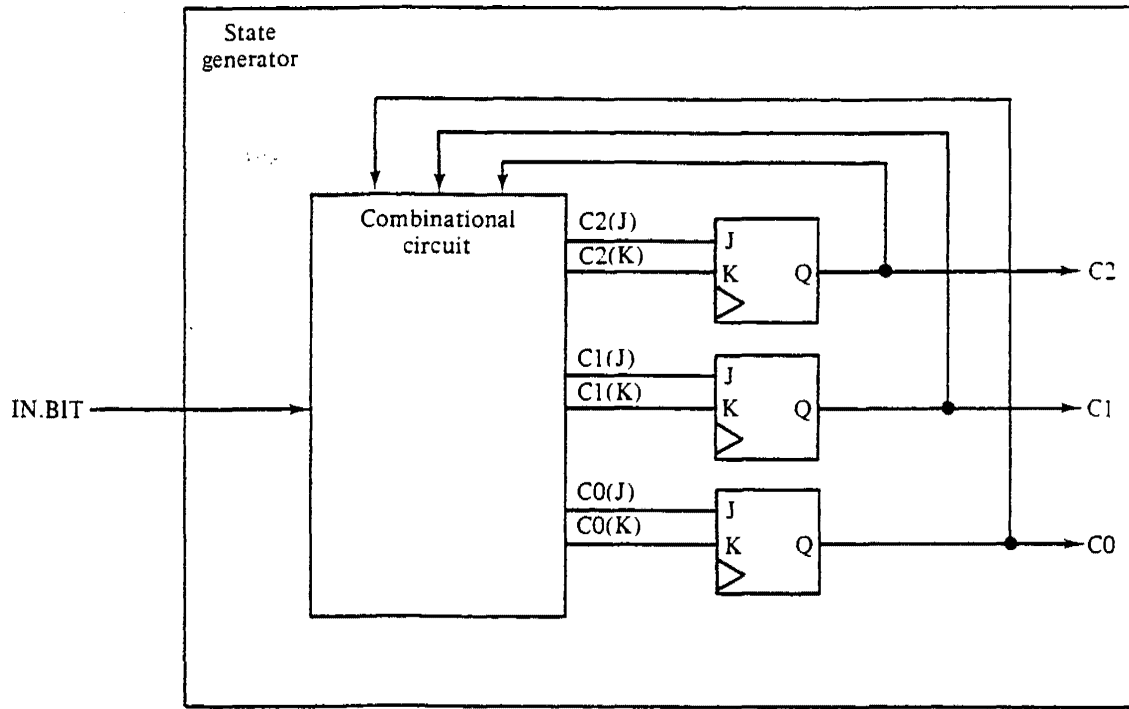


Figure 7.10 State generator circuit for the controller of Fig. 7.9.

from C_0 and C_0^+ . The result is shown in Table 7.4. Incidentally, note in Table 7.4 that for the unused state codes 101, 110, and 111 (rows 11 through 16), the next-state code is arbitrarily specified as 000.

Using Table 7.4 we can derive the logic equations for $C_2(J)$, $C_2(K)$, $C_1(J)$, $C_1(K)$, $C_0(J)$, and $C_0(K)$ as functions of C_2 , C_1 , C_0 , and $IN.BIT$. The K-maps are shown in

TABLE 7.4 J AND K INPUTS FOR PRODUCING THE STATE TRANSITIONS

C_2	C_1	C_0	$IN.BIT$	C_2^+	C_1^+	C_0^+	$C_2(J)$	$C_2(K)$	$C_1(J)$	$C_1(K)$	$C_0(J)$	$C_0(K)$
0	0	0	0	0	0	0	0	X	0	X	0	X
0	0	0	1	0	0	1	0	X	0	X	1	X
0	0	1	0	0	1	0	0	X	1	X	X	1
0	0	1	1	0	0	1	0	X	0	X	X	0
0	1	0	0	0	0	0	0	X	X	1	0	X
0	1	0	1	0	1	1	0	X	X	0	1	X
0	1	1	0	0	1	0	0	X	X	0	X	1
0	1	1	1	1	0	0	1	X	X	1	X	1
1	0	0	0	0	1	0	X	1	1	X	0	X
1	0	0	1	0	0	1	X	1	0	X	1	X
1	0	1	0	0	0	0	X	1	0	X	X	1
1	0	1	1	0	0	0	X	1	0	X	X	1
1	1	0	0	0	0	0	X	1	X	1	0	X
1	1	0	1	0	0	0	X	1	X	1	0	X
1	1	1	0	0	0	0	X	1	X	1	X	1
1	1	1	1	0	0	0	X	1	X	1	X	1

		C2			
		C0\C1			
IN.BIT	00	00	01	11	10
	00	0	0	X	X
	01	0	0	X	X
	11	0	1	X	X
	10	0	0	X	X

$$C2(J) = C1 \cdot C0 \cdot \text{IN.BIT}$$

		C2			
		C0\C1			
IN.BIT	00	X	X	1	1
	01	X	X	1	1
	11	X	X	1	1
	10	X	X	1	1

$$C2(K) = 1$$

		C2			
		C0\C1			
IN.BIT	00	0	X	X	1
	01	0	X	X	0
	11	0	X	X	0
	10	1	X	X	0

$$C1(J) = \overline{C2} \cdot C0 \cdot \text{IN.BIT} + C2 \cdot \overline{C0} \cdot \text{IN.BIT}$$

		C2			
		C0\C1			
IN.BIT	00	X	1	1	X
	01	X	0	1	X
	11	X	1	1	X
	10	X	0	1	X

$$C1(K) = C2 + \overline{C0} \cdot \text{IN.BIT} + C0 \cdot \text{IN.BIT}$$

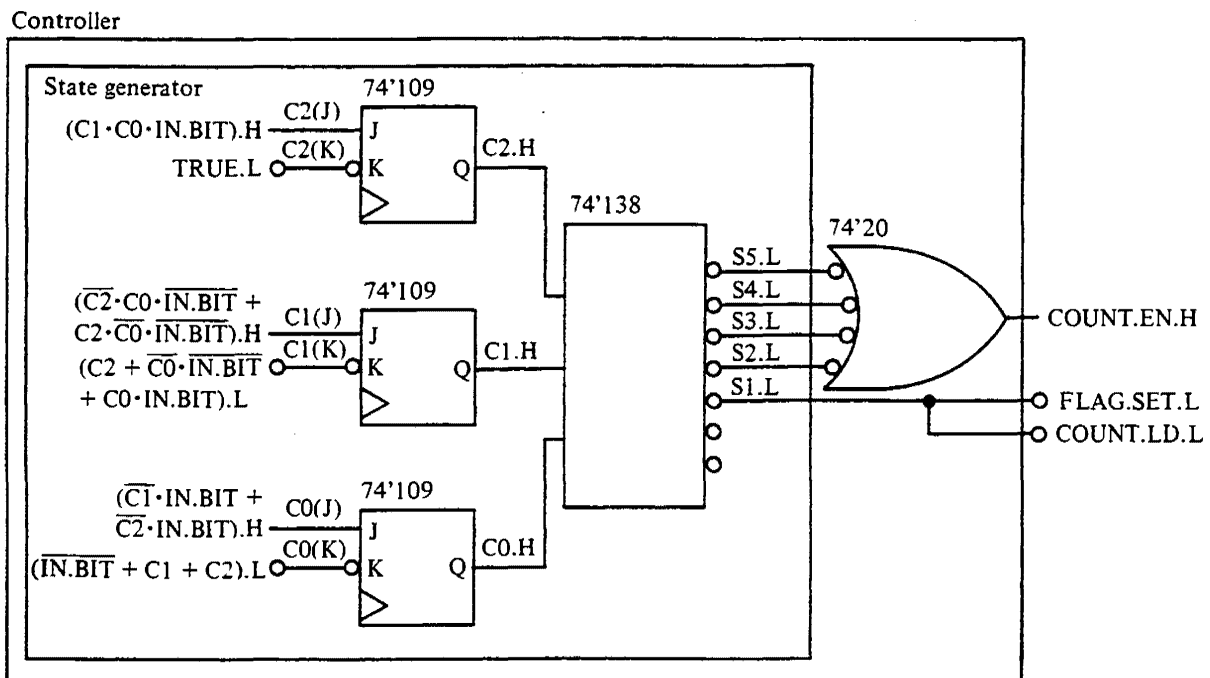
		C2			
		C0\C1			
IN.BIT	00	0	0	0	0
	01	1	1	0	1
	11	X	X	X	X
	10	X	X	X	X

$$C0(J) = \overline{C1} \cdot \text{IN.BIT} + \overline{C2} \cdot \text{IN.BIT}$$

		C2			
		C0\C1			
IN.BIT	00	X	X	X	X
	01	X	X	X	X
	11	0	1	1	1
	10	1	1	1	1

$$C0(K) = \text{IN.BIT} + C1 + C2$$

(a)



(b)

Figure 7.11 J-K flip-flop controller realization.

Fig. 7.11(a), and the resulting circuit diagram, along with the decoded state signals and ASM outputs, is given in Fig. 7.11(b).

7.7 TRADITIONAL STATE MACHINES

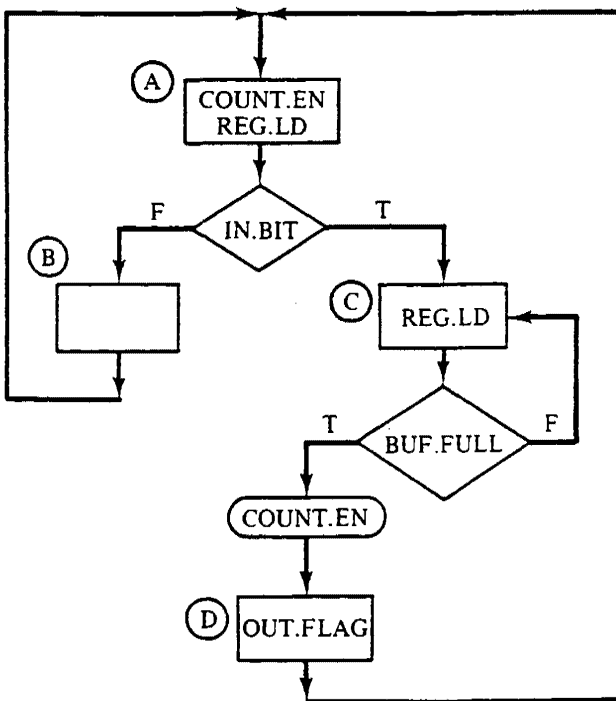
As stated in Sec. 7.4, two characteristics are essential for the notation used to represent the control algorithm of a digital circuit:

1. For the designer to effectively use it, the notation must provide a clear description of the algorithm, and in terms to which the designer can relate.
2. The notation must support a direct translation into a hardware realization of the control algorithm.

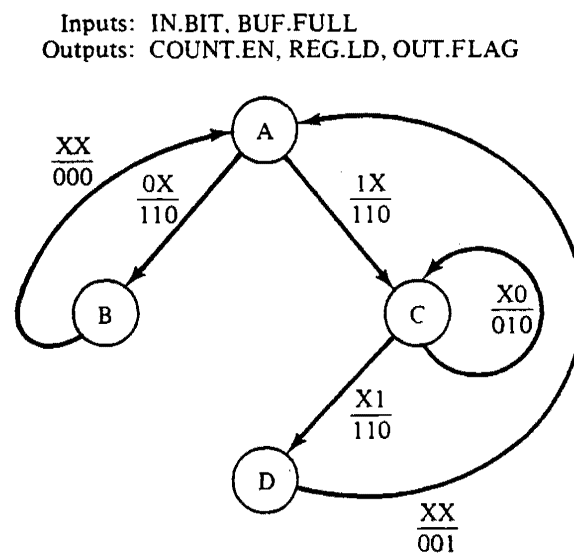
Traditional state-diagram methods, such as the Mealy and Moore state machines, satisfy the second condition. In fact, translation from a traditional state diagram to a hardware realization is practically identical to that for the ASM chart presented in the preceding section. With traditional state diagrams, however, it is difficult to clearly represent complex control algorithms, especially if there are more than a limited number of input and output signals. Furthermore, traditional state diagrams are often not as flexible as the ASM charts, as we will soon see. Even so, traditional state diagrams are still in common use. Therefore, it is beneficial for the reader to gain some exposure to them.

7.7.1 Mealy State Machine

The Mealy state machine is essentially an Algorithmic State Machine (ASM) in which all outputs are represented as conditional outputs. Shown in Fig. 7.12(b) is a Mealy state diagram (or graph) that is equivalent to the ASM chart of Fig. 7.12(a), which is copied from Fig. 7.2(a). A state is represented in a Mealy state graph by a circle. Therefore the four circles in Fig. 7.12(b) represent the four states: A, B, C, and D. As in the ASM



(a) ASM chart



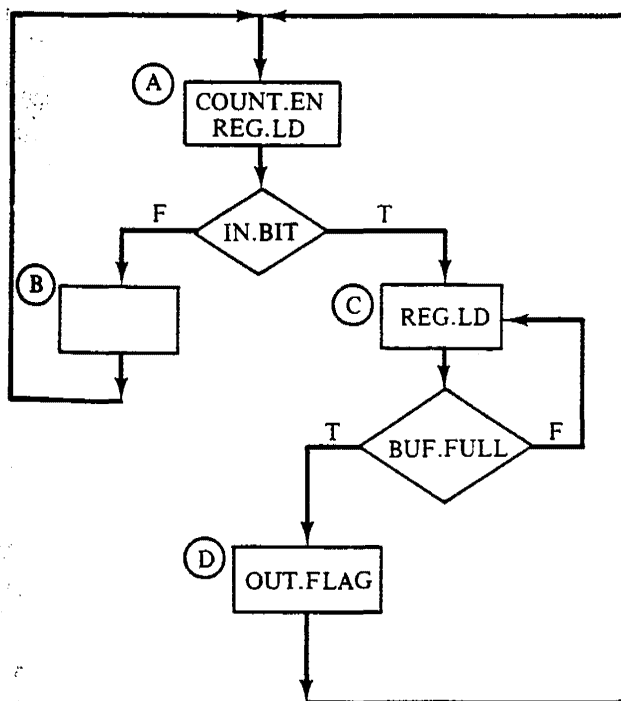
(b) Corresponding Mealy state graph

Figure 7.12 Mealy state graph illustration.

chart, a state transition is represented by an arrow from one state to a next state. Further, specified for each transition arrow are the present values of every input signal and every conditional output signal, with the input values specified on the top and the output values on the bottom. The correspondences of these values to the actual signals are implicit through the ordering. Here, the ordering is IN.BIT, BUF.FULL on each top, and COUNT.EN, REG.LD, OUT.FLAG on each bottom. Again, note that all outputs for the Mealy state diagram are represented as conditional outputs. Therefore, to represent unconditional outputs such as COUNT.EN and REG.LD for state A, we have to specify those output values at both transition arrows from state A, as shown.

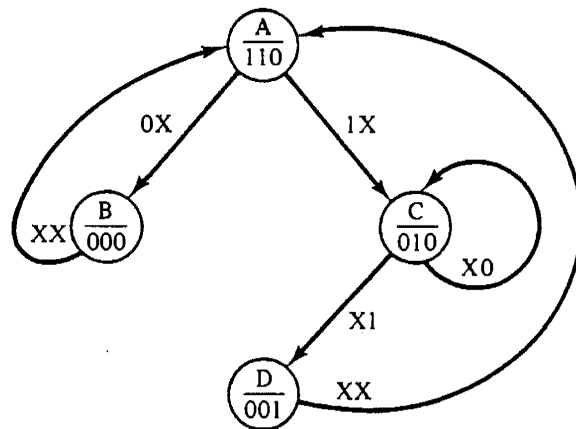
7.7.2 Moore State Machine

The Moore state machine is essentially an ASM in which all outputs are represented as unconditional outputs. Shown in Fig. 7.13(b) is a Moore state diagram that is equivalent to the ASM chart of Fig. 7.13(a). Just as for a Mealy state diagram, a state in a Moore state diagram is represented by a circle, and a state transition is represented by an arrow. However, the specification for each transition arrow has only the present values of the input signals used to determine the next state; there are no output signal values. Instead, the outputs are associated unconditionally with states, being specified inside each state circle under the state name. As a result, a Moore state machine cannot be used to represent the ASM chart of Fig. 7.2 or 7.12(a), where conditional and unconditional outputs are present.



(a) ASM chart

Inputs: IN.BIT, BUF.FULL
Outputs: COUNT.EN, REG.LD, OUT.FLAG



(b) Corresponding Moore state graph

Figure 7.13 Moore state graph illustration.

7.8 DESIGN EXAMPLES

In this section we will consider design examples to illustrate the digital circuit design fundamentals that have been presented in this chapter. In studying these examples the reader is particularly urged to keep in mind the two main concepts in the design of a digital circuit:

1. A digital circuit is conceptually divided into the following:
 - (a) A set of circuit elements.
 - (b) A controller that controls the inputs to these circuit elements.
2. A digital circuit design is stepwise refined through the following phases:
 - (a) The preliminary design phase—which results in block diagrams of the set of circuit elements and a flowchart of the controller.
 - (b) The refinement phase—which results in a set of detailed circuit elements with completely defined functions and signals, and an ASM chart of the controller.
 - (c) The realization phase—which results in a hardware realization with commercially available ICs of the circuit elements and the ASM chart.

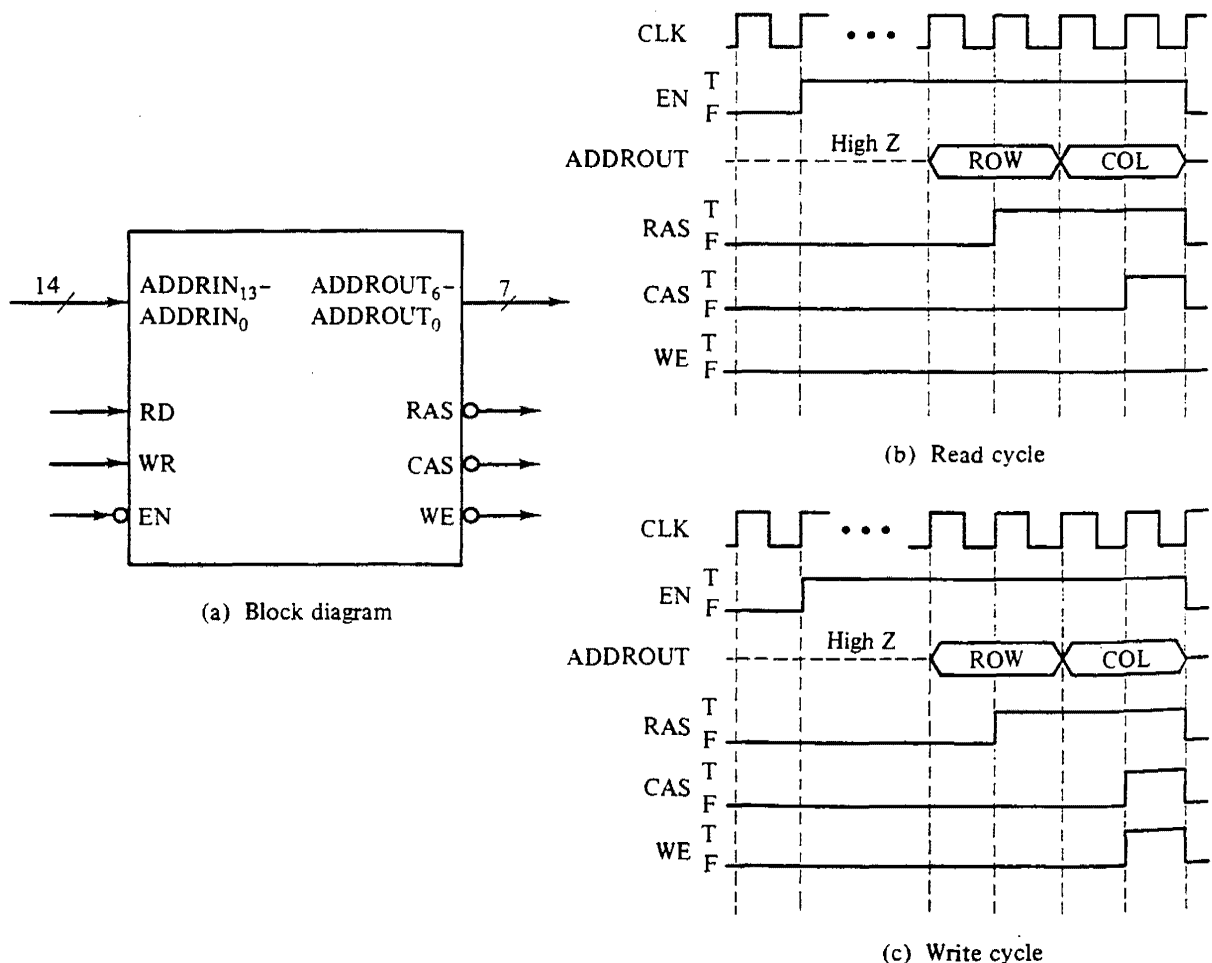
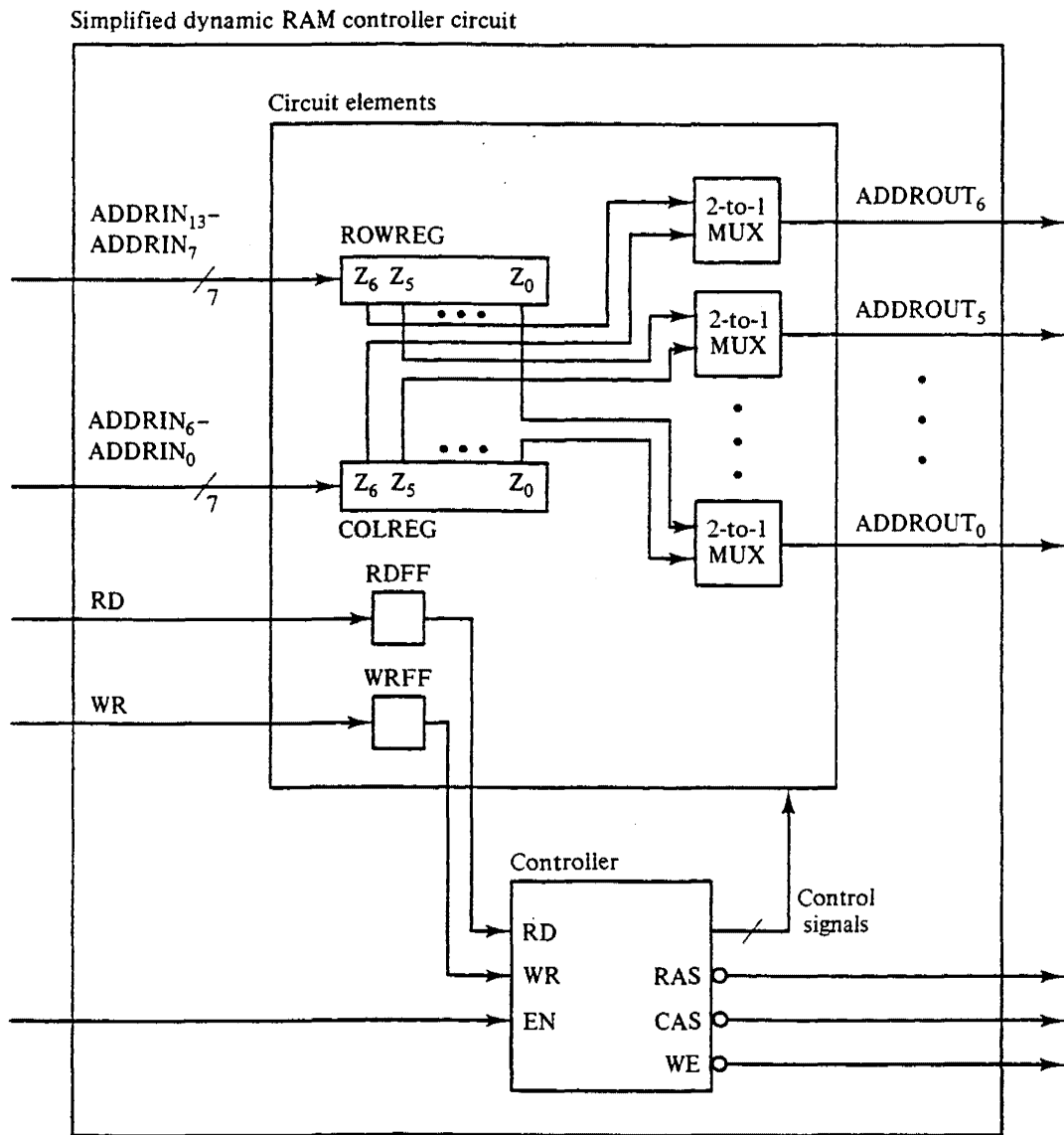


Figure 7.14 Problem specification for the simplified dynamic RAM controller.

