PRINCIPLES OF MODERN DIGITAL DESIGN

PARAG K. LALA

PRINCIPLES OF MODERN DIGITAL DESIGN

Parag K. Lala
Cary and Lois Patterson Chair of Electrical Engineering Texas A&M University–Texarkana

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

PRINCIPLES OF MODERN DIGITAL DESIGN

The Wiley Bicentennial–Knowledge for Generations

Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!

William J. Pesce
President and Chief Executive Officer

Peter Booth Wiley
Chairman of the Board

PRINCIPLES OF MODERN DIGITAL DESIGN

Parag K. Lala
Cary and Lois Patterson Chair of Electrical Engineering Texas A&M University–Texarkana

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data:

Lala, Parag K., 1948–
Principles of modern digital design / by Parag K. Lala.
p. cm.
Includes index.
ISBN 978-0-470-07296-7 (cloth/cd)
1. Logic design. 2. Logic circuits—Design and construction. 3. Digital electronics. I. Title
TK7868. L6L3486 2007
621.39′5--dc22 2006032483

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To Mrs. Mithilesh Tiwari and Miss Shakuntala Tiwari for their love

"*Full many a gem of purest ray serene,*
The dark unfathomed caves of ocean bear:
Full many a flower is born to blush unseen,
And waste its sweetness on the desert air."

Thomas Gray

CONTENTS

PREFACE

This book covers all major topics needed in a modern digital design course. A number of good textbooks in digital design are currently available. Some of these introduce VHDL before students get a good grasp of the fundamentals of digital design. VHDL is a language that is used to describe the function of digital circuits/systems. In the author's opinion, students benefit more from VHDL only when they can appreciate the advantages of using it in digital design. In this book, VHDL is introduced only after a thorough coverage of combinational circuit design and a discussion of the fundamental concepts of sequential circuits.

The complexity of modern digital systems is such that they have to be designed using computer-aided design (CAD) synthesis and minimization tools. The techniques used in some of the CAD tools, for example computer-aided minimization, multilevel logic design, and state assignment are inadequately covered or not covered at all in current undergraduate text books. In this book, the basic concepts of some of these important techniques are introduced in appropriate chapters. The material has been discussed in a tutorial form, although the nature of certain topics makes an abstract discussion unavoidable. The objective is not to achieve understanding at the expense of avoiding necessary theory, but to clarify the theory with illustrative examples in order to establish the theoretical basis for practical implementations.

The book is subdivided into ten chapters.

Chapter 1 provides coverage of number representations and considers various number formats. It also discusses binary arithmetic operations such as addition, subtraction, multiplication, and division.

Chapter 2 provides a comprehensive coverage of a miscellany of basic topics in discrete mathematics required for understanding material presented in later chapters. Also, the operations of various gates used to construct logic circuits are discussed.

Chapter 3 provides an in-depth coverage of combinational logic circuit analysis, minimization, and design techniques. The representation of Boolean functions using cubes is explained and the concept of tautology is discussed. The principles of heuristic minimization, different types of don't cares and multilevel logic synthesis is explained with many examples. A detailed coverage of all types of arithmetic circuits including BCD addition/subtraction algorithms and carry-save addition techniques is provided. Multiplication and division are thoroughly discussed. Combinational logic implementation using Programmable Logic Devices (PLDs) is also covered.

Chapter 4 presents the basic concepts of sequential circuits. The operation of memory elements is analyzed. The use of state diagrams and state tables to represent the behavior of sequential circuits is discussed. Also, the distinction between synchronous and asynchronous operation of sequential circuits is clarified.

It is quite routine in the electronics industry to use a hardware description language such as VHDL to describe the function of digital circuits. Chapter 5 introduces the language in sufficient detail so that readers can write VHDL code for representing digital circuits.

Several examples are given to clarify different ways of representing digital circuit using VHDL. This chapter is not meant to be an exhaustive guide to VHDL; a number of excellent books that deal exclusively with VHDL have been published in recent years.

Chapter 6 builds on the previous chapter and focuses on VHDL code for computer-aided synthesis of combinational logic circuits. Certain features of the VHDL that result in more efficient code for combinational logic circuits are presented. All these are illustrated with complete VHDL codes that have been compiled and synthesized using Altera Corporation's Quartus II software package.

Chapter 7 provides a clear picture of how sequential circuits are designed using fundamental building blocks (e.g., latches and flip-flops) rather than presenting a rigorous mathematical structure of such circuits. Algorithms that are used in some of the currently popular computer-aided state assignment techniques are discussed. A good coverage of partition algebra for deriving state assignment has been included. A detailed discussion of sequential circuit implementation using PLDs is also presented.

Chapter 8 provides comprehensive coverage of counters. Counters are important in many digital applications. Several design examples and illustrations are provided to clarify the design of various types of counters.

Chapter 9 presents VHDL coding of sequential circuits. The coding style for sequential circuits is different from that of combinational circuits. Combinational circuits are usually coded using *concurrent* VHDL statements whereas sequential circuits use mainly *sequential* VHDL statements. Many examples of VHDL coding of sequential circuits are included; these codes have been compiled and synthesized using Quartus II.

Chapter 10 covers design principles for traditional fundamental mode non-synchronous sequential circuits. The concepts of race and hazard are clarified with examples, and state assignment techniques to avoid these are also discussed.

All modern digital systems are implemented using CMOS technology. A short introduction to CMOS logic is provided in Appendix A.

A Quartus II CD ROM from Altera Corporation is included in the book. All the examples in the book have been compiled and synthesized using this state-of-the-art and user-friendly software package.

This book is primarily intended as a college text for a two-semester course in logic design for students in electrical/computer engineering and computer science degree programs, or in electrical/computer technology. It does not require any previous knowledge of electronics; only some general mathematical ability is assumed.

In the first (introductory) course the following sequence of chapters may be covered: Chapter 1, Chapter 2, Chapter 3 (3.1 to 3.4, 3.8, 3.12 to 3.14), Chapter 4, Chapter 7 (Sections 7.1–7.5), Chapter 8.

In the second (more advanced) course the suggested sequence of chapters is: Chapter 3 (Sections 3.5 to 3.7, 3.9 to 3.11), Chapter 5, Chapter 6, Chapter 7 (Section 7.6), Chapter 9 and Chapter 10.

Although the book is meant to be used for a two-semester course sequence, certain sections can be omitted to fit the material in a typical one-semester course. Individual instructors may select chapters at their discretion to suit the needs of a particular digital design course they are teaching.

This book should also be extremely useful for practicing engineers who took logic design courses five or more years ago, to update their knowledge. Electrical engineers who are not logic designers by training but wish to become one, can use this book for self-study.

I am grateful to Dr. Karen Panetta of the Department of Electrical and Computer Engineering, Tufts University for her constructive review and suggestions, and for permitting me to use problems from her laboratory curriculum in VHDL.

I would also like to thank my former students in several universities who took digital design courses I taught over the years. I made references to class projects of some of them in appropriate sections of the book.

I am greatly indebted to my wife, Meena, for her patience. She has been a constant source of support throughout the writing of the book. Finally I would like to thank my children Nupur and Kunal for their quiet encouragement and for being who they are.

PARAG K. LALA

1 Number Systems and Binary Codes

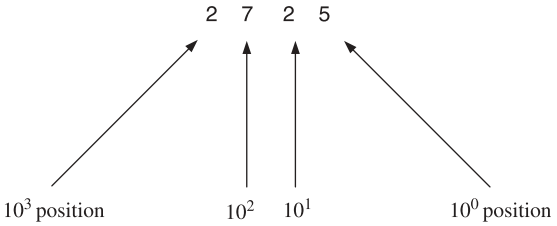
1.1 INTRODUCTION

In conventional arithmetic, a number system based on ten units (0 to 9) is used. However, arithmetic and logic circuits used in computers and other digital systems operate with only 0's and 1's because it is very difficult to design circuits that require ten distinct states. The number system with the basic symbols 0 and 1 is called *binary*. Although digital systems use binary numbers for their internal operations, communication with the external world has to be done in decimal systems. In order to simplify the communication, every decimal number may be represented by a unique sequence of binary digits; this is known as *binary encoding*. In this chapter we discuss number systems in general and the binary system in particular. In addition, we consider the octal and hexadecimal number systems and fixed- and floating-point representation of numbers. The chapter ends with a discussion on weighted and nonweighted binary encoding of decimal digits.

1.2 DECIMAL NUMBERS

The invention of decimal number systems has been the most important factor in the development of science and technology. The term decimal comes from the Latin word for "ten." The decimal number system uses positional number representation, which means that the value of each digit is determined by its position in a number.

The base (also called radix) of a number system is the number of symbols that the system contains. The decimal system has ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; in other words it has a base of 10. Each position in the decimal system is 10 times more significant than the previous position. For example, consider the four-digit number 2725:



Notice that the 2 in the 10^3 position has a different value than the 2 in the 10^1 position. The value of a decimal number is determined by multiplying each digit of the number by the

Principles of Modern Digital Design, by Parag K. Lala

value of the position in which the digit appears and then adding the products. Thus the number 2725 is interpreted as

$$2 \times 1000 + 7 \times 100 + 2 \times 10 + 5 \times 1 = 2000 + 700 + 20 + 5$$

that is, two thousand seven hundred twenty-five. In this case, 5 is the least significant digit (LSD) and the leftmost 2 is the most significant digit (MSD).

In general, in a number system with a base or radix r, the digits used are from 0 to $r - 1$. The number can be represented as

$$N = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r^1 + a_0 r^0 \tag{1.1}$$

where, for $n = 0, 1, 2, 3, \ldots,$

$r =$ base or radix of the number system
$a =$ number of digits having values between 0 and $r - 1$

Thus, for the number 2725, $a_3 = 2$, $a_2 = 7$, $a_1 = 2$, and $a_0 = 5$. Equation (1.1) is valid for all integers. For numbers between 0 and 1 (i.e., fractions), the following equation holds:

$$N = a_{-1} r^{-1} + a_{-2} r^{-2} + \cdots + a_{-n+1} r^{-n+1} + a_{-n} r^{-n} \tag{1.2}$$

Thus for the decimal fraction 0.8125,

$$\begin{aligned} N &= 0.8000 + 0.0100 + 0.0020 + 0.0005 \\ &= 8 \times 10^{-1} + 2 \times 10^{-2} + 1 \times 10^{-3} + 8 \times 10^{-4} \\ &= a_{-1} \times 10^{-1} + a_{-2} \times 10^{-2} + a_{-3} \times 10^{-3} + a_{-4} \times 10^{-4} \end{aligned}$$

where

$$\begin{aligned} a_{-1} &= 8 \\ a_{-2} &= 1 \\ a_{-3} &= 2 \\ a_{-1} &= 5 \end{aligned}$$

1.3 BINARY NUMBERS

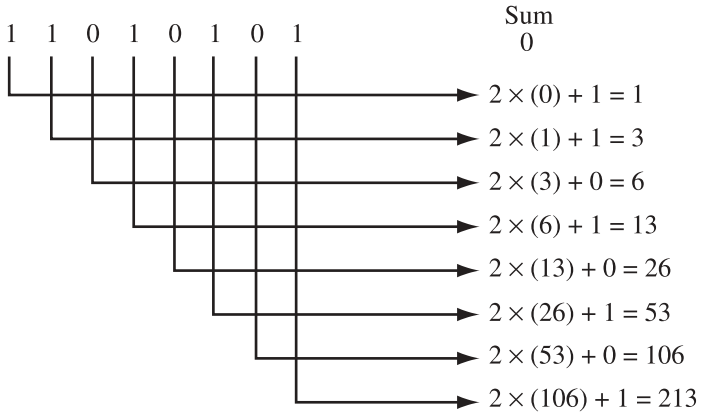
The binary numbers has a radix of 2. As $r = 2$, only two digits are needed, and these are 0 and 1. A binary digit, 0 or 1, is called a bit. Like the decimal system, binary is a positional system, except that each bit position corresponds to a power of 2 instead of a power of 10. In digital systems, the binary number system and other number systems closely related to it are used almost exclusively. However, people are accustomed to using the decimal number system; hence digital systems must often provide conversion between decimal and binary numbers. The decimal value of a binary number can be formed by multiplying each power of 2 by either 1 or 0, and adding the values together.

Example 1.1 Let us find the decimal equivalent of the binary number 101010.

$$
\begin{aligned}
N &= 101010 \\
&= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \quad \text{(using Eq. (1.1))} \\
&= 32 + 0 + 8 + 0 + 2 + 0 \\
&= 42
\end{aligned}
$$

An alternative method of converting from binary to decimal begins with the leftmost bit and works down to the rightmost bit. It starts with a sum of 0. At each step the current sum is multiplied by 2, and the next digit to the right is added to it.

Example 1.2 The conversion of 11010101 to decimal would use the following steps:



The reverse process, the conversion of decimal to binary, may be made by first decomposing the given decimal number into two numbers—one corresponding to the positional value just lower than the original decimal number and a remainder. Then the remainder is decomposed into two numbers: a positional value just equal to or lower than itself and a new remainder. The process is repeated until the remainder is 0. The binary number is derived by recording 1 in the positions corresponding to the numbers whose summation equals the decimal value.

Example 1.3 Let us consider the conversion of decimal number 426 to binary:

$$
\begin{aligned}
426 &= 256 + 170 \\
&= 256 + 128 + 42 \\
&= 256 + 128 + 32 + 10 \\
&= 256 + 128 + 32 + \ 8 + 2 \\
&\qquad\ \uparrow \quad\ \uparrow \quad\ \uparrow \quad\ \uparrow \quad\ \uparrow \\
&\qquad 2^8 \quad 2^7 \quad 2^5 \quad 2^3 \quad 2^1
\end{aligned}
$$

Thus $426_{10} = 110101010_2$ (the subscript indicates the value of the radix).

An alternative method for converting a decimal number to binary is based on successive division of the decimal number by the radix number 2. The remainders of the divisions, when written in reverse order (with the first remainder on the right), yield the binary equivalent to the decimal number. The process is illustrated below by converting 353_{10} to binary,

$$\frac{353}{2} = 176, \text{ remainder } 1$$
$$\frac{176}{2} = 88, \text{ remainder } 0$$
$$\frac{88}{2} = 44, \text{ remainder } 0$$
$$\frac{44}{2} = 22, \text{ remainder } 0$$
$$\frac{22}{2} = 11, \text{ remainder } 0$$
$$\frac{11}{2} = 5, \text{ remainder } 1$$
$$\frac{5}{2} = 2, \text{ remainder } 1$$
$$\frac{2}{2} = 1, \text{ remainder } 0$$
$$\frac{1}{2} = 0, \text{ remainder } 1$$

Thus $353_{10} = 101100001_2$.

So far we have only considered whole numbers. Fractional numbers may be converted in a similar manner.

Example 1.4 Let us convert the fractional binary number 0.101011 to decimal. Using Eq. (1.2), we find

$$\begin{aligned} N &= 0.101011 \\ &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} \end{aligned}$$

where $a_{-1} = 1$, $a_{-2} = 0$, $a_{-3} = 1$, $a_{-4} = 0$, $a_{-5} = 1$, $a_{-6} = 1$.

Thus

$$\begin{aligned} N &= 0.101011 \\ &= \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \frac{1}{64} = 0.671875 \end{aligned}$$

A decimal fraction can be converted to binary by successively multiplying it by 2; the integral (whole number) part of each product, 0 or 1, is retained as the binary fraction.

Example 1.5 Derive the binary equivalent of the decimal fraction 0.203125. Successive multiplication of the fraction by 2 results in

$$\begin{array}{rr} & 0.203125 \\ & \underline{\qquad 2} \\ a_{-1} = 0 & 0.406250 \\ & \underline{\qquad 2} \\ a_{-2} = 0 & 0.812500 \\ & \underline{\qquad 2} \\ a_{-3} = 1 & 0.625000 \\ & \underline{\qquad 2} \\ a_{-4} = 1 & 0.250000 \\ & \underline{\qquad 2} \\ a_{-5} = 0 & 0.500000 \\ & \underline{\qquad 2} \\ a_{-6} = 1 & 0.000000 \end{array}$$

Thus the binary equivalent of 0.203125_{10} is 0.001101_2. The multiplication by 2 is continued until the decimal number is exhausted (as in the example) or the desired accuracy is achieved. Accuracy suffers considerably if the conversion process is stopped too soon. For example, if we stop after the fourth step, then we are assuming 0.0011 is approximately equal to 0.20315, whereas it is actually equal to 0.1875, an error of about 7.7%.

1.3.1 Basic Binary Arithmetic

Arithmetic operations using binary numbers are far simpler than the corresponding operations using decimal numbers due to the very elementary rules of addition and multiplication. The rules of binary addition are

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \ \text{(carry 1)} \end{aligned}$$

As in decimal addition, the least significant bits of the addend and the augend are added first. The result is the sum, possibly including a carry. The carry bit is added to the sum of the digits of the next column. The process continues until the bits of the most significant column are summed.

Example 1.6 Let us consider the addition of the decimal numbers 27 and 28 in binary.

Decimal	Binary	
27	11011	Addend
$+28$	$+11100$	Augend
55	110111	Sum
	11000	Carry

To verify that the sum is correct, we convert 110111 to decimal:

$$\begin{aligned} &1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 32 + 16 + 0 + 4 + 2 + 1 \\ &= 55 \end{aligned}$$

Example 1.7 Let us add -11 to -19 in binary. Since the addend and the augend are negative, the sum will be negative.

Decimal	Binary	
19	10011	
11	01011	
30	11110	Sum
	00011	Carry

In all digital systems, the circuitry used for performing binary addition handles two numbers at a time. When more than two numbers have to be added, the first two are added, then the resulting sum is added to the third number, and so on.

Binary subtraction is carried out by following the same method as in the decimal system. Each digit in the *subtrahend* is deducted from the corresponding digit in the *minuend* to obtain the difference. When the minuend digit is less than the subtrahend digit, then the radix number (i.e., 2) is added to the minuend, and a *borrow* 1 is added to the next subtrahend digit. The rules applied to the binary subtraction are

$$\begin{aligned} &0 - 0 = 0 \\ &0 - 1 = 1 \quad \text{(borrow 1)} \\ &1 - 0 = 1 \\ &1 - 1 = 0 \end{aligned}$$

Example 1.8 Let us consider the subtraction of 21_{10} from 27_{10} in binary:

Decimal	Binary	
27	11011	Minuend
-21	-10101	Subtrahend
6	00110	Difference
	00100	Borrow

It can easily be verified that the difference 00110_2 corresponds to decimal 6.

Example 1.9 Let us subtract 22_{10} from 17_{10}. In this case, the subtrahend is greater than the minuend. Therefore the result will be negative.

Decimal	Binary	
17	10001	
-22	-10110	
-5	-00101	Difference
	00001	Borrow

Binary multiplication is performed in the same way as decimal multiplication, by multiplying, then shifting one place to the left, and finally adding the partial products. Since the multiplier can only be 0 or 1, the partial product is either zero or equal to the multiplicand. The rules of multiplication are

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

Example 1.10 Let us consider the multiplication of the decimal numbers 67 by 13 in binary:

Decimal	Binary	
67	1000011	Multiplicand
$\times$ 13	1101	Multiplier
871	1000011	First partial product
	0000000	Second partial product
	1000011	Third partial product
	1000011	Fourth partial product
	1101100111	Final product

Example 1.11 Let us multiply 13.5 by 3.25.

Decimal	Binary	
13.5	1101.10	Multiplicand
$\times$ 3.25	11.01	Multiplier
43.875	110110	First partial product
	000000	Second partial product
	110110	Third partial product
	110110	Fourth partial product
	101011.1110	Final product

The decimal equivalent of the final product is $43 + 0.50 + 0.25 + 0.125 = 43.875$.

The process of binary division is very similar to standard decimal division. However, division is simpler in binary because when one checks to see how many times the divisor fits into the dividend, there are only two possibilities, 0 or 1.

Example 1.12 Let us consider the division of 101110 (46_{10}) by 111 (7_{10})

```
                 0001     Quotient
Divisor 111 ⌈ 101110     Dividend
              0111
              ----
               100
```

Since the divisor, 111, is greater than the first three bits of the dividend, the first three quotient bits are 0. The divisor is less than the first four bits of the dividend; therefore

the division is possible, and the fourth quotient bit is 1. The difference is less than the divisor, so we bring down the net bit of the dividend:

$$\begin{array}{rr} & 00011 \\ 111 & \overline{)\,101110} \\ & \underline{0111} \\ & 1001 \\ & \underline{111} \\ & 10 \end{array}$$

The difference is less than the divisor, so the next bit of the dividend is brought down:

$$\begin{array}{rr} & 000110 \\ 111 & \overline{)\,101110} \\ & \underline{0111} \\ & 1001 \\ & \underline{111} \\ & 100 \end{array} \quad \text{Remainder}$$

In this case the dividend is less than the divisor; hence the next quotient bit is 0 and the division is complete. The decimal conversion yields $46/7 = 6$ with remainder 4, which is correct.

The methods we discussed to perform addition, subtraction, multiplication, and division are equivalents of the same operations in decimal. In digital systems, all arithmetic operations are carried out in modified forms; in fact, they use only addition as their basic operation.

1.4 OCTAL NUMBERS

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, *octal* and *hexadecimal*, are used for representing large binary numbers. The octal number system uses a base or radix of 8; thus it has digits from 0 to $r - 1$, or $8 - 1$, or 7. As in the decimal and binary systems, the positional value of each digit in a sequence of numbers is definitely fixed. Each position in an octal number is a power of 8, and each position is 8 times more significant than the previous position. The number 375_8 in the octal system therefore means

$$\begin{aligned} 3 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 &= 192 + 56 + 5 \\ &= 253_{10} \end{aligned}$$

Example 1.13 Let us determine the decimal equivalent of the octal number 14.3.

$$\begin{aligned} 14.3_g &= 1 \times 8^1 + 4 \times 8^0 + 3 \times 8^{-1} \\ &= 8 + 4 + 0.375 \\ &= 12.375 \end{aligned}$$

The method for converting a decimal number to an octal number is similar to that used for converting a decimal number to binary (Section 1.2), except that the decimal number is successively divided by 8 rather than 2.

Example 1.14 Let us determine the octal equivalent of the decimal number 278.

$$\frac{278}{8} = 34, \text{ remainder } 6$$
$$\frac{34}{8} = 4, \text{ remainder } 2$$
$$\frac{4}{8} = 0, \text{ remainder } 4$$

Thus $278_{10} = 426_8$.