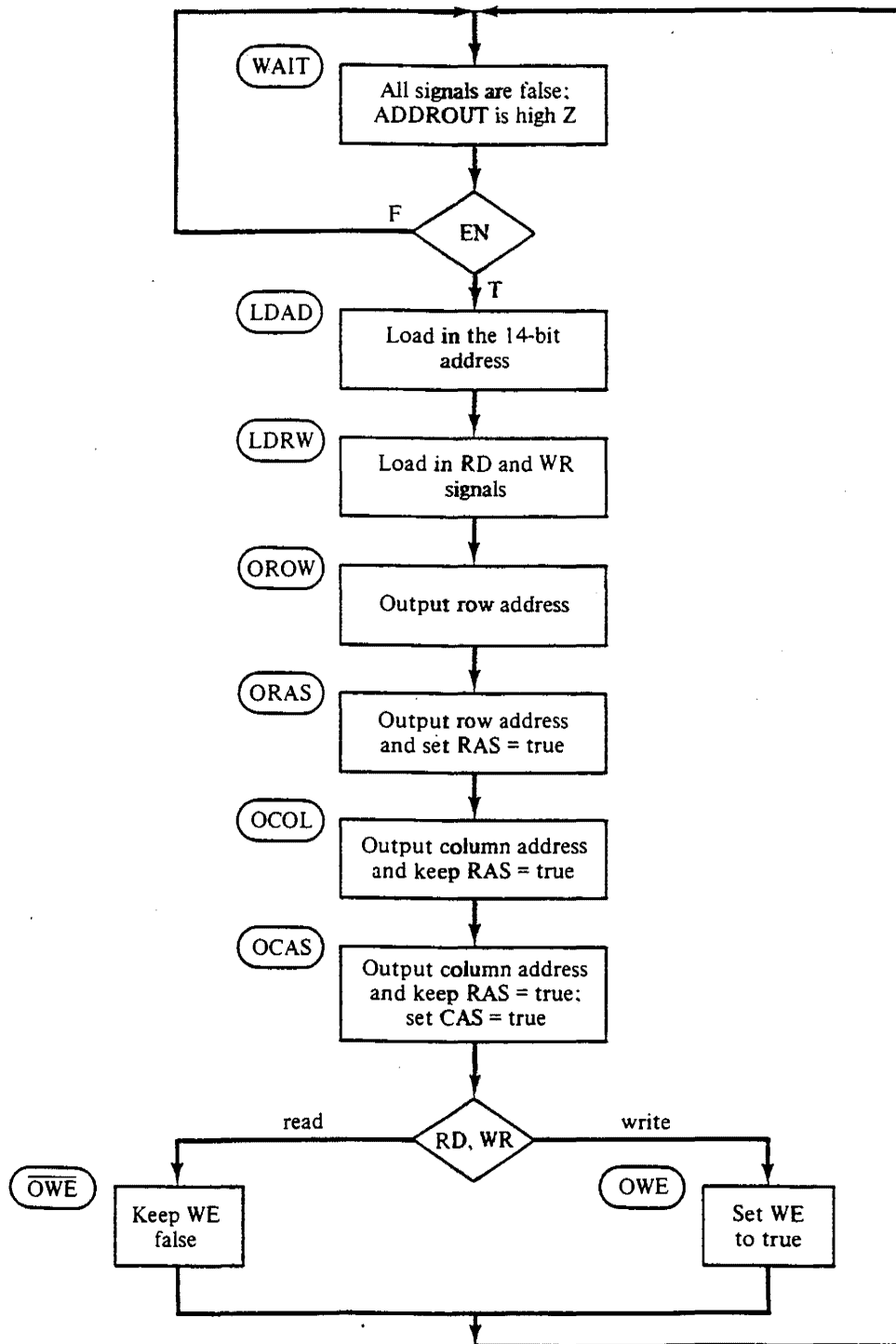
7.8.1 Simplified Dynamic RAM Controller

For this example we are to design a simplified dynamic RAM controller circuit, the block diagram of which is shown in Fig. 7.14(a). This circuit has the following specifications: The inputs to the circuit are a 14-bit address ($ADDRIN_{13}$ – $ADDRIN_0$), a read signal (RD), a write signal (WR), and an enable signal (EN). This circuit does not function until EN is true. When EN becomes true, the 14-bit ADDRIN is loaded in as a row address ($ADDRIN_{13}$ – $ADDRIN_7$) and a column address ($ADDRIN_6$ – $ADDRIN_0$). Also when EN becomes true, the values of RD and WR are latched. Subsequently, the row address is outputted (at ADDRROUT) along with the row address strobe (RAS) signal (at RAS). Then, the column address is outputted (at ADDRROUT) along with the column address strobe (CAS) signal (at CAS). Finally, if the operation is a write operation (RD = false, WR = true), then the WE output is true. Otherwise, for a read operation (RD = true, WR = false), the WE output remains false.



(a) Block diagram of the circuit elements and the controller

Figure 7.15 Preliminary design for the dynamic RAM controller circuit.



(b) Flowchart for the controller

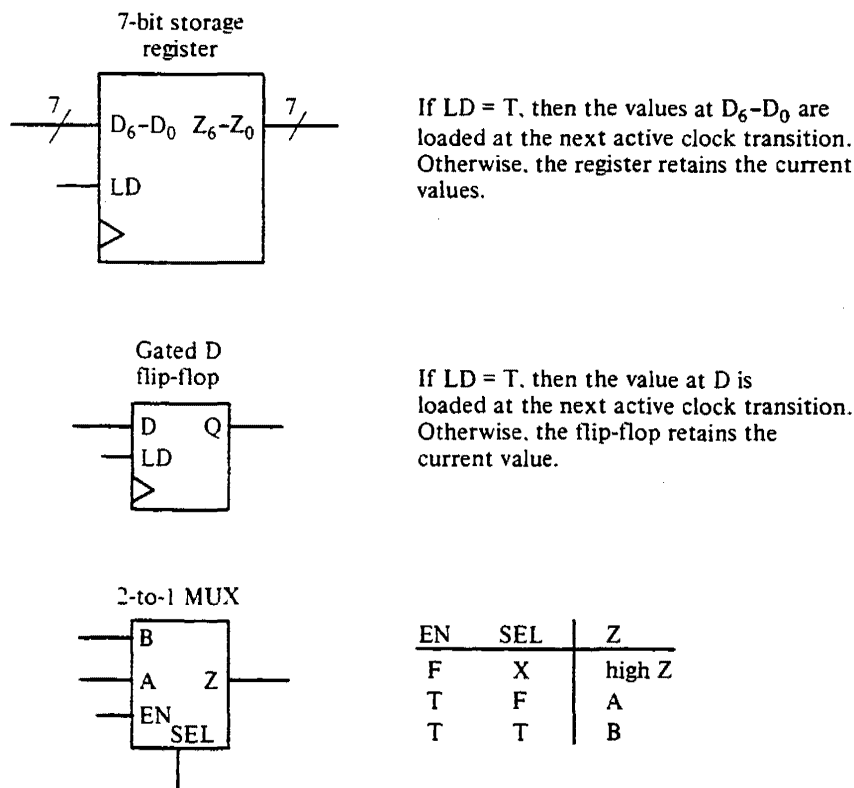
Figure 7.15 (cont.)

The timing of this digital circuit is specified more precisely in the timing diagrams of Figs. 7.14(b) and (c). Note that these diagrams are similar to those of Fig. 6.31 for the read and write cycles of the dynamic RAM presented in Sec. 6.5.3. However, the timing diagrams in Fig. 7.14 are specified at the logic level (T and F), whereas the ones in Fig. 6.31 are specified at the voltage level (H and L). Note also from the timing diagrams in Figs. 7.14(b) and (c) that the row address is outputted at ADDRROUT some

time after EN becomes true. The circuit should be designed so that this time delay is as small as possible.

A preliminary design of the simplified dynamic RAM controller circuit is shown in Fig. 7.15. It should be obvious that we need two 7-bit storage registers to store the row address (ROWREG) and the column address (COLREG), respectively. Also needed are two 1-bit storage registers (flip-flops) to latch the values of RD and WR. All these registers should have a synchronous load input so that when EN becomes true, the controller can generate the load signals to these registers to load in the values. After the row and column addresses are loaded, they are time-multiplexed at the ADDR0UT outputs. More specifically, the row address is outputted first, and then the column address, with both appearing at the same outputs (ADDR0UT). As a result, seven 2-to-1 multiplexers are needed to select either the row address or the column address to be outputted. It is the function of the controller to perform this selection (via the select inputs of the MUXs) along with outputting the appropriate RAS and CAS outputs. Additionally, the controller should output $WE = \text{true}$ for a write operation and $WE = \text{false}$ for a read operation. The control algorithm for the preliminary design is specified in the form of a flowchart in Fig. 7.15(b).

The next phase of the design process is the refinement phase. The first step in it is the refining of the definitions of the circuit elements. At this point of the design we have a good idea of the functions that are required for each of the circuit elements. Consequently, a detailed definition of them is possible. The result is shown in Fig. 7.16(a). Next, we see that the control signals outputted from the controller are now



(a) Detailed specifications for the circuit elements

Figure 7.16 Refined design of the dynamic RAM controller circuit.

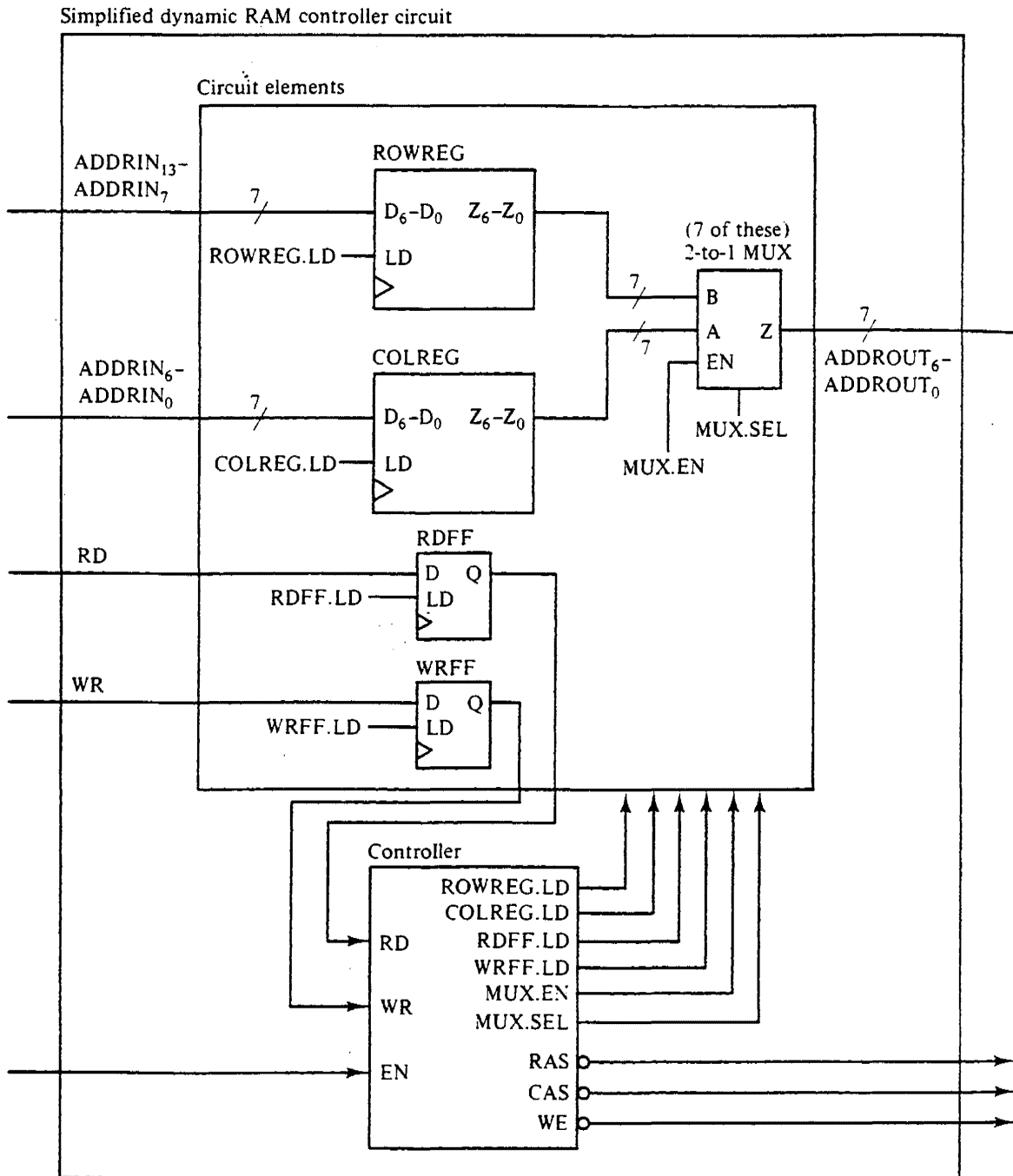
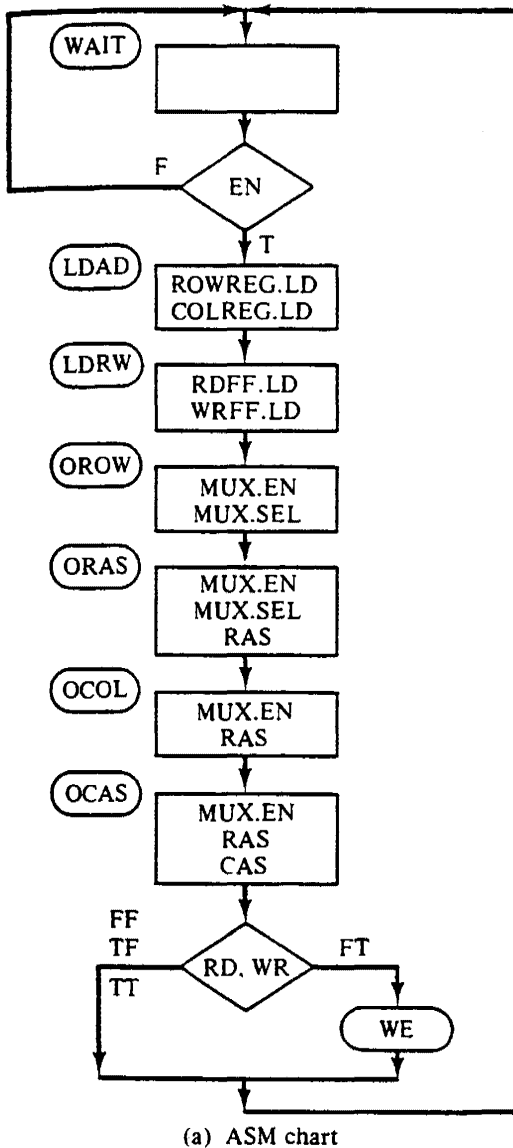


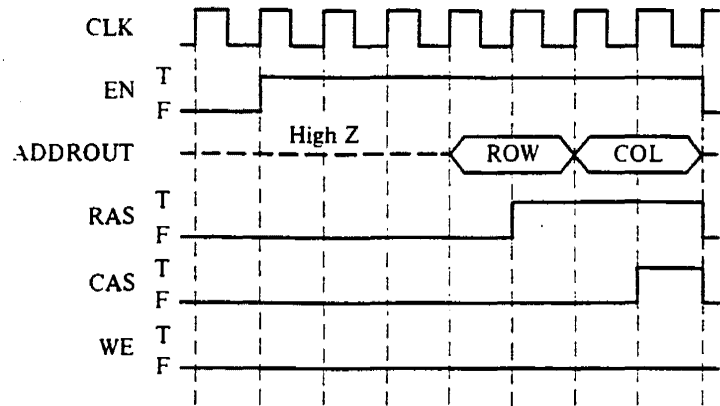
Figure 7.16 (cont.)

defined. They correspond to the control inputs of the circuit elements as shown in Fig. 7.16(b).

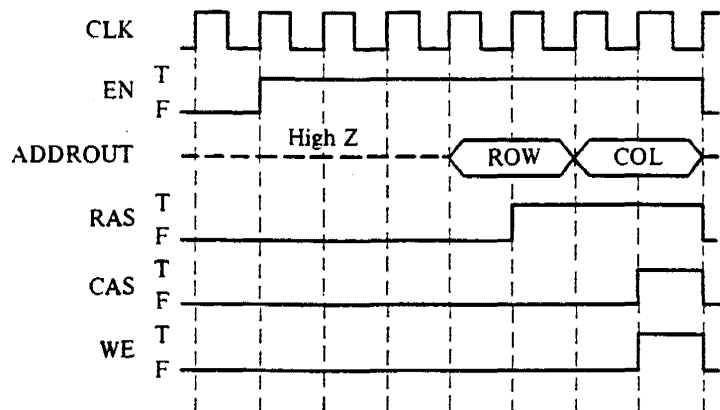
Based on the refined set of circuit elements shown in Fig. 7.16(b), the flowchart of Fig. 7.15(b) can be converted to the ASM chart shown in Fig. 7.17(a). Different from a flowchart, an ASM chart precisely specifies the timing, with each state corresponding to a clock period. Additionally, unconditional and conditional control outputs are speci-



(a) ASM chart



(b) Read cycle



(c) Write cycle

Figure 7.17 An ASM chart for the controller of the dynamic RAM controller circuit.

ified for each state to accomplish the required functions for that state. For example, in the state LDAD, the row address and column address are synchronously loaded into ROWREG and COLREG, respectively, by applying true load signals ROWREG.LD and COLREG.LD during that clock cycle. Note that the state OWE in the original flowchart is changed in the ASM chart to a conditional output that is associated with the state OCAS. This is necessary because we want to output $WE = T$ in the same clock cycle as the CAS output. Also note that in the state OCAS, if $RD, WR = FF$ or TT , there is a default to a read operation ($WE = F$). This is a designer's choice. We could have just as well decided to make the default a write operation. The precise timing of the ASM chart in Fig. 7.17(a) is shown in the corresponding timing diagrams of Figs. 7.17(b) and (c). Note that for a controller realized from this ASM chart, the row address is not outputted until three clock cycles after EN becomes true.

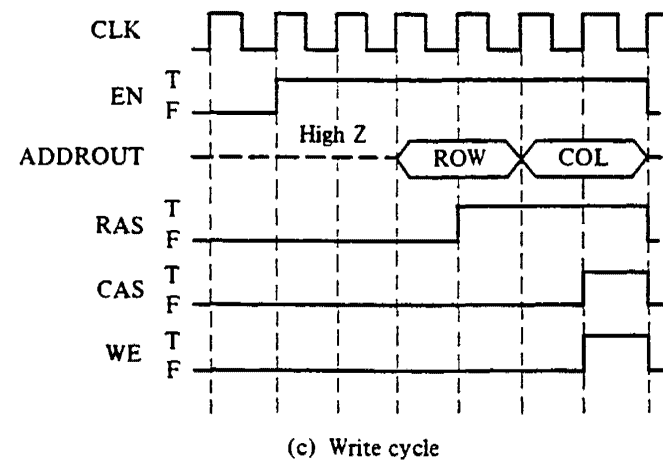
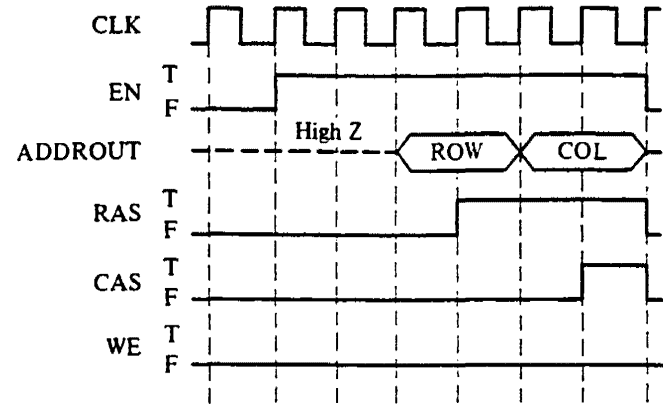
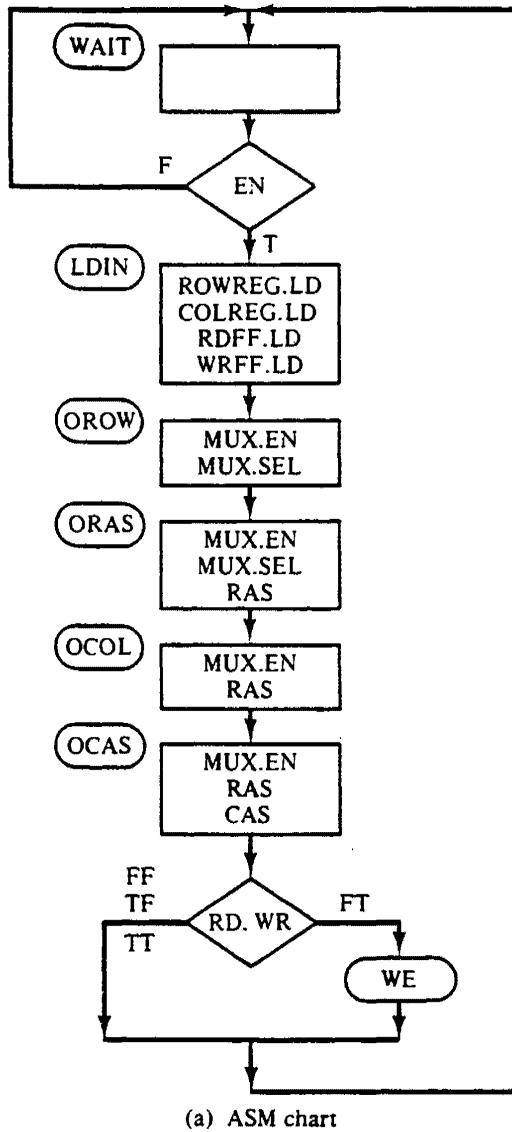


Figure 7.18 First refinement of the ASM chart for the dynamic RAM controller circuit.