Decimal fractions can be converted to octal by progressively multiplying by 8; the integral part of each product is retained as the octal fraction. For example, 0.651_{10} is converted to octal as follows:

$$\begin{array}{rl} & 0.651 \\ & \underline{\quad\quad 8} \\ 5 & 0.208 \\ & \underline{\quad\quad 8} \\ 1 & 0.664 \\ & \underline{\quad\quad 8} \\ 5 & 0.312 \\ & \underline{\quad\quad 8} \\ 2 & 0.496 \\ & \underline{\quad\quad 8} \\ 3 & 0.968 \\ & \text{etc.} \end{array}$$

According to Eq. (1.2), $a_{-1} = 5$, $a_{-2} = 1$, $a_{-3} = 5$, $a_{-4} = 2$, and $a_{-5} = 3$; hence $0.651_{10} = 0.51523_8$. More octal digits will result in more accuracy.

A useful relationship exists between binary and octal numbers. The number of bits required to represent an octal digit is three. For example, octal 7 can be represented by binary 111. Thus, if each octal digit is written as a group of three bits, the octal number is converted into a binary number.

Example 1.15 The octal number 324_8 can be converted to a binary number as follows:

$$\begin{array}{ccc} 3 & 2 & 4 \\ \downarrow & \downarrow & \downarrow \\ 011 & 010 & 100 \end{array}$$

Hence $324_8 = 11010100_2$; the most significant 0 is dropped because it is meaningless, just as 0123_{10} is the same as 123_{10}.

The conversion from binary to octal is also straightforward. The binary number is partitioned into groups of three starting with the least significant digit. Each group of three binary digits is then replaced by an appropriate decimal digit between 0 and 7 (Table 1.1).

TABLE 1.1 Binary to Octal Conversion

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Example 1.16 Let us convert 110011101001_2 to octal:

$$\underbrace{110}_{6}\ \underbrace{011}_{3}\ \underbrace{101}_{5}\ \underbrace{001}_{1}$$

The octal representation of the binary number is 6351_8. If the leftmost group of a partitioned binary number does not have three digits, it is padded on the left with 0's. For example, 1101010 would be divided as

$$\underbrace{001}_{1}\ \underbrace{101}_{5}\ \underbrace{010}_{2}$$

The octal equivalent of the binary number is 152_8. In case of a binary fraction, if the bits cannot be grouped into 3-bit segments, the 0's are added on the right to complete groups of three. Thus 110111.1011 can be written

$$\underbrace{110}_{6}\ \underbrace{111}_{7}\ .\ \underbrace{101}_{5}\ \underbrace{100}_{4}$$

As shown in the previous section, the binary equivalent of a decimal number can be obtained by successively dividing the number by 2 and using the remainders as the answer, the first remainder being the lowest significant bit, and so on. A large number of divisions by 2 are required to convert from decimal to binary if the decimal number is large. It is often more convenient to convert from decimal to octal and then replace each digit in octal in terms of three digits in binary. For example, let us convert 523_{10} to binary by going through octal.

$$\frac{523}{8} = 65, \text{ remainder } 3$$

$$\frac{65}{8} = 8, \text{ remainder } 1$$

$$\frac{8}{8} = 1, \text{ remainder } 0$$

$$\frac{1}{8} = 0, \text{ remainder } 1$$

Thus

$$\begin{aligned}(523)_{10} &= (1 \quad 0 \quad 1 \quad 3)_8 \\ &= (001 \quad 000 \quad 001 \quad 011)_2\end{aligned}$$

It can be verified that the decimal equivalent of 001000001011_2 is 523_{10}:

$$\begin{aligned}1 \times 2^9 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 &= 512 + 8 + 2 + 1 \\ &= 523_{10}\end{aligned}$$

Addition and subtraction operations using octal numbers are very much similar to that use in decimal systems. In octal addition, a carry is generated when the sum exceeds 7_{10}. For example,

$$\begin{array}{r} 153_8 \\ +\,327_8 \\ \hline 502_8 \end{array}$$

$3 + 7 = 10_{10} = 2 + 1$ carry ⟵ first column
$5 + 2 + 1$ carry $= 0 + 1$ carry ⟵ second column
$1 + 3 + 1$ carry $= 5$ ⟵ third column

In octal subtraction, a borrow requires that 8_{10} be added to the minuend digit and a 1_{10} be added to the left adjacent subtrahend digit.

$$\begin{array}{r} 670_8 \\ -\,125_8 \\ \hline 543_8 \end{array}$$

$0 - 5 = (8 - 5 + 1\,\text{borrow})_{10} = 3 + 1$ borrow ⟵ first column
$7 - (2 + 1\,\text{borrow}) = 7 - 3 = 4$ ⟵ second column
$6 - 1 = 5$ ⟵ third column

1.5 HEXADECIMAL NUMBERS

The hexadecimal numbering system has a base 16; that is, there are 16 symbols. The decimal digits 0 to 9 are used as the first ten digits as in the decimal system, followed by the letters A, B, C, D, E, and F, which represent the values 10, 11, 12, 13, 14, and 15, respectively. Table 1.2 shows the relationship between decimal, binary, octal, and hexadecimal number systems. The conversion of a binary number to a hexadecimal number consists of partitioning the binary numbers into groups of 4 bits, and representing each group with its hexadecimal equivalent.

TABLE 1.2 Number Equivalents

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Example 1.17 The binary number 1010011011110001 is grouped as

$$1010 \quad 0110 \quad 1111 \quad 0001$$

which is shown here in hexadecimal:

$$\text{A6F1}_{\text{H}}$$

The conversion from hexadecimal to binary is straightforward. Each hexadecimal digit is replaced by the corresponding 4-bit equivalent from Table 1.2. For example, the binary equivalent of $4AC2_H$ is

$$\begin{array}{cccc} 4 & A & C & 2 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 0100 & 1010 & 1110 & 0010 \end{array}$$

Thus $4AC2_H = 0100101011100010_2$.

Sometimes it is necessary to convert a hexadecimal number to decimal. Each position in a hexadecimal number is 16 times more significant than the previous position. Thus the decimal equivalent for $1A2D_H$ is

$$\begin{aligned} &1 \times 16^3 + \text{A} \times 16^2 + 2 \times 16^1 + \text{D} \times 16^0 \\ &= 1 \times 16^3 + 10 \times 16^2 + 2 \times 16^1 + 13 \times 16^0 \\ &= 6701 \end{aligned}$$

Hexadecimal numbers are often used in describing the data in a computer memory. A computer memory stores a large number of words, each of which is a standard size collection

of bits. An 8-bit word is known as a *byte*. A hexadecimal digit may be considered as half of a byte. Two hexadecimal digits constitute one byte, the rightmost 4 bits corresponding to half a byte, and the leftmost 4 bits corresponding to the other half of the byte. Often a half-byte is called a *nibble*.

Hexadecimal addition and subtraction are performed as for any other positional number system.

Example 1.18 Let us find the sum of 688_H and 679_H.

$$\begin{array}{r} 688_H \\ 679_H \\ \hline D01_H \end{array}$$

$$\begin{aligned} 8 + 9 &= 17_{10} = 1 + 1 \text{ carry} \longleftarrow \text{first column} \\ 8 + 7 + 1 \text{ carry} &= 16_{10} = 0 + 1 \text{ carry} \longleftarrow \text{second column} \\ 6 + 6 + 1 \text{ carry} &= 13_{10} = D \longleftarrow \text{third column} \end{aligned}$$

Hexadecimal subtraction requires the same need to carry digits from left to right as in octal and decimal.

Example 1.19 Let us compute $2A5_H - 11B_H$ as shown:

$$\begin{array}{r} 2A5_H \\ 11B_H \\ \hline 18A_H \end{array}$$

$$\begin{aligned} 5 - B = (21 - 11 + 1 \text{ borrow})_{10} &= 10 + 1 \text{ borrow} \\ &= A + 1 \text{ borrow} \longleftarrow \text{first column} \\ A - (1 + 1 \text{ borrow}) = (10 - 2)_{10} &= 8 \longleftarrow \text{second column} \\ 2 - 1 &= 1 \longleftarrow \text{third column} \end{aligned}$$

1.6 SIGNED NUMBERS

So far, the number representations we considered have not carried sign information. Such unsigned numbers have a magnitude significance only. Normally a prefix + or − may be placed to the left of the magnitude to indicate whether a number is positive or negative. This type of representation, known as *sign-magnitude representation*, is used in the decimal system. In binary systems, an additional bit known as sign bit, is added to the left of the most significant bit to define the sign of a number. A 1 is used to represent − and a 0 to represent +. Table 1.3 shown 3-bit numbers in terms of signed and unsigned equivalents. Notice that there are two representations of the number 0, namely, +0 and −0. The range of integers that can be expressed in a group of three bits is from $-(2^2 - 1) = -3$ to $+(2^2 - 1) = +3$, with one bit being reserved to denote the sign.

TABLE 1.3 Signed and Unsigned Binary Numbers

	Decimal Equivalent	
Binary	Signed	Unsigned
000	+0	0
001	+1	1
010	+2	2
011	+3	3
100	−0	4
101	−1	5
110	−2	6
111	−3	7

Although the sign-magnitude representation is convenient for computing the negative of a number, a problem occurs when two numbers of opposite signs have to be added. To illustrate, let us find the sum of $+2_{10}$ and -6_{10}.

$$+2_{10} = 0010$$
$$-6_{10} = \underline{1110}$$
$$10000$$

The addition produced a sum that has 5 bits, exceeding the capability of the number system (sign +3 bits); this results in an *overflow*. Also, the sum is wrong; it should be −4 (i.e., 1100) instead of 0.

An alternative representation of negative numbers, known as the *complement* form, simplifies the computations involving signed numbers. The complement representation enjoys the advantage that no sign computation is necessary. There are two types of complement representations: diminished radix complement and radix complement.

1.6.1 Diminished Radix Complement

In the decimal system ($r = 10$) the complement of a number is determined by subtracting the number from $(r - 1)$, that is, 9. Hence the process is called finding the 9's complement. For example,

9's complement of 5 $(9 - 5) = 4$
9's complement of 63 $(99 - 63) = 36$
9's complement of 110 $(999 - 110) = 889$

In binary notation ($r = 2$), the diminished radix complement is known as the *1's complement*. A positive number in 1's complement is represented in the same way as in sign-magnitude representation. The 1's complement representation of a negative number x is derived by subtracting the binary value of x from the binary representation of $(2^n - 1)$, where n is the number of bits in the binary value of x.

Example 1.20 Let us compute the 1's complement form of -43. The binary value of $43 = 00101011$. Since $n = 8$, the binary representation of $2^8 - 1$ is

$$\begin{array}{r} 2^8 = 100000000 \\ -1 \\ \hline 2^8 - 1 = \quad 11111111 \end{array}$$

Hence the 1's complement form of -43 is

$$\begin{array}{r} 2^8 - 1 = \quad 11111111 \\ -43 = -00101011 \\ \hline 11010100 \end{array}$$

Notice that the 1's complement form of -43 is a number that has a 0 in every position that $+43$ has a 1, and vice versa. Thus the 1's complement of any binary number can be obtained by complementing each bit in the number.

A 0 in the most significant bit indicates a positive number. The sign bit is not complemented when negative numbers are represented in 1's complement form. For example, the 1's complement form of -25 will be represented as follows:

$$\begin{aligned} -25 &= 111001 \quad \text{(sign-magnitude form)} \\ &= 100110 \quad \text{(1's complement form)} \end{aligned}$$

Table 1.4 shows the comparison of 3-bit unsigned, signed, and 1's complement values. The advantage of using 1's complement numbers is that they permit us to perform subtraction by actually using the addition operation. It means that, in digital systems, addition and subtraction can be carried out by using the same circuitry.

The addition operation for two 1's complement numbers consists of the following steps:

(i) Add the two numbers including the sign bits.
(ii) If a carry bit is produced by the leftmost bits (i.e., the sign bits), add it to the result. This is called *end-around carry*.

TABLE 1.4 Comparison of 3-Bit Signed, Unsigned, and 1's Complement Values

	Decimal Equivalent		
Binary	Unsigned	Sign-Magnitude	1's Complement
000	0	+0	+0
001	1	+1	+1
010	2	+2	+2
011	3	+3	+3
100	4	−0	−3
101	5	−1	−2
110	6	−2	−1
111	7	−3	−0

Example 1.21 Let us add −7 to −5:

1's complement form of –7 = 11000
1's complement form of –5 = 11010
Sum 110010
Carry → 1
1's complement sum 10011

The result is a negative number. By complementing the magnitude bits we get

11100, that is, –12

Thus the sum of −7 and −5 is −12, which is correct.

The subtraction operation for two 1's complement numbers can be carried out as follows:

(i) Derive the 1's complement of the subtrahend and add it to the minuend.
(ii) If a carry bit is produced by the leftmost bits (i.e., the sign bits), add it to the result.

Example 1.22 Let us subtract +21 from +35:

Minuend = +35 = 0100011
Subtrahend = +21 = 1101010 (in 1's complement form)
Sum = 10001101
Carry → 1
1's complement sum = 0001110

The sign is correct, and the decimal equivalent +14 is also correct.

The 1's complement number system has the advantage that addition and subtraction are actually one operation. Unfortunately, there is still a dual representation for 0. With 3-bit numbers, for example, 000 is *positive zero* and 111 is *negative zero*.

1.6.2 Radix Complement

In the decimal system, the radix complement is the 10's complement. The 10's complement of a number is the difference between 10 and the number. For example,

10's complement of 5, $10 - 5 = 5$
10's complement of 27, $100 - 27 = 73$
10's complement of 48, $100 - 48 = 52$

In binary number system, the radix complement is called the 2's complement. The *2's*

complement representation of a positive number is the same as in sign-magnitude form. The 2's complement representation of a negative number is obtained by complementing the sign-magnitude representation of the corresponding positive number and adding a 1 to the least significant position. In other words, the 2's complement form of a negative number can be obtained by adding 1 to the 1's complement representation of the number.

Example 1.23 Let us compute the 2's complement representation of −43:

$$\begin{aligned} +43 &= 0101011 && \text{(sign-magnitude form)} \\ &= 1010100 && \text{(1's complement form)} \\ &\quad\ +1 && \text{(add 1)} \\ &\overline{\quad 1010101} && \text{2's complement form} \end{aligned}$$

Table 1.5 shows the comparisons of four representations of 3-bit binary numbers. The bit positions in a 2's complement number have the same weight as in a conventional binary number except that the weight of the sign bit is negative. For example, the 2's complement number 1000011 can be converted to decimal in the same manner as a binary number:

$$\begin{aligned} -2^6 + 2^1 + 2^0 &= -64 + 2 + 1 \\ &= -61 \end{aligned}$$

A distinct advantage of 2's complement form is that, unlike 1's complement form, there is a unique representation of 0 as can be seen in Table 1.5. Moreover, addition and subtraction can be treated as the same operation as in 1's complement; however, the carry bit can be ignored and the result is always in correct 2's complement notation. Thus addition and subtraction are easier in 2's complement than in 1's complement.

An *overflow* occurs when two 2's complement numbers are added, if the carry-in bit into the sign bit is different from the carry-out bit from the sign bit. For example, the

TABLE 1.5 Various Representations of 3-Bit Binary Numbers

	Decimal Equivalent			
Binary	Unsigned	Signed	1's Complement	2's Complement
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	−0	−3	−4
101	5	−1	−2	−3
110	6	−2	−1	−2
111	7	−3	−0	−1

following addition will result in an overflow:

```
                0 ◄— Carry-in
  101010        (−22)
  101001        (−23)
 1010011
Carry-out ——┘
```

Hence the result is invalid.

Example 1.24 Let us derive the following using 2's complement representation:

```
(i)  +13    (ii)  −15    (iii)  −9    (iv)  −5
     −7           −6           +6          −1
```

```
                1 ◄— Carry-in
(i)  +13        01101
     −7         11001
     +6        100110
Carry-out ——┘
```

Since the carry-in is equal to the carry-out, there is no overflow; the carry-out bit can be ignored. The sign bit is positive. Thus the result is +6.

```
                    0 ◄— Carry-in
(ii)   −15          10001
       −6           11010
                   101011
Carry-out ——┘
```

The carry-out bit is not equal to the carry-in bit. Thus there is an overflow and the result is invalid.

```
                    0 ◄— Carry-in
(iii)  −9           10111
       +6           00110
       −3          011101
Carry-out ——┘
```

There is no overflow, and the sign bit is negative. The decimal equivalent of the result is $-2^4 + 2^3 + 2^2 + 2^0 = -3$.

```
                    1 ◄— Carry-in
(iv)   −5           1011
       −1           1111
       −6          11010
Carry-out ——┘
```

There is no overflow, and the sign bit is negative. The result, as expected, is $-2^3 + 2^1 = -6$.

An important advantage of 2's complement numbers is that they can be *sign-extended* without changing their values. For example, if the 2's complement number 101101 is

shifted right (i.e., the number becomes 1101101), the decimal value of the original and the shifted number remains the same (−19 in this case).

1.7 FLOATING-POINT NUMBERS

Thus far, we have been dealing mainly with fixed-point numbers in out discussion. The word *fixed* refers to the fact that the radix point is placed at a fixed place in each number, usually either to the left of the most significant digit or to the right of the least significant digit. With such a representation, the number is always either a fraction or an integer. The main difficulty of fixed-point arithmetic is that the range of numbers that could be represented is limited. Figure 1.1 illustrates the fixed-point representation of a signed four-digit decimal number; the range of numbers that can be represented using this configuration is 9999. In order to satisfy this limited range, scaling has to be used.

For example, to add +50.73 to +40.24 we have to multiply both numbers by 100 before addition and then adjust the sum, keeping in mind that there should be a decimal point two places from the right. The scale factor in this example is 100.

An alternative representation of numbers, known as the *floating-point* format, may be employed to eliminate the scaling factor problem. In a floating-point number, the radix point is not placed at a fixed place; instead, it "floats" to various places in a number so that more digits can be assigned to the left or to the right of the point. More digits on the left of the radix point permit the representation of larger numbers, whereas more digits on the right of the radix point result in more digits for the fraction. Numbers in floating point format consist of two parts—a fraction and an exponent; they can be expressed in the form

$$\text{fraction} \times \text{radix}^{\text{exponent}}$$

The fraction is often referred to as the *mantissa* and can be represented in sign-magnitude, diminished radix complement, or radix complement form. For example, the decimal number 236,000 can be written 0.236×10^6. In a similar manner, very small numbers may be represented using negative exponents. For example, 0.00000012 may be written 0.12×10^{-6}. By adjusting the magnitude of the exponent, the range of numbers covered can be considerably enlarged. Leading 0's in a floating-point number may be removed by shifting the mantissa to the left and decreasing the exponent accordingly; this process is known as *normalization* and floating-point numbers without leading 0's are called normalized. For example, the normalized of floating-point number 0.00312×10^5 is 0.312×10^3. Similarly, a binary fraction such as 0.001×2^4 would be normalized to 0.1×2^2.

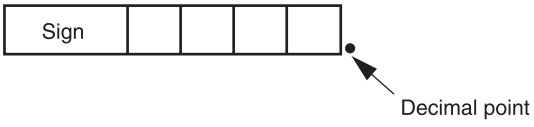


FIGURE 1.1 Fixed-point number representation.

1.8 BINARY ENCODING

In Section 1.1 it was shown how decimal numbers can be represented by equivalent binary numbers. Since there are ten decimal numbers (0, 1, . . . , 9), the minimum number of bits required to represent a decimal number is four, giving 16 ($=2^4$) possible combinations, of which only ten are used. Binary codes are divided into two groups—*weighted* and *nonweighted*.

1.8.1 Weighted Codes

In a weighted code each binary digit is assigned a weight w; the sum of the weights of the I bits is equal to the decimal number represented by the four-bit combination. In other words, if d_j ($i = 0, \ldots, 3$) are the digit values and w_j ($i = 0, \ldots, 3$) are the corresponding weights, then the decimal equivalent of a 4-bit binary number is given by

$$d_3w_3 + d_2w_2 + d_1w_1 + d_0w_0$$

If the weight assigned to each binary digit is exactly the same as that associated with each digit of a binary number (i.e., $w_0 = 2^\circ = 1$, $w_1 = 2^1 = 2$, $w_2 = 2^2 = 4$, and $w_3 = 2^3 = 8$), then the code is called the *BCD (binary-coded decimal)* code. The BCD code differs from the conventional binary number representation in that, in the BCD code, each decimal digit is binary coded. For example, the decimal number 15 in conventional binary number representation is

1111

whereas in the BCD code, 15 is represented by

```
0001 0101
'-'-'  '-'-'
  1      5
```

the decimal digits 1 and 5 each being binary coded.

Several forms of weighted codes are possible, since the codes depend on the weight assigned to the binary digits. Table 1.6 shows the decimal digits and their weighted

TABLE 1.6 Weighted Binary Codes

Decimal Number	8421	7421	4221	8421
0	0000	0000	0000	0000
1	0001	0001	0001	0111
2	0010	0010	0010	0110
3	0011	0011	0011	0101
4	0100	0100	1000	0100
5	0101	0101	0111	1011
6	0110	0110	1100	1010
7	0111	1000	1101	1001
8	1000	1001	1110	1000
9	1001	1010	1111	1111

code equivalents. The 7421 code has fourteen 1's in its representation, which is the minimum number of 1's possible. However, if we represent decimal 7 by 0111 instead of 1000, the 7421 code will have sixteen 1' instead of fourteen. In the 4221 code, the sum of the weights is exactly $9(=4+2+2+1)$. Codes whose weights add up to 9 have the property that the 9's complement of a number (i.e., $9-N$, where N is the number) represented in the code can be obtained simply by taking the 1's complement of its coded representation. For example, in the 4221 code shown in Table 1.6, the decimal number 7 is equivalent to the code word 1101; the 9's complement of 7 is 2 $(=9-7)$, and the corresponding code word is 0010, which is the 1' complement of 1101. Codes having this property are known as *self-complementing* codes. Similarly, the 8421 is also a self-complementing code.

Among the weighted codes, the BCD code is by far the most widely used. It is useful in applications where output information has to be displayed in decimal. The addition process in BCD is the same as in simple binary as long as the sum is decimal 9 or less. For example,

Decimal	BCD
6	0110
+3	+0011
9	1001

However, if the sum exceeds decimal 9, the result has to be adjusted by adding decimal 6 (0110) to it. For example, let us add 5 to 7:

Decimal	BCD	
7	0111	
+5	+0101	
12	1100	12 (not a legal BCD number)
	+0110	Add 6
	0001 0010	
	1 2	

As another example, let us add 9 to 7:

Decimal		BCD
9		1001
+7		+0111
16	0001	0000
	1	0

Although the result consists of two valid BCD numbers, the sum is incorrect. It has to be corrected by adding 6 (0110). This is required when there is a carry from the most significant bit of a BCD number to the next higher BCD number. Thus the correct result is

	0001	0000
+		0110
	0001	0110
	1	6

Other arithmetic operations in BCD can also be performed.