Figures 7.18 and 7.19 illustrate stepwise refinements of the controller ASM chart. Note in the ASM chart of Fig. 7.18(a) that the states LDAD and LDRW, from Fig. 7.17(a), are combined into a single state LDIN. The result is that both the row and column addresses are loaded during the same state as the RD and WR signals. From the circuit elements shown in Fig. 7.16(b) we see that this is physically possible and does not introduce any timing problem. Clearly, this joint loading is desirable since it reduces the delay before the row address can be outputted, as shown in Figs. 7.18(b) and (c).

The ASM chart can be further optimized by eliminating the state LDIN by associating conditional outputs with the state WTLD as shown in Fig. 7.19(a). Again, this elimination does not introduce any timing problem. In addition, it reduces the row address output delay by one more clock pulse, as shown in Figs. 7.19(b) and (c).

One more refinement for the digital circuit can be made. We observe that the four load signals ROWREG.LD, COLREG.LD, RDFF.LD, and WRFF.LD are applied in the

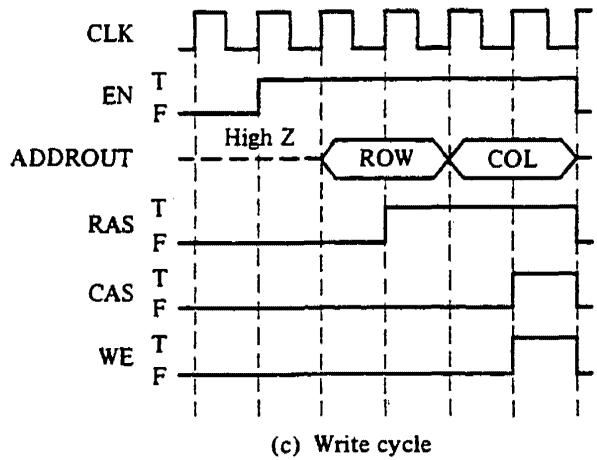
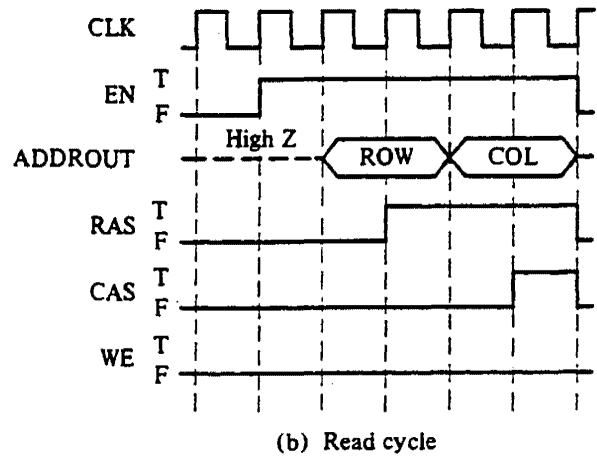
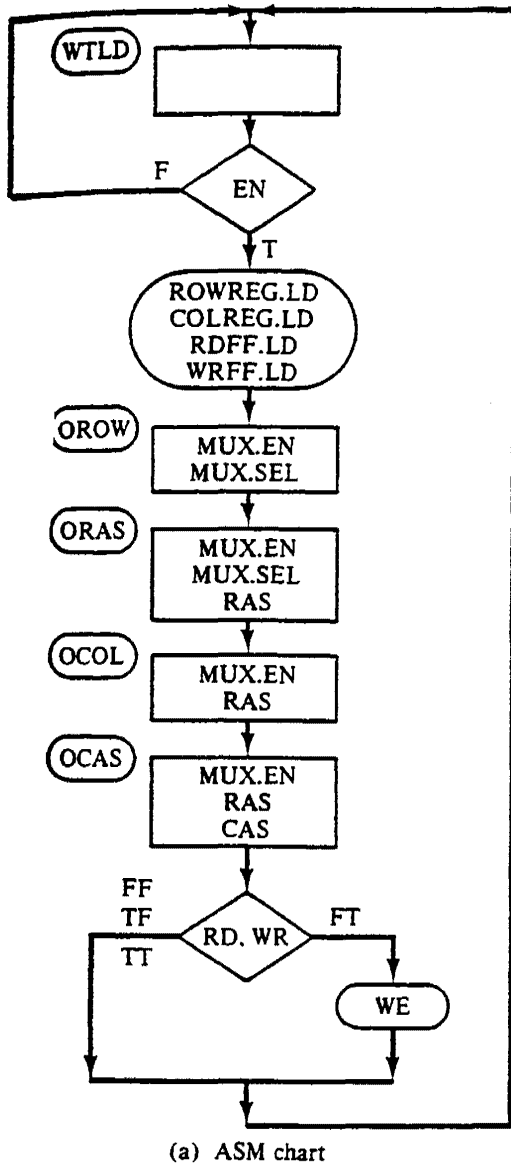


Figure 7.19 Final refinement of the ASM chart for the dynamic controller circuit.

same state and under the same condition. Therefore, they can be replaced by a single load signal, LOAD. So, ROWREG.LD = COLREG.LD = RDFF.LD = WRFF.LD = LOAD. The final ASM chart for the controller is shown in Fig. 7.20.

All the ASM charts in Figs. 7.17(a), 7.18(a), 7.19(a), and 7.20 will function as specified in the original problem statement. However, the ASM charts of Figs. 7.19(a) and 7.20 provide the best performances.

Having gone from the flowchart of Fig. 7.15(b) to the final ASM chart of Fig. 7.20, we can now appreciate the general method for designing the controller of a digital circuit. First, the algorithm for the controller is specified in the form of a flowchart. Then, the flowchart is converted into an ASM chart in which the timing problems are resolved. Finally, the ASM chart is refined and optimized by combining and removing states and by using conditional outputs wherever possible.

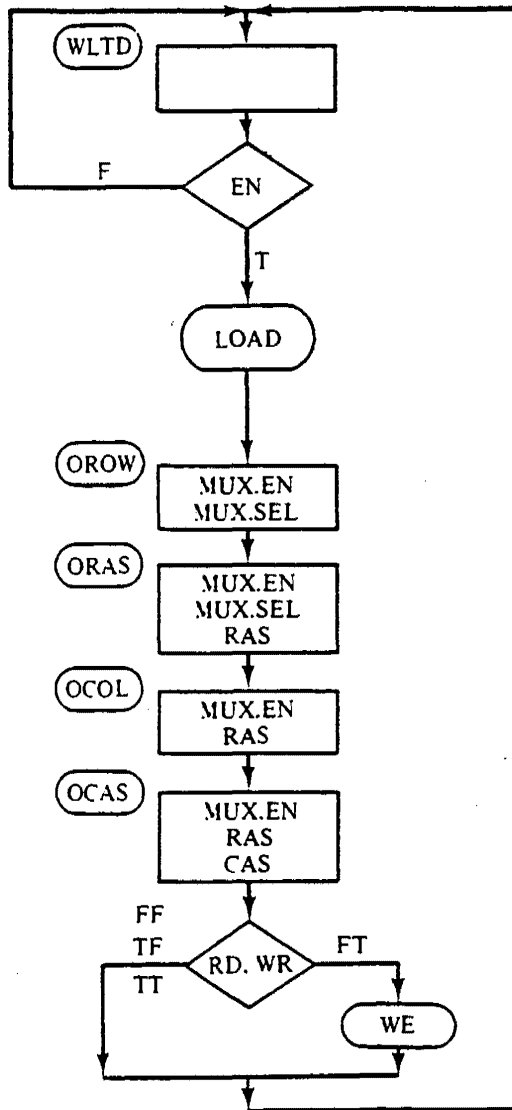


Figure 7.20 Final ASM chart for the dynamic RAM controller.

The final phase of the design process is the realization phase. For this circuit, this phase consists of using commercially available ICs to realize the circuit elements in Fig. 7.16(b) and the controller represented by the ASM chart in Fig. 7.20. As has been mentioned, the realization phase is straightforward. At this point of the design process, the hard work is over. The realization of the simplified dynamic RAM controller circuit is summarized in Figs. 7.21 and 7.22.

As shown in Fig. 7.21, the circuit element ROWREG is realized with two 74'163 counters utilized as storage registers, with enable inputs of $ET = EP = \text{false}$ to prevent counting. The circuit element COLREG is realized in the same manner. The 2-to-1 multiplexers with three-state outputs are directly available commercially as ICs (74'257). The RDFF and WRFF are D flip-flops with a synchronous load input. This type of flip-flop is not directly available as a commercial IC; nevertheless, a gated D flip-flop can be easily designed and, in fact, was designed as an example in Sec. 5.3.4 and shown in

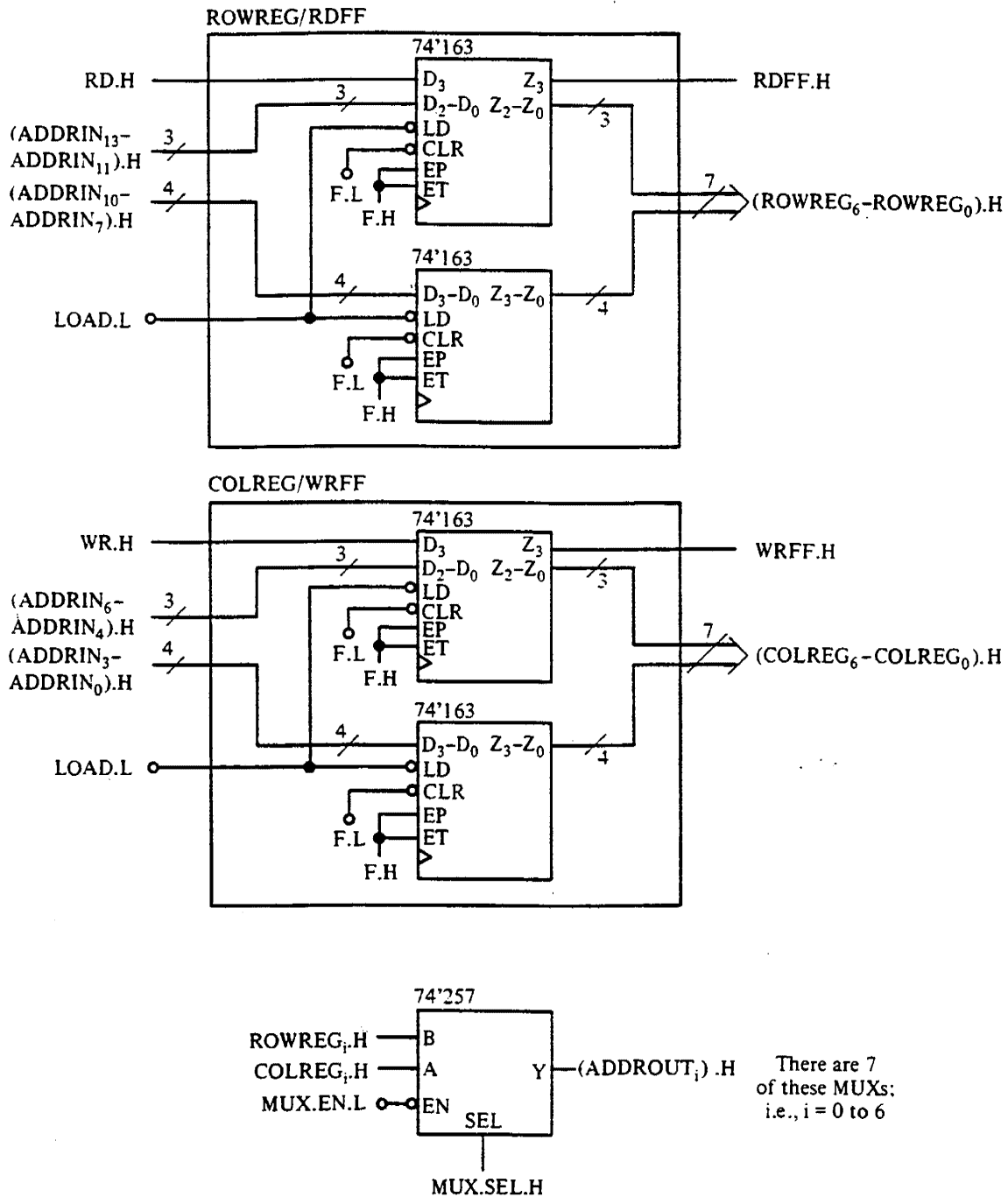
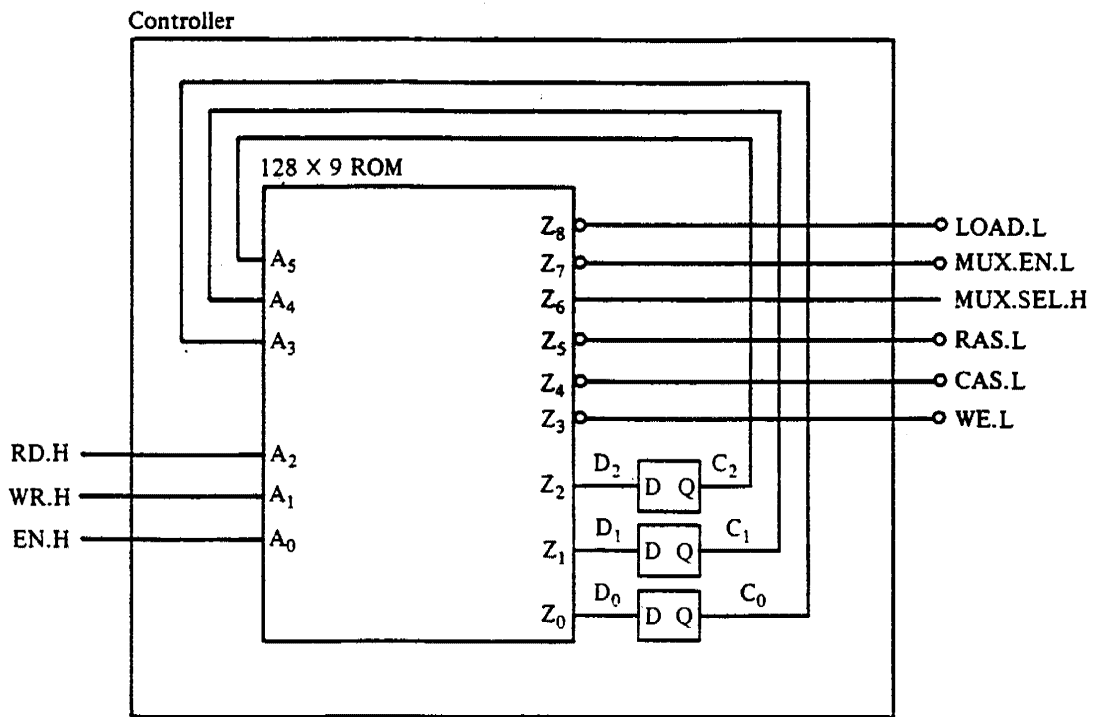


Figure 7.21 Realization of the circuit elements.

Fig. 5.10. This is, however, not necessary since in the 74'163 storage registers used for ROWREG and COLREG there are two unused cells that are perfectly suitable for realizing the RDFF and WRFF.

Any of the ASM realization methods presented in this chapter can be used to realize the ASM chart of Fig. 7.20. As an illustration, the ROM method will be used. The result is summarized in Fig. 7.22. The block diagram of the controller and the state assignments are shown in Fig. 7.22(a). The next-state and output table, derived from the ASM chart of Fig. 7.20, is shown in Fig. 7.22(b). Finally, the corresponding contents of the ROM are given in Fig. 7.22(c). Of course, the ROM addresses correspond to C₂, C₁, C₀, RD,



State assignments

	C ₂	C ₁	C ₀
WTLD	0	0	0
OROW	0	0	1
ORAS	0	1	0
OCOL	0	1	1
OCAS	1	0	0

(a) Block diagram and state assignments

Present-state code			Input values for the present state			Output values for the present state						Next state			D flip-flop input values		
C ₂	C ₁	C ₀	RD	WR	EN	LOAD	MUX. EN	MUX. SEL	RAS	CAS	WE	C ₂ ⁺	C ₁ ⁺	C ₀ ⁺	D ₂	D ₁	D ₀
0	0	0	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	X	X	1	1	0	0	0	0	0	0	0	1	0	0	1
0	0	1	X	X	X	0	1	1	0	0	0	0	1	0	0	1	0
0	1	0	X	X	X	0	1	1	1	0	0	0	1	1	0	1	1
0	1	1	X	X	X	0	1	0	1	0	0	1	0	0	1	0	0
1	0	0	0	0	X	0	1	0	1	1	0	0	0	0	0	0	0
1	0	0	0	1	X	0	1	0	1	1	1	0	0	0	0	0	0
1	0	0	1	X	X	0	1	0	1	1	0	0	0	0	0	0	0
1	0	1	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0
1	1	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0

Active low
Active high
Active low

Where 0 = false, 1 = true, and X = don't care

(b) Next-state and output table

Figure 7.22 Realization of the controller.

ROM address in decimal	ROM address in hexadecimal	Contents in binary (where 0 = low, 1 = high) $Z_8 \cdots Z_0$								
0	0 0	1	1	0	1	1	1	0	0	0
1	0 1	0	1	0	1	1	1	0	0	1
2	0 2	1	1	0	1	1	1	0	0	0
3	0 3	0	1	0	1	1	1	0	0	1
4	0 4	1	1	0	1	1	1	0	0	0
5	0 5	0	1	0	1	1	1	0	0	1
6	0 6	1	1	0	1	1	1	0	0	0
7	0 7	0	1	0	1	1	1	0	0	1
8-15	08-0F	1	0	1	1	1	1	0	1	0
16-23	10-17	1	0	1	0	1	1	0	1	1
24-31	18-1F	1	0	0	0	1	1	1	0	0
32-33	20-21	1	0	0	0	0	1	0	0	0
34-35	22-23	1	0	0	0	0	0	0	0	0
36-39	24-27	1	0	0	0	0	1	0	0	0
40-63	28-3F	1	1	0	1	1	1	0	0	0

(c) ROM contents

Figure 7.22 (cont.)

WR, EN and the ROM contents to LOAD, MUX.EN, MUX.SEL, RAS, CAS, WE, D_2 , D_1 , D_0 . The correspondences between the rows of the next-state and output table and the rows of the ROM table are apparent from an inspection. For example, the first two rows of the next-state and output table correspond in an alternate fashion to the ROM address rows 0 through 7. Row 3 of the next-state and output table corresponds to ROM address rows 8 through 15, and so on. Note that for the active-low outputs (LOAD, MUX.EN, RAS, CAS, and WE), the ROM content values are inverted. In other words, the value true is 0 (low) and the value false is 1 (high).

7.8.2 Modified Counter

For this example, we are to design the digital circuit shown in block diagram form in Fig. 7.23(a). Overall, the circuit is a modified 4-bit counter with outputs Z_3 - Z_0 . There is also a FLAG output. Under normal operation, the circuit counts with the frequency of the input clock signal, and produces a binary output Z_3 - Z_0 corresponding to . . . , 0, 1, 2, . . . , 14, 15, 0, 1, An input IN.BIT can, however, interrupt the normal counting sequence and cause the count to jump to a number corresponding to another input: D_3 - D_0 . Specifically, when IN.BIT, which is a serial string of 0s and 1s, contains the pattern 1011, the circuit does the following at the *next* active transition (leading edge in this case) of the clock signal:

1. Parallel loads the input D_3 - D_0 into the counter component of the circuit so that the subsequent count sequence begins from there
2. Sets the output signal FLAG to true for the next clock period, after which the signal returns to false

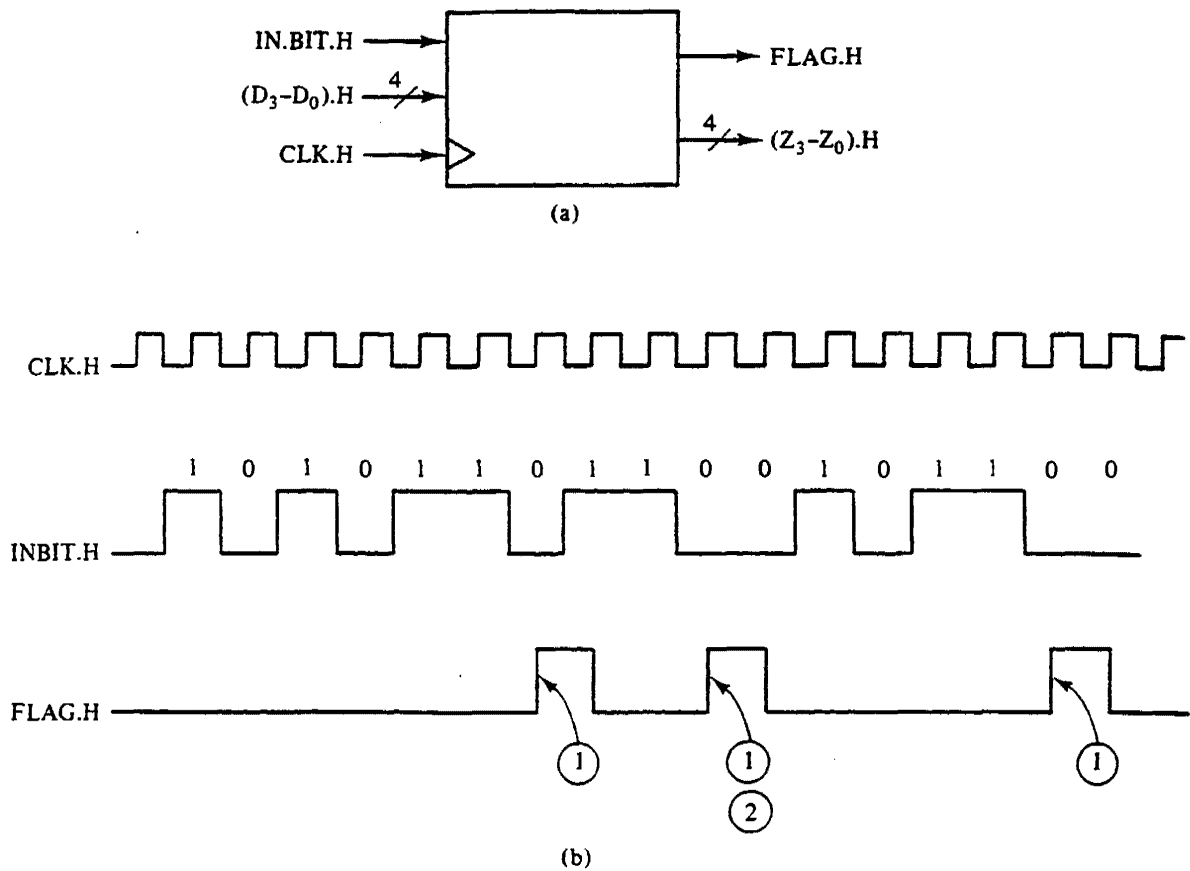
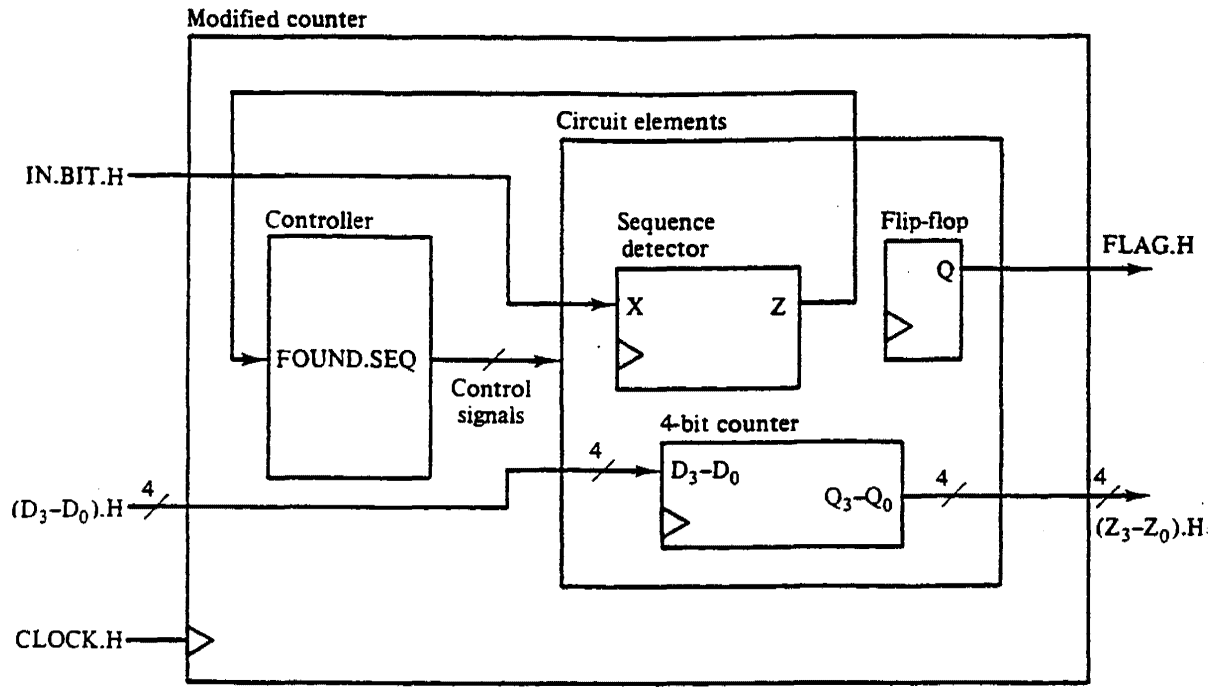


Figure 7.23 Block diagram and timing diagram specifications for the counter example.

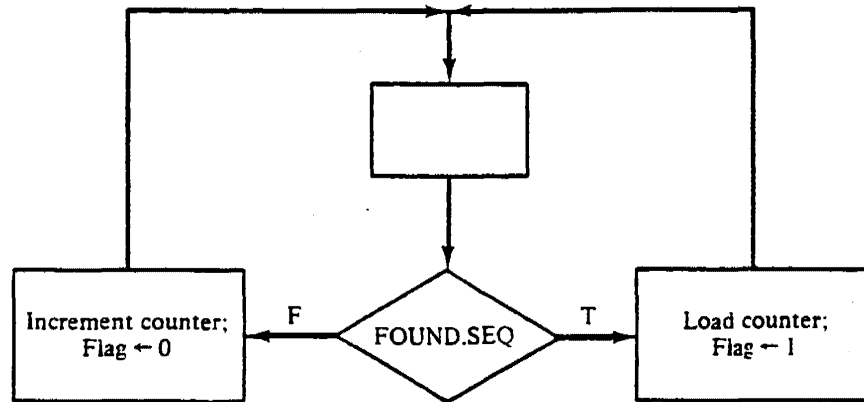
For a better understanding of the desired operation, consider Fig. 7.23(b), which is a timing diagram of a sample input string for IN.BIT and of the resulting FLAG output of the circuit. Note that at the times designated by ① a 1011 pattern is detected. Thus the circuit loads D_3-D_0 into the counter component, which means that $D_3-D_0 \rightarrow Z_3-Z_0$. Also, the circuit causes the output FLAG to go from false to true. Further note that at time ② this circuit action occurs even though there is no separate preceding 1011 IN.BIT sequence, which means that activating 1011 sequences can overlap.

With a little thought, we see that we need a 4-bit binary counter with a parallel load capability for producing the Z_3-Z_0 output and for loading the D_3-D_0 input. Also, we can use a flip-flop for generating the FLAG output. Finally, we need a sequence detector for examining the serial input IN.BIT and for generating a feedback signal FOUND.SEQ when it detects an input 1011 sequence. As is shown in Fig. 7.24(a), our identification of these three main components completes the selection of the circuit elements for the preliminary design of the circuit.

As is indicated by the flowchart of Fig. 7.24(b), the controller action is quite simple. Under normal operation, when the sequence detector has not detected the sequence 1011, the detector output is false (FOUND.SEQ = F). In this case the counter counts normally and the FLAG flip-flop has a false output. But when the sequence detector detects the sequence 1011, its output becomes true (FOUND.SEQ = T). Then, at the next active clock transition, the counter loads in D_3-D_0 . Also at this transition, the FLAG flip-flop is set to true.



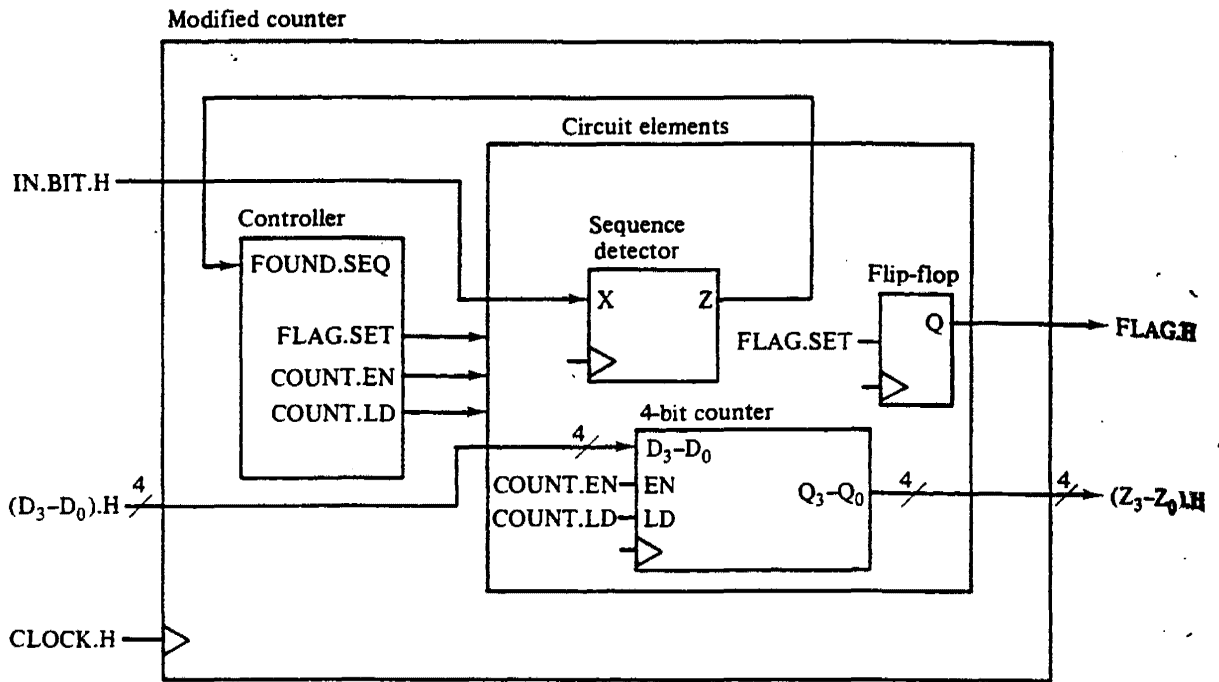
(a) Block diagram design



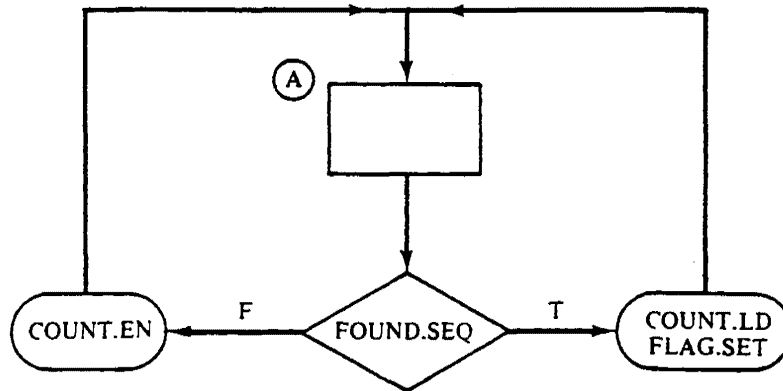
(b) Flowchart for the controller

Figure 7.24 Preliminary design for the modified counter.

In the refinement phase of the design we define the circuit elements in more detail. The result is shown in Fig. 7.25(a). As is typical, the counter has enable (EN) and load (LD) inputs, in addition to the clock input. For a normal count, the control inputs are $EN = \text{true}$ and $LD = \text{false}$. And for the loading of data from D_3-D_0 , these inputs are just the opposite: $EN = \text{false}$ and $LD = \text{true}$. Also, the FLAG flip-flop has a SET input in addition to the clock input. To set this flip-flop we make $SET = \text{true}$ before the next active clock transition. And to clear it we make $SET = \text{false}$ before the next active clock transition. For the sequence detector we will specify that it makes $FOUND.SEQ = \text{true}$ when it detects the sequence 1011. Note in Fig. 7.25(a) that we do not assign voltage



(a) Block diagram design



(b) ASM chart for the controller

Figure 7.25 Refined design for the modified counter.

representations for the signals FOUND.SEQ, FLAG.SET, COUNT.EN, and COUNT.LD because at this point in the design we do not know whether active-high or active-low signals are required.

Now having the refined circuit elements of Fig. 7.25(a), we can convert the preliminary controller flowchart of Fig. 7.24(b) into the ASM chart shown in Fig. 7.25(b). Note the use of conditional outputs. In this case the conditional outputs are not used simply for performance (as in the case of the dynamic RAM controller example), but also are necessary to achieve the required timing. The reader is urged to verify the timing of this ASM chart by comparing it with the timing diagram shown in Fig. 7.23(b).

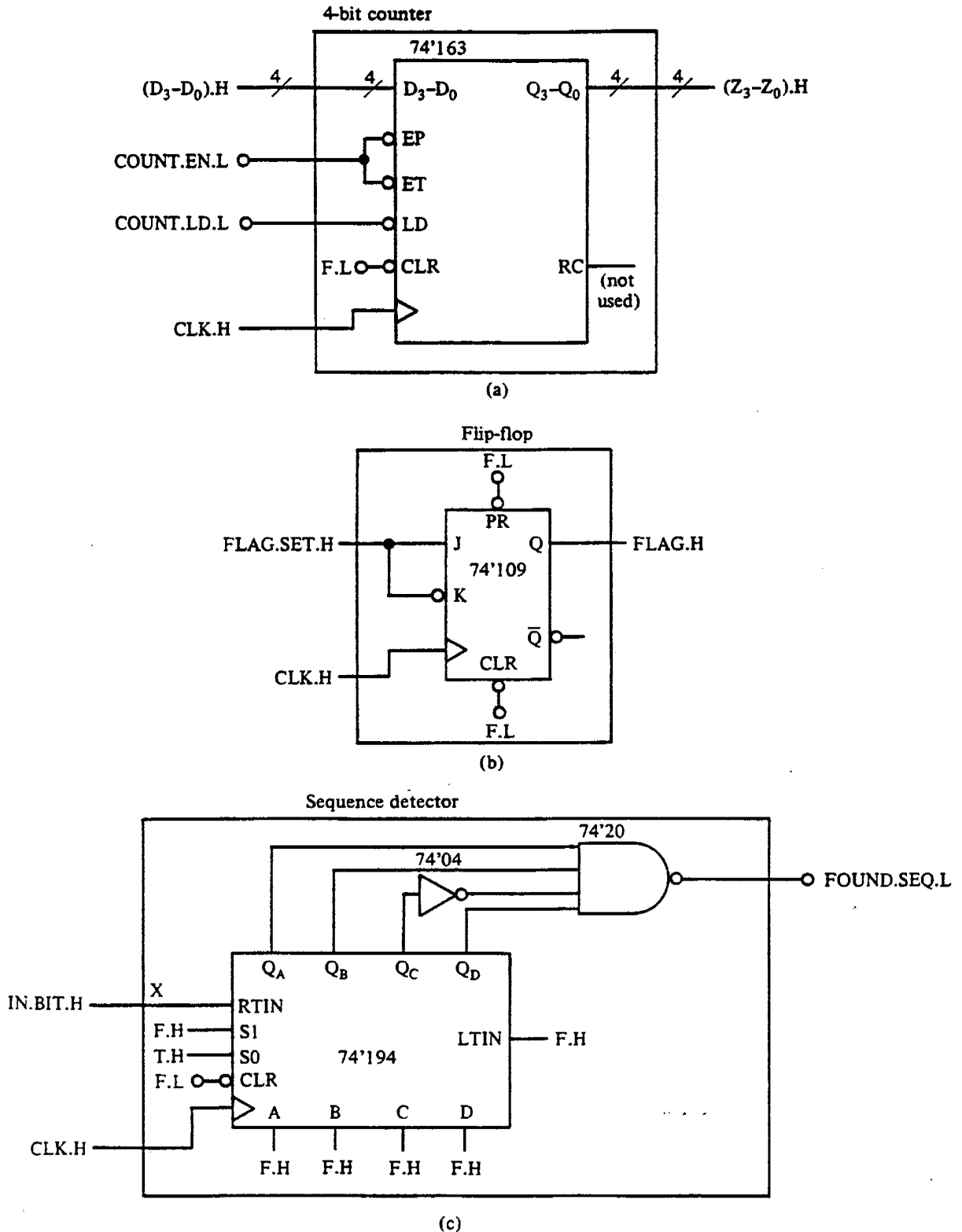


Figure 7.26 Circuit components for the modified counter.

The final phase of the design process is the realization phase. The realization of the circuit elements is summarized in Fig. 7.26. From a TTL data book we can find that the 74'161 and the 74'163 are both 4-bit counters with parallel load capability. They differ only in that the 74'161 has an asynchronous clear and the 74'163 a synchronous

clear. Since we do not need the clear function for this design, either counter is satisfactory. We will arbitrarily select the 74'163, which requires the signals shown in Fig. 7.26(a). Note that once the actual integrated circuit is selected, the voltage representations of the signals and terminals are fixed. For the 74'163, for example, both COUNT.EN and COUNT.LD are active-low. As is shown in Fig. 7.26(b), the 74'109 is suitable for the flip-flop circuit element.

The only remaining element of our circuit is the sequence detector. Unfortunately, there is no 74'XX sequence detector in any TTL data book. But we can realize the sequence detector circuit element in a rather obvious manner by using a 74'194 shift register, as shown in Fig. 7.26(c). We should have had this fact in mind when we decided to use a sequence detector circuit element in our earlier design steps, for there is no sense in designing with circuit elements that we do not have or cannot realize with the available integrated circuits.

From the techniques presented in this chapter, the realization of the controller should be straightforward. In fact, this particular realization is quite simple since the ASM chart for this controller [Fig. 7.25(b)] has only one state—state A. For this special case, no state flip-flop is required and the ASM outputs are dependent only on the input FOUND.SEQ, as follows:

$$\begin{aligned} \text{COUNT.EN} &= \overline{\text{FOUND.SEQ}} \\ \text{COUNT.LD} &= \text{FOUND.SEQ} \\ \text{FLAG.SET} &= \text{FOUND.SEQ} \end{aligned}$$

The resultant realization of the one-state controller is shown in Fig. 7.27.

7.8.3 Alternative Design for the Modified Counter

For the modified counter specification of Fig. 7.23 suppose that we did not happen to think of using a shift register to realize the sequence detector. In fact, suppose that our train of thought went in another direction, and we decided to use as our major circuit elements only a 4-bit counter and a flip-flop, as shown in Fig. 7.28.

For the modified counter circuit of Fig. 7.28 to operate as specified in Fig. 7.23, the controller must provide control signals FLAG.SET, COUNT.EN, and COUNT.LD

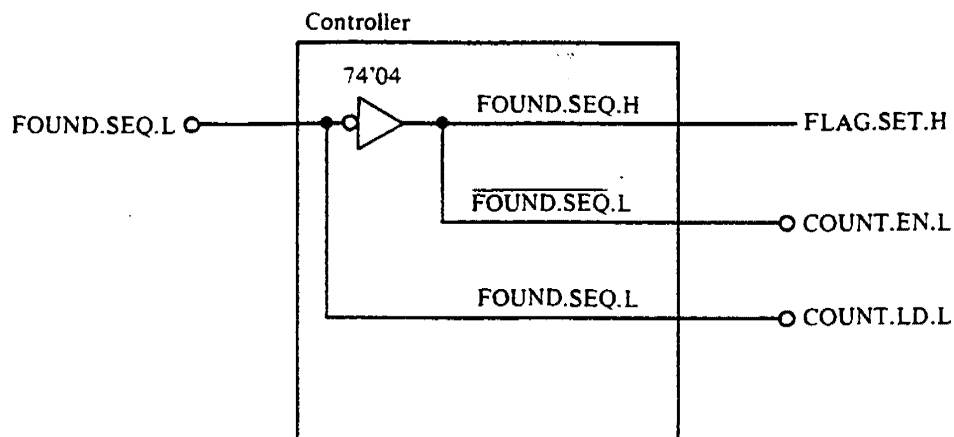


Figure 7.27 Controller for the modified counter.

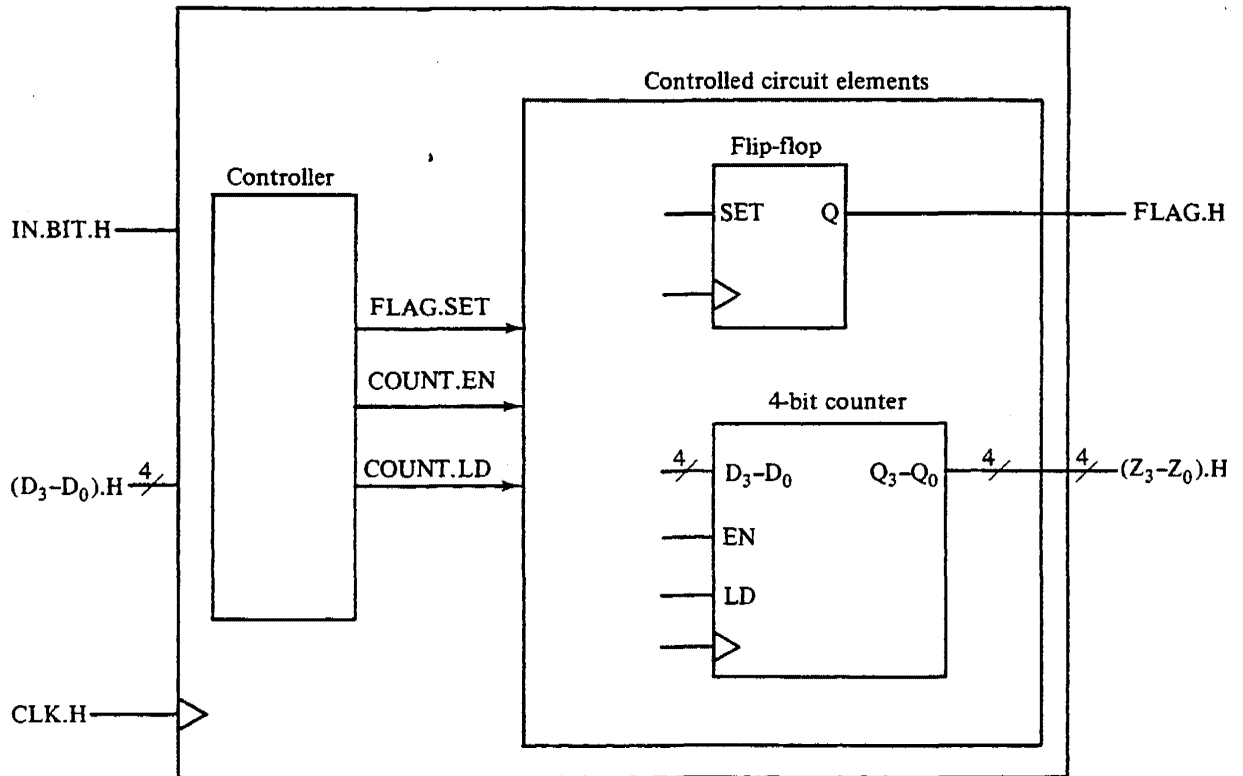


Figure 7.28 Preliminary design for the alternative modified counter.

that are correct for each moment in time. In other words, for every moment in time, the functions of the controller are as follows:

1. To determine the current situation of the input sequence, based on the present value of IN.BIT along with a memory of the past values of IN.BIT leading up to the present. (Note that we no longer can use the signal FOUND.SEQ since we no longer are using the sequence detector circuit element.)
2. For that situation, provide the appropriate input control signals COUNT.EN and COUNT.LD for the counter and FLAG.SET for the flip-flop.

Obviously, the controller for this circuit will be more complex than that of the original design. First, we need to determine all the *distinct* situations for the input sequence. Fortunately, there are only a finite number of them:

(S0) No part of the valid sequence has been detected. Example of such a sequence:

... XXXXX00 (where X is either a 0 or a 1)
 ↑
 last value of IN.BIT received

(S1) A sequence of 1 has already been detected. This implies that this could be the beginning of a valid sequence. Example:

... XXXABC1 in which $ABC \neq 101$ and $BC \neq 10$
 ↑

(S2) A sequence of 10 has already been detected. This implies that a valid sequence is possible within two more clock cycles. Example:

... XXXXX10
 ↑

(S3) A sequence of 101 has already been detected. This implies that if the current value of IN.BIT is 1, then we will have found the valid sequence. Example:

... XXX101
 ↑

(S4) The sequence of 1011 has already been detected.

These are the only distinct situations, or *states*, for this problem environment. No other situation is relevant to the design of the controller.

With the states of the controller identified, the controller design becomes less formidable. The steps for developing the corresponding ASM chart are shown in Fig. 7.29. In Fig. 7.29(a) the states for the ASM chart are identified and a state box is used to represent each of the states. Then as shown in Fig. 7.29(b), the state transition for each state is determined and specified. For example, state S2 represents the situation that a sequence of 10 has already been detected. If, while the circuit is in state S2, the current input IN.BIT is a 1, then the resulting partial sequence becomes 101, which is represented by state S3. Consequently, the state transition from state S2 is to state S3. If, however, when the circuit is in state S2, the current input IN.BIT is a 0, then the resulting sequence becomes 100, which is no longer a part of the valid sequence 1011. Consequently, the state transition is to state S0, the reset state, where no part of the valid sequence has been detected. The state transitions for each of the states can be determined in this manner.