TABLE 1.7 Excess-3 Code

Decimal	Excess-3
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

1.8.2 Nonweighted Codes

In the nonweighted codes there are no specific weights associated with the digits, as was the case with weighted codes. A typical nonweighted code is the *excess-3* code. It is generated by adding 3 to a decimal number and then converting the result to a 4-bit binary number. For example, to encode the decimal number 6 to excess-3 code, we first add 3 to 6. The resulting sum, 9, is then represented in binary (i.e., 1001). Table 1.7 lists the excess-3 code representations of the decimal digits.

Excess-3 code is a self-complementing code and is useful in arithmetic operations. For example, consider the addition of two decimal digits whose sum will result in a number greater than 9. If we use BCD code, no carry bit will be generated. On the other hand, if excess-3 code is used, there will be a natural carry to the next higher digit; however, the sum has to be adjusted by adding 3 to it. For example, let us add 6 to 7:

Decimal	Excess-3	
7	1010	
+6	1001	
13	10011	
Carry ↗	00110011	Add 3 to both sum
	0100 0110	and carry bit
	1 3	

In excess-3 code, if we add two decimal digits whose sum is 9 or less, then the sum should be adjusted by subtracting 3 from it. For example,

Decimal	Excess-3	
6	1001	
+2	+0101	
8	1110	
	0011	Subtract 3
	1011	
	8	

For subtraction in excess-3 code, the difference should be adjusted by adding 3 to it. For example,

Decimal	Excess-3		
17	0100	1010	
−11	0100	0100	
6	0000	0110	
		0011	Add 3
		1001	
		6	

Another code that uses four unweighted binary digits to represent decimal numbers is the *cyclic code*. Cyclic codes have the unique feature that the successive codewords differ only in one bit position. Table 1.8 shows an example of such a code.

One type of cyclic code is the *reflected code*, also known as the *Gray code*. A 4-bit Gray code is shown in Table 1.9. Notice that in Table 1.9, except for the most significant bit

TABLE 1.8 Cyclic Code

Decimal	Cyclic
0	0000
1	0001
2	0011
3	0010
4	0110
5	0100
6	1100
7	1110
8	1010
9	1000

TABLE 1.9 Gray Code

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

position, all columns are "reflected" about the midpoint; in the most significant bit position, the top half is all 0's and the bottom half all 1's.

A decimal number can be converted to Gray code by first converting it to binary. The binary number is converted to the Gray code by performing a modulo-2 sum of each digit (starting with the least significant digit) with its adjacent digit. For example, if the binary representation of a decimal number is

$$b_3 \quad b_2 \quad b_1 \quad b_0$$

then the corresponding Gray code word, $G_3G_2G_1G_0$, is

$$\begin{aligned} G_3 &= b_3 \\ G_2 &= b_3 \oplus b_2 \\ G_1 &= b_2 \oplus b_1 \\ G_0 &= b_1 \oplus b_0 \end{aligned}$$

where $\oplus$ indicates exclusive-OR operation (i.e., modulo-2 addition), according to the following rules:

$$\begin{aligned} 0 \oplus 0 &= 0 \\ 0 \oplus 1 &= 1 \\ 1 \oplus 0 &= 1 \\ 1 \oplus 1 &= 0 \end{aligned}$$

As an example let us convert decimal 14 to Gray code.

Decimal	Binary			
	b_3	b_2	b_l	b_0
14	1	1	1	0

Therefore

$$\begin{aligned} G_3 &= b_3 &&= 1 \\ G_2 &= b_3 \oplus b_2 &&= 1 \oplus 1 = 0 \\ G_1 &= b_2 \oplus b_1 &&= 1 \oplus 1 = 0 \\ G_0 &= b_1 \oplus b_0 &&= 1 \oplus 0 = 1 \end{aligned}$$

Thus the Gray code word for decimal 14 is

G_3	G_2	G_1	G_0
1	0	0	1

The conversion of a Gray code word to its decimal equivalent is done by following this sequence in reverse. In other words, the Gray code word is converted to binary and

then the resulting binary number is converted to decimal. To illustrate, let us convert 1110 from Gray to decimal:

$$
\begin{aligned}
&G_3 \quad G_2 \quad G_1 \quad G_0 \\
&1 \quad\ 1 \quad\ 1 \quad 0 \\
&b_3 = G_3 = 1 \\
&G_2 = b_3 \oplus b_2 \\
&\therefore\ b_2 = G_2 \oplus b_3 \quad (\text{since } G_2 \oplus b_3 = b_3 \oplus b_2 \oplus b_3) \\
&\qquad = 1 \oplus 1 \quad (\text{since } G_2 = 1 \text{ and } b_3 = 1) \\
&\qquad = 0 \\
&G_1 = b_2 \oplus b_1 \\
&\therefore\ b_1 = G_1 \oplus b_2 \\
&\qquad = 1 \oplus 0 \\
&\qquad = 1 \\
&G_0 = b_1 \oplus b_0 \\
&\therefore\ b_0 = 1
\end{aligned}
$$

Thus the binary equivalent of the Gray code word 1110 is 1011, which is equal to decimal 11. The first ten codewords of the Gray code shown in Table 1.9 can be utilized as reflected BCD code if desired. The middle ten codewords can be used as reflected excess-3 code.

EXERCISES

1. Convert the following decimal numbers to binary:
   a. 623
   b. 73.17
   c. 53.45
   d. 2.575

2. Convert the following binary numbers to decimal:
   a. 10110110
   b. 110000101
   c. 100.1101
   d. 1.001101

3. Convert the following binary numbers to hexadecimal and octal:
   a. 100101010011
   b. 001011101111
   c. 1011.111010101101
   d. 1111.100000011110

4. Convert the following octal numbers to binary and hexadecimal.
   a. 1026

b. 7456
c. 5566
d. 236.2345

5. Convert the following hexadecimal numbers to binary and octal:
a. EF69
b. 98AB5
c. DAC.IBA
d. FF.EE

6. Perform the addition of the following binary numbers:

a. 100011 + 1101
b. 10110110 + 11100011
c. 10110011 + 1101010

7. Perform the following subtractions, where each of the numbers is in binary form:

a. 101101 − 111110
b. 1010001 − 1001111
c. 10000110 − 1110001

8. Add the following pairs of numbers, where each number is in hexadecimal form:

a. ABCD + 75EF
b. 129A + AB22
c. EF23 + C89

9. Repeat Exercise 8 using subtraction instead of addition.

10. Add the following pairs of numbers, where each number is in octal form:

a. 7521 + 4370
b. 62354 + 3256
c. 3567 + 2750

11. Repeat Exercise 10 using subtraction instead of addition.

12. Derive the 6-bit *sign-magnitude*, *1's complement*, *and 2's complement* of the following decimal numbers:
a. $+22$
b. -31
c. $+17$
d. -1

13. Find the sum of the following pairs of decimal numbers assuming 8-bit 1's complement representation of the numbers:
a. $+61 + (-23)$
b. $-56 + (-55)$
c. $+28 + (+27)$
d. $-48 + (+35)$

14. Repeat Exercise 13 assuming 2's complement representation of the decimal numbers.

15. Assume that X is the *2's complement* of an n-bit binary number Y. Prove that the 2's complement of X is Y.

16. Find the floating-point representation of the following numbers:

a. $(326.245)_{10}$

b. $(101100.100110)_2$

c. $(-64.462)_8$

17. Normalize the following floating-point numbers:

a. 0.000612×10^6

b. 0.0000101×2^4

18. Encode each of the ten decimal digits using the weighted binary codes.

a. 4, 4, 1, −2

b. 7, 4, 2, −1

19. Given the following *weighted* codes determine whether any of these is *self-complementing*.

a. (8, 4, −3, −2)

b. (7, 5, 3, −6)

c. (6, 2, 2, 1)

20. Represent the decimal numbers 535 and 637 in

a. BCD code

b. Excess-3 code

Add the resulting numbers so that the sum in each case is appropriately encoded.

21. Subtract 423 from 721, assuming the numbers are represented in *excess-3* code.

22. Determine two forms for a cyclic code other than the one shown in Table 1.8.

23. Assign a binary code, using the minimum number of bits, to encode the 26 letters in the alphabet.

2 Fundamental Concepts of Digital Logic

2.1 INTRODUCTION

The objective of this chapter is to familiarize readers with the basic concepts of set theory, relations, graphs, and Boolean algebra. These are necessary to understand material presented in later chapters. Boolean algebra is used for the analysis and design of electrical switching circuits (popularly known as digital logic circuits). We are mainly interested in the application of Boolean algebra in constructing *digital circuits* using elements called *gates*.

2.2 SETS

A *set* is a collection of objects. The objects comprising a set must have at least one property in common—for example, all male students in a class, all integers less than 15 but greater than 5, or inputs to a logic circuit. A set that has a finite number of elements is described by listing the elements of the set between brackets. Thus the set of all positive integers less than 15 but greater than 5 can be written

$$\{6, 7, 8, 9, 10, 11, 12, 13, 14\}$$

The order in which the elements of a set are listed is not important. This set can also be represented as follows:

$$\{9, 10, 6, 7, 14, 13, 8, 11, 12\}$$

In general, uppercase letters are used to represent a set and lowercase letters are used to represent the numbers (or elements) of a set.

If a is an element of a set S, we represent that fact by writing

$$a \in S$$

On the other hand, if a is not an element of S, we can indicate that by writing

$$a \notin S$$

For example, if $S = \{5, 7, 8, 9\}$, then $5 \in S$, $7 \in S$, $8 \in S$, and $9 \in S$, but $6 \notin S$.

Principles of Modern Digital Design, by Parag K. Lala

A set may have only one element; it is then called a *singleton*. For example, {6} is the set with 6 as its only element. There is also a set that contains no element; it is known as the *empty* or *null set* and is denoted by $\varnothing$.

A set may also be defined in terms of some property that all members of the set are required to have. Thus the set $A = \{2, 4, 6, 8, 10\}$ may be defined as

$$A = \{x | x \text{ is an even positive integer not greater than } 10\}$$

In general, a set of objects that share common properties may be represented as

$$\{x | x \text{ possesses certain properties}\}$$

Sets with an infinite number of members are specified in this way, for example,

$$B = \{x | x \text{ is an even number}\}$$

A set P is a subset of a set Q if every element of P is also an element in Q. This may also be defined as P is included in Q and is represented with a new symbol:

$$P \subseteq Q$$

For example, the set $\{x, y\}$ is a subset of the set $\{a, b, x, y\}$ but is not a subset of the set $\{x, c, d, e\}$. Note that every set is a subset of itself and the null set is a subset of every set. If P is a subset of Q and there is at least one element in Q that is not in P, then P is said to be a proper subset of Q. For example, the set $\{a, b\}$ is a proper subset of the set $\{x, y, a, b\}$. The notation $P \subseteq Q$ is used to denote that P is a proper subset of Q.

When two sets P and Q contain exactly the same elements (in whatever order), they are equal. For example, the two sets $P = \{w, x, y, z\}$ and $Q = \{x, y, w, z\}$ are equal. One may also say that two sets are equal if $P \subseteq Q$ and $Q \subseteq P$. The collection of all subsets of a set A is itself a set, called the *power set* of A, and is denoted by $P(A)$. For example, if $A = \{x, y, z\}$, then $P(A)$ consists of the following subsets of A:

$$\varnothing, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}$$

Sets can be combined by various operations to form new sets. These operations are analogous to such familiar operations as addition on numbers. One such operation is *union*. The union of two sets P and Q, denoted by $P \cup Q$, is the set consisting of all elements that belong to either P or Q (or both). For example, if

$$P = \{d, c, e, f\} \qquad Q = \{b, c, a, f\}$$

then

$$P \cup Q = \{a, b, c, d, e, f\}$$

The *intersection* of two sets P and Q, denoted by $P \cap Q$, is the set that contains those elements that are common to both sets. For example, if

$$P = \{a, c, e, g\} \qquad Q = \{c, e, i, k\}$$

then

$$P \cap Q = \{c, e\}$$

Two sets are *disjoint* if they have no element in common; that is, their intersection is empty. For example, if

$$B = \{b, e, f, r, s\} \qquad C = \{a, t, u, v\}$$

then

$$B \cap C = \varnothing$$

In other words, there are no elements that belong to both P and Q.

If P and Q are two sets, then the complement of Q with respect to P, denoted by $P - Q$, is the set of elements that belong to P but not to Q. For example, if

$$P = \{u, w, x\} \quad \text{and} \quad Q = \{w, x, y, z\}$$

then

$$P - Q = \{u\} \quad \text{and} \quad Q - P = \{y, z\}$$

Certain properties of the set operations follow easily from their definitions. For example, if P, Q, and R are sets, the following laws hold.

Commutative Law

1. $P \cup Q = Q \cup P$
2. $P \cap Q = Q \cap P$

Associative Law

3. $P \cup (Q \cup R) = (P \cup Q) \cup R$
4. $P \cap (Q \cap R) = (P \cap Q) \cap R$

Distributive Law

5. $P \cap (Q \cup R) = (P \cap Q) \cup (P \cap R)$
6. $P \cup (Q \cap R) = (P \cup Q) \cap (P \cup R)$

Absorption Law

7. $P \cap (P \cup Q) = P$
8. $P \cup (P \cap Q) = P$

Idempotent Law

9. $P \cup P = P$
10. $P \cap P = P$

DeMorgan's Law

11. $P - (Q \cup R) = (P - Q) \cap (P - R)$
12. $P - (Q \cap R) = (P - Q) \cup (P - R)$

2.3 RELATIONS

A *relation* is used to express some association that may exist between certain objects. For example, among a group of students, two students may be considered to be related if they have common last names. This section discusses the concept of binary relation and the different properties such a relation may possess. To formally define a binary relation it is helpful first to define ordered pairs of elements. An ordered pair of elements is a pair of elements arranged in a prescribed order. The notation (a, b) is used to denote the ordered pair in which the first element is a and the second element is b. The order of elements in an ordered pair is important. Thus the ordered pair (a, b) is not the same as the ordered pair (b, a). Besides, the two elements of an ordered pair need not be distinct. Thus (a, a) is a valid ordered pair. Two ordered pairs (a, b) and (c, d) are equal only when $a = c$ and $b = d$.

The *Cartesian product* of two sets A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$. For example, if $A = \{a, b\}$ and $B = \{1, 2, 3, 4\}$, then $A \times B = \{(a, 1), (a, 2), (a, 3), (a, 4), (b, 1), (b, 2), (b, 3), (b, 4)\}$. A binary relation from A to B is a subset of $A \times B$. For example, $\{(a, 1), (a, 4), (b, 2), (b, 3)\}$ is a binary relation on $A = \{a, b\}$ and $B = \{1, 2, 3, 4\}$. A binary relation can be represented in graphical form as shown in Figure 2.1, where the points on the left-hand side are the elements in A, the points on the right-hand side are the elements on B, and an arrow indicates that the corresponding element in A is related to the corresponding element in B.

The points used to represent the elements in sets A and B are known as vertices of the graph in Figure 2.1, and the directed lines $a \rightarrow 1$, $a \rightarrow 4$, $b \rightarrow 2$, and $b \rightarrow 3$ are called the *edges* of the graph. Thus it is possible to represent any relation with a directed graph. For example, if $A = \{0, 1, 2\}$ and we consider the relation "not equal to" on set A and itself,

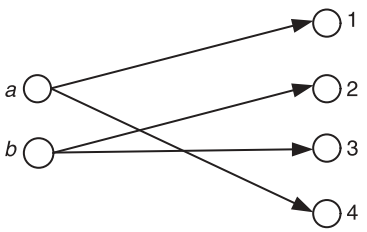


FIGURE 2.1 Graphical form of the relation $\{(a, 1), (a, 4), (b, 2), (b, 3)\}$.

then the directed graph representation of the relation can be derived directly from the Cartesian product of A with itself:

$$A \times A = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$$

Figure 2.2 shows the resulting directed graph. Note the absence of the loop from a node to itself; this is because it is not possible for an element in A not to be equal to itself.

In general, in many applications we encounter only binary relations on a set and itself, rather than relations from one set to a different set. The remainder of this section deals with the properties that these relations satisfy. A relation R on a set A (i.e., $R \subseteq A \times A$) is *reflexive* if (a, a) is in R for every a in A. In other words, in a reflexive relation every element in A is related to itself.

For example, let $A = \{1, 2, 3, 4\}$ and let $R = \{(1, 1), (2, 4), (3, 3), (4, 1), (4, 4)\}$. R is not a reflexive relation, since $(2, 2)$ does not belong to R.

Note that all ordered pairs (a, a) must belong to R in order for R to be reflexive. A relation that is not reflexive is called *irreflexive*. As another example, let A be a set of triangles and let the relation R on A be "a is similar to b." The relation is reflexive, since every triangle is similar to itself.

A relation R on a set is *symmetric* if $(a, b) \in R$ whenever $(b, a) \in R$; that is, if a is related to b, then b is also related to a. For example, let $A = \{1, 2, 3, 4\}$ and let $R = \{(1, 3), (4, 2), (2, 4), (2, 3), (3, 1)\}$. R is not a symmetric relation, since

$$(2, 3) \in R \quad \text{but} \quad (3, 2) \notin R.$$

On the other hand, if A is a set of people and the relation R on A is defined as

$$R = \{(x, y) \in A \times A | x \text{ is a cousin of } y\}$$

then R is symmetric, because if x is a cousin of y, y is a cousin of x.

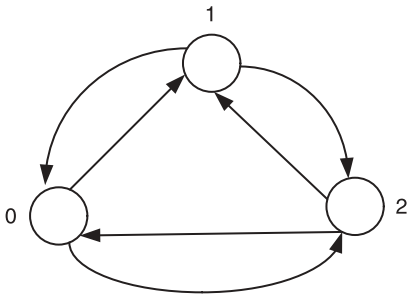


FIGURE 2.2 Directed graph representation of the relation "not equal to" on set $A = \{0, 1, 2\}$ and itself.

A relation R on a set A is *antisymmetric* if

$$(a, b) \in R \quad \text{and} \quad (b, a) \in R \ \text{ implies } \ a = b$$

In other words, if $a \neq b$, then a may be related to b or b may be related to a, but not both.

For example, if A is a set of people, then the relation

$$\{(a, b) \in A \times A | a \text{ is the father of } b\}$$

is antisymmetric because the reverse (i.e., b is the father of a) is not true. As a further example, let N be the set of natural numbers and let R be the relation on N defined by "x divides y." The relation R is antisymmetric, since x divides y and y divides x, which implies $x = y$.

A relation R on a set A is *transitive* if

$$(a, b) \in R \quad \text{and} \quad (b, c) \in R \text{ implies } (a, c) \in R$$

In other words, if a is related to b and b is related to c, then a is related to c. For example, let R be the relation "perpendicular to" on the set of lines $\{a, b, c\}$ in a plane. R is not transitive, since if a, b, c are lines such that "a is perpendicular to b" and "b is perpendicular to c," then a is not perpendicular to c; in fact, a is parallel to c. On the other hand, if R is the relation "less than or equal to" on the set of integers $A = \{4, 6, 8\}$, then $R = \{(4, 6), (6, 8), (4, 8), (6, 6)\}$ is transitive.

A relation R on a set A is called an *equivalence relation* if

(i) R is reflexive; that is, for every $a \in A$, $(a, a) \in R$.
(ii) R is symmetric; that is, $(a, b) \in R$ implies $(b, a) \in R$.
(iii) R is transitive; that is, $(a, b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$.

For example, the relation "working in the same office" on a set of employees in a given company is an equivalence relation (assuming that no employee works in more than one office).

1. It is reflexive, because each employee works in the same office as himself/herself.
2. It is symmetric, because if a works in the same office as b, then certainly b works in the same office as a.
3. It is transitive, because if a works in the same office as b, and b works in the same office as c then a works in the same office as c.

2.4 PARTITIONS

A partition, denoted by P, on any set A is a collection of nonempty subsets $A_1, A_2, \ldots, A_k$ of A such that their set union is A. In other words,

$$P = (A_1, A_2, \ldots, A_k)$$

such that

(i) $A_1 \cup A_2 \cup \ldots, A_k = A$.
(ii) $A_i \cup A_j = \varnothing$ if $i \neq j$.

The subsets A_j are called *blocks* of the partition.

For example, let $A = \{s, t, u, v, w, x, y, z\}$. Let us consider the following subsets of A:

$$A_1 = \{s, t, w, x\} \qquad A_2 = \{u, v, y, z\} \qquad A_3 = \{s, w, x\}$$

Then (A_1, A_3) is not a partition, since $A_1 \cap A_3 \neq \varnothing$. Also, $A_2 \cap A_3$ is not a partition, since it does not belong to either A_2 or A_3. The collection (A_1, A_2) is a partition of A. The partition in which each block has a single member of a set A is known as the *null partition*, denoted as $P(0)$. The partition in which elements of a set are in a single block is known as the *unity partition*, denoted as $P(I)$.

For example, let $A = \{w, x, y, z\}$; then

$$P(0) = \{(w), (x), (y), (z)\} \quad \text{and} \quad P(I) = \{(w, x, y, z)\}$$

It is evident that an equivalence relation on a set A induces a partition on A, since every two elements in a block are related. The blocks in the partition are called the *equivalence classes*. Conversely, a partition of a nonempty set A induces an equivalence relation on A; two elements are related if they belong to the same block of the partition.

For example, let $A = \{a, b, c, d, e\}$ and P_1 be a partition on A, where

$$P_1 = \{(a, c, e), (b, d)\}$$

The equivalence classes of the elements of A are the given blocks of the partition, that is, (a, c, e) and (b, d). Let R be an equivalence relation that induces the preceding partition. From the equivalence class (a, c, e) and the fact that R is an equivalence relation, we find that

$$(a, a), (a, c), (a, e), (c, c), (c, a), (c, e), (e, e), (e, a), (e, c) \in R$$

Similarly, equivalence class $(b, d) \in R$.