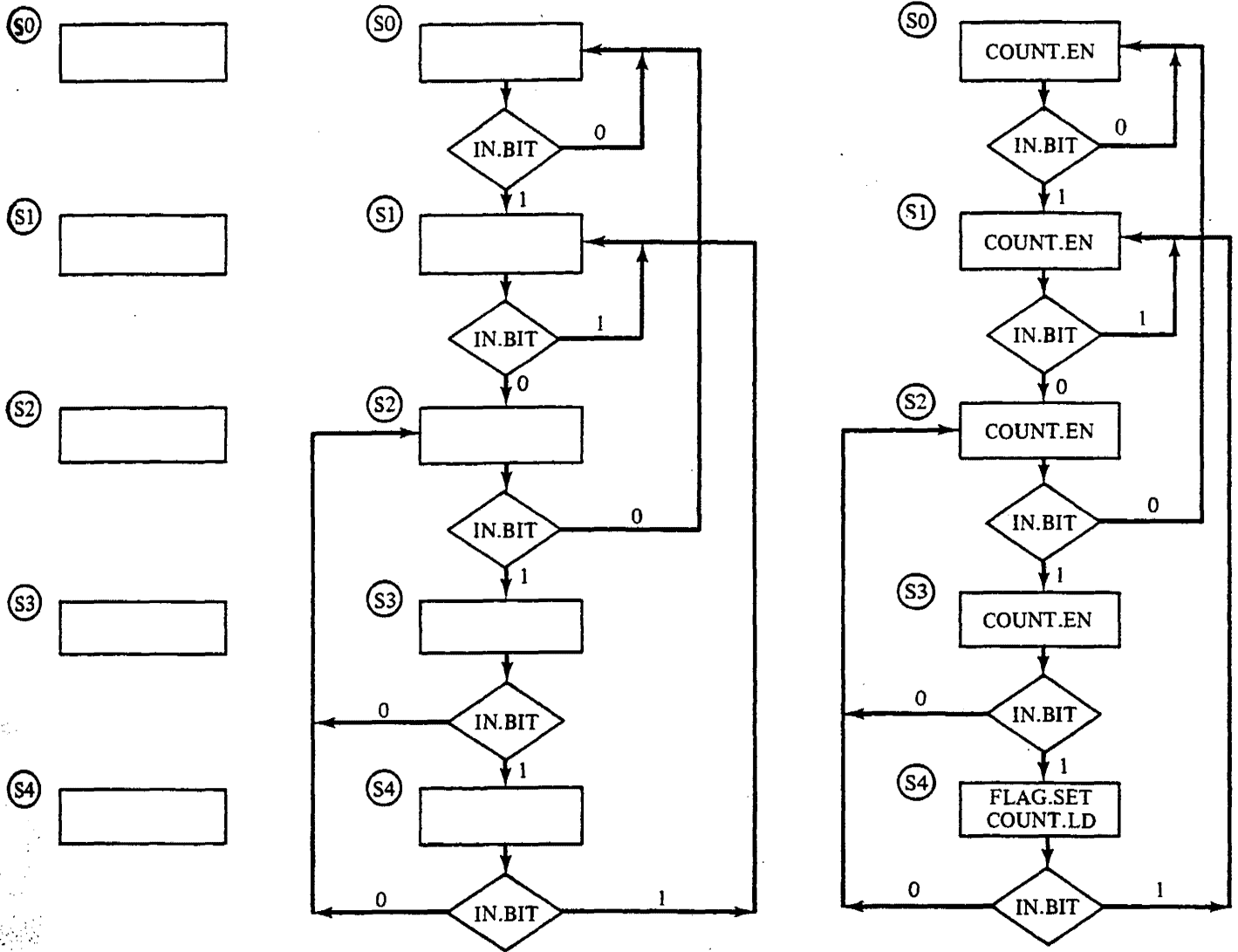
Next, the control outputs for each state of the ASM chart have to be determined and specified, which is quite straightforward for this controller. The result is shown in Fig. 7.29(c). In state S4, where the valid sequence has been detected, the counter needs to be loaded (COUNT.LD) and the flip-flop needs to be set (FLAG.SET) at the next active clock transition. For the remainder of the states, where the valid sequence has not been detected, all that is needed is to increment the counter (COUNT.EN).

Finally, the ASM chart is reviewed to determine whether the timing is correct and whether it can be optimized. This is normally accomplished through the use of conditional outputs. In this case, however, the ASM chart shown in Fig. 7.29(c) is the final one.

The final phase of the design process is the realization phase. The realization of the circuit elements of this circuit (counter and flip-flop) is the same as that shown in Figs. 7.26(a) and (b). The realization of the ASM chart has already been accomplished in Sec. 7.6 and Fig. 7.11.

7.8.4 Hardware Multiplier

In this example we are to design a hardware multiplier circuit that can multiply two 4-bit unsigned binary numbers to produce an 8-bit product. The block diagram is shown in Fig. 7.30. The inputs to this circuit are a 4-bit multiplicand



(a) Identification of states

(b) Specification of state transitions

(c) Specification of control outputs

Figure 7.29 Development of the ASM chart for the controller.

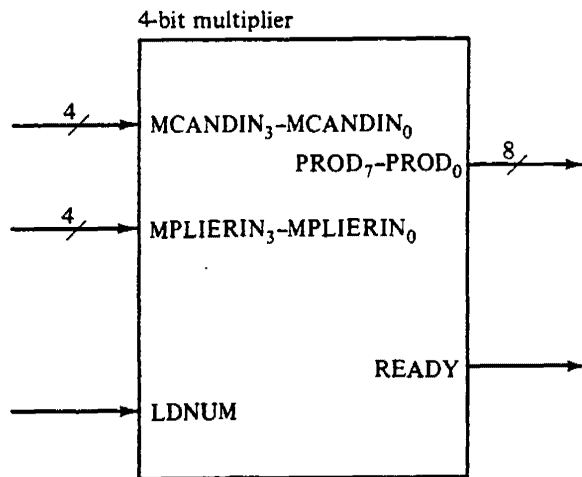


Figure 7.30 Block diagram of the multiplier.

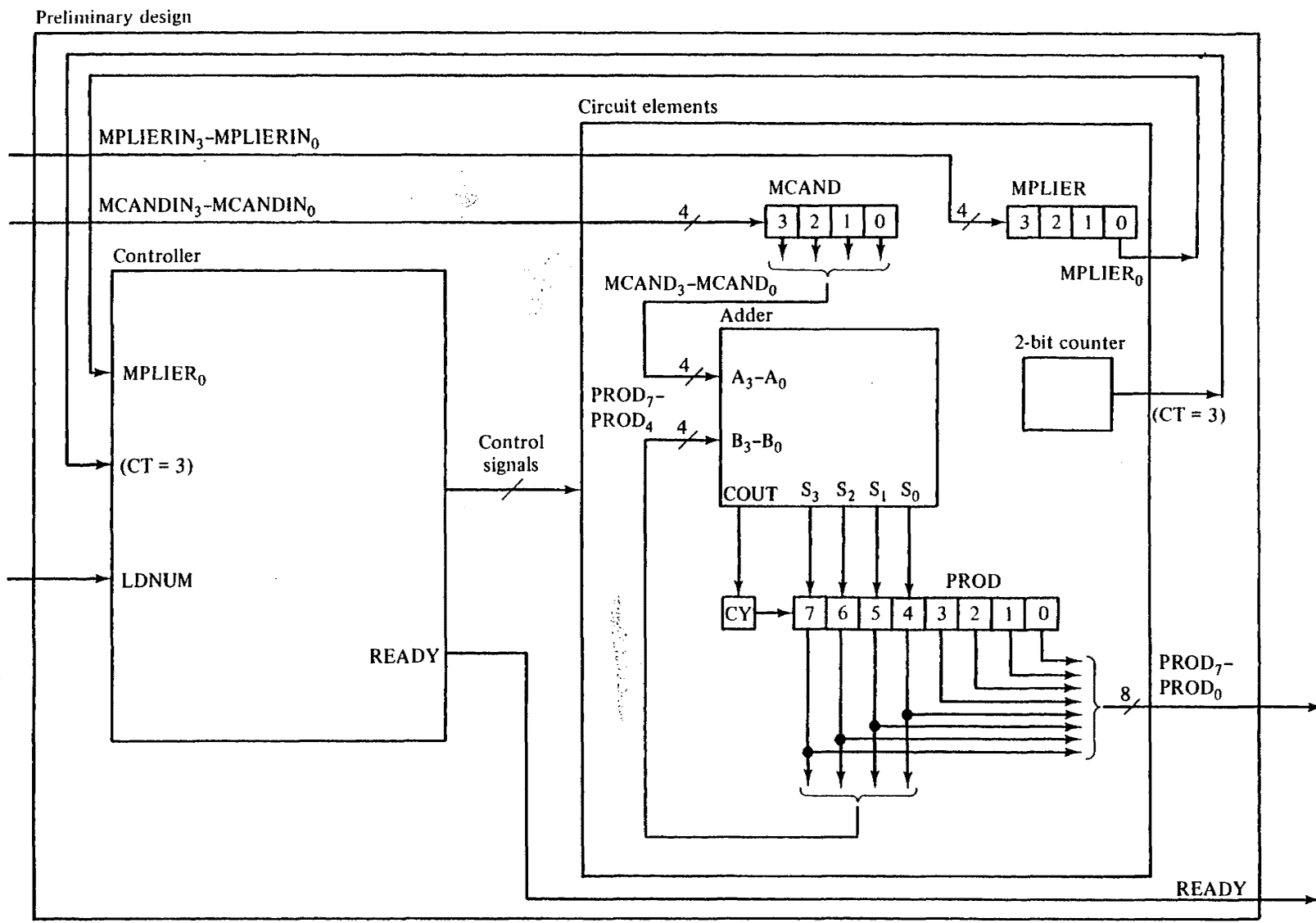
1 0 1 0	(Multiplicand)
X 1 0 1 1	(Multiplier)
0 0 0 0	[1 0 1 0] Multiplier(0) = 1: multiplicand is shifted 0 times and added to the partial product
0 0 0	[1 0 1 0] 0 Multiplier(1) = 1: multiplicand is shifted 1 time and added to the partial product
0 0	[0 0 0 0] 0 0 Multiplier(2) = 0: 0 is added to the partial product
0	[1 0 1 0] 0 0 0 Multiplier(3) = 1: multiplicand is shifted 3 times and added to the partial product
0 1 1 0 1 1 1 0	(Product)

Figure 7.31 Illustration of hand multiplication of unsigned binary numbers.

(MCANDIN₃–MCANDIN₀), a 4-bit multiplier (MPLIERIN₃–MPLIERIN₀), and a control input (LDNUM). The outputs from this circuit are an 8-bit product (PROD₇–PROD₀) and a status output (READY). It is further specified that when the circuit is ready for a multiplication operation, the READY output is true. Then, when the input LDNUM becomes true, the two 4-bit values at MCANDIN and MPLIERIN are loaded into the multiplier circuit and the multiplication process begins. During the multiplication process the READY output is false, and subsequent inputs at MCANDIN and MPLIERIN are ignored. When the multiplication process is completed, the READY output becomes true to indicate that the 8-bit product is available at the PROD outputs and also that the circuit is ready for another multiplication operation.

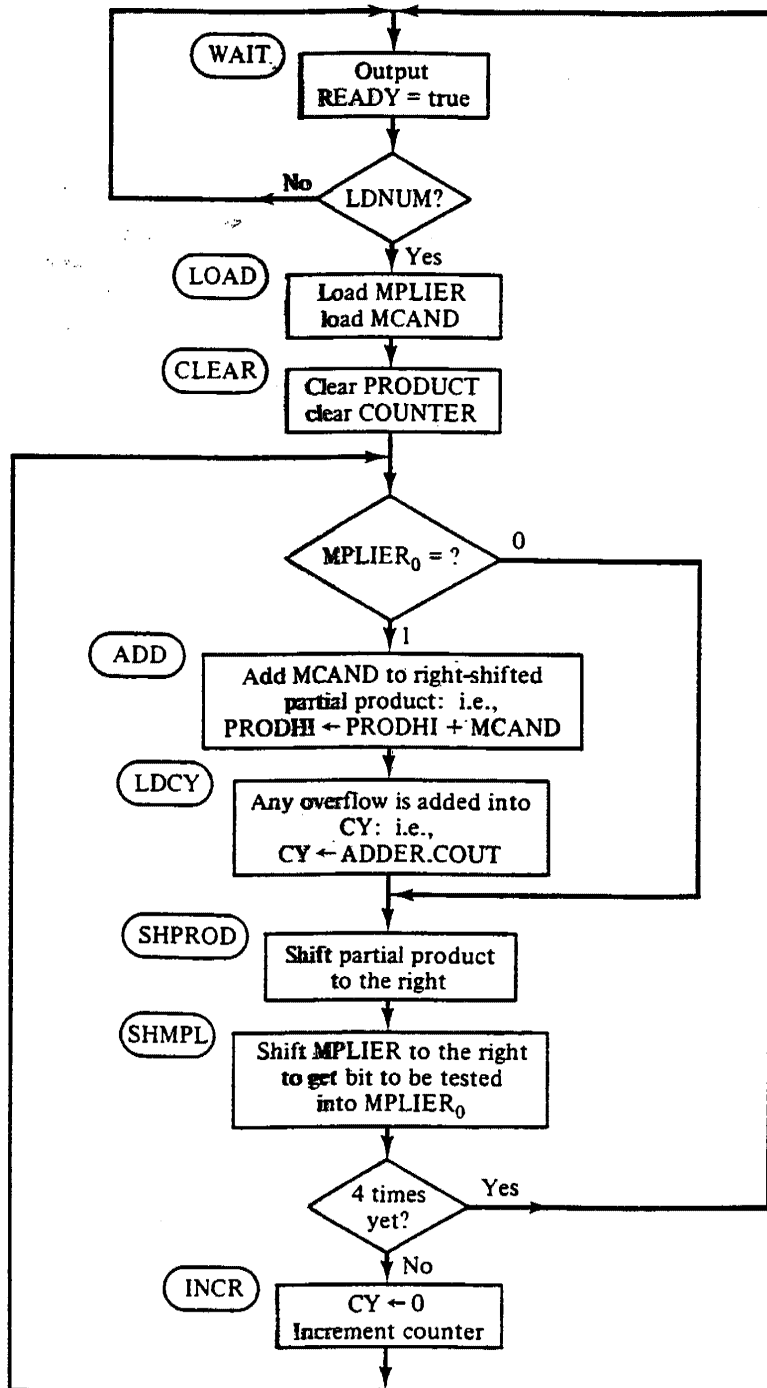
Recall that in the hand multiplication of two unsigned binary numbers, such as that illustrated in Fig. 7.31, each bit of the multiplier is examined. If the current multiplier bit is 1, then the shifted multiplicand is added to the partial product. The purpose of the left-shifting of the multiplicand is to account for the weight of the multiplier bit. The algorithm used for the multiplier circuit of Fig. 7.30 will follow this basic multiplication algorithm.

A preliminary design of the multiplier circuit is shown in the two parts of Fig. 7.32. Specifically, the circuit elements are shown in Fig. 7.32(a), and the algorithm for the control of the multiplication is given in the form of a flowchart in Fig. 7.32(b). As can be seen from the flowchart, with references to the circuit diagram, in the WAIT state the circuit outputs READY = true to indicate that the circuit is ready for another multiplication operation. And, the circuit remains in the WAIT state as long as the LDNUM input is false. But when LDNUM becomes true, the values at the inputs MCANDIN and MPLIERIN are loaded into two 4-bit registers MCAND and MPLIER, respectively. Also, the circuit is initialized to begin the multiplication by clearing the 8-bit PROD register and a 2-bit counter. This PROD register holds the partial product during the multiplication and then the final product at the end. For reasons that will be seen, it is divided into two parts: PRODHI (PROD₇–PROD₄) and PRODLO (PROD₃–PROD₀). The 2-bit counter is used to ensure in each multiplication that the shift-add process is performed for exactly four times (COUNT = 00, 01, 10, 11). The algorithm for the multiplication process is based on the basic algorithm for hand multiplication as illustrated in the example shown in Fig. 7.31. However, the left-shifting of the multiplicand of the hand multiplication is replaced by the functionally equivalent right-shifting of the partial product. Then, only a 4-bit adder is required instead of an 8-bit adder. The reader is urged to work through a simple multiplication example based on the algorithm in Fig. 7.32(b) to verify the correctness of this algorithm.



(a) Block diagrams of the circuit elements and the controller

Figure 7.32 Preliminary design of the multiplier circuit.



(b) Flowchart for the controller

Figure 7.32 (cont.)

The next phase of the design process is the refinement phase. After some thought and several iterations on the design, we can specify in more detail the functions that are required for each circuit element. The result is shown in Fig. 7.33. MCAND is a 4-bit storage register with a synchronous load input. PRODHI, PRODLO, and MPLIER all are shift-right registers, with the incoming bit value applied at the serial input (SI). Furthermore, each of these shift registers has a synchronous load and/or clear input, as dictated by the control algorithm. ADDER is a standard 4-bit adder. COUNT is a standard

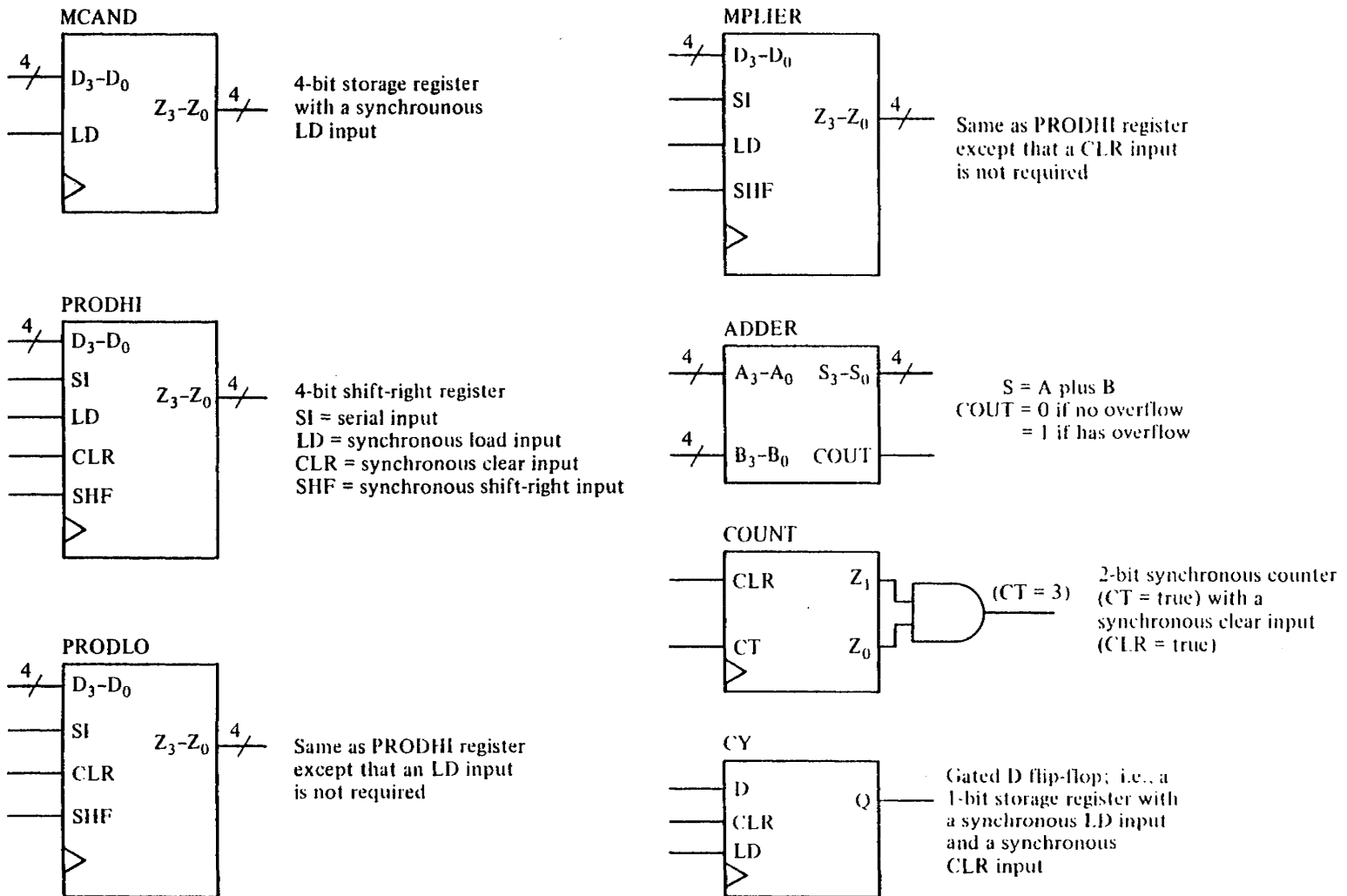
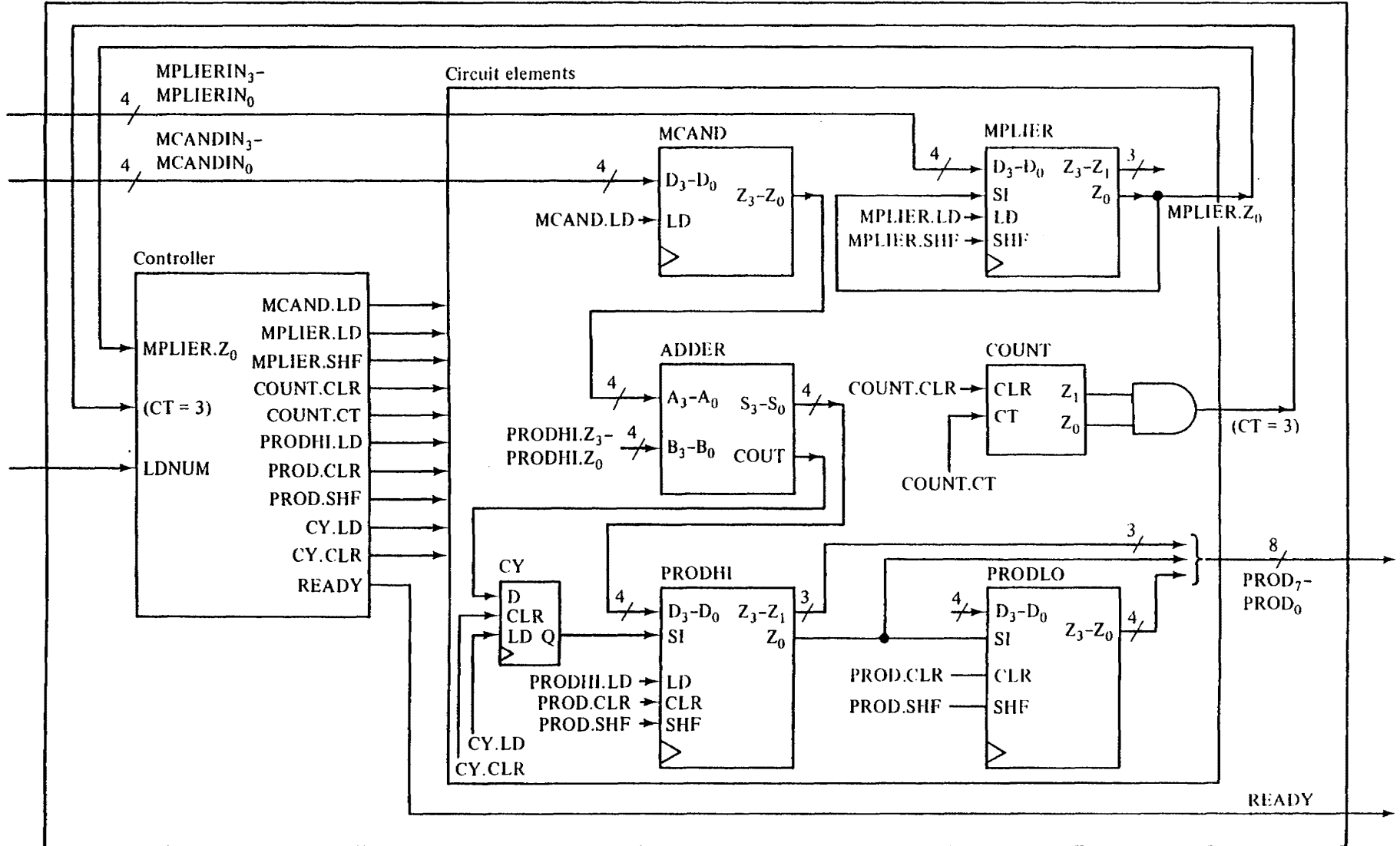


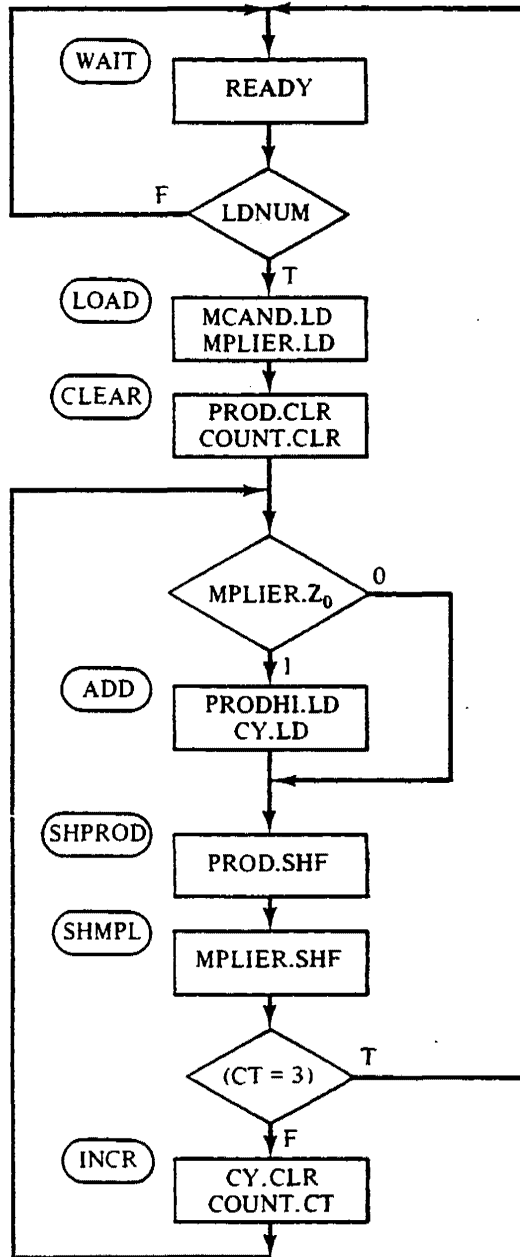
Figure 7.33 Detailed specifications for the circuit elements.