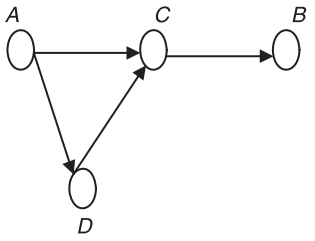
2.5 GRAPHS

A graph consists of a set of points called *nodes* or *vertices* and a set of interconnecting line segments called *arcs* or *edges*. If the edges in a graph have directions (orientation), then the graph is *directed*, otherwise it is *nondirected*.

An example of a directed graph, also known as a *digraph*, is shown in Figure 2.3*a*. Here the vertices are represented by points and there is a directed edge heading from A to C, A to D, D to C, and C to B. If there is a directed edge from vertex u to vertex v, u is *adjacent* to v. Thus in Figure 2.3*a*, A is adjacent to C, C is adjacent to B, and so on.

All connections in a directed graph can be described by the *connection matrix* of the directed graph. The connection matrix T is a square matrix of dimension n, where n is equal to the number of vertices in the directed graph. The entry t_{ij} at the intersection of row i and column j is 1 if there is an edge from node i to node j; otherwise, t_{ij} is 0. The connection matrix for the directed graph of Figure 2.3*a* is shown in Figure 2.3*b*.

In a directed graph, a *path* is a sequence of edges such that the terminal vertex of an edge coincides with the initial vertex of the following edge. For example, in Figure 2.4 *ADCB* is a path. A path with the same initial and final vertices is known as a *cycle*. In Figure 2.4, *ACDA* is a cycle. A directed graph without cycles is said to be *acyclic*.



(*a*)

	A	*B*	*C*	*D*
A	0	0	1	0
B	0	0	0	0
C	0	1	0	0
D	0	0	1	0

(*b*)

FIGURE 2.3 (*a*) Directed graph and (*b*) connection matrix.

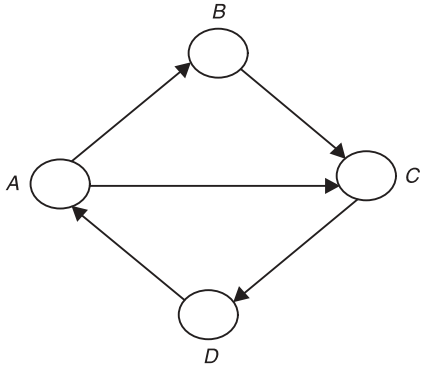


FIGURE 2.4 A directed graph.

The *in-degree* of a vertex in a directed graph is the number of edges terminating in the vertex. For example, in Figure 2.4 the in-degree of vertex *C* is 2, the in-degree of vertex *D* is 1, and so on. The number of edges leaving a vertex is called its *out-degree*. For example, the out-degree of vertex *D* is 1. An acyclic graph in which one vertex has an in-degree of 0 and all other vertices have an in-degree of 1 is called a *tree*. The vertex with in-degree 0 is called the root of the tree. The vertices with out-degree 0 are called the *leaves* of the tree. The edges in a tree are known as *branches.* Figure 2.5 shows a tree in which vertex *A* has

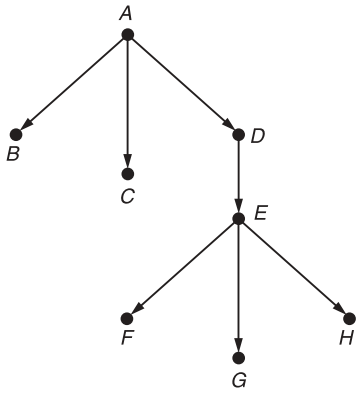


FIGURE 2.5 A tree.

an in-degree of 0, and vertices B, C, F, G, and H have an out-degree of 0. Thus A is the root of the tree, and B, C, F, G, and H are its leaves. Note that vertices D and E are not leaves of the tree because they have an out-degree of 1 and 3, respectively.

2.6 BOOLEAN ALGEBRA

Boolean algebra may be defined for a set A (which could be finite or infinite) in terms of two binary operations $+$ and $\cdot$. The symbols $+$ and $\cdot$ are called the inclusive OR and AND, respectively; they should not be confused with the addition and multiplication operations in conventional algebra. The operations in Boolean algebra are based on the following axioms or postulates, known as *Huntington's postulates*:

Postulate 1. If $x, y \in A$, then

$$x + y \in A; \quad x \cdot y \in A$$

This is called the *closure property*.

Postulate 2. If $x, y \in A$, then

$$x + y = y + x; \quad x \cdot y = y \cdot x$$

that is, $+$ and $\cdot$ operations are commutative.

Postulate 3. If $x, y, z \in A$, then

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$
$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

that is, $+$ and $\cdot$ operations are distributive.

Postulate 4. Identity elements, denoted as 0 and 1, must exist such that $x + 0 = x$ and $x \cdot 1 = x$ for all elements of A.

Postulate 5. For every element x in A there exists an element $\bar{x}$, called the *complement* of x, such that

$$x + \bar{x} = 1 \qquad x \cdot \bar{x} = 0$$

Note that the basic postulates are grouped in paris. One postulate can be obtained from the other by simply interchanging all OR and AND operations, and the identity elements 0 and 1. This property is known as *duality*. For example,

$$\begin{array}{c} x + (y \cdot z) = (x + y) \cdot (x + z) \\ \downarrow \quad \downarrow \qquad\quad \downarrow \quad \downarrow \quad \downarrow \\ x \cdot (y + z) = (x \cdot y) \; + \; (x \cdot z) \end{array}$$

Several theorems used for the manipulation of Boolean expressions are given below:

Theorem 1. The identity elements 0 and 1 are unique

Theorem 2. The Idempotent Laws

(i) $x + x = x$

(ii) $x \cdot x = x$

Theorem 3.

(i) $x + 1 = x$

(ii) $x \cdot 0 = 0$

Theorem 4. The Absorption Laws

(i) $x + xy = x$

(ii) $x(x + y) = x$

Theorem 5. Every element in set A has a unique complement

Theorem 6. Involution Theorem

$$(\bar{\bar{x}}) = x$$

Theorem 7.

(i) $x + \bar{x}y = x + y$

(ii) $x(\bar{x} + y) = xy$

Theorem 8. DeMorgan's Theorem

(i) $\overline{(x + y)} = \bar{x} \cdot \bar{y}$

(ii) $\overline{xy} = \bar{x} + \bar{y}$

We shall prove part i of DeMorgan's theorem. By definition of the complement (Postulate 5) and its uniqueness (Theorem 5) it is obvious that

$$(x + y) + \bar{x}\bar{y} = 1 \quad \text{and} \quad (x + y)\,\bar{x}\bar{y} = 0$$

$$\begin{aligned}
(x + y) + \bar{x}\bar{y} &= [(x + y) + \bar{x}]\,[(x + y) + \bar{y}] && \text{(by Postulate 3)} \\
&= [y + (x + \bar{x})]\,[x + (y + \bar{y})] && \text{(by associativity)} \\
&= [y + 1]\,[x + 1] && \text{(by Postulate 5)} \\
&= 1 \cdot 1 && \text{(by Theorem 3)} \\
&= 1
\end{aligned}$$

$$\begin{aligned}
(x + y)\,\bar{x}\bar{y} &= \bar{x}\,\bar{y}\,(x + y) && \text{(by commutativity)} \\
&= \bar{x}\bar{y} \cdot x + \bar{x}\bar{y} \cdot y && \text{(by distributivity)} \\
&= \bar{y}\,(\bar{x}x) + \bar{x}\,(\bar{y}y) && \text{(by associativity)} \\
&= \bar{y}\,(x\bar{x}) + \bar{x}\,(y\bar{y}) && \text{(by commutavity)} \\
&= \bar{y} \cdot 0 + \bar{x} \cdot 0 && \text{(by Postulate 5)} \\
&= 0
\end{aligned}$$

Since $(x + y) + \bar{x}\bar{y} = 1$ and $(x + y) \cdot \bar{x}\bar{y} = 0$, by Postulate 5 $\overline{(x + y)} = \bar{x}\bar{y}$.

We may generalize to include more than two elements:

(i) $\overline{a + b + \cdots + z} = \bar{a}\bar{b} \cdots \bar{z}$.
(ii) $\overline{ab \cdots z} = \bar{a} + \bar{b} + \cdots + \bar{z}$.

Example 2.1 Let us complement the following expression using Theorem 8

$$
\begin{aligned}
& a + b\,(\bar{c} + u\bar{v}) \\
\overline{a + b(\bar{c} + u\bar{v})} &= \bar{a} \cdot \overline{b(\bar{c} + u\bar{v})} \\
&= \bar{a}(\bar{b} + (\overline{\bar{c} + u\bar{v}})) \\
&= \bar{a}(\bar{b} + \bar{\bar{c}}(\overline{u\bar{v}})) \\
&= \bar{a}(\bar{b} + c(\bar{u} + v))
\end{aligned}
$$

It can be seen from the example that the complement of an expression can be obtained by replacing + (OR) with · (AND) and vice versa, and replacing each element by its complement.

Theorem 9. Consensus

(i) $xy + \bar{x}z + yz = xy + \bar{x}z$.
(ii) $(x + y)\,(\bar{x} + z)\,(y + z) = (x + y)\,(\bar{x} + z)$.

Proof:

$$
\begin{aligned}
xy + \bar{x}z + yz &= xy + \bar{x}z + 1 \cdot yz && \text{(by Postulate 4)} \\
&= xy + \bar{x}z + (x + \bar{x})yz && \text{(by Postulate 5)} \\
&= xy + \bar{x}z + xyz + \bar{x}yz && \text{(by Postulate 3)} \\
&= (xy + xyz) + (\bar{x}z + \bar{x}zy) \\
&= xy + \bar{x}z && \text{(by Theorem 4)}
\end{aligned}
$$

Example 2.2 Let us simplify the following expression using Theorem 9

$$(\bar{x} + y)\,wz + x\bar{y}v + vwz$$

Assume $x\bar{y} = a$ and $wz = b$. Then

$$
\begin{aligned}
(\bar{x} + y)\,wz + x\bar{y}v + vwz &= \bar{a}b + av + bv \\
&= \bar{a}b + av && \text{(by Theorem 9i)} \\
&= (\bar{x} + y)\,wz + x\bar{y}v && \text{(by replacing the values of } a \text{ and } b\text{)}
\end{aligned}
$$

Theorem 10.

(i) $xy + x\bar{y}z = xy + xz$.
(ii) $(x + y)\,(x + \bar{y} + z) = (x + y)\,(x + z)$.

Proof:

$$\begin{aligned} xy + x\bar{y}z &= x(y + \bar{y}z) && \text{(by Postulate 3)} \\ &= x(y + z) && \text{(by Theorem 7i)} \\ &= xy + xz && \text{(by Postulate 3)} \end{aligned}$$

Example 2.3 The following expression can be represented in a simplified form using Theorem 10.

$$\begin{aligned} (a + \bar{b})\,(a + b + c) &= (a + \bar{b})\,(a + c) \\ &= a + \bar{b}c && \text{(by Postulate 3)} \end{aligned}$$

Theorem 11.

(i) $xy + \bar{x}z = (x + z)\,(\bar{x} + y)$.

(ii) $(x + y)(\bar{x} + z) = xz + xy$.

Proof:

$$\begin{aligned} xy + \bar{x}z &= (xy + \bar{x})(xy + z) && \text{(by Postulate 3)} \\ &= (x + \bar{x})(y + \bar{x})(x + z)(y + z) && \text{(by Postulate 3)} \\ &= 1 \cdot (\bar{x} + y)(x + z)(y + z) && \text{(by Postulate 5)} \\ &= (\bar{x} + y)(x + z)(y + z) && \text{(by Postulate 4)} \\ &= (\bar{x} + y)(x + z) && \text{(by Theorem 9ii)} \end{aligned}$$

Example 2.4 Let us show the application of Theorem 11 in changing the form of the following Boolean expression.

$$\begin{aligned} & (\bar{a}b + ac)(a + \bar{b})(\bar{a} + \bar{c}) \\ &= (\bar{a} + c)(a + b)(a + \bar{b})(\bar{a} + \bar{c}) && \text{(by Theorem 11i)} \\ &= (\bar{a} + c)(\bar{a} + \bar{c})(a + b)(a + \bar{b}) && \text{(by Postulate 2)} \\ &= (\bar{a} + c \cdot \bar{c})(a + b \cdot \bar{b}) && \text{(by Postulate 3)} \\ &= (\bar{a} + 0)(a + 0) && \text{(by Postulate 5)} \\ &= \bar{a} \cdot a && \text{(by Postulate 5)} \\ &= 0 \end{aligned}$$

The postulates and theorems of Boolean algebra presented here relate to elements of a finite set A. If the set A is restricted to contain just two elements 0 and 1, then Boolean algebra is useful in the analysis and design of digital circuits. The two elements 0 and 1 are not binary numbers; they are just two symbols that are used to represent data in digital circuits. Two-valued Boolean algebra is often referred to as *switching algebra*. Henceforth, we shall use Boolean algebra for the set $\{0, 1\}$ unless otherwise specified.

2.7 BOOLEAN FUNCTIONS

In Boolean algebra, symbols are used to represent statements or propositions that may be true or false. These statements or propositions are connected together by operations AND, OR, and NOT. If 0 is used to denote a false statement and 1 is used to denote a true statement, then the AND $(\cdot)$ combination of two statements can be written as follows:

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

In tabular form it can be represented as

$\cdot$	0	1
0	0	0
1	0	1

The AND combination of two statements is known as the *product* of the statements.

The OR $(+)$ combination of two statements can be written as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

In tabular form it can be represented as

$+$	0	1
0	0	1
1	1	1

The OR combination of two statements is known as the *sum* of the statements.

The NOT operation of a statement is true if and only if the statement is false. The NOT operation on a statement can be stated as follows:

$$\bar{0} = 1$$
$$\bar{1} = 0$$

The NOT operation is also known as *complementation*.

A Boolean function $f(x_1, x_2, x_3, \ldots, x_n)$ is a function of n individual statements x_1, x_2, x_3, $\ldots$, x_n combined by AND, OR, and NOT operations. The statements x_1, x_2, x_3, $\ldots$, x_n are also known as *Boolean variables* and can be either *true* or *false*. In other words, each statement represents either the element 0 or 1 of the Boolean algebra.

TABLE 2.1 Truth Table for $f(x,y,z) = xy + \bar{x}z + \bar{y}\bar{z}$

x	y	z	$f(x, y, z)$
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	1

For example,

$$f(x, y, z) = xy + \bar{x}z + \bar{y}\bar{z}$$

is a function of three Boolean variables x, y, and z. In this function if $x = 0$, $y = 0$, and $z = 1$, then $f = 1$, as verified below:

$$\begin{aligned} f(0, 0, 1) &= 0 \cdot 0 + \bar{0} \cdot 1 + \bar{0} \cdot \bar{1} \\ &= 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

A Boolean function of n variables may also be described by a *truth table*. Since each of the Boolean variables can independently assume either a true (1) or a false (0) value, there are 2^n combinations of values for n variables. For each combination of values, a function can have a value of either 0 or 1. A truth table displays the value of a function for all possible 2^n combinations of its variables. The truth table for the function $f(x, y, z) = xy + \bar{x}z + \bar{y}\bar{z}$ is shown in Table 2.1.

It should be noted that a truth table describes only one Boolean function, although this function may be expressed in a number of ways.

The complement function $\bar{f}(x_1, x_2, \ldots, x_n)$ of a Boolean function $f(x_1, x_2, \ldots, x_n)$ has a value 1 whenever the value of f is 0 and a value 0 when $f = 1$. The truth table of a function can be used to derive the complement of the function by complementing each entry in column f. For example, by replacing each entry in the column f of Table 2.1 by its complement, we get the truth table of Table 2.2. It is the truth table for the function

TABLE 2.2 Truth Table for $f(x, y, z) = \overline{xy + \bar{x}z + \bar{y}\bar{z}}$

x	y	z	$f(x, y, z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

$f(x, y, z) = \bar{x}y\bar{z} + x\bar{y}z$, which is the complement of the function $f(x, y, z) = xy + \bar{x}z + \bar{y}\bar{z}$. This can be proved by deriving the complement of the function $f(x, y, z) = xy + \bar{x}z + \bar{y}\bar{z}$ using DeMorgan's theorem:

$$\begin{aligned}
\overline{f(x, y, z)} &= \overline{xy + \bar{x}z + \bar{y}\bar{z}} \\
&= \overline{(xy)} \cdot \overline{(\bar{x}z)} \cdot \overline{(\bar{y}\bar{z})} \\
&= (\bar{x} + \bar{y})(x + \bar{z})(y + z) \\
&= (\bar{x}x + \bar{x}\bar{z} + \bar{y}x + \bar{y}\bar{z})(y + z) \\
&= (0 + \bar{x}\bar{z} + \bar{y}x + \bar{y}\bar{z})(y + z) \\
&= (\bar{x}\bar{z} + \bar{y}x + \bar{y}\bar{z})(y + z) \\
&= \bar{x}\bar{z}y + \bar{y}xy + \bar{y}\bar{z}y + \bar{x}\bar{z}z + \bar{y}xz + \bar{y}\bar{z}z \\
&= \bar{x}y\bar{z} + x\bar{y}y + y\bar{y}\bar{z} + \bar{x}\bar{z}z + x\bar{y}z + \bar{y}\bar{z}z \\
&= \bar{x}y\bar{z} + x \cdot 0 + 0 \cdot \bar{z} + \bar{x} \cdot 0 + x\bar{y}z + \bar{y} \cdot 0 \\
&= \bar{x}y\bar{z} + 0 + 0 + 0 + x\bar{y}z + 0 \\
&= \bar{x}y\bar{z} + x\bar{y}z
\end{aligned}$$

2.8 DERIVATION AND CLASSIFICATION OF BOOLEAN FUNCTIONS

The behavior of a digital circuit is usually specified in plain English. Therefore this specification must be formulated into a truth table format before the circuit can actually be designed. As an example, let us consider the behavior of a circuit.

A circuit for controlling the lighting of a room is to be designed. The lights may be switched on or off from any of three switch points. Let the three on/off switches be X, Y, and Z and the light on or off condition be represented $f = 1$ or 0, respectively. The desired circuit is shown as a box in Figure 2.6. From the word description it is possible to tabulate the output condition (i.e., light on or off) for each combination of switch inputs. This is shown in Table 2.3.

Note that if the light is already on, turning on a switch will turn the light off. The truth table may be used to derive the Boolean function by extracting from the table those combination of XYZ that make $Z = 1$ (i.e., turn the light on). Thus

$$f(X, Y, Z) = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

A Boolean function in this form is known as a *sum of products*. A product term is a Boolean product (i.e., AND) of complemented or uncomplemented variables. As can be

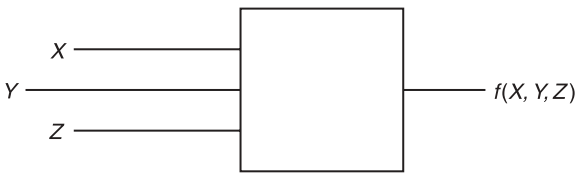


FIGURE 2.6 Lighting circuit to be designed.

TABLE 2.3 Truth Table for the Lighting Circuit (Switch On = 1; Switch Off = 0)

X	Y	Z	$f(X, Y, Z)$
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

seen, the above function is the sum (OR) of the products of the variables X, Y, and Z. The following expression is another example of the sum-of-products form of a Boolean function:

$$f(A, B, C, D) = AB\bar{C}\bar{D} + A\bar{B}CD + \bar{A}BC\bar{D} + \bar{A}\bar{B}C\bar{D}$$

This function has four variables and four product terms.

The *product-of-sums* form of a Boolean function can be derived from the truth table by extracting those combinations of input variables that produce an output of logic 0. For example, in Table 2.3, the product terms $\overline{XYZ}, \bar{X}YZ, X\bar{Y}Z$, and $XY\bar{Z}$ make the output $f = 0$ (i.e., do not turn the light on). Thus

$$\overline{f(X, Y, Z)} = \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}Z + XY\bar{Z}$$

By applying DeMorgan's theorem we obtain

$$\overline{\overline{f(X, Y, Z)}} = \overline{\bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}Z + XY\bar{Z}}$$

$$f(X, Y, Z) = \overline{\bar{X}\bar{Y}\bar{Z}} \cdot \overline{\bar{X}YZ} \cdot \overline{X\bar{Y}Z} \cdot \overline{XY\bar{Z}}$$

$$= (X + Y + Z)(X + \bar{Y} + \bar{Z})(\bar{X} + Y + \bar{Z})(\bar{X} + \bar{Y} + Z)$$

the product-of-sums form of the Boolean function for the lighting circuit.

Either the sum-of-products or the product-of-sums form may be used to represent a Boolean function. A product-of-sums form of a Boolean function can be obtained from its sum-of-products form by using the following steps:

Step 1. Derive the dual of the sum-of-products expression.

Step 2. Convert the resulting product-of-sums expression into a sum-of-products expression.

Step 3. Derive the dual of the sum-of-products expression to obtain the product-of-sums expression.

Let us convert the following sum-of-products expression into the product-of-sums form using the above procedure:

$$f(X, Y, Z) = XY + X\bar{Z} + YZ$$

The dual of the given expression is

$$(X + Y)(X + \bar{Z})(Y + Z)$$

The corresponding sum-of-products expression is

$$XY + XZ + Y\bar{Z}$$

The dual of this expression is the desired product of sums:

$$(X + Y)(X + Z)(Y + \bar{Z})$$

2.9 CANONICAL FORMS OF BOOLEAN FUNCTIONS

For any Boolean function, either sum of products or product of sums, there exists a standard or *canonical* form. In these two alternative forms, every variable appears, in either complemented or uncomplemented form, in each product term or sum term. A product that has this property is known as a *minterm*, whereas a sum term possessing this property is known as a *maxterm*.

A Boolean function composed completely of minterms is said to be in *canonical sum-of-products form*. For example,

$$f(X, Y, Z) = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}Z + XYZ \tag{2.1}$$

is a canonical function of three variables. Every product term in this expression contains all the variables in the function. A variable v and its complement $\bar{v}$ in an expression are known as *literals.* Note that although v and $\bar{v}$ are not two different variables, they are considered to be different literals.

If a Boolean function is composed completely of maxterms, then it is said to be in *canonical product-of-sums form*. For example,

$$f(X, Y, Z) = (X + Y + Z)(X + \bar{Y} + \bar{Z})(\bar{X} + Y + \bar{Z})(\bar{X} + \bar{Y} + Z) \tag{2.2}$$

is a canonical function of three variables with four maxterms.

For a Boolean function of n variables, there are 2^n minterms and 2^n maxterms. For example, minterms and maxterms for the three-variable Boolean function $f(X, Y, Z)$ are

Minterms	Maxterms
$\bar{X}\bar{Y}\bar{Z}$	$X + Y + Z$
$\bar{X}\bar{Y}Z$	$X + Y + \bar{Z}$
$\bar{X}Y\bar{Z}$	$X + \bar{Y} + Z$
$\bar{X}YZ$	$X + \bar{Y} + \bar{Z}$
$X\bar{Y}\bar{Z}$	$\bar{X} + Y + Z$
$X\bar{Y}Z$	$\bar{X} + Y + \bar{Z}$
$XY\bar{Z}$	$\bar{X} + \bar{Y} + Z$
XYZ	$\bar{X} + \bar{Y} + \bar{Z}$

From this minterm/maxterm list we note that the complement of any minterm is a maxterm and vice versa.

In order to simplify the notation for minterms, they are usually coded in decimal numbers. This is done by assigning 0 to a complemented variable and 1 to an uncomplemented variable, which results in the binary representation of a minterm. The corresponding decimal number d is derived, and the minterm is represented by m_d. For example, the minterm $X\bar{Y}\bar{Z}$ may be written 100 (=4); hence the minterm can be denoted by m_4. Thus the Boolean function represented by Eq. (2.1) may be written

$$f(X, Y, Z) = m_1 + m_2 + m_5 + m_7$$

This equation can be written in the minterm list form

$$f(X, Y, Z) = \sum m(1, 2, 5, 7)$$

The notation for maxterms is also simplified by coding them in decimal numbers. However, in this case, 0 is assigned to an uncomplemented variable and 1 to a complemented variable. The maxterms are denoted by M_d, where d is the decimal equivalent of the binary number. Thus the Boolean function represented by Eq. (2.2) may be written

$$f(X, Y, Z) = M_0 \cdot M_3 \cdot M_5 \cdot M_6$$

The above equation can be written in the maxterm list form:

$$f(X, Y, Z) = \prod M(0, 3, 5, 6)$$

As an example, let us derive the minterm list and the maxterm list for the Boolean function specified by the following truth table:

X	Y	Z	$f(X, Y, Z)$
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	1
1	1	1	1

The sum-of-products form of the function is

$$\begin{aligned} f(X, Y, Z) &= \underset{111}{XYZ} + \underset{110}{XY\bar{Z}} + \underset{100}{X\bar{Y}\bar{Z}} + \underset{010}{\bar{X}Y\bar{Z}} + \underset{001}{\bar{X}\bar{Y}Z} \\ &= m_7 + m_6 + m_4 + m_2 + m_1 \\ &= \sum m(1,\ 2,\ 4,\ 6,\ 7) \end{aligned}$$

The product-of-sums form of the function is

$$\begin{aligned} f(X,Y,Z) &= \overline{\bar{X}\bar{Y}\bar{Z} + \bar{X}YZ + X\bar{Y}Z} \\ &= \overline{\bar{X}\bar{Y}\bar{Z}} \cdot \overline{\bar{X}YZ} \cdot \overline{X\bar{Y}Z} \\ &= \underset{0}{(X} + \underset{0}{Y} + \underset{0}{Z})\left(\underset{0}{X} + \underset{1}{\bar{Y}} + \underset{1}{\bar{Z}}\right)\left(\underset{1}{\bar{X}} + \underset{0}{Y} + \underset{1}{\bar{Z}}\right) \quad \text{Maxterm code} \\ &= M_0 \cdot M_3 \cdot M_5 \\ &= \prod M(0,3,5) \end{aligned}$$

It can be seen from the example that the minterm and maxterm list of a Boolean function can be written directly from the truth table by inspection. The minterm list is the summation of all minterms for which the function has a value 1, whereas the maxterm list is the product of all decimal integers that are missing from the minterm list. Thus the conversion from one canonical form to the other is straightforward.

The following function is expressed in sum-of-products form:

$$f(X,Y,Z) = \sum m(1,3,6,7)$$

Its conversion to product-of-sums form results in

$$f(X,Y,Z) = \prod M(0,2,4,5)$$

A noncanonical Boolean function can be expanded to canonical form through repeated use of Postulate 5 (Section 2.6). Let us expand the following noncanonical sum-of-products form of the Boolean function:

$$\begin{aligned} f(X,Y,Z) &= XY + X\bar{Z} + YZ \\ &= XY(Z + \bar{Z}) + X(Y + \bar{Y})\bar{Z} + (X + \bar{X})YZ \\ &= XYZ + XY\bar{Z} + \cancel{XY\bar{Z}} + X\bar{Y}\bar{Z} + \cancel{XYZ} + \bar{X}YZ \end{aligned}$$

Since by Theorem 2, $x + x = x$, the duplicated terms can be deleted from the expression. Hence

$$\begin{aligned} f(X,Y,Z) &= \underset{111}{XYZ} + \underset{110}{XY\bar{Z}} + \underset{100}{X\bar{Y}\bar{Z}} + \underset{011}{\bar{X}YZ} \\ &= m_7 + m_6 + m_4 + m_3 \\ &= \sum m(3,4,6,7) \end{aligned}$$

Similarly, the following noncanonical product-of-sums form of the Boolean function may be expanded into the canonical form:

$$f(X,Y,Z) = (X + Y)\ (\bar{X} + Z)$$

In the first sum term Z is missing, and in the second Y is missing. Since by Theorem 2

$x \cdot \bar{x} = 0$, we introduce $Z\bar{Z} = 0$ and $Y\bar{Y} = 0$ in the first and second sum terms, respectively. Hence

$$\begin{aligned} f(X,Y,Z) &= (X + Y + Z\bar{Z})(\bar{X} + Y\bar{Y} + Z) \\ &= \begin{pmatrix} X & Y & Z \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} X & Y & \bar{Z} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \bar{X} & Y & Z \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \bar{X} & \bar{Y} & Z \\ 1 & 1 & 0 \end{pmatrix} \\ &= M_0 \cdot M_1 \cdot M_4 \cdot M_6 \\ &= \prod M(0,1,4,6) \end{aligned}$$

The canonical sum-of-products and product-of-sums forms of a Boolean function are unique. This property can be used to determine whether two noncanonical forms of a Boolean function are equal or not.

As an example, let us determine whether or not the following Boolean expressions are equal:

$$f(X,Y,Z) = XY + YZ + \bar{X}Z + \bar{X}\bar{Y} \tag{2.3}$$

$$f(X,Y,Z) = XY + \bar{X}\bar{Y} + \bar{X}YZ \tag{2.4}$$

We can expand Eq. (2.3) to its canonical form:

$$\begin{aligned} f(X,Y,Z) &= XY(Z + \bar{Z}) + (X + \bar{X})YZ + \bar{X}(Y + \bar{Y})Z + \bar{X}\bar{Y}(Z + \bar{Z}) \\ &= XYZ + XY\bar{Z} + XYZ + \bar{X}YZ + \bar{X}YZ + \bar{X}\bar{Y}Z + \bar{X}\bar{Y}Z + \bar{X}\bar{Y}\bar{Z} \\ &= XYZ + XY\bar{Z} + \bar{X}YZ + \bar{X}\bar{Y}Z + \bar{X}\bar{Y}\bar{Z} \end{aligned}$$

Next, we expand Eq. (2.4) to its canonical form:

$$\begin{aligned} f(X,Y,Z) &= XY + \bar{X}\bar{Y} + \bar{X}YZ \\ &= XY(Z + \bar{Z}) + \bar{X}\bar{Y}(Z + \bar{Z}) + \bar{X}YZ \\ &= XYZ + XY\bar{Z} + \bar{X}\bar{Y}Z + \bar{X}\bar{Y}\bar{Z} + \bar{X}YZ \end{aligned}$$

Since the canonical forms of both Eqs. (2.3) and (2.4) are identical, they represent the same Boolean function.

2.10 LOGIC GATES

The circuit elements used to realize Boolean functions are known as logic gates. There are AND, OR, and NOT (inverter) gates corresponding to AND, OR, and NOT operations, respectively.

AND The AND gate produces a 1 output if and only if all outputs are 1's. For example, the circuit in Figure 2.7 requires that both switches X and Y, which are normally open, must be closed before the light (L) comes on. In terms of Boolean algebra, a closed switch may correspond to a 1 and an open switch to a 0. Similarly, the light off and on conditions may be represented by 0 and 1, respectively. Thus the truth table for the AND gate is as shown in Table 2.4.

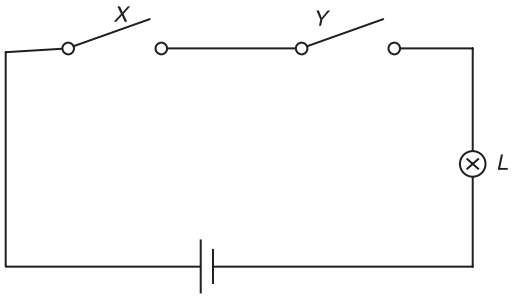
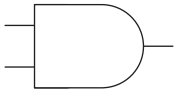


FIGURE 2.7 AND circuit.

TABLE 2.4 Truth Table for AND Circuit

X	*Y*	*L*
0	0	0
0	1	0
1	0	0
1	1	1

The AND circuit is symbolized as



The AND gate shown above has a *fan-in* of two (i.e., two inputs). However, it is possible for an AND gate to have more than two inputs; all inputs must be 1 for the output to be 1. Under any other condition the output will be 0.

OR The OR gate produces a 1 output if at least one of the inputs is 1. This is represented by switches in parallel, as shown in Figure 2.8. The truth table corresponding to the OR circuit is shown in Table 2.5.

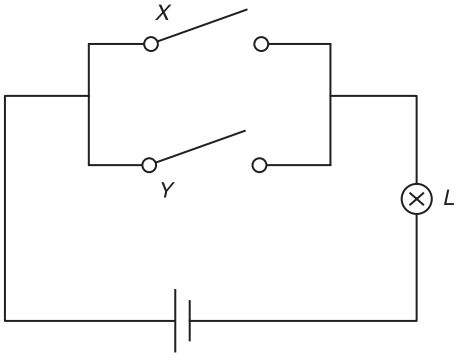
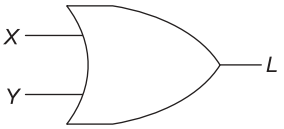


FIGURE 2.8 OR circuit.

TABLE 2.5 Truth Table for OR Circuit

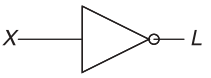
X	*Y*	*L*
0	0	0
0	1	1
1	0	1
1	1	1

The symbolic representation for an OR gate is



Again, the OR gate shown above has two inputs; theoretically, an OR gate can have any number of inputs. In such a gate the output will be 0 only if all the inputs are 0; otherwise the output will be 1.

NOT The NOT gate produces an output of 1 when the input is 0, and an output of 0 when the input is 1. The circuit representation of the NOT gate is shown in Figure 2.9. If the switch X is closed, the light remains on. In terms of Boolean algebra this means if $X = 0$, $f = 1$). When the switch is operated (i.e., $X = 1$) the circuit is broken, causing the light to be off (i.e., $f = 0$). The truth table of the NOT circuit is shown in Table 2.6. The NOT gate is also known as an inverter and is represented by the following symbol:



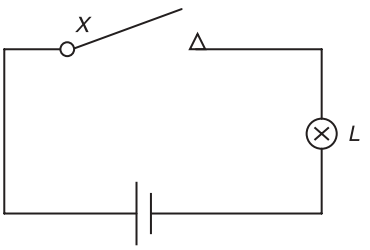


FIGURE 2.9 NOT circuit.

TABLE 2.6 Truth Table for NOT Circuit

X	*L*
0	1
1	0

The three basic gates give rise to two further compound gates: the *NOR* gate and the *NAND* gate. A NOR gate is formed by combining a NOT gate with an OR gate (Fig. 2.10). The truth table for the NOR gate is shown in Table 2.7. It can be seen from the truth table that if a 1 appears at any input, the output will be 0. The output is 1 if and only if both inputs are 0 (i.e., $L = XY$). Thus a NOR gate may also be formed by combining two NOT gates with an AND gate as shown in Figure 2.11.

A NAND gate is formed by combining a NOT gate with an AND gate (Fig. 2.12). The truth table for the NAND gate is shown in Table 2.8. It can be seen from the truth table that the output of the NAND gate will be 1 if at least one of the inputs is 0. The output is 0 if and only if the inputs are 1; that is,

$$L = \overline{X \cdot Y}$$

By DeMorgan's law,

$$L = \bar{X} + \bar{Y}$$

Thus a NAND gate may also be formed by combining two NOT gates with an OR gate, as shown in Figure 2.13.

So far we have discussed five logic gates: OR, AND, NOT, NOR, and NAND. These gates can be used to design any digital circuit. Two additional types of gates are also frequently used in digital circuit design; they are exclusive-OR (EX-OR) and exclusive-NOR (EX-NOR) gates.

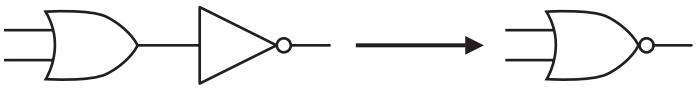


FIGURE 2.10 NOR gate.

TABLE 2.7 Truth Table for NOR Gate

X	Y	L
0	0	1
0	1	0
1	0	0
1	1	0

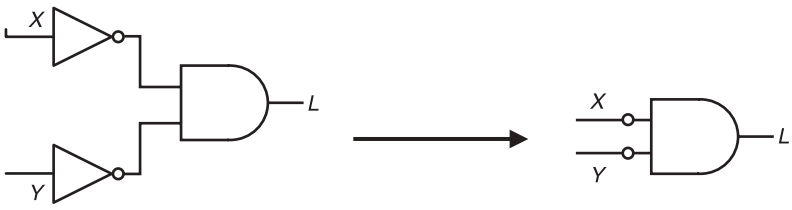


FIGURE 2.11 Alternative form of NOR gate.

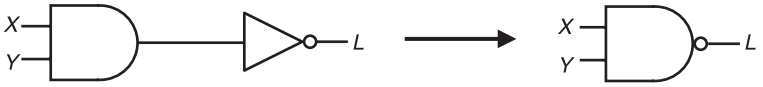


FIGURE 2.12 NAND gate.

TABLE 2.8 Truth Table for NAND Gate

X	*Y*	*L*
0	0	1
0	1	1
1	0	1
1	1	0

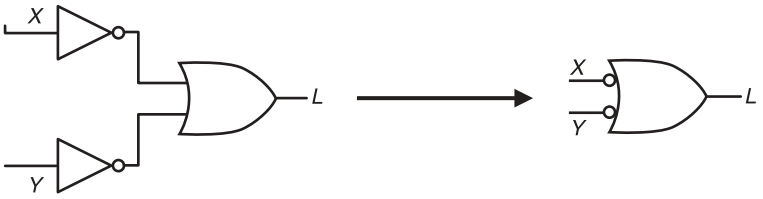


FIGURE 2.13 Alternative form of NAND gate.

The EX-OR gate produces a 1 output when *either* of the inputs is 1, but *not both.* This is different from the traditional OR gate, which produces a 1 output when either one or both of the inputs are 1. The truth table for an EX-OR gate is shown in Table 2.9.

In order to distinguish an EX-OR from the conventional or *inclusive* OR, a different symbol ($\oplus$) is used for an EX-OR operation.

The Boolean function corresponding to the truth table for the EX-OR gate is

$$f(X, Y) = \bar{X}Y + X\bar{Y} = X \oplus Y$$

thus an EX-OR gate can be formed from a combination of AND, OR, and NOT gates as shown in Figure 2.14.

The EX-OR gate is symbolized as

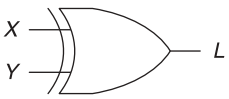


TABLE 2.9 Truth Table for EX-OR Gate

X	*Y*	*L*
0	0	0
0	1	1
1	0	1
1	1	0

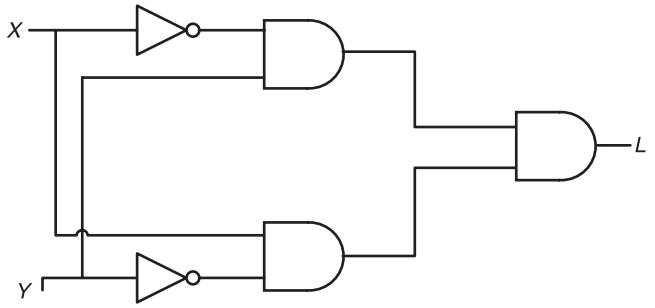


FIGURE 2.14 EX-OR gate.

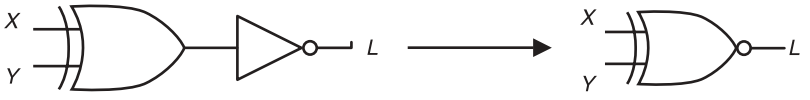


FIGURE 2.15 EX-NOR gate.

TABLE 2.10 Truth Table for EX-NOR Gate

X	*Y*	*L*
0	0	1
0	1	0
1	0	0
1	1	1

An EX-NOR gate can be formed by combining a NOT gate with an EX-OR gate (Fig. 2.15). The truth table for an EX-NOR gate is shown in Table 2.10. It can be seen from the truth table that the output of the EX-NOR gate is 1 only when both the inputs are either 0 or 1. For this reason an EX-NOR gate is also known as an *equivalence* or *coincidence* gate. It is represented by $\odot$.

EXERCISES

1. Let $S = \{a, b, c\}$. What are the subsets of S?

2. Consider the following sets: $W = \{a, b, c\}$, $Y = \{a, b\}$, and $Z = \{c\}$.
   a. Which of these sets are subsets of others or of themselves?
   b. How many proper subsets does each set have?

3. Write a specification by properties for each of the following sets:

a. {4, 8, 12, 16, . . .}

b. {3, 4, 7, 8, 11, 12, 15, 16, 19, 20, . . .}

c. {3, 13, 23, 33, . . .}

4. Let $W = \{w, x, y\}$, $X = \{y, d\}$, and $Y = \{y, e, f\}$. Determine the following:

a. $W \cup (X \cap Y)$

b. $W \cap X \cap Y$

c. $W - X$

5. Describe in words the following sets:

a. $\{x \mid x \text{ is oddly divisible by 3 and } 25 < x\}$

b. $\{x \mid x^2 - x - 6 = 0\}$

c. {5, 6, (2, 3)}

6. If $A \subseteq B$, what is $B \cup A$? What is $A \cap B$?

7. If C and D are disjoint, what is $C \cap D$? What is $C - D$?

8. Determine which of the properties *reflexive*, *transitive*, and *symmetric* apply to the following relations between integers x and y.

a. $x \leq y$

b. $x < y$

c. $x = y$

9. Let $S = \{1, 2, 3, 4, \ldots, 14, 15\}$, and let $a\ R\ b$ mean $a = b \bmod n$, where n is a positive integer.

a. Prove R is an equivalent relation on S.

b. List the equivalence classes into which R partitions S.

10. Let $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and let

$$P_1 = \{1, 2, 3, 4\} \qquad P_2 = \{5, 6, 7\}$$
$$P_3 = \{4, 5, 7, 9\} \qquad P_4 = \{4, 8, 10\}$$
$$P_5 = \{8, 9, 10\} \qquad P_6 = \{1, 2, 3, 6, 8, 10\}$$

Which of the following are partitions of S?

a. $\{P_1, P_2, P_5\}$

b. $\{P_1, P_3, P_5\}$

c. $\{P_3, P_6\}$

d. $\{P_4, P_3, P_2\}$

11. Let $P_1 = (1, 2)(4, 5)(6, 7)(3, 8, 9)$ and $P_2 = (4, 5, 7)(1, 2)(3, 9)(8)$. Find

a. $P_1 + P_2$

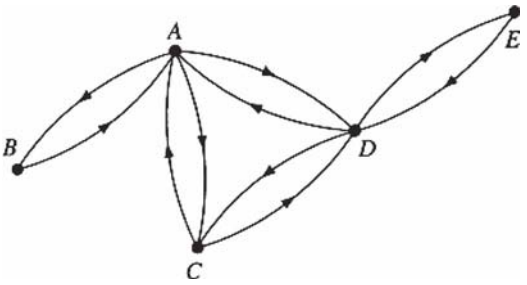
b. $P_1 \cdot P_2$

12. Let $A =$ the set of people belonging to a sports club. Let $a\ R\ b$ if and only if a and b play tennis; let $a\ S\ b$ if and only if a and b play golf. Determine $R \cap S$. (R and S are relations.)

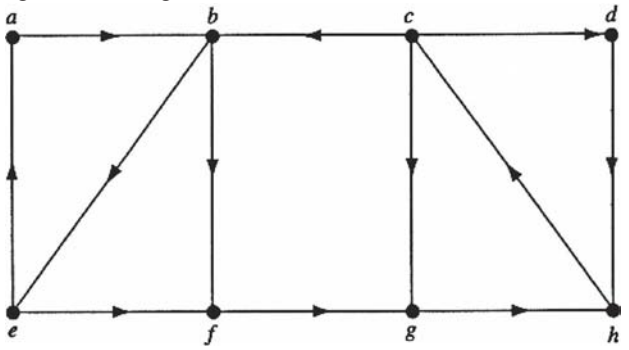
13. In the directed graph shown below, identify the following:

a. The set of vertices

b. The set of arcs



14. Find the number of distinct cycles of length 3 in the following graph. Are there any cycles of length 4 and length 5?