Refined design



(a) Block diagram of the circuit elements and the controller

Figure 7.34 Refined design of the multiplier circuit.



(b) ASM chart for the controller

Figure 7.34 (cont.)

2-bit binary counter plus an AND gate for generating the $(CT = 3)$ signal. Finally, CY is a D flip-flop with synchronous load and clear inputs.

Using this detailed specification for the circuit elements, we can obtain the refined design of Fig. 7.34. As shown in Fig. 7.34(a), the control signals outputted from the controller are now defined. They correspond to the control inputs of the circuit elements. Based on the refined set of circuit elements shown in Fig. 7.34(a), the flowchart of Fig. 7.32(b) can be converted to the ASM chart shown in Fig. 7.34(b). Again, unlike the flowchart, the ASM chart precisely specifies the timing. Since the control outputs are now defined, their logic values can be specified for each state to accomplish the required functions for that state. While paying particular attention to the timing of the circuit, the

reader is urged to work through a simple multiplication example based on the ASM chart of Fig. 7.34(b) to verify the correctness of this chart.

The ASM chart in Fig. 7.34(b) can be refined to the ASM chart in Fig. 7.35. We observe from Figs. 7.34(a) and (b) that the loading of MCAND and MPLIER and the clearing of PROD and COUNT can be performed together in one state. Consequently, the two states LOAD and CLEAR in Fig. 7.34(b) can be combined into one state. Furthermore, this state can be eliminated by using conditional outputs without introducing any timing problems, as is shown in Fig. 7.35. The result is an increase in the performance of the multiplier circuit by two clock cycles. Similarly, states SHPROD and SHMPL can be combined into one state, SHIFT. And states ADD and INCR can be eliminated through the use of conditional outputs. Then, however, a new state TSBIT has to be added. (Why?) In all, a net gain of four clock cycles in performance is achieved through the refinement of the original ASM chart of Fig. 7.34(b).

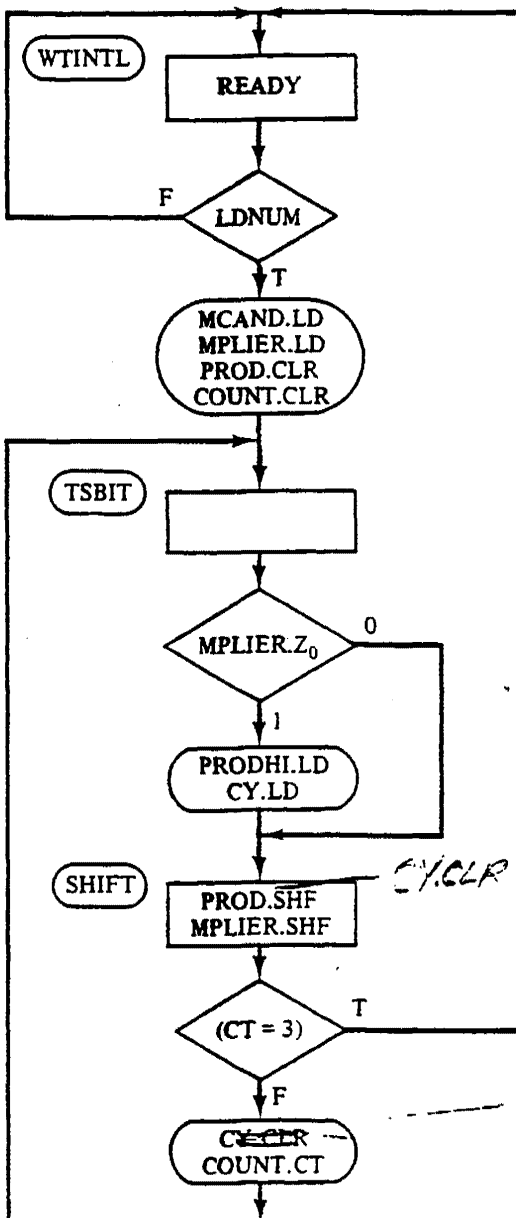
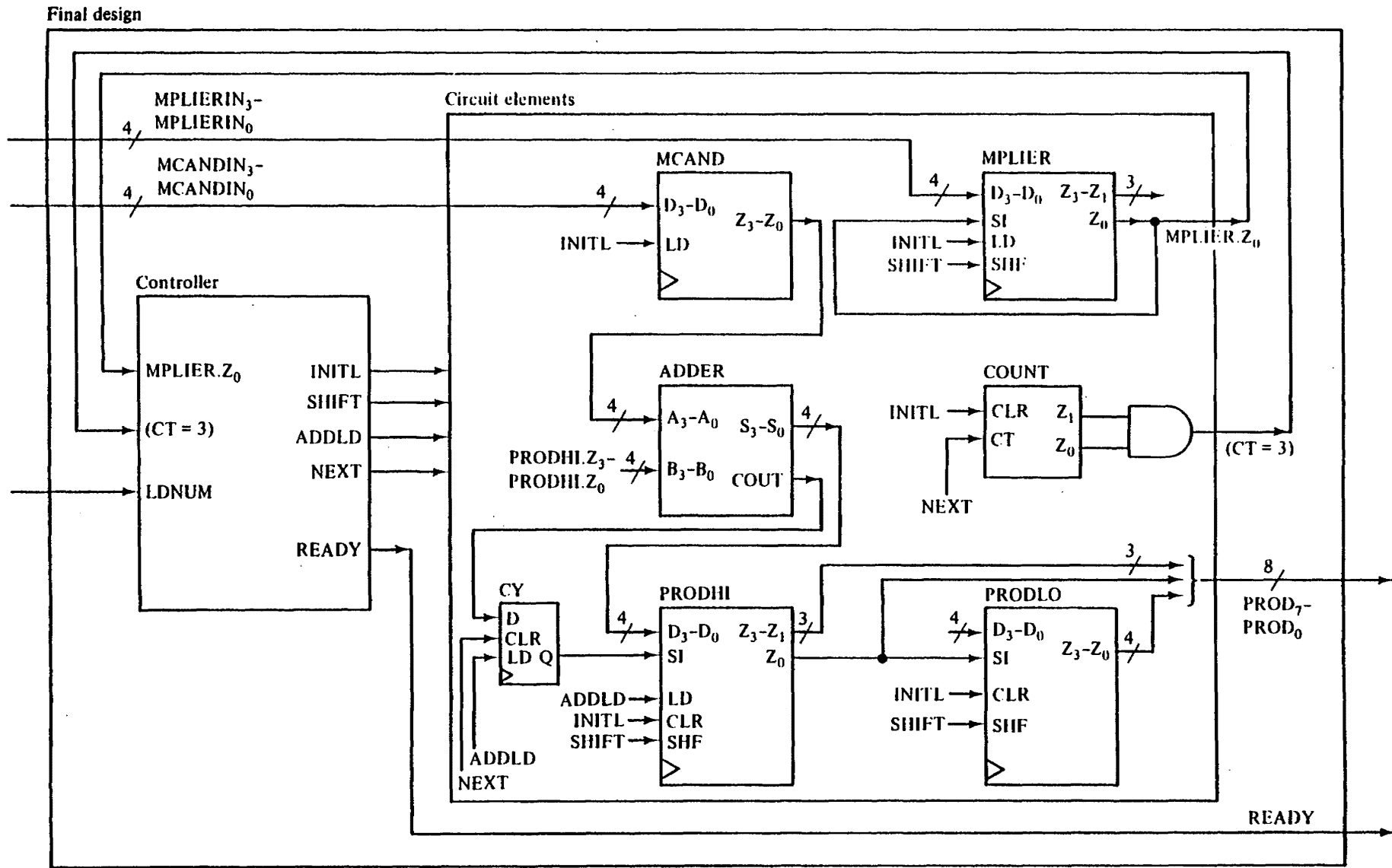
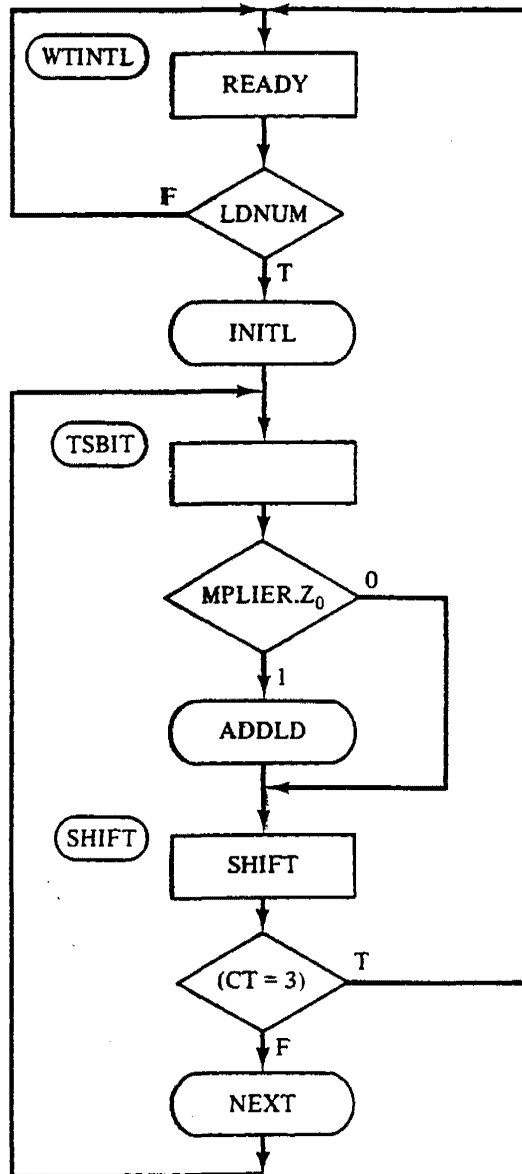


Figure 7.35 Refinement of the ASM chart for the controller.



(a) Circuit elements and the controller

Figure 7.36 Final design for the multiplier circuit.



(b) ASM chart for the controller

Figure 7.36 (cont.)

Some final refinements of the multiplier circuit can be made. We observe that the four initialization signals (MCAND.LD, MPLIER.LD, PROD.CLR, and COUNT.CLR) are applied in the same state and under the same condition. They can, therefore, be replaced by a single initialization signal, INITL. Similarly, the two signals PRODHI.LD and CY.LD can be replaced by a single load signal, ADDLD. Also, the two signals PROD.SHF and MPLIER.SHF can be replaced by a single signal, SHIFT. Finally, the two signals CY.CLR and COUNT.CT can be replaced by NEXT. These refinements in the design are shown in Fig. 7.36.

The final phase of the design process is the realization phase. For this circuit, we need to use commercially available ICs to realize the circuit elements in Fig. 7.36(a) and the controller represented by the ASM chart in Fig. 7.36(b). The realization of the circuit elements is summarized in Fig. 7.37. As shown, MCAND is realized by using a 74'163 configured as a storage register ($EP = ET = \text{false}$). PRODHI is realized by using a

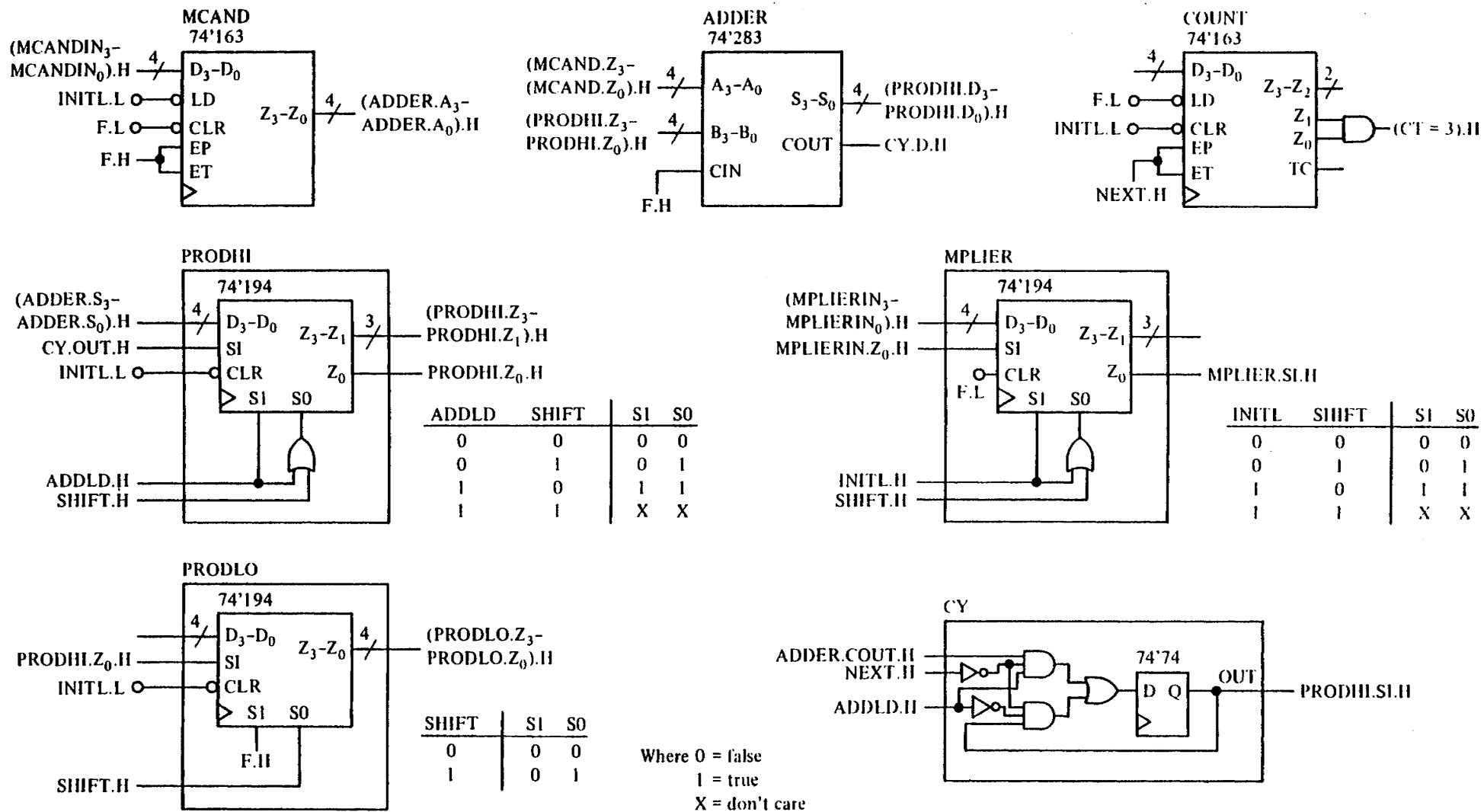
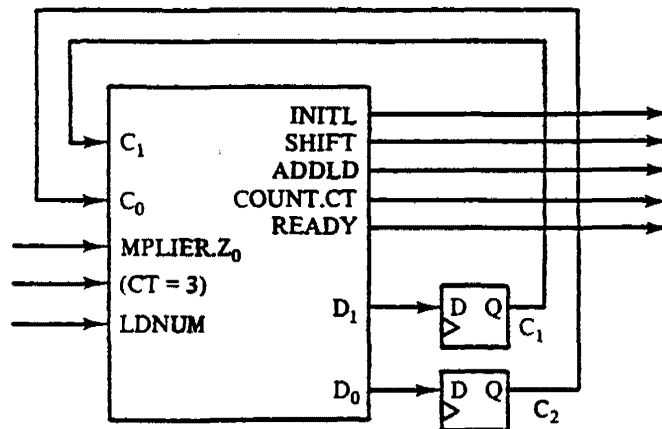


Figure 7.37 Realization of the circuit elements.

74'194 shift register with some external gating (an OR gate) to convert the ADDLD and SHIFT input into S1 and S0 inputs as is required for the 74'194. Similarly, PRODLO and MPLIER are realized with 74'194 shift registers. The 4-bit ADDER is realized directly with a 74'283. COUNT is realized with a 74'163 with some additional gating (an AND gate) to generate the (CT = 3) signal. Finally, CY is realized as a gated D flip-flop with a synchronous clear input. The design of CY is similar to that of the gated D flip-flop described in Sec. 5.3.4 (Fig. 5.10).

The realization of the controller is straightforward with the techniques presented in this chapter. The block diagram of it and the resulting next-state and output table are shown in Fig. 7.38.

In summary, the digital design process requires a combination of creativity, experience, and understanding of the general design principles. Our purpose in this chapter



State assignments:

	C_1	C_0
WTINIT	0	0
TSBIT	0	1
SHIFT	1	0

(a) Block diagram and state assignment

present state		inputs			outputs					next state		D flip-flop inputs	
C_1	C_0	$MPLIER.Z_0$	(CT = 3)	$LDNUM$	INITL	SHIFT	ADDLD	COUNT.CT	READY	C_1^+	C_0^+	D_1	D_0
0	0	X	X	0	0	0	0	0	1	0	0	0	0
0	0	X	X	1	1	0	0	0	1	0	1	0	1
0	1	0	X	X	0	0	0	0	0	1	0	1	0
0	1	1	X	X	0	0	1	0	0	1	0	1	0
1	0	X	0	X	0	1	0	1	0	0	1	0	1
1	0	X	1	X	0	1	0	0	0	0	0	0	0
1	1	X	X	X	0	0	0	0	0	0	0	0	0

Where 0 = false
 1 = true
 X = don't care

(b) Next-state and output table

Figure 7.38 Outline of the realization of the controller.

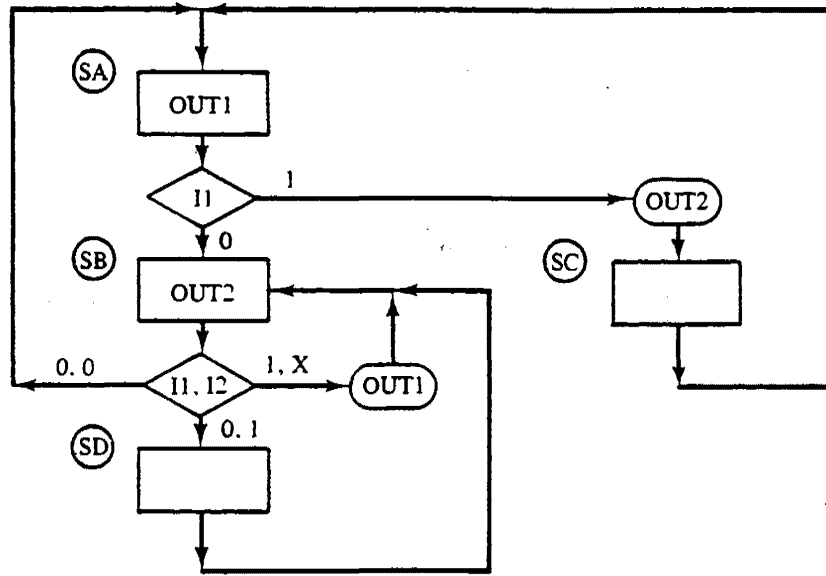
has been to present and illustrate the general principles of digital design and to consider various tools that are useful in the design process. These design principles and tools form a solid foundation on which to build experience and to exercise creativity.

SUPPLEMENTARY READING (see Bibliography)

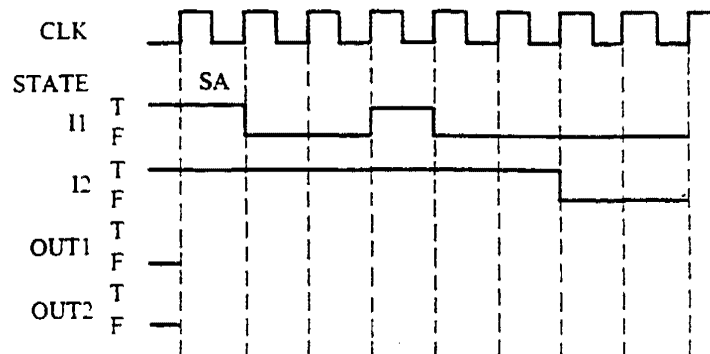
[Clare 73], [Fletcher 80], [Kline 83], [Mano 79], [Mano 84], [Peatman 80], [Prosser 87], [Wiatrowski 80]

PROBLEMS

- 7.1. Explain the general model for a digital circuit design shown in Fig. 7.1.
- 7.2. The digital design process can be divided into three major phases: the preliminary design phase, the refinement phase, and the realization phase. At the conclusion of each phase, what are the expected end products in terms of the controller and the controlled circuit elements?
- 7.3. Given the ASM chart of Fig. 7.39(a), complete the corresponding timing diagram of Fig. 7.39(b).