15. The set of vertices (V) and the set of edges (E) for three separate graphs are given below. In each case determine if the graph is a tree, and if it is, find the root.

$$G_1: \quad V = \{a, b, c, d, e, f\} \quad E = \{a - d, b - c, c - a, d - e\}$$
$$G_2: \quad V = \{a, b, c, d, e, f\} \quad E = \{a - b, c - e, f - a, f - c, f - d\}$$
$$G_3: \quad V = \{a, b, c, d, e, f\} \quad E = \{b - a, b - c, c - d, d - e, d - f\}$$

16. Simplify the following expressions using the postulates and theorems of Boolean algebra.

a. $ab + \bar{a}c + \bar{a}b\bar{c}$

b. $(a + \bar{c}) + abc + ac\bar{d} + cd$

c. $\overline{(\bar{a}(b + \bar{c}))}(a + \bar{b} + \bar{c})\overline{(abc)}$

d. $(a + b)(a + c)b$

e. $\bar{a}\,\bar{b}(ac + \bar{b}) + (a + b)(a\bar{b}\,\bar{c} + \bar{a}bc)$

17. Derive the dual of the following Boolean functions:

a. $f(a, b, c) = \bar{a}\,\bar{b}c + b\bar{c} + ac$

b. $f(a, b, c, d) = (\bar{a}\,\bar{c} + d)(ab + \bar{c})(\bar{b} + d)$

c. $f(a, b, c, d, e) = (a\bar{c} + bd + e)(\bar{a} + de)(a + \bar{d}\,\bar{e})(b + \bar{c})$

18. Prove that the complement of the EX-OR function is equal to its dual.

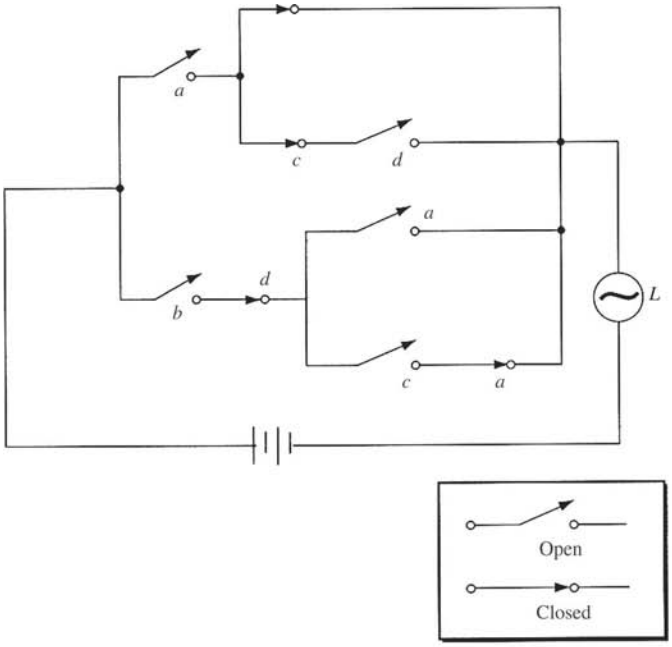
19. Find the complements of the following Boolean functions:

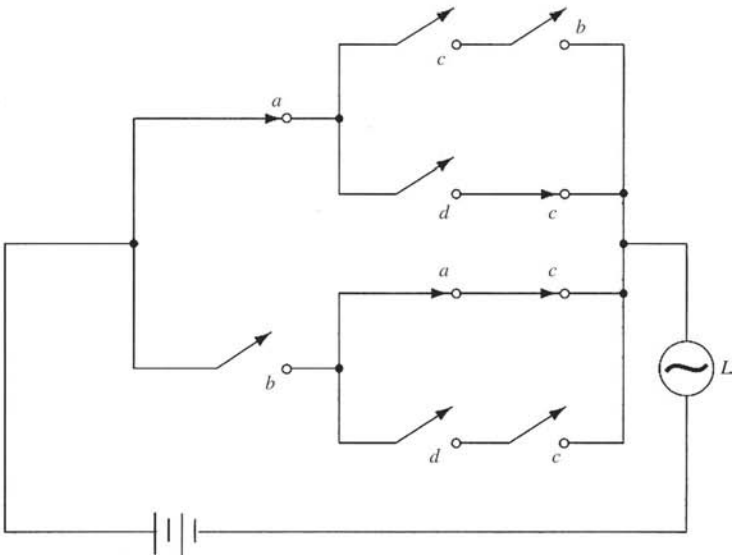
a. $f(a, b, c) = ab\bar{c} + a\bar{b}c + \bar{a}c$

b. $f(a, b, c) = ab \oplus c \oplus \bar{b}\,\bar{c}$

c. $f(a, b, c, d) = (b\bar{c} + \bar{a}d)(\bar{b}\,\bar{d} + ac)(ab + cd)$

20. Represent each of the following circuits using AND, OR and NOT gates. Derive the truth table for each of the following circuits.



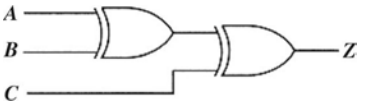


21. Derive the truth tables for the following gates:

a. 3-input NAND

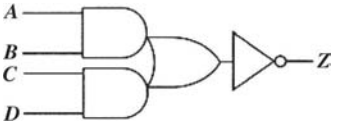
b. 3-input NOR

22. A 3-input EX-OR gate can be formed as shown:



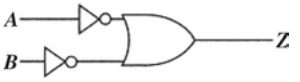
Derive the truth table for the gate.

23. A 4-input AOI (AND–OR–INVERT) gate is shown:



Derive the truth table for the gate.

24. Prove that the NAND function can be implemented as shown:



3 Combinational Logic Design

3.1 INTRODUCTION

Logic circuits are classified into two categories: *combinational* and *sequential*. In a combinational logic circuit the output is a function of the present input only. It does not depend on the past values of the inputs. If the output is a function of past inputs (memory) as well as the present inputs, then the circuit is known as a sequential logic circuit. This chapter focuses on combinational circuit design.

The main objective of combinational circuit design is to construct a circuit utilizing the minimum number of gates and inputs from the behavioral specification of the circuit. The first step in the design process is to construct a truth table of the circuit from its specification. The sum-of-products or product-of-sums form of the Boolean expression is then derived from the truth table and simplified where possible. The simplified expression is then implemented into the actual circuit by using appropriate gates.

Let us consider the design of a combinational circuit to meet the following specification. "The circuit has four inputs and one output. The output will be 1 if any two or more inputs are 1; otherwise the output will be 0." We begin by constructing the truth table that shows all the possible input combinations and the resulting output (Table 3.1). It is assumed that A, B, C, and D are the four inputs to the circuit and Z is the output. Writing down those input combinations that produce an output of 1, we obtain the canonical sum-of-products Boolean expression for the circuit:

$$Z = \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + \bar{A}BC\bar{D} + \bar{A}BCD + A\bar{B}C\bar{D} + A\bar{B}\bar{C}D$$
$$+ A\bar{B}CD + ABC\bar{D} + AB\bar{C}\bar{D} + ABCD + AB\bar{C}D$$

The next step in the design process is to simplify the sum-of-products expression (if possible). The expression can be rewritten in the following form, introducing certain redundant terms (underlined):

$$\begin{aligned} Z = {} & \bar{A}\bar{B}CD + \bar{A}BCD + A\bar{B}CD + ABCD + AB\bar{C}\bar{D} + AB\bar{C}D + \underline{ABCD} \\ & + ABC\bar{D} + \bar{A}B\bar{C}D + \underline{AB\bar{C}D} + \underline{\bar{A}BCD} + ABCD + A\bar{B}\bar{C}D + \underline{A\bar{B}CD} \\ & + \underline{AB}\bar{C}D + \underline{A}BC\underline{D} + \bar{A}BC\bar{D} + \underline{\bar{A}BCD} + \underline{ABC\bar{D}} + ABCD + A\bar{B}C\bar{D} \\ & + \underline{A\bar{B}CD} + \underline{ABC\bar{D}} + \underline{ABCD} \end{aligned}$$

Principles of Modern Digital Design, by Parag K. Lala

TABLE 3.1 A Truth Table

Input				Output
A	*B*	*C*	*D*	*Z*
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

$$\begin{aligned} Z &= (\bar{A}\bar{B} + \bar{A}B + A\bar{B} + AB)CD + AB(\bar{C}\bar{D} + \bar{C}D + CD + C\bar{D}) + BD(\bar{A}\bar{C} \\ &\quad + A\bar{C} + \bar{A}C + AC) + AD(\bar{B}\bar{C} + \bar{B}C + B\bar{C} + BC) + BC(\bar{A}\bar{D} \\ &\quad + \bar{A}D + A\bar{D} + AD) + AC(\bar{B}\bar{D} + \bar{B}D + B\bar{D} + BD) \\ &= CD + AB + BD + AD + BC + AC \end{aligned}$$

Note that the inclusion of redundant terms does not affect the expression, since by Theorem 2 (Chapter 2) of Boolean algebra, $x + x = x$.

As can be seen from the final expression, these additional terms helped considerably in simplifying the original Boolean expression. It can be implemented in AND and OR gates to give the required combinational logic circuit as shown in Figure 3.1.

3.2 MINIMIZATION OF BOOLEAN EXPRESSIONS

The formal specification of combinational logic circuits leads to canonical Boolean expressions. In most cases, these expressions must be simplified in order to reduce the number of gates required to implement the corresponding circuits. There are two steps that may be used to simplify a Boolean expression:

Step 1. Reduce the number of terms in the expression.

Step 2. Reduce the number of literals in the expression.

The first step corresponds to the reduction of the number of gates; the second step corresponds to the reduction of inputs to the gates in the resulting combinational logic circuit.

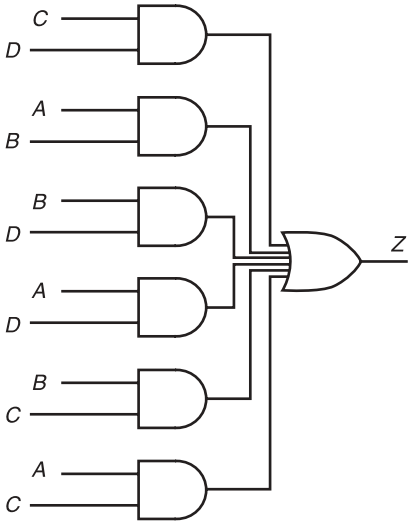


FIGURE 3.1 Implementation of the Boolean expression $Z = CD + AB + BD + AD + BC + AC$.

As an illustration, let us consider the minimization of the following Boolean expression:

$$Z = ABD + AB\bar{D} + \bar{A}C + \bar{A}BC + ABC$$

The expression has 5 product terms and 14 literals. Direct implementation of the expression would require 5 AND gates and 1 OR gate, assuming that the complemented variables are already available. Figure 3.2*a* shows the direct implementation of the expression.

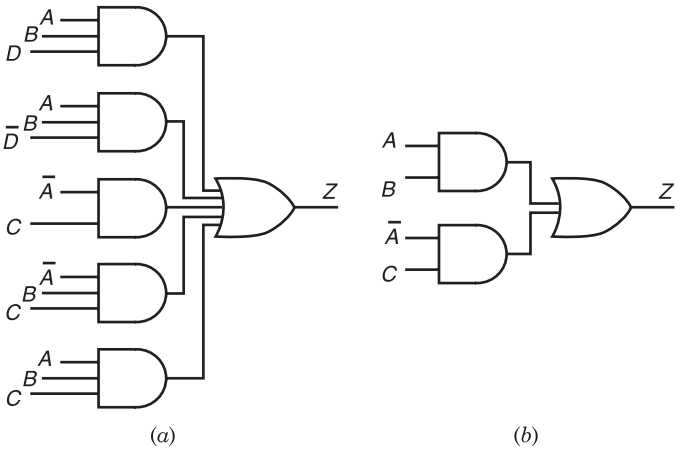


FIGURE 3.2 Circuit implementation of sum-of-products expression.

The expression can be simplified in the following way:

$$\begin{aligned} Z &= ABD + AB\bar{D} + \bar{A}C + \bar{A}BC + ABC \\ &= AB(D + \bar{D}) + \bar{A}C(1 + B) + ABC \\ &= AB + \bar{A}C + ABC \quad \text{(by Postulate 5 and Theorem 3, Chapter 2)} \\ &= AB(1 + C) + \bar{A}C \\ &= AB + \bar{A}C \end{aligned}$$

The minimized expression has two product terms and four literals. The variable D has been found to be redundant and has been eliminated. The minimized circuit is shown in Figure 3.2*b*.

So far we have considered the minimization of the sum-of-products form of the Boolean expression. The product-of-sums expressions can be minimized in a similar manner. For example, let us minimize the following expression:

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

We first take the complement of the product-of-sums expression,

$$\begin{aligned} Z &= \overline{(A + B + C)(A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C})} \\ &= \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC + ABC \\ &= \bar{A}\bar{C}(\bar{B} + B) + (\bar{A} + A)BC \\ &= \bar{A}\bar{C} + BC \end{aligned}$$

The next step is to take the complement of the resulting sum-of-products expression,

$$\begin{aligned} \bar{\bar{Z}} &= \overline{\bar{A}\bar{C} + BC} \\ Z &= \overline{\bar{A}\bar{C}} \cdot \overline{BC} \\ &= (A + C)(\bar{B} + \bar{C}) \end{aligned}$$

Thus

$$\begin{aligned} Z &= (A + B + C)(A + \bar{B} + C)(A + \bar{B} + \bar{C})(\bar{A} + \bar{B} + \bar{C}) \\ &= (A + C)(\bar{B} + \bar{C}) \end{aligned}$$

The original product-of-sums expression had 4 sum terms and 12 literals, whereas the minimized expression has 2 sum terms and 4 literals. Figure 3.3*a* and 3.3*b* show the implementations of the original and minimized expressions, respectively.

Sometimes the dual of an expression provides and easier way of minimizing a product-of-sums expression. The expression to be minimized is first converted to its

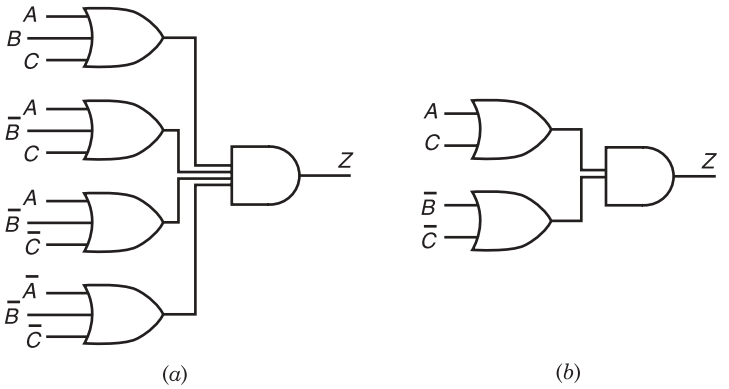


FIGURE 3.3 Circuit implementation of product-of-sums expression.

dual. The dual expression is minimized and then converted to *its* dual. For example, the Boolean expression

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

can be minimized by simplifying its dual expression,

$$\begin{aligned} Z_d &= \bar{A}B + \bar{A}C + \bar{B}\bar{C} \\ &= \bar{A}(B + C) + \bar{B}\bar{C} \end{aligned}$$

The dual of Z_d is

$$\begin{aligned} Z &= (\bar{A} + BC)(\bar{B} + \bar{C}) \\ &= \bar{A}\bar{B} + \bar{A}\bar{C} + BC \cdot \bar{B} + BC \cdot \bar{C} \\ &= \bar{A}\bar{B} + \bar{A}\bar{C} \\ &= \bar{A}(\bar{B} + \bar{C}) \end{aligned}$$

3.3 KARNAUGH MAPS

Boolean expressions can be graphically depicted and simplified with the use of Karnaugh maps. In a Karnaugh map 2^n possible minterms of an n-variable Boolean function are represented by means of separate squares or cells on the map. For example, the Karnaugh map of two variables A and B will consist of 2^2 squares—one for each possible combination of A and B as shown in Figure 3.4. Each square of the Karnaugh map is designated by a decimal number written on the right-hand upper corner of the square. The decimal number corresponds to the minterm number of the Boolean function.

For a Boolean function of n variables, the Karnaugh map is a $2^{n/2} \times 2^{n/2}$ square array if n is even. Thus if $n = 2$, the Karnaugh map is a 2×2 array as shown in Figure 3.4. If n is

	$\bar{B}$	B
$\bar{A}$	0 $\bar{A}\bar{B}$	1 $\bar{A}\bar{B}$
A	2 $A\bar{B}$	3 AB

FIGURE 3.4 Karnaugh map for a two-variable Boolean function.

	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	0	1	3	2
A	4	5	7	6

FIGURE 3.5 Karnaugh map for a three-variable Boolean function.

odd, the Karnaugh map is a $2^{(n-1)/2} \times 2^{(n+1)/2}$ rectangular array. Thus for a three-variable function, the Karnaugh map is a 2×4 array is shown in Figure 3.5. Figure 3.6 shows the Karnaugh map for a four-variable Boolean function, which contains 16 squares.

Boolean expressions may be plotted on Karnaugh maps if they are expressed in canonical form. For example, the following Boolean expression may be represented by the Karnaugh map shown in Figure 3.7.

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

Note that 1's are entered in cells 4, 3, 5, and 2, which correspond to the minterms $A\bar{B}\bar{C}$, $\bar{A}BC$, $A\bar{B}C$, and $\bar{A}B\bar{C}$ respectively; 0's are entered in all other cells.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	3	2
$\bar{A}B$	4	5	7	6
AB	12	13	15	14
$A\bar{B}$	8	9	11	10

FIGURE 3.6 Karnaugh map for a four-variable Boolean function.

	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	0 0	1 0	3 1	2 1
A	4 1	5 1	7 0	6 0

FIGURE 3.7 Karnaugh map for $Z = A\bar{B}\bar{C} + \bar{A}BC + A\bar{B}C + \bar{A}B\bar{C}$.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0 1	1	3 1	2 1
$\bar{A}B$	4	5 1	7	6
AB	12	13	15	14
$A\bar{B}$	8	9 1	11 1	10

FIGURE 3.8 Karnaugh map for $Z = \Sigma m(0, 2, 3, 5, 9, 11)$.

A further simplification is frequently made on Karnaugh maps by representing zeros by blank squares. Thus, a blank square means that the corresponding minterm is not included in the Boolean function. For example, the function

$$\begin{aligned} Z(A,B,C,D) &= \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}D + A\bar{B}\bar{C}D + A\bar{B}CD \\ &= \Sigma m(0,2,3,5,9,11) \end{aligned}$$

can be plotted on a Karnaugh map as shown in Figure 3.8.

The main feature of a Karnaugh map is that each square on the map is logically adjacent to the square that is physically adjacent to it. In other words, minterms corresponding to physically adjacent squares differ by a single variable. For example, in Figure 3.8 squares 9 and 11 are physically adjacent; square 9 represents minterm $A\bar{B}\bar{C}D$ and square 11 represents $A\bar{B}CD$, which are the same except in variable C. It should be noted that the first and last rows and the first and last columns in a Karnaugh map are also logically adjacent. For example square 3 (minterm $\bar{A}\bar{B}CD$) and square 11 (minterm $A\bar{B}CD$) are logically adjacent; similarly, square 0 (minterm $\bar{A}\bar{B}\bar{C}\bar{D}$) and square 2 (minterm $\bar{A}\bar{B}C\bar{D}$) are also adjacent.

Boolean functions on the Karnaugh maps can be simplified by using the property of adjacency. Thus, two minterms that are similar in all but one of their variables an be replaced by their common factor.

Example 3.1 The Boolean functions represented by the Karnaugh map of Figure 3.8 can be reduced to

$$Z(A,B,C,D) = \bar{A}\bar{B}\bar{D} + \bar{B}CD + A\bar{B}D + \bar{A}B\bar{C}D$$

Five of the six minterms that make up Z combine into three pairs:

$$m_0 + m_2 = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} = \bar{A}\bar{B}\bar{D}$$

$$m_3 + m_{11} = \bar{A}\bar{B}CD + A\bar{B}CD = \bar{B}CD$$

$$m_9 + m_{11} = A\bar{B}\bar{C}D + A\bar{B}CD = A\bar{B}D$$

Minterm 5 cannot be combined with any other minterm. Note that minterm 11 has been combined with two separate minterms, 3 and 9; this is possible because of Theorem 2

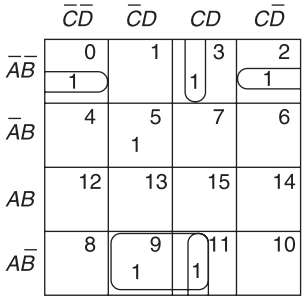


FIGURE 3.9 Looping of adjacent squares.

(Chapter 2). Thus a cell may be used in as many pairings as desired. The paintings of the minterms in the Karnaugh map for Z are shown by the loops in Figure 3.9. These loops indicate which two minterms have been combined to produce a simple term. As can be seen in Figure 3.9, when combining two minterms within a loop, the variable that changes from 0 to 1, or vice versa, is eliminated from the minterms. Thus if we combine minterms 3 and 11, the variable A is eliminated because it changes from 0 to 1. Similarly, when we combine minterms 9 and 11, the variable C is eliminated, and combining minterms 0 and 2 eliminates the variable C. In other words, the variables that are constant for a loop define the product term corresponding to the loop.

So far, we have considered only the grouping of two cells that are adjacent, either vertically or horizontally. Large numbers of cells can also be grouped, provided the number of cells in the group is a power of 2 (i.e., 4 cells, 8 cells, etc.). In fact, the larger the group of cells, the fewer will be the number of literals in the resulting product term. To illustrate, let us plot the four-variable Boolean function $f(A, B, C, D) = \Sigma m(0, 2, 5, 7, 8, 10, 13, 15)$ on the Karnaugh map (Fig. 3.10).

There are two groups, each containing four cells on the map, as shown by the loops in Figure 3.11. We have enclosed the terms $\bar{A}B\bar{C}D$, $\bar{A}BCD$, $AB\bar{C}D$, and $ABCD$; these combine to form $\bar{A}BD(C + \bar{C})$ and $ABD(\bar{C} + C)$, the results of which may then be combined to give $BD(\bar{A} + A) = BD$. Now, grouping terms 0 and 8 together and terms 2 and 10 together eliminated the variable A from each pair. As can be seen in the map (Fig. 3.11), these four terms reduce to the two terms $\bar{B}\bar{C}\bar{D}$ and $\bar{B}C\bar{D}$. These two terms, which represent combined minterms 0 and 8, and 2 and 10, respectively, can be grouped to eliminated the variable C,

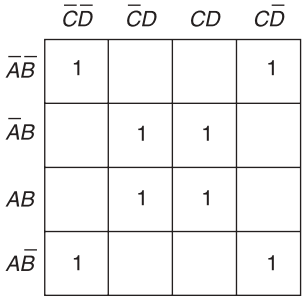


FIGURE 3.10 Karnaugh map for $f = \Sigma m(0, 2, 5, 7, 8, 10, 13, 15)$.