(a)



(b)

Figure 7.39 ASM chart and timing diagram for Problem 7.3.

7.4. Given the ASM chart and block diagram of Fig. 7.40(a), complete the corresponding timing diagram of Fig. 7.40(b).

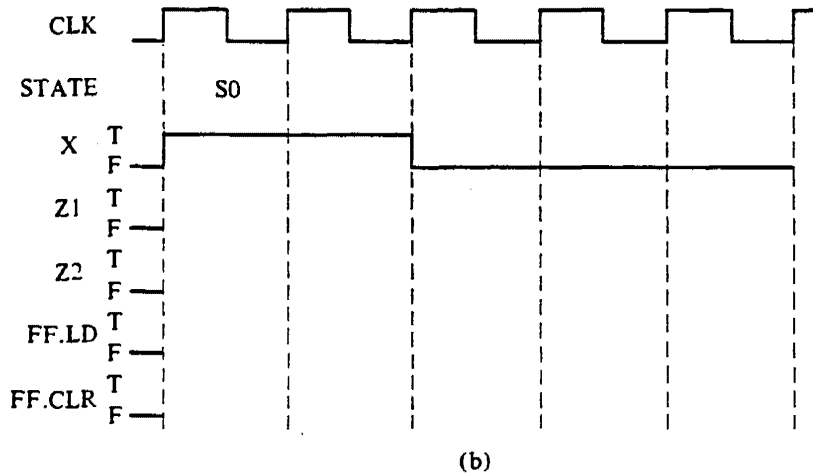
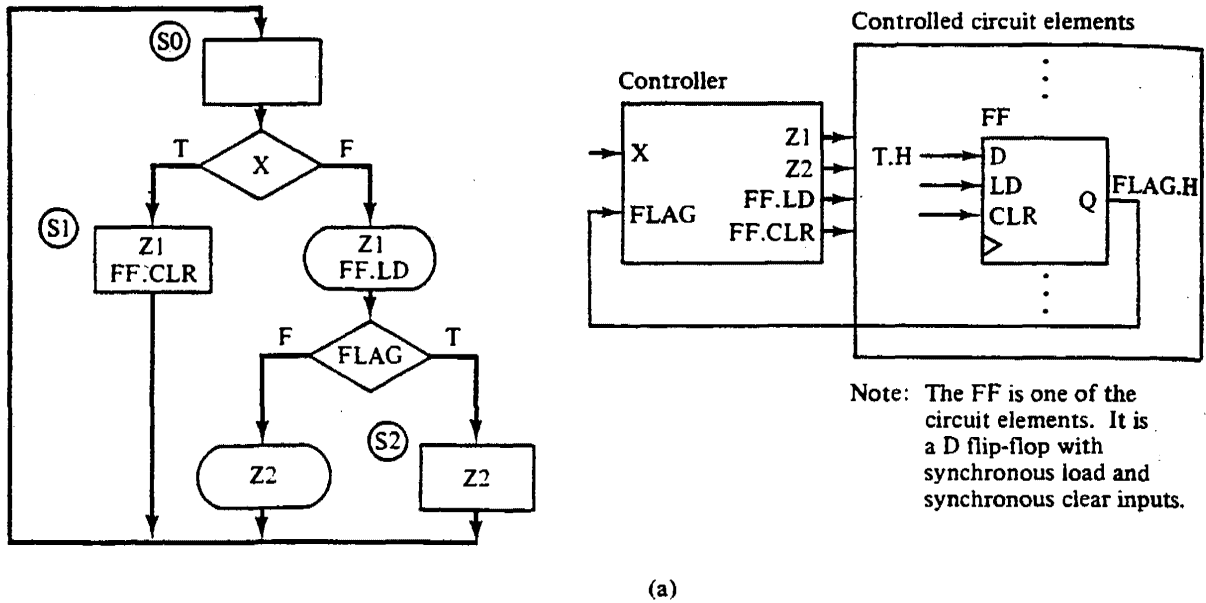


Figure 7.40 ASM chart, block diagram, and timing diagram for Problem 7.4.

7.5. Given the timing diagram of Fig. 7.41, reconstruct the ASM chart that corresponds to it.

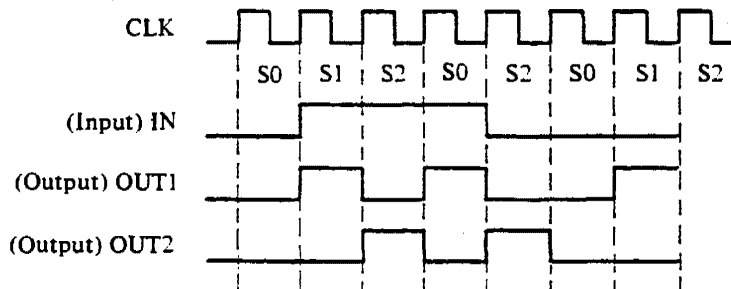


Figure 7.41 Timing diagram for Problem 7.5.

7.6. Given the timing diagram of Fig. 7.42, reconstruct the ASM chart that corresponds to it.

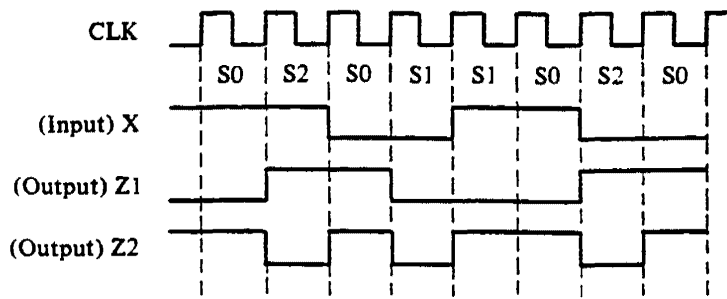


Figure 7.42 Timing diagram for Problem 7.6.

- 7.7. For the ASM chart specified in Fig. 7.40(a),
- Make a block diagram design (similar to the one shown in Fig. 7.3) of the corresponding state generator, specifying the inputs, outputs, and the state flip-flops (D type).
 - Construct the next-state table for the state generator, given the following state code assignment:

	C1	C0
S0	1	0
S1	0	0
S2	0	1

- Realize the controller by the traditional method. Use D flip-flops, and draw a detailed circuit diagram of the controller.

- 7.8. For the ASM chart of Fig. 7.43,

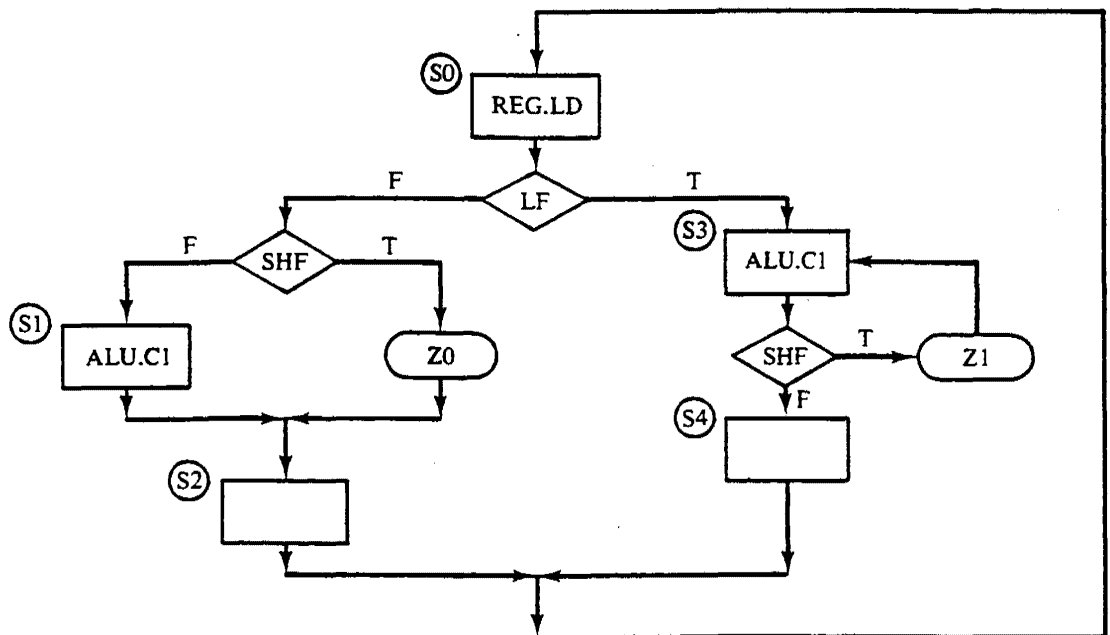


Figure 7.43 ASM chart for Problem 7.8.

- Make a block diagram of the corresponding controller, specifying the controller inputs and outputs.
- Make a block diagram design (similar to the one shown in Fig. 7.3) of the corresponding state generator, specifying the inputs, outputs, and the state flip-flops (D type).

- (c) Determine the next-state table for the state generator, given the following state code assignment:

	C2	C1	C0
S0	0	0	0
S1	0	0	1
S2	0	1	0
S3	0	1	1
S4	1	0	0

- (d) Realize the controller by the traditional method. Use D flip-flops, and draw a detailed circuit diagram of the controller.

- 7.9. Given the ASM chart of Fig. 7.4,

- (a) Specify the ASM outputs as a function of the state code and the ASM inputs. In other words, complete the following truth table:

C1	C0	IN. BIT	BUF. FULL	COUNT. EN	REG. LD	OUT. FLAG
0	0	0	0			
0	0	0	1			
0	0	1	0			
1	1	1	1			

- (b) Using gates, realize the ASM outputs directly as a function of the state code and the ASM inputs. Compare your realization with the one shown in Fig. 7.6, which has a circuit for decoding the state code.

- 7.10. Given the ASM chart of Fig. 7.40(a) and the code assignment of Problem 7.7(b),

- (a) Specify, in the form of a truth table, the ASM outputs (Z1, Z2, FF.LD, and FF.CLR) as a function of the state code (C1, C0) and the ASM inputs (X, FLAG).

- (b) Using gates, realize the ASM outputs directly as a function of the state code and the ASM inputs. Compare this realization with your solution to Problem 7.7.

- 7.11. For the ASM chart of Fig. 7.40(a) and the code assignment of Problem 7.7(b), realize the corresponding controller by using a PAL16R4. Compare it with your solution to Problem 7.7.

- 7.12. For the ASM chart of Fig. 7.43 and the code assignment of Problem 7.8, realize the corresponding controller by using a PAL16R4. Compare it with your solution to Problem 7.8.

- 7.13. For the ASM chart of Fig. 7.40(a) and the code assignment of Problem 7.7(b), realize the corresponding controller by using the ROM method and D flip-flops. That is,

- (a) Make a block diagram design of the controller, specifying the state flip-flops and all the appropriate connections to the inputs and outputs of the ROM. In other words, make a block diagram design similar to the one shown in Fig. 7.8, but without the actual ROM contents.

- (b) Specify the ROM contents in hexadecimal.

- 7.14. Repeat Problem 7.13 for the ASM chart of Fig. 7.43 and the code assignment of Problem 7.8(c).

7.15. Figure 7.44 shows a block diagram design of a controller based on the ROM method and with D flip-flops. The ROM contents are also given. Derive the corresponding ASM chart.

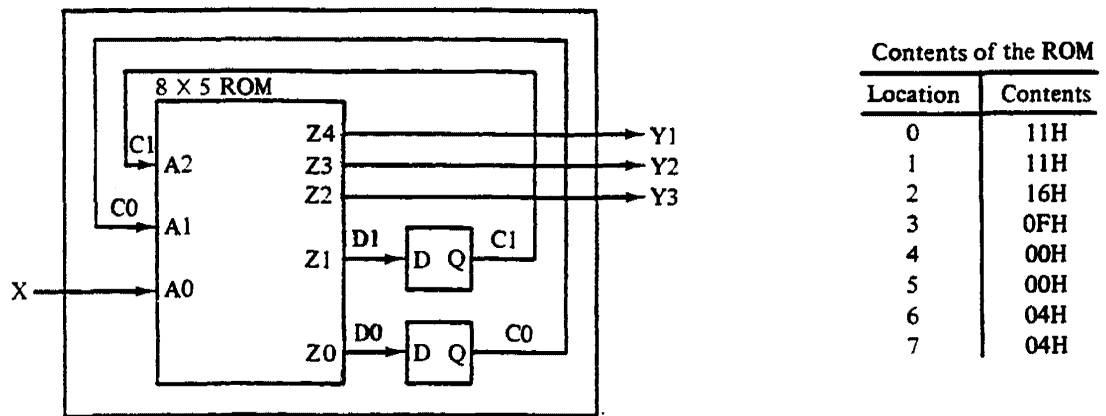


Figure 7.44 Controller block diagram and ROM contents for Problem 7.15.

7.16. For the ASM chart of Fig. 7.39(a),

- (a) Draw a block diagram of the corresponding controller, specifying the controller inputs and outputs.
- (b) Make a block diagram design (similar to the one of Fig. 7.10) for the corresponding state generator, specifying the inputs, outputs, and the state flip-flops (J-K type).
- (c) Determine the next-state table for the state generator, given the following state code assignment:

	C1	C0
SA	0	0
SB	0	1
SC	1	0
SD	1	1

- (d) Realize the controller by the traditional method. Use J-K flip-flops, and draw a detailed circuit diagram of the controller.

7.17. For the ASM chart of Fig. 7.39(a) and the state code assignment of Problem 7.16(c), realize the corresponding controller by using the ROM method and J-K flip-flops: that is,

- (a) Make a block diagram design of the controller, specifying the state flip-flops and all the appropriate connections to the ROM inputs and outputs.
- (b) Specify the ROM contents in hexadecimal.

7.18. Convert the Mealy state graph of Fig. 7.45 into an equivalent ASM chart. The inputs are X1 and X2, and the outputs are Z1 and Z2.

7.19. Convert the Moore state graph of Fig. 7.46 into an equivalent ASM chart. The inputs are X1 and X2, and the outputs are Z1, Z2, and Z3.

7.20. Assume that the design of the dynamic RAM controller circuit described in Sec. 7.8.1 is to be modified. In addition to the already specified functions, if the RD and WR signals are both true, then the function of the circuit is to be described by the timing diagram of Fig. 7.47. Specifically,

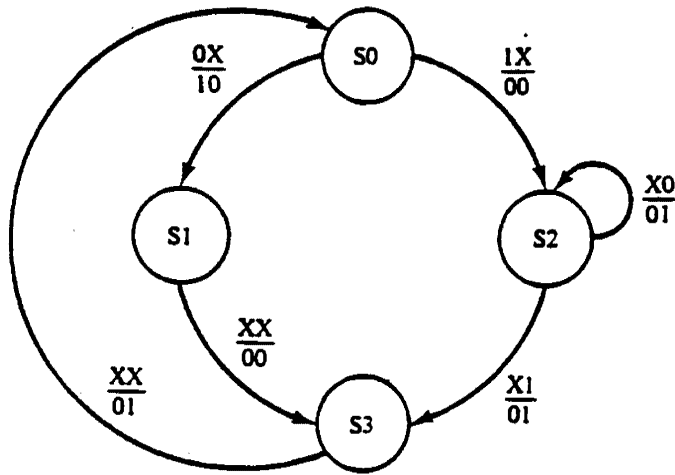


Figure 7.45 Mealy state graph for Problem 7.18.

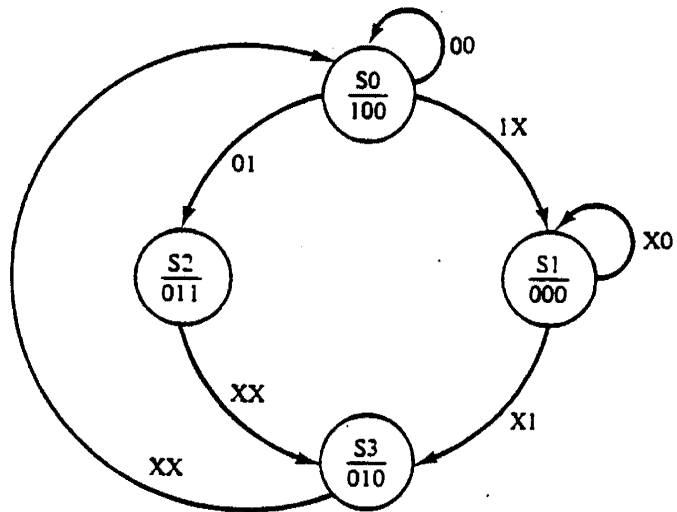


Figure 7.46 Moore state graph for Problem 7.19.

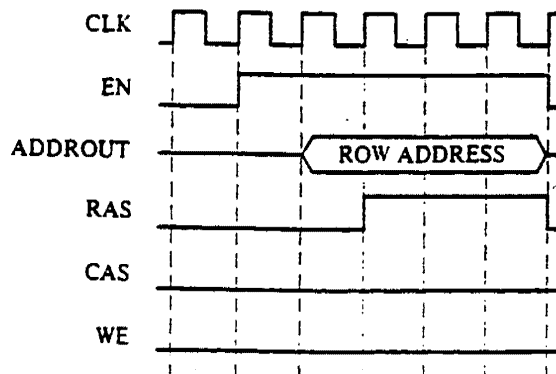


Figure 7.47 Timing diagram for Problem 7.20.

- (a) Modify the ASM chart of Fig. 7.20 to include this function.
 (b) Realize this modified ASM chart by using the ROM method and D flip-flops.
- 7.21. An 8-bit parallel-to-serial converter circuit, the block diagram of which is shown in Fig. 7.48, is to be designed and realized. This circuit remains in an idle state as long as the START input is false. But when this START input becomes true, the 8-bit data BYTE is loaded into the shift register and the right-shifting of the data begins. After the 8 bits are shifted out, the circuit returns to the idle state.
- (a) The major circuit elements for the parallel-to-serial converter circuit are given in Fig. 7.48. Make any reasonable assumptions for the circuit elements that are required and derive the ASM chart for the controller. (*Hint*: A two-state ASM chart can be derived for this circuit.)
- (b) Realize the circuit elements with commercially available chips.
- (c) Realize the ASM chart by using any of the methods presented in this chapter.
- 7.22. A lock to a certain safe can be opened with a correct combination or with a key. If the combination feature is used, then the lock must be supplied with the correct combination within three attempts. Otherwise an alarm will sound. Specifications for the lock circuit are as follows:
1. After the first unsuccessful attempt, output message 1 (MSG1).
 2. After the second unsuccessful attempt,
 - (a) output message 2 (MSG2), and
 - (b) wait for 10 seconds before being ready to be tried again (READY).
 3. After the third successive unsuccessful attempt, sound an alarm.
 4. When the safe is opened, reset to allow three more tries.
 5. When the key is used, stop the alarm and reset to allow three more tries.
- Your part in this problem is to design the module of Fig. 7.49. Make a detailed design of this module, using a counter for the major circuit element. Realize the controller with the ROM method and D flip-flops.
- 7.23. Shown in Fig. 7.50(a) are the circuit elements for a digital circuit that functions as follows:

If $OP = 00$, then $REGB \leftarrow REGA$: followed by $REGA \leftarrow IN + 1$.

If $OP = 01$, then $REGB \leftarrow 0$; $REGA \leftarrow REGA + 1$.

If $OP = 10$, $REGA \leftarrow IN$; $REGB \leftarrow IN$; followed by $REGB \leftarrow REGB + 1$.

If $OP = 11$, then the contents of $REGA$ are doubled; $REGB \leftarrow REGB + 1$.

Notation: An arrow (\leftarrow) designates to load the data at the next active clock transition. For example, $REGA \leftarrow IN + 1$ designates to add 1 to the value of IN and load the sum into $REGA$ at the next active clock transition. In the function statements, more than one operation can be performed within a single state unless otherwise stated, as designated by "followed by."

The flowchart for the controller is given in Fig. 7.50(b). Derive the corresponding ASM chart for it. Optimize it by using the least number of states. In other words, if more than one operation can be performed within a single state, then do so. Also, make use of conditional outputs.

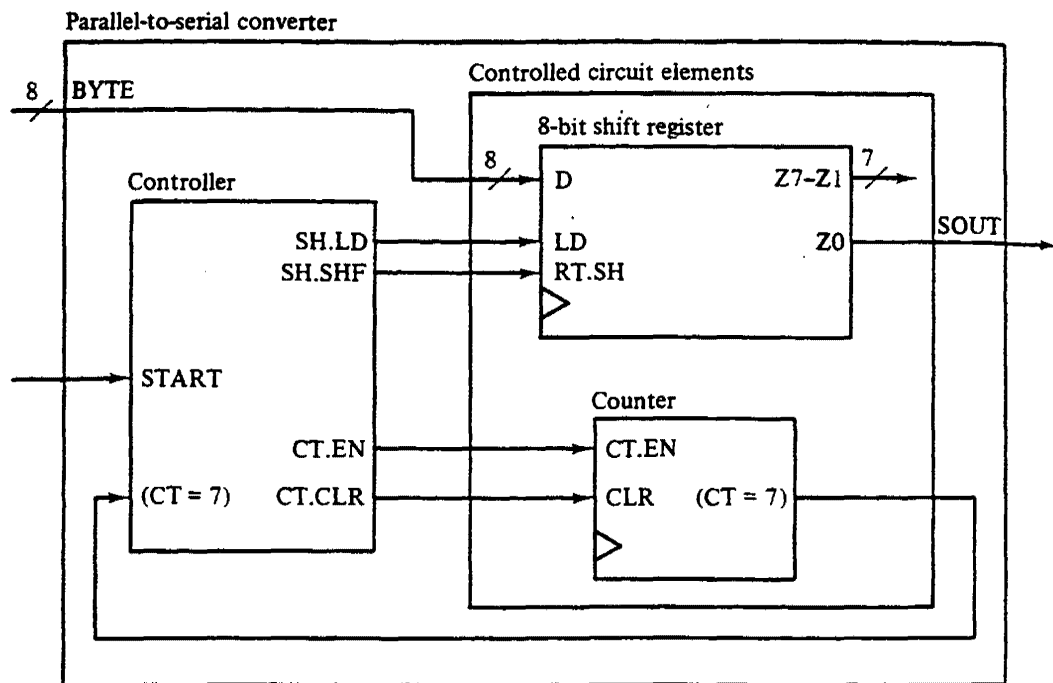
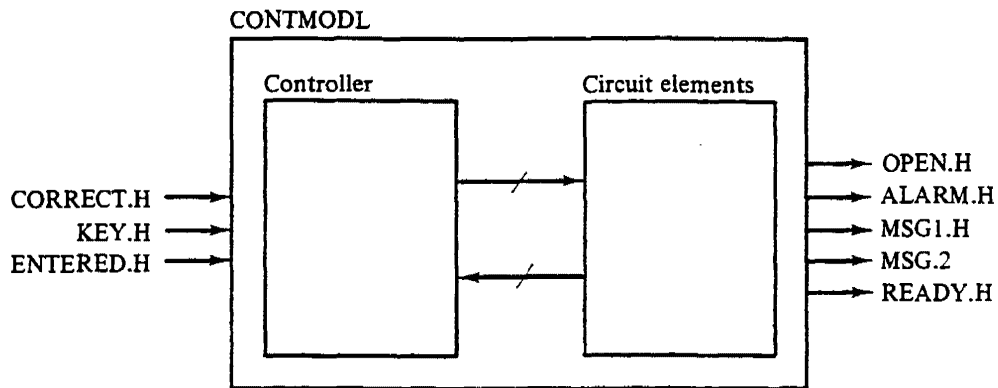


Figure 7.48 Parallel-to-serial converter circuit for Problem 7.21.



Signal specifications

CORRECT (from another comparator module): T = correct combination;
F = incorrect combination

KEY: T = a key is used to open the safe; F = no key is used

ENTERED: T = another combination is entered;

F = another attempt has not been made

OPEN: an output signal for another module to open the safe

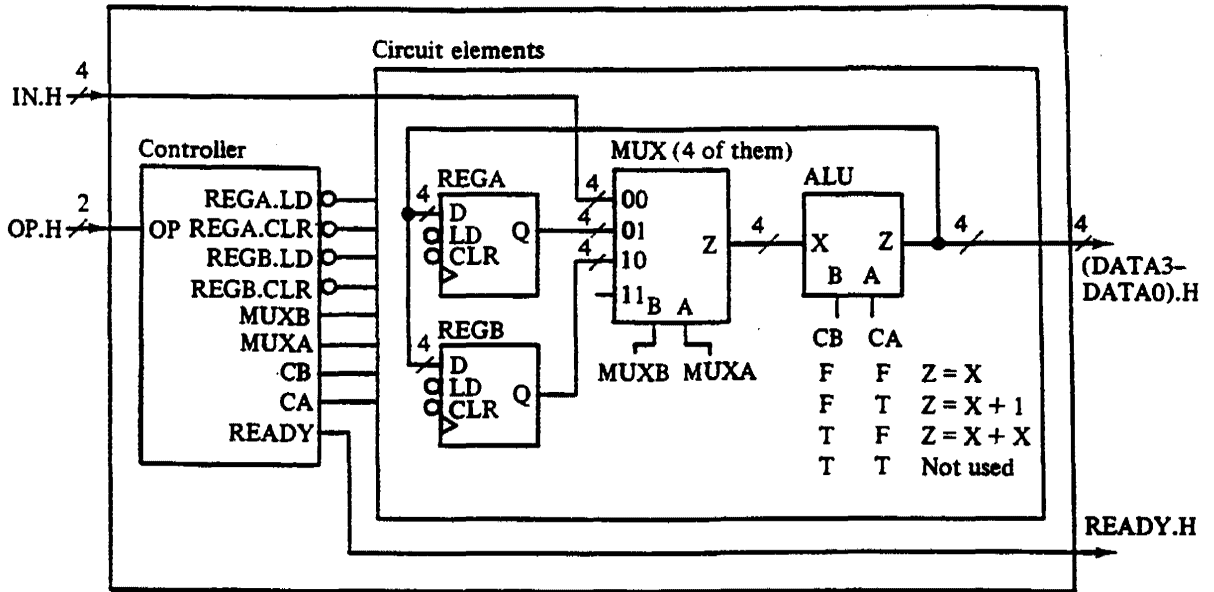
ALARM: an output signal for another module to sound the alarm

MSG1: an output signal for another module to output message 1

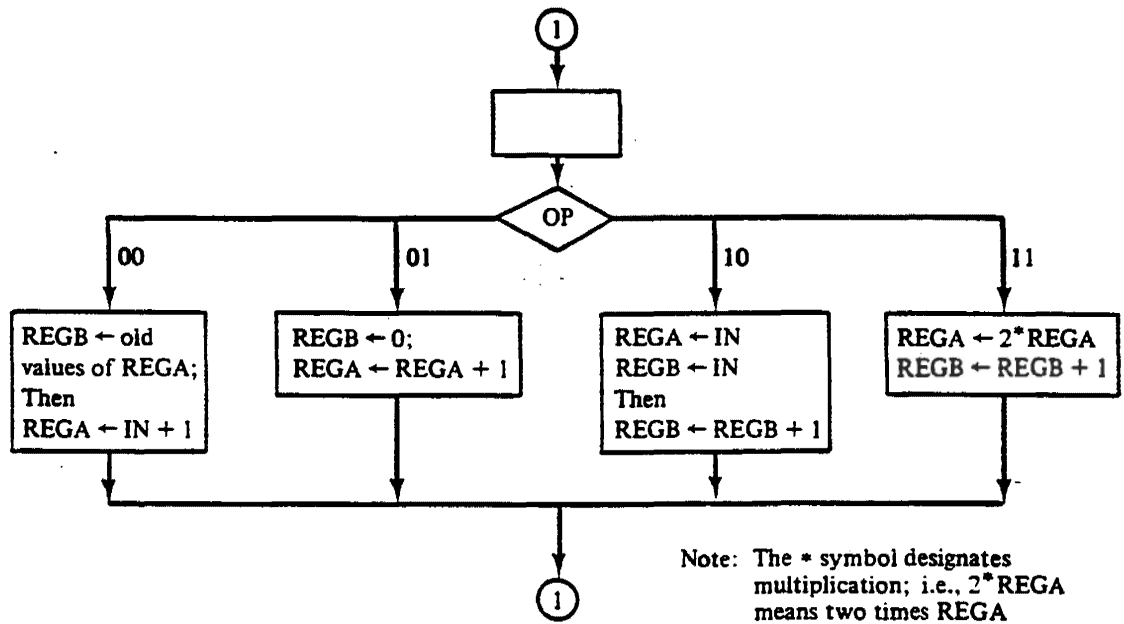
MSG2: an output signal for another module to output message 2

READY: an output signal for another module to allow another attempt

Figure 7.49 Module for Problem 7.22.



(a)



(b)

Figure 7.50 Circuit elements and flowchart for Problem 7.23.

7.24. Figure 7.51(a) shows circuit elements for a digital circuit that is to be designed. They are organized in a common bus structure.

(a) It is important not to have more than one set of Z outputs connected to the common bus at any one time. Why?

(b) With the shown connections of circuit elements, can we perform the following operations within a single state?

$$\text{REGA} \leftarrow \text{REGD} \quad \text{and} \quad \text{REGC} \leftarrow \text{REGA}$$

Explain your answer.

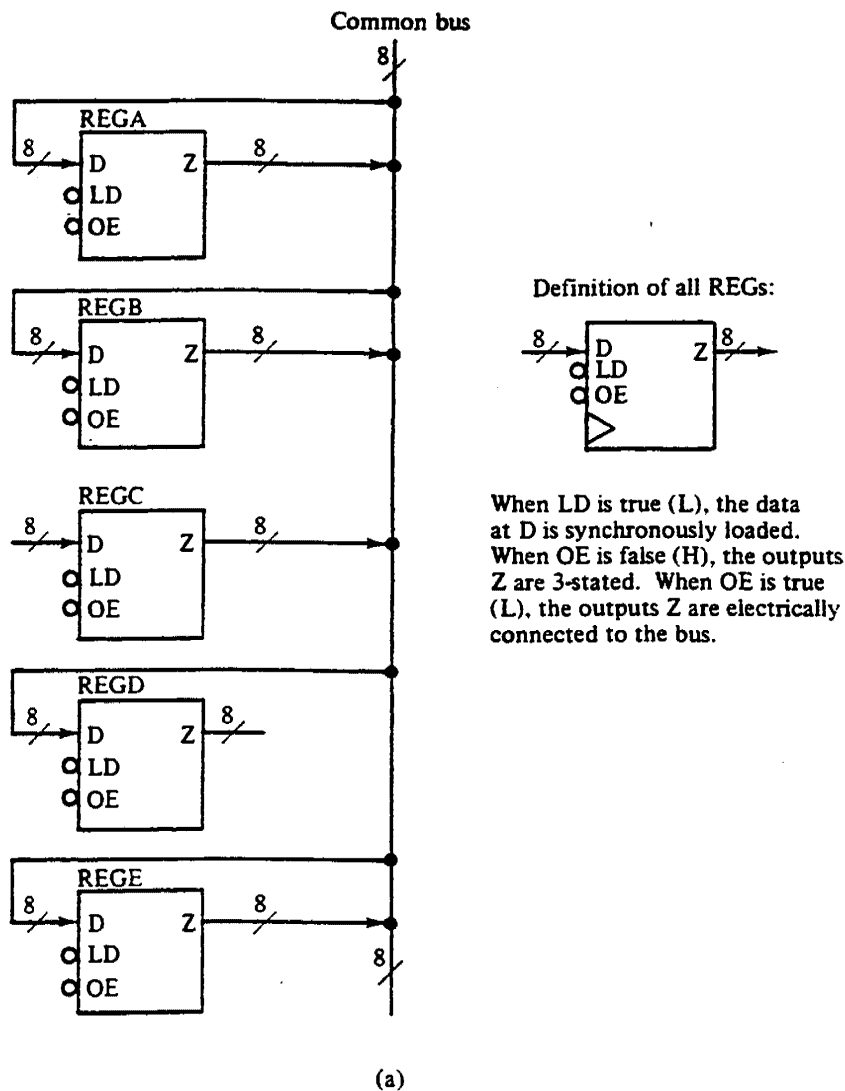
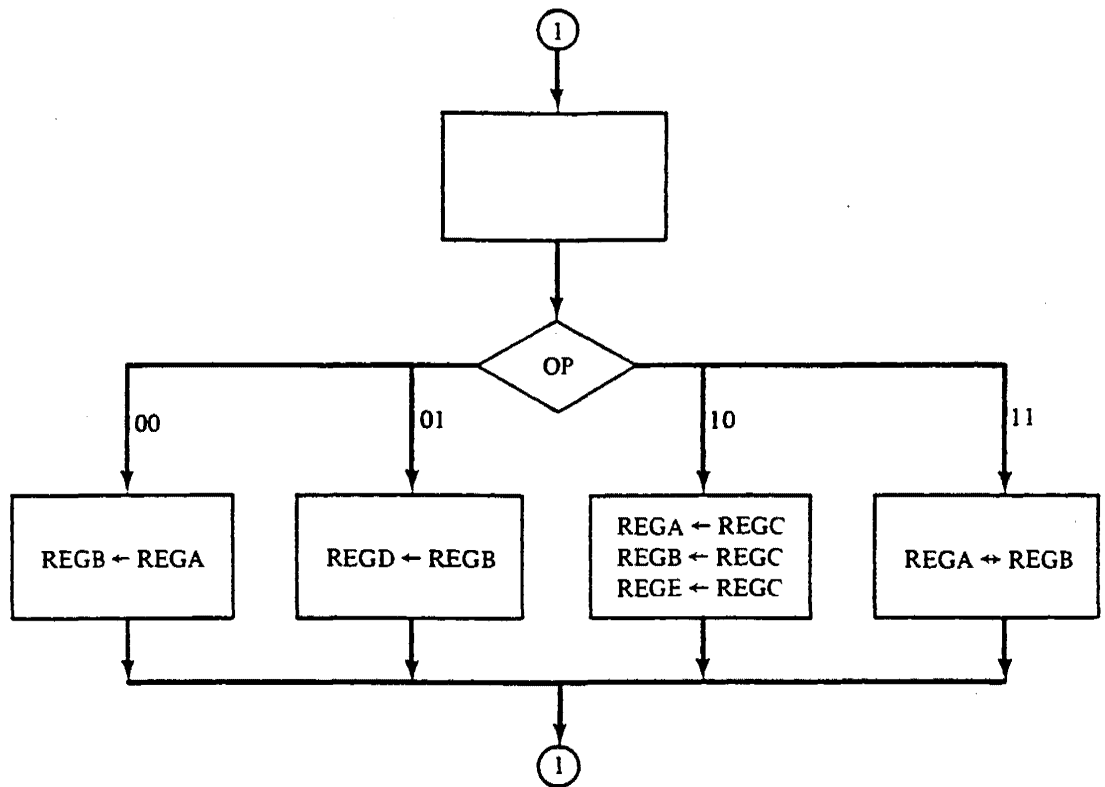


Figure 7.51 Circuit elements and flowchart for Problem 7.24.

- (c) The flowchart for the controller is given in Fig. 7.51(b). Derive the corresponding ASM chart for it. Optimize it by using the least number of states. In other words, if more than one operation can be performed in a single state, then do so. Also, make use of conditional outputs.
- 7.25. (a) Given in Fig. 7.52(a) are the circuit elements for a digital circuit that is to be designed. They are organized in a common bus structure. Assume that the access time of MEM is 125 ns (nanoseconds), and that the clock period for the ASM is 100 ns. Then, how many states are required to perform a MEM read or MEM write operation?
- (b) The flowchart for the controller is given in Fig. 7.52(b). Derive the corresponding ASM chart for it. Optimize the ASM chart by using the least number of states. In other words, if more than one operation can be performed in a single state, then do so. Also, make use of conditional outputs.



Note: REGA ↔ REGB means to interchange the contents of REGA and REGB

(b)

Figure 7.51 (cont.)

7.26. A hardware stack module, the block diagram of which is shown in Fig. 7.53, is to be designed and implemented. The function of this hardware stack module is defined as follows:

If STKENBL = 0, do nothing.

If STKENBL = 1, then there are four possible operations, depending on OP:

OP = 00 DEFINE a new top of stack; i.e., $SP \leftarrow IN$.

OP = 01 PUSH the stack; i.e., increment SP; $MEM(SP) \leftarrow IN$.

OP = 10 POP the stack; i.e., $MEM(SP)$ is connected to OUT until STKENBL becomes 0; decrement SP.

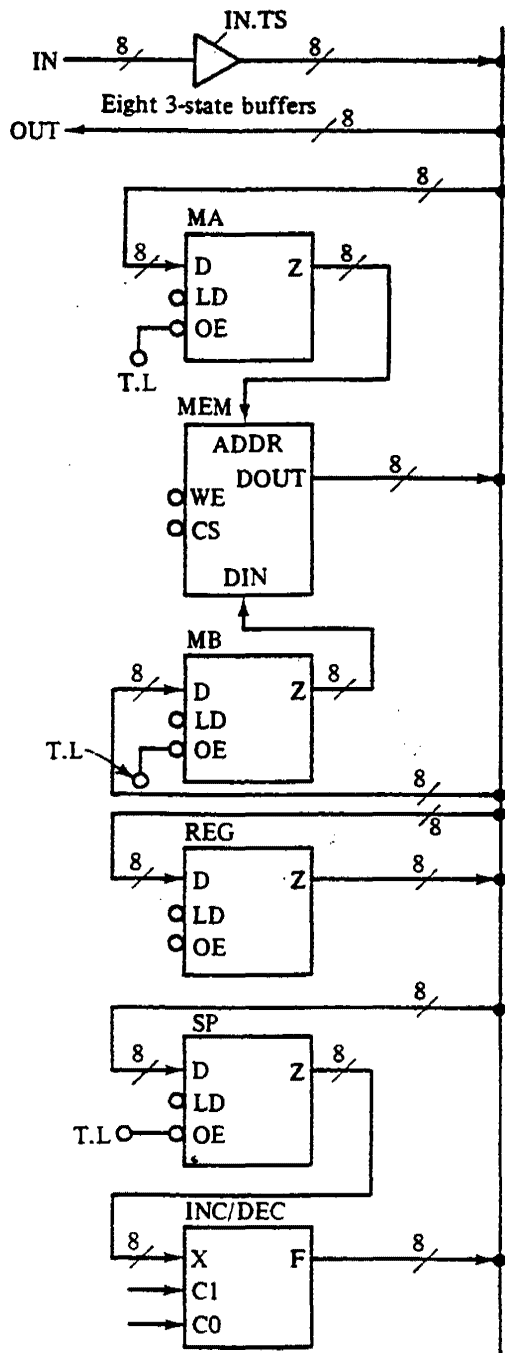
OP = 11 READ the top of the stack; i.e., SP is connected to OUT until STKENBL becomes 0.

Notes:

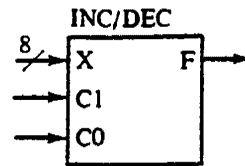
1. SP contains the address (8 bits) of the top of the stack.
2. MEM(SP) is the memory content of that address.
3. Do not worry about the stack being empty or full.

(a) Using the circuit elements shown in Fig. 7.52(a), derive the ASM chart for the controller for the hardware stack module. Optimize it by using the minimum number of states.

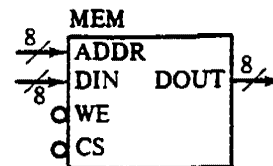
(b) Using any commercially available chips, realize your design.



Definition of circuit elements:
All registers (MA, MB, REG, and SP) are defined the same as the registers in Fig. 7.51(a).



C1	C0	Function
0	0	Outputs F are 3-stated
0	1	$F = X$
1	0	$F = X + 1$
1	1	$F = X - 1$



MEM is a 256 X 8 RAM module.

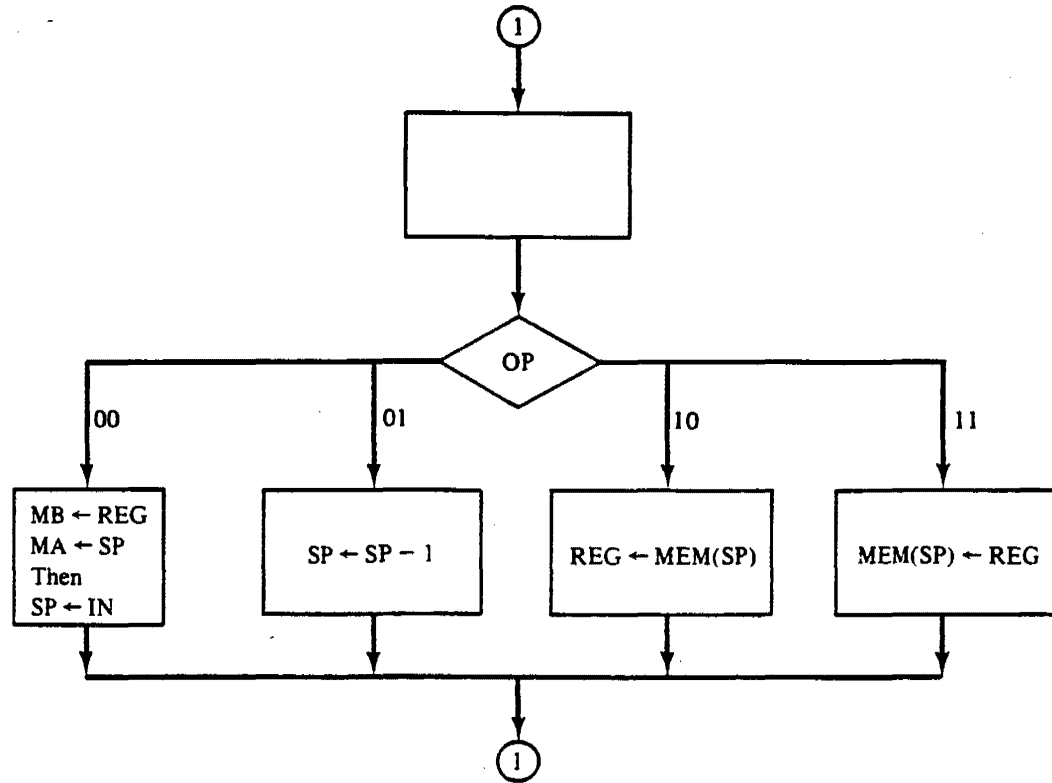
WE	CS	Function
X	H	Outputs DOUT are 3-stated.
L	L	Data applied at DIN is written to the RAM location specified by ADDR. Also, DOUT outputs are 3-stated.
H	L	Contents of RAM location specified by ADDR are outputted at DOUT.

(a)

Figure 7.52 Circuit elements and controller flowchart for Problem 7.25.

7.27. Repeat Problem 7.26 and take care of the problem of whether the stack is empty (for the POP operation) or full (for the PUSH operation).

7.28. Design and realize the hardware multiplier of Sec. 7.8.4. modified as follows: Instead of using two separate 4-bit registers (MPLIER and PRODLO) to store the multiplier and the low-nibble product, as shown in Fig. 7.36(a), use PRODLO for storing both the multiplier and low-nibble product. Specifically, use PRODLO initially for storing the multiplier since



Note: MEM(SP) means the contents of a memory location whose address is stored in the SP register.

(b)

Figure 7.52 (cont.)

this register is not used initially to store the low-nibble product. As the multiplication process proceeds, have the multiplier shifted out of PRODLO one bit at a time. At the same time, have the low-nibble product shifted into PRODLO from PRODHI. This is a more elegant design that saves the use of a register.

- 7.29. (a) Draw a block diagram for an enhanced hardware multiplier and specify its functions. You have the flexibility of incorporating any features that you desire. For example, you

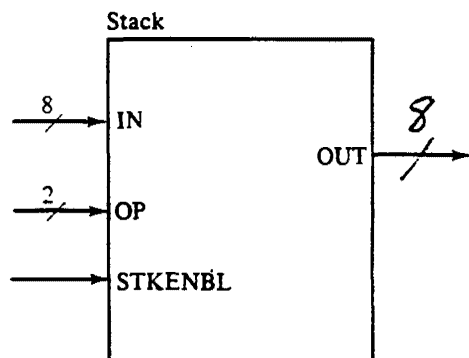


Figure 7.53 Hardware stack module for Problem 7.26.

may want your multiplier to be able to perform the multiplication of signed binary numbers, or of BCD numbers, or have additional handshaking capabilities. and so forth.

- (b) Design and realize this enhanced multiplier.
- 7.30. A stack, as described in Problem 7.26, is a first-in, last-out (FILO) structure. In other words, when an 8-bit data is stored (pushed) onto the stack, it cannot be retrieved (popped) until all the data that has been subsequently stored is popped first, much like a stack of trays in a cafeteria. On the other hand, a queue is a first-in, first-out (FIFO) structure. In other words, the first 8-bit data stored is the first to be retrieved, much like the servicing of a line of customers in a cafeteria. With this background in mind,
- (a) Draw a block diagram for a hardware queue module and specify its functions. You have the flexibility of incorporating any features that you desire.
- (b) Design and realize this hardware queue module.
- 7.31. For the transmission of asynchronous serial data, a start bit must be inserted before each data byte and a stop bit inserted after the data byte. Also, for error-checking purposes, sometimes a parity bit is inserted between the data byte and the stop bit. In this problem you are to redesign the parallel-to-serial converter of Fig. 7.48 so that it will perform these insertions. Specifically, when the START input is false, the converter is to be in an idle state in which it outputs a "high" at SOUT. But when START becomes true, the converter loads the 8-bit data BYTE into the shift register and shifts out the start bit first, after which it begins the shifting out of the data. After shifting out the 8 data bits, it shifts out the parity bit, followed by the stop bit.

Notes:

1. The value of the start bit is L.
2. The value of the stop bit is H.
3. The value of the parity bit has to be determined as follows: the parity bit = L if the number of ones in the data byte is an even number; the parity bit = H if the number of ones in the data byte is an odd number.