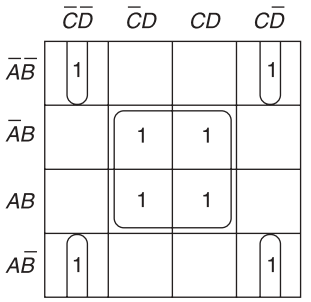


FIGURE 3.11 Looping of squares.

thus reducing the four terms to one terms, $\bar{B}\bar{D}$. Note that four terms have been combined to eliminate two liberals. Thus the reduced form of the above Boolean function is

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

Example 3.2 Simplify the following four-variable function:

$$f(A, B, C, D) = \Sigma m(0, 1, 4, 5, 7, 8, 9, 12, 13, 15)$$

The Karnaugh map for the function is shown in Figure 3.12. The reduced form of the function can be derived directly from the Karnaugh map:

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

In four-variable Karnaugh maps, the top and bottom rows are logically adjacent and so are the left and right columns. We saw one example of grouping four cells that were not physically adjacent (Fig. 3.11). Figure 3.13 shows a few more left–right column, top–bottom row adjacencies.

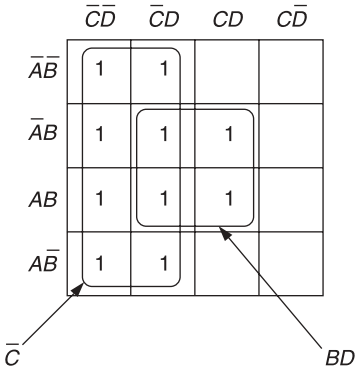


FIGURE 3.12 Karnaugh map for the logic function of Example 3.2.

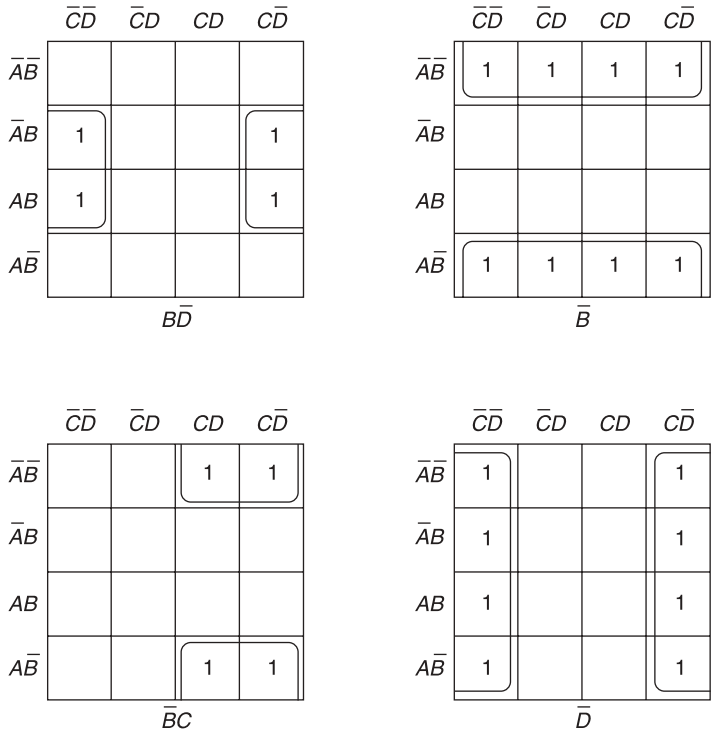


FIGURE 3.13 Examples of adjacencies in Karnaugh maps.

3.3.1 Don't Care Conditions

In certain Boolean functions it is not possible to specify the output for some input combinations. It means that these particular input combinations have no relevant effect on the output. These input combinations or conditions are called *don't care conditions*, and the minterms corresponding to these input combinations are called don't care terms. Functions that include don't care terms are said to be *incompletely specified* functions. The don't care minterms are labeled d instead of m.

Example 3.3 Let us consider the following function,

$$f(A, B, C) = \Sigma m(0, 4, 7) + d(1, 2, 6)$$

where 1, 2, and 6 are the don't care terms. Since the don't care combinations cannot occur, the output corresponding to these input combinations can be assigned either 0 or 1 at will. It is often possible to utilize the don't care terms to aid in the simplification of Boolean functions. For example, the Karnaugh map resulting from the above Boolean function is as shown in Figure 3.14*a*. Note that the simplified function

$$f(A, B, C) + \bar{C} + AB$$

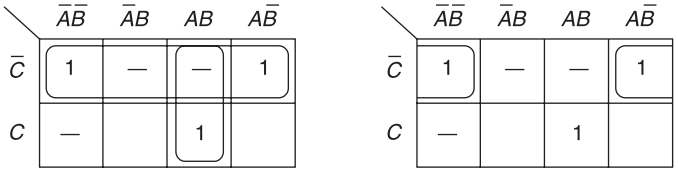


FIGURE 3.14 (*a*) Minimization using don't cares, and (*b*) minimization without using don't cares.

is obtained by grouping minterms 0 and 4 with don't cares 2 and 6, and grouping minterm 7 with don't care 6. Don't care 2 is not used.

If the don't care terms are not included in the minimization, only minterms 0 and 4 can be grouped, as shown in Figure 3.14*b*. In that case, the function simplifies to

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

which contains more literals than the simplified function obtained by considering the don't care terms.

The best way to utilize don't cares in minimizing Boolean functions is to assume don't cares to be 1 if that results in grouping a larger number of cells on the map than would be possible otherwise. In other words, only those don't cares that aid in the simplification of a function are taken into consideration.

Example 3.4 Let us minimize the following Boolean function using a Karnaugh map:

$$f(A, B, C, D) = \Sigma m(0, 1, 5, 7, 8, 9, 12, 14, 15) + d(3, 11, 13)$$

The Karnaugh map is shown in Figure 3.15. From this the minimized function is given by

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

Again, it is emphasized that while considering don't care terms on Karnaugh maps, it is not necessary to use all the terms (or even one term), unless their inclusion in a group would assist the minimization process.

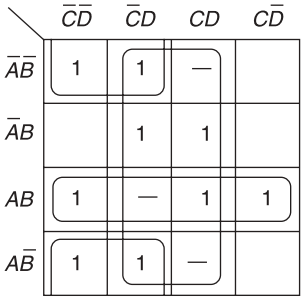


FIGURE 3.15 Karnaugh map for the function in Example 3.4.

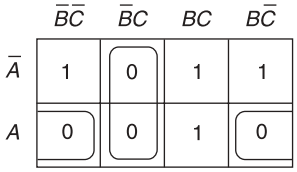


FIGURE 3.16 Karnaugh map for a three-variable function.

3.3.2 The Complementary Approach

Sometimes it is more convenient to group the 0's on a Karnaugh map rather than the 1's. The resultant sum of products is the complement of the desired expression. This sum of products is then complemented by using DeMorgan's theorem, which results in a minimum product-of-sums expression. This is known as the complementary approach.

Example 3.5 Let us consider the three-variable Karnaugh map shown in Figure 3.16. The grouping of 0's yields the function

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

Hence $f = \bar{\bar{f}} = \overline{A\bar{C} + \bar{B}C} = (\bar{A} + C)(B + \bar{C})$.

Occasionally the complement function gives a better minimization.

Example 3.6 The minimized form of the Boolean function

$$f(A,B,C,D) = \Sigma m(0,1,2,4,5,6,8,9,10)$$

is $f = \bar{A}\bar{C} + \bar{B}\bar{C} + \bar{A}\bar{D} + \bar{B}\bar{D}$; this is derived from the Karnaugh map of Figure 3.17*a*.

The complement function is derived by grouping the 0's in the Karnaugh map (Fig. 3.17*b*). Thus the complement function is

$$\bar{f} = AB + CD$$

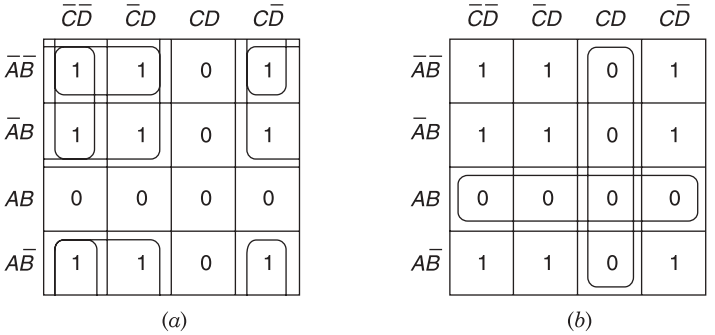


FIGURE 3.17 (*a*) Groupings of ones, (*b*) groupings of zeroes.

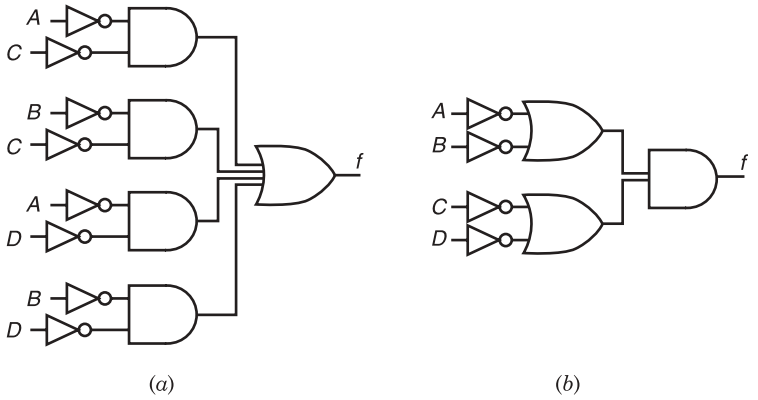


FIGURE 3.18 Examples of adjacencies in Karnaugh.

Inverting the complement function, we find

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

The implementation of the minimized sum-of-products and the product-of-sums functions are shown in Figure 3.18*a* and 3.18*b*, respectively. The product-of-sums form gives a simpler solution in this case. Thus both the simplified sum-of-products and the product-of-sums forms for a given function must be examined before a decision can be made as to which form will be cheaper to implement.

The complementary approach may also be utilized to expand a general product-of-sums function to canonical form. Let us illustrate this by obtaining the canonical product-of-sums form for the function

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

The complement of the function is

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

which leads to the Karnaugh map shown in Figure 3.19. The canonical form of $\bar{f}$ is

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

	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	0	0	1	0
A	0	0	1	1

FIGURE 3.19 Examples of adjacencies in Karnaugh.

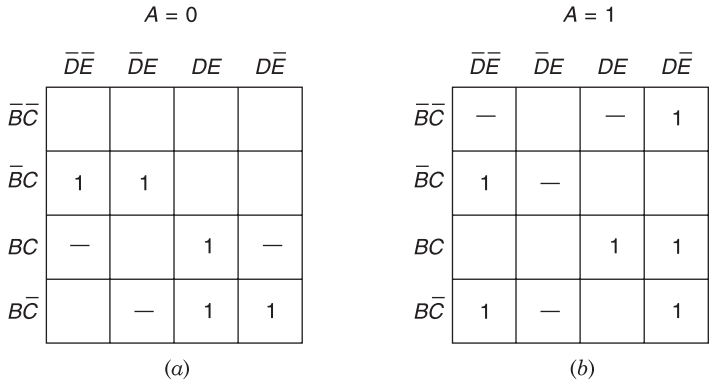


FIGURE 3.20 Examples of adjacencies in Karnaugh.

Thus the canonical product-of-sums form for f is

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

A Karnaugh map with four input variables is quite straightforward. With five or more variables, however, the map becomes complicated and the adjacencies are difficult to recognize. However, it is possible to minimize a five-variable Boolean function with a four-variable Karnaugh map.

Example 3.7 Let us minimize the Boolean function

$$f(A, B, C, D, E) = \Sigma m(4, 5, 10, 11, 15, 18, 20, 24, 26, 30, 31) + d(9, 12, 14, 16, 19, 21, 25)$$

The Karnaugh map for the function is shown in Figure 3.20. The five variables are divided between two four-variable maps. Note that the difference between the $A = 0$ map and the $A = 1$ map is that for $A = 0$, the entry in cell $B\bar{C}DE$ is 1, whereas the corresponding entry is 0 in the $A = 1$ map. In addition, the entries in cells $\bar{B}\bar{C}\bar{D}\bar{E}, \bar{B}\bar{C}D\bar{E}$, and $B\bar{C}\bar{D}\bar{E}$ in the

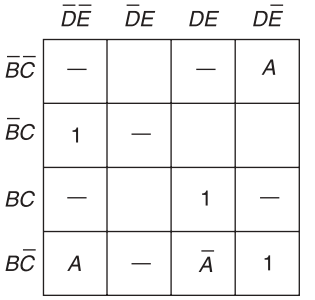


FIGURE 3.21 Equivalent Karnaugh map for Figure 3.20*a* and 3.20*b*.

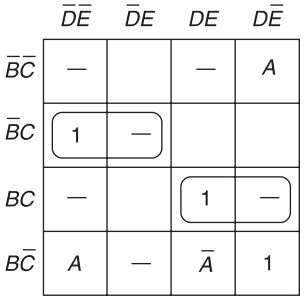


FIGURE 3.22 $f(A, B, C, D, E) = \bar{B}C\bar{D} + BCD$.

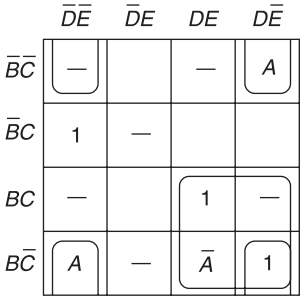


FIGURE 3.23 $f(A, B, C, D, E) = A\bar{C}\bar{E} + \bar{B}C\bar{D} + BCD + \bar{A}BD$.

$A = 1$ map are –, 1, and 1, respectively, whereas the corresponding entries in the $A = 0$ map are 0's. Thus the four-variable Karnaugh map with $\bar{A}$ in cells $B\bar{C}DE$ and A in cells $\bar{B}\bar{C}D\bar{E}$ and $B\bar{C}\bar{D}\bar{E}$ shown in Figure 3.21 is equivalent to the maps of Figure 3.20.

The map is then reduced in two steps.

Step 1. Group all terms employing 1's and –'s. The letter variable terms are ignored at this step. Figure 3.22 shown the relevant groupings on the map of Figure 3.21.

Step 2. Group the letter variable(s) with the adjacent 1's and –'s. The resulting terms are then ORed with the terms derived in step 1 to obtain the minimized function (shown in Figure 3.23).

3.4 QUINE–McCLUSKEY METHOD

The Karnaugh map approach is not suitable for minimizing Boolean functions having more than six variables. For functions with a large number of variables, a tabular method known as the Quine–McCluskey method is much more effective. The method consists of two steps:

1. Generation of all prime implicants.
2. Selection of a minimum subset of prime implications, which will represent the original function.

A *prime implicant* is a product term that cannot be combined with any other product term to generate a term with fewer literals than the original term.

As an example, consider a Boolean function

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

which after minimization becomes

$$f(A, B, C) = AB + BC + AC + \bar{A}\bar{B}\bar{C}$$

The product terms AB, BC, AC, and $\bar{A}\bar{B}\bar{C}$ are all prime implicants because none of them can be ombined with any other term in the function to yield a term with fewer literals. A prime implicants is called an *essential prime implicant* if it covers at least one minterm that is not covered by any other prime implicant of the function.

Example 3.8 Let us minimize the following Boolean function:

$$f(A, B, C, D) = \Sigma m(1, 4, 5, 10, 12, 13, 14, 15)$$

The Karnaugh map for the function is shown in Figure 3.24.

The prime implicants for the function are $B\bar{C}$, AB, $\bar{A}\bar{C}D$, and $AC\bar{D}$. The minimized function is

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

The prime implicant $\bar{A}\bar{C}\bar{D}$ is an essential prime implicant because it covers minterm $\bar{A}\bar{B}\bar{C}D$, which is not covered by any other prime implicant. Similarly, only $AC\bar{D}$ covers minterm $A\bar{B}C\bar{D}$, $B\bar{C}$ covers $\bar{A}B\bar{C}\bar{D}$, and AB covers $ABCD$; in other words, $AC\bar{D}$, $B\bar{C}$, and AB are also essential prime implicants.

The Quine–McCluskey method for minimization can be formulated as follows:

Step 1. Tabulate all the minterms of the function by their binary representations.

Step 2. Arrange the minterms into groups according to the number of 1's in their binary representation. For example, if the first group consists of minterms with n 1's, the

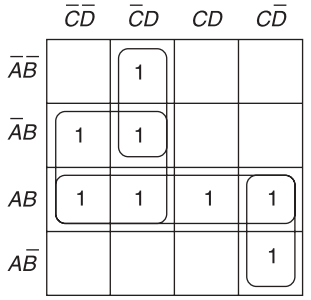


FIGURE 3.24 Examples of adjacencies in Karnaugh.

second group will consist of minterms with $(n+1)$ 1's and so on. Lines are drawn between different group to simplify identification.

Step 3. Compare each minterm in a group with each of the minterms in the group below it. If the compared pair is adjacent (i.e., if they differ by one variable only), they are combined to form a new term. The new term has a dash in the position of the eliminated variable. Both combining terms are checked off in the original list indicating that they are not prime implicatnts.

Step 4. Repeat the above step for all groups of minterms in the list. This results in a new list of terms with dashes in place of eliminated variables.

Step 5. Compare terms in the new list in search for further combinations. This is done by following step 3. In this case a pair of terms can be combined only if they have dashes in the same positions. As before, a term is checked off if it is combined with another. This step is repeated until no new list can be formed. All terms that remain unchecked are prime implications.

Step 6. Select a minimal subset of prime implicants that cover all the terms of the original Boolean function.

Example 3.9 Let us minimize the following Boolean function using the Quine–McCluskey procedure:

$$f(A,B,C,D,E) = \Sigma m(0,1,2,9,11,12,13,27,28,29)$$

The minterms are first tabulated according to step 1.

Minterm	A	B	C	D	E
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	0
9	0	1	0	0	1
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
27	1	1	0	1	1
28	1	1	1	0	0
29	1	1	1	0	1

The minterms are then grouped according to the number of 1's contained in each term, as specified in step 2. This results in list 1 of Figure 3.25. In list 1, terms of group 1 are combined with those of group 2, terms of group 2 are combined with those of group 3, and so on, using step 3. For example, 0(00000) is adjacent to 1(00001). So they are combined to form 0000– which is the first term in list 2. Both combined terms are checked off in list 1. Since 0(00000) is also adjacent to 2(00010) they are combined to form the term 000–0, which is also entered in list 2. A line is then drawn under the two terms in list 2 in order to identify them as a distinct group.

The next step is to compare the two terms in group 2 of list 1 with the two terms in group 3. Only terms 1(00001) and 9(01001) combine to give 0–001; all other terms

	List 1			List 2			List 3		
	Minterm	*ABCDE*		*Minterms*	*ABCDE*		*Minterms*	*ABCDE*	
Group 1	0	00000	✓	0,1	0000–	PI_2	12,13,28,29	–110–	PI_1
	1	00001	✓	0,2	000–0	PI_3			
Group 2	2	00010	✓	1,9	0–001	PI_4			
	9	01001	✓	9,13	01–01	PI_5			
Group 3	12	01100	✓	9,11	010–1	PI_6			
	13	01101	✓	12,13	0110–	✓			
Group 4	11	01011	✓	12,28	–1100	✓			
	28	11100	✓	13,29	–1101	✓			
	29	11101	✓	11,27	–1011	PI_7			
Group 5	27	11011	✓	28,29	1110–	✓			

FIGURE 3.25 Determination of prime implicants.

differ in more than one variable and therefore do not combine. As a result, the second group of list 2 contains only one combination. The two terms in group 3 are now compared with the three terms in group 4. Terms 9(01001) and 11(01011) combine to give 010–1, terms 9(01001) and 13(01101) combine to give 01–01, terms 12(01100) and 13(01101) combine to give 0110–, and terms 12(01100) and 28(11100) combine to give –1100. Thus the third group of list 2 contains four terms. Finally, the three terms in group 4 of list 1 are compared with the two terms in group 5. Terms 13(01101) and 29(11101) combine to give –1101, terms 11(01011) and 27(11011) combine to give –1011, and terms 28(11100) and 29(11101) combine to give 1110–. Therefore the fourth group of list 2 contains three terms.

The process of combining terms in adjacent groups is continued for list 2. This results in list 3. It can be seen in Figure 3.25 that certain terms cannot be combined further in list 2. These correspond to the prime implicants of the Boolean function and are labeled $PI_1, \ldots, PI_7$.

The final step of the Quine–McCluskey proedure is to find a minimal subset of the prime implicants which can be used to realize the original function. The complete set of prime implicants for the given function can be derived from Figure 3.25; are

$$(BC\bar{D},\ \bar{A}\bar{B}\bar{C}\bar{D},\ \bar{A}\bar{B}\bar{C}\bar{E},\ \bar{A}\bar{C}\bar{D}E,\ \bar{A}B\bar{D}E,\ \bar{A}B\bar{C}E,\ B\bar{C}DE)$$

In order to select the smallest number of prime implicants that account for all the original minterms, a *prime implicant chart* is formed as shown in Figure 3.26. A prime implicant

	0	*1*	*2*	*9*	*11*	*12*	*13*	*27*	*28*	*29*
PI_1*						X	X		X	X
PI_2	X	X								
PI_3*	X		X							
PI_4		X		X						
PI_5				X			X			
PI_6				X	X					
PI_7*					X			X		

FIGURE 3.26 Prime implicant chart.

	1	9
PI_2	X	
PI_4	X	X
PI_5		X
PI_6		X

FIGURE 3.27 Examples of adjacencies in Karnaugh.

chart has a column for each of the original minterms and a row for each prime implicant. For each prime implicant row, an X is placed in the columns of those minterms that are accounted for by the prime implicant. For example, in Figure 3.26 prime implicant PI_1, comprising minterms 12, 13, 28, and 29, has X's in columns 12, 13, 28, and 29. To choose a minimum subset of prime implicants, it is first necessary to identify the essential prime implicants. A column with a single X indicates that the prime implicant row is the only one covering the minterm corresponding to the column; therefore the prime implicant is essential and must be included in the minimized function. Figure 3.26 has three essential prime implicants, and they are identified by asterisks. The minterms covered by the essential prime implicants are marked with asterisks.

The next step is to select additional prime implicants that can cover the remaining column terms. This is usually done by forming a reduced prime implicant chart that contains only the minterms that have not been covered by the essential prime implicants. Figure 3.27 shows the reduced prime implicant chart derived from Figure 3.26.

Prime implicant PI_4 covers the minterms 1 and 9. Therefore the minimum sum-of-products equivalent to the original function is

$$\begin{aligned} f(A,B,C,D,E) &= PI_1 + PI_3 + PI_4 + PI_7 \\ &= -110 - + 000 - 0 + 0 - 001 + -1011 \\ &= BC\bar{D} + \bar{A}\bar{B}\bar{C}\bar{E} + \bar{A}\bar{C}\bar{D}E + B\bar{C}DE \end{aligned}$$

For some functions, the prime implicant chart may not contain any essential prime implicants. In other words, in every column of a prime implicant chart there are two or more X's. Such a chart is said to be *cyclic*.

Example 3.10 The following Boolean function has a cyclic prime implicant chart:

$$f(A,B,C) = \Sigma m(1,2,3,4,5,6)$$

The prime implicants of the function can be derived as shown in Figure 3.28. The resulting prime implicant chart as shown in Figure 3.29 is cyclic; all columns have two X's. As can

Minterm	ABC		Minterm	ABC	
1	001	✓	1, 3	0–1	PI_1
2	010	✓	1, 5	–01	PI_2
4	100	✓	2, 3	01–	PI_3
3	011	✓	2, 6	–10	PI_4
5	101	✓	4, 5	10–	PI_5
6	110	✓	4, 6	1–0	PI_6

FIGURE 3.28 Derivation of prime implicants.

	1	2	3	4	5	6
PI_1	X		X			
PI_2	X				X	
PI_3		X	X			
PI_4		X				X
PI_5				X	X	
PI_6				X		X

FIGURE 3.29 Prime implicant chart.

be seen, there is no simple way to select the minimum number of prime implicants from the cyclic chart. We can proceed by selecting prime implicant PI_1, which covers minterms 1 and 3. After crossing out row PI_1 and columns 1 and 3, we see that PI_4 and PI_5 cover the remaining columns (Fig. 3.30). Thus the minimum sum-of-products form of the given Boolean function is

$$\begin{aligned} f(A, B, C) &= PI_1 + PI_4 + PI_5 \\ &= \bar{A}C + B\bar{C} + A\bar{B} \end{aligned}$$

This is not a unique minimum sum of products for the function. For example,

$$\begin{aligned} f(A, B, C) &= PI_6 + PI_2 + PI_3 \\ &= A\bar{C} + \bar{B}C + \bar{A}B \end{aligned}$$

is also a minimal form of the original function. It can be verified from the Karnaugh map of the function (Fig. 3.31) that these are the minimum sum-of-products forms. Note that each minterm in the Karnaugh map can be grouped within two different loops, which ndicates that two different prime implicants can over the same minterm.

3.4.1 Simplification of Boolean Function with Don't Cares

The Quine–McClusky procedure for minimizing Boolean functions containing don't care minterms is similar to the conventional procedure in that all the terms, including don't cares, are used to produce the complete set of prime implicants. However, don't care terms are not listed as column headings in the prime implicant chart because they need not be included in the final expression.

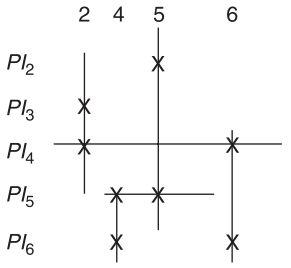


FIGURE 3.30 Examples of adjacencies in Karnaugh.

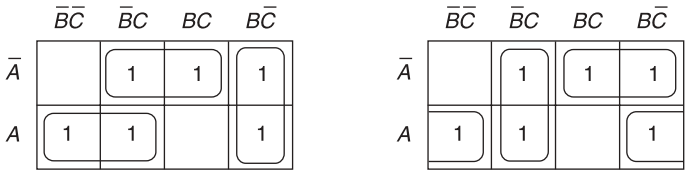


FIGURE 3.31 Examples of adjacencies in Karnaugh.

Example 3.11 Let us minimize the following Boolean function:

$$f(A, B, C, D) = \Sigma m(3, 7, 9, 14) + d(1, 4, 6, 11)$$

Both the minterms and don't cares are listed in the minimizing table and combined in the manner discussed previously.

List 1			*List 2*					
Minterm	*ABCD*		*Minterm*	*ABCD*		*Minterm*	*ABCD*	
1	0001	✓	1,3	00–1	✓	1,3,9,11	–0–1	PI_1
4	0100	✓	1,9	–001	✓			
3	0011	✓	4,6	01–0	PI_2			
6	0110	✓	3,7	0–11	PI_3			
9	1001	✓	3,11	–011	✓			
7	0111	✓	6,7	011–	PI_4			
14	1110	✓	6,14	–110	PI_5			
11	1011	✓	9,11	10–1	✓			

A prime implicant chart is then obtained that contains only the minterms.

	3	*7*	*9*	*14*
PI_1	X		X	
PI_3	X	X		
PI_4		X		
PI_5				X

It can be seen from the chart that PI_1 and PI_5 are essential prime implicants. Since only minterm 7 is not covered by the essential prime implicants, a reduced prime implicant chart is not required. Thus a minimal form of the Boolean function is

$$f(A, B, C, D) = PI_1 + PI_3 + PI_5 = \bar{B}D + \bar{A}CD + BC\bar{D}$$

3.5 CUBICAL REPRESENTATION OF BOOLEAN FUNCTIONS

A variable or its complement in a Boolean expression is known as a *literal*. A *cube* is a product of literals. A sum-of-products Boolean expression can be represented by a list of cubes, one cube for each product term in the expression. In such a representation the cubes are part of the ON-set of the function; each cube in the set produces a logic "1" for the function. For example, the Boolean expression

$$f(a, b, c) = \bar{a}b + a\bar{b}c + \bar{a}\bar{c}$$

can be represented by the ON-set

$$\{\bar{a}b,\ a\bar{b}c,\ \bar{a}\bar{c}\}$$

Since three variables in the function can produce up to eight product terms, cubes that are not in the ON-set belong either to the OFF-set or the DC (don't care)-set. The cubes in the OFF-set produce a logic "0" for the function. For example, the OFF-set for the above function is

$$\{a\bar{b}\bar{c},\ \bar{a}\bar{b}c\}$$

Cube that are not in either the ON-set or the OFF-set belong to the DC-set. Thus the DC-set for the above function is

$$\{ab\}$$

A cube is an *implicant* of a function if it does not contain any member of the OFF-set of the function; the names cubes and implicants are often used interchangeably to define a function. An implicant is *prime* if it is not contained in any other implicant in the function. Similarly, a *prime cube* is not contained by any other cube of the function.

A cube that is the product of all the variables in a function is a *minterm*. For example, $a\bar{b}c$ in the above ON-set is a minterm.

An n-variable Boolean function can be represented by an n-dimensional graph. The graphs for 2- and 3-variable functions are shown in Figure 3.32*a* and 3.32*b*, respectively. Each vertex in a graph corresponds to a minterm of the function and an edge connecting two minterms indicates that they are adjacent, like two adjacent cells in a Karnaugh map. Figure 3.32*c* shows the graph for the above 3-variable function; the minterms corresponding to the function are identified as shaded circles in the graph. The edges in bold indicate that the minterms connected by the edges can be reduced to smaller implicants. Thus minterms $\bar{a}bc$ and $\bar{a}b\bar{c}$ are reduced to $\bar{a}b$, and minterms $\bar{a}b\bar{c}$ and $\bar{a}\bar{b}\bar{c}$ are reduced to $\bar{a}\bar{c}$. Minterm $a\bar{b}c$ cannot be reduced.

A *cover* of a function is a set of cubes, none of which is contained by any cube in the OFF-set of the function. Thus a possible cover of the above function is

$$\{ac, bc, \bar{a}\bar{c}\}$$

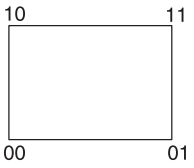
The number of cubes in a cover is known as the *size* or *cardinality* of the cover.

A cover is *irredundant* or *minimal* if the removal of a cube from the corresponding set results in a set that is no longer a cover.

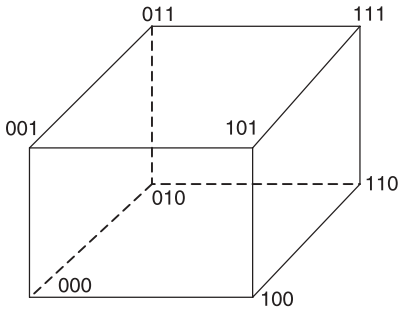
A cover is *prime* if it contains only prime cubes. A cover is *minimum* if there is no other cover that has fewer cubes.

Cubes are usually expressed by replacing each literal with a 2-bit code; the code to be assigned to a literal (l) is determined based on its value as given below:

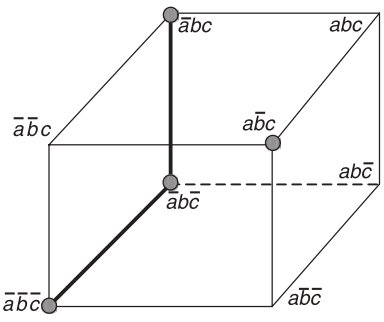
Literal l	Code
0	10
1	01
– (don't care)	11
φ (void)	00



(*a*)



(*b*)



(*c*)

FIGURE 3.32 (*a*) Two-dimensional graph, (*b*) three-dimensional graph, and (*c*) implementation of $f(a, b, c) = \bar{a}b + a\bar{b}c + \bar{a}\bar{c}$.

This is known as *positional cube notation* (PCN) of a cube. If 00 is present in a cube, the cube is removed from the cover of the function.

To illustrate, let us derive the PCN of the cubes belonging to the following cover of a function $\{\bar{a}c, a\bar{b}c, b\bar{c}\}$. In cube $\bar{a}c, a = 0, b = -$, and $c = 1$; thus the PCN of the cube is

10 11 01

The PCN for the remaining cubes in the cover can be determined in a similar manner; thus the PCN for the cover is

10 11 01	$\bar{a}c$
01 10 01	$a\bar{b}c$
11 01 10	bc

For example, let us consider the following function:

$$f(a,b,c) = \bar{a}b + \bar{b}\bar{c} + c$$

The three implicants in the function are represented in PCN as follows [1]:

$\bar{a}b$	10 01 11
$\bar{b}\bar{c}$	11 10 10
c	11 11 01

The PCN representation of cubes simplifies Boolean operations on them. For example, the *intersection* ($\cap$) of two cubes is obtained by taking bit-by-bit AND of the two cubes. The resulting cube contains only the common minterms in both cubes. The pairwise intersections of the above cubes are as follows:

$$\bar{a}b \cap \bar{b}\bar{c} = 10\ 01\ 11 \cap\ 11\ 10\ 10 = 10\ 00\ 10 = 0\,\varphi\,0 \quad \text{void}$$

$$\bar{a}b \cap c = 10\ 01\ 11 \cap 11\ 11\ 01 = 10\ 01\ 01 = 011$$

$$\bar{b}\bar{c} \cap c = 11\ 10\ 10 \cap\ 11\ 11\ 01 = 11\ 10\ 00 = x0\varphi \quad \text{void}$$

Note that the intersections resulted in only a single valid cube.

The *supercube* of two cubes includes only those literals that are in both cubes; it is generated by taking bit-by-bit OR of the two cubes. For example, the supercube corresponding to cubes ab and $b\bar{c}$ is $01\ 01\ 11 \cap 11\ 01\ 10 = 11\ 01\ 11$. The supercube of a set of cubes is derived in a similar manner.

3.5.1 Tautology

Tautology checking is utilized in all currently used computer-aided logic optimization techniques to determine whether a cube is contained in the cover of a function. A function is a tautology if the OFF-set of the function is empty. In other words, the output(s) of the function is 1 for all inputs. For example, the following function is a tautology:

$$f(a,b,c) = bc + ac + \bar{a}\bar{b} + \bar{c}$$

whereas

$$f(a,b,c) = ac + bc + ab$$

is not a tautology because the function produces 1 only for $abc = 111$, 110, 101, and 011.

An efficient approach for checking whether a function is a tautology, without applying all possible inputs, is based on determining whether the *cofactors* of the function with respect to a variable and its complement are a tautology [1]. The cofactor of a function f with respect to a variable x, denoted as f_x, is obtained by replacing x by a "1." Thus for the function

$$f(a,b,c) = \bar{a}\bar{c} + ab + \bar{b}c$$

the cofactor of f with respect to variable a is

$$f_a = 0 \cdot \bar{c} + 1 \cdot b + b\bar{c} = b + b\bar{c} = b + c$$

Similarly, the cofactor of f with respect to $\bar{a}$ is obtained by replacing a with a "0"; thus

$$f_{\bar{a}} = 1 \cdot \bar{c} + 0 \cdot b + \bar{b}c = \bar{c} + \bar{b}c = \bar{b} + \bar{c}$$

Note that a function f can be represented using the cofactors of the function derived using variable x and its complement:

$$f = xf_x + \bar{x}f_{\bar{x}}$$

This representation is known as *Shannon's expansion* of a function with respect to a variable.

The cofactors of a function with respect to a cube are obtained by sequentially deriving the cofactors of the function with respect to each literal in the cube. For example, the cofactor of the above three-variable function with respect to cube $a\bar{b}$ is

$$f_a = b + c$$
$$f_{a\bar{b}} = c$$

If the cofactors of a function with respect to any variable and its complement are a tautology, then the function itself is a tautology. This can be verified from Shannon's expansion. If both f_x and $f_{\bar{x}}$ in the expansion are 1, then

$$f = x + \bar{x} = 1$$

In other words, function f is a tautology.

A set of rules is utilized to determine whether the cover of a function represented in PCN form is a tautology or not.

Rule 1. If the cover contains a row of all 1's, it is a tautology.

Rule 2. If there is a column in the cover that is all 1's or all 0's, it is not a tautology.

Rule 3. If two columns corresponding to a variable have both 0's and 1's and the remaining columns are all 1's, the cover is a tautology.

To illustrate, let us determine whether the following function is a tautology:

$$f(a, b, c, d) = a + \bar{b}d + \bar{a}b + \bar{a}\bar{d}$$

The cover of the function using PCN is

01 11 11 11
11 10 11 01
10 01 11 11
10 11 11 10

None of the above rules are directly applicable to the cover. Since variable a affects more cubes than the other two variables, the cofactors of the function with respect to a and $\bar{a}$ are

derived to check whether they are a tautology. The cofactor of f with respect to a (i.e., f_a) has only one row—11 11 11 11—which by Rule 1 is a tautology.

The cofactor of f with respect to $\bar{a}$ (i.e., $f_{\bar{a}}$) is

11 10 11 01
11 01 11 11
11 11 11 10

None of the above rules are directly applicable to $f_{\bar{a}}$. The cofactors of $f_{\bar{a}}$ with respect to b are derived next; $f_{\bar{a}b}$ has only one row, 11 11 11 11, which is a tautology. The cofactor of $f_{\bar{a}\bar{b}}$ is

11 11 11 01
11 11 11 10

which by Rule 3 is a tautology. Since both $f_{\bar{a}b}$ and $f_{\bar{a}\bar{b}}$ are tautologies, $f_{\bar{a}}$ is also a tautology. Thus both f_a and $f_{\bar{a}}$ are tautologies; hence f is a tautology.

Tautology checking can also be used to determine whether a particular cube is contained in the cover of a function. This is true if the cofactor of the function with respect to the cube is a tautology. For example, let us consider the function

$$f(a, b, c) = \bar{a}c + a\bar{c} + ab$$

To determine whether cube bc is contained in the function, f_{bc} is derived:

10 11 11
01 11 11

which by Rule 3 is a tautology; hence bc is contained in the function.

3.5.2 Complementation Using Shannon's Expansion

Shannon's expansion of a function f with respect to a variable x of the function is given by

$$f = xf_x + \bar{x}f_{\bar{x}}$$

The complement of the expression is

$$\bar{f} = x\bar{f}_x + \bar{x}\bar{f}_{\bar{x}}$$

Notice that $f \cdot \bar{f} = 0$ and $f + \bar{f} = 1$, which proves that $\bar{f}$ is the complement of f.

The procedure to derive the complement of a function f using Shannon's expansion consists of the following steps:

Step 1. Derive the cofactor of f with respect to a variable (x).

Step 2. Derive the cofactor of f_x with respect to another variable (y) and its complement ($\bar{y}$); continue this step with other variables until the results are single cubes or a constant (1 or 0).

Step 3. Compute the complements of the single cubes and form $\bar{f}_x$.

Step 4. Repeat steps 1–3 using variable $\bar{x}$ and form $f_{\bar{x}}$.

Step 5. Combine the results of step 3 to get the complement of f.

To illustrate, let us derive the complement of the following function:

$$f(a, b, c) = a\bar{b} + bc + \bar{a}b\bar{c}$$

The cofactor of f with respect to a is

$$f_a = \bar{b} + c$$

The next step is to derive f_{ab} and $f_{a\bar{b}}$:

$$f_{ab} = c \quad \text{and} \quad f_{a\bar{b}} = 1$$

The complement of f_{ab} and $f_{a\bar{b}}$ are

$$\bar{f}_{ab} = \bar{c} \quad \text{and} \quad \bar{f}_{a\bar{b}} = 0$$

Therefore $\bar{f}_a = 1$ when $a = 1$, $b = 1$, and $c = 0$.

Next, the complement of $f_{\bar{a}}$ is computed. The cofactor of f with respect to $\bar{a}$

$$f_{\bar{a}} = b$$

Thus the complement of $f_{\bar{a}}$ is $\bar{b}$. In other words, $\bar{f}_{\bar{a}}$ is 1 when $a = 0$ and $b = 0$.

Combining the results for $\bar{f}_a$ and $f_{\bar{a}}$, the complement of function f is

$$\bar{f} = ab\bar{c} + \bar{a}\bar{b}$$

Note that the selection of a variable to derive the initial cofactor can be arbitrary; subsequent cofactors can also be derived using randomly selected variables of the function.

3.6 HEURISTIC MINIMIZATION OF LOGIC CIRCUITS

A major problem with the Quine–McCluskey technique is that all prime implicants for a function have to be computed. This becomes computationally very expensive for a function with a large number of inputs. Heuristic minimization techniques apply several operators to minimize a function, thus avoiding the cost of generating all prime implicants. ESPRESSO utilizes such heuristics. However, the resulting minimized functions may not necessarily be minimal. The primary operators used in ESPRESSO are EXPAND, REDUCE, and IRREDUNDANT [1].

3.6.1 EXPAND

The EXPAND operator aims to maximize the *size* of cubes in a cover. The bigger a cube the more minterms it covers, thereby making them redundant. The expansion of a cube involves removing a literal from a cube. To illustrate, let us consider cube $ab\bar{c}$ in

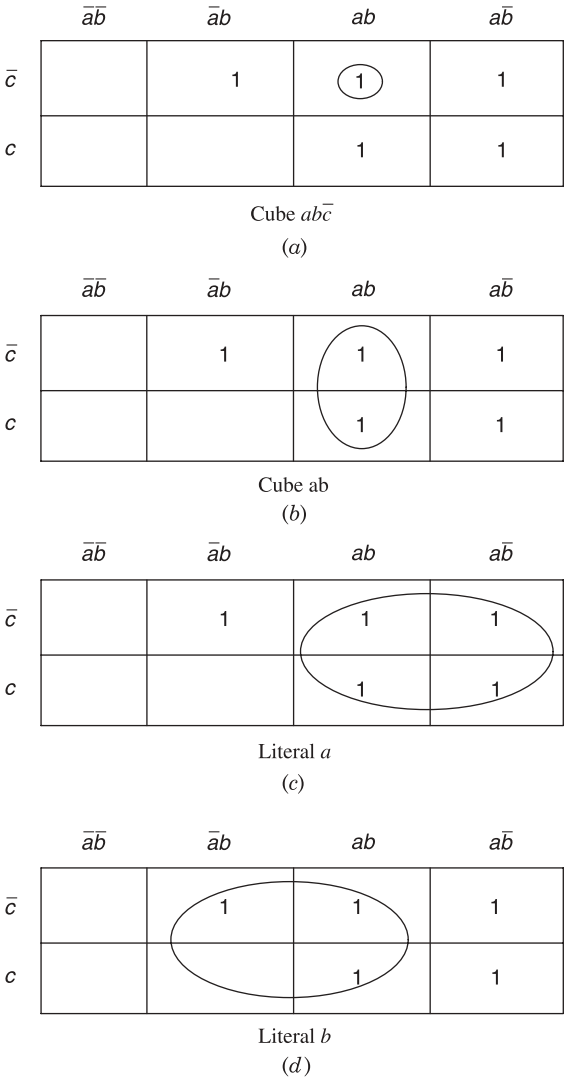


FIGURE 3.33 (*a*) Cube $ab\bar{c}$, (*b*) cube ab, (*c*) literal a, and (*d*) literal b.

Figure 3.33*a*. If literal $\bar{c}$ is removed (i.e., made a don't care in the cube), the resulting cube ab covers minterms $ab\bar{c}$ and abc in the ON-set as shown in Figure 3.33*b*. If literal b is removed from cube ab, the resulting literal a covers minterms $ab\bar{c}$, abc, $a\bar{b}\bar{c}$, and $a\bar{b}c$ as shown in Figure 3.33*c*. However, if instead of literal b, literal a is removed from cube ab, the resulting literal a covers minterms $\bar{a}b\bar{c}$, $ab\bar{c}$, and abc from the ON-set and minterm $\bar{a}bc$ from the OFF-set as shown in Figure 3.33*d*; hence ab cannot be expanded by eliminating literal a.

The order of expansion of cubes in a function is prioritized in terms of their *weights* from low to high. A cube with a lower weight covers fewer minterms than one with a

higher weight. For example, cube $a\bar{b}c$ has lower weight than cube ac. It should be clear that a higher weight cube has more don't cares than a lower weight cube, as can be seen from the positional cube notation of $a\bar{b}c$ and ac:

$$\begin{array}{cc} a\bar{b}c & 01\ 10\ 01 \\ ac & 01\ 11\ 01 \end{array}$$

The weight of a cube is determined using the following steps:

Step 1. Arrange the ON-set of the function as a matrix.

Step 2. Add up the 1's in each column of the matrix and form a sum vector.

Step 3. Transpose the sum vector and multiply the ON-set matrix with the vector; this provides the weight of each cube.

The derivation of the weight is illustrated using the following ON-set:

$$\begin{vmatrix} 10 & 11 & 01 \\ 11 & 01 & 10 \\ 01 & 11 & 10 \end{vmatrix}$$

$$\text{Sum vector} \quad \begin{bmatrix} 22 & 23 & 21 \end{bmatrix}$$

The DC set for the function is assumed to be $\{\bar{a}\bar{b}\bar{c}, \bar{a}bc\}$.

The multiplication of the ON-set matrix by the transposed sum vector results in the following:

$$\begin{vmatrix} 10 & 11 & 01 \\ 11 & 01 & 10 \\ 01 & 11 & 10 \end{vmatrix} * \begin{matrix} 2 \\ 2 \\ 2 \\ 3 \\ 2 \\ 1 \end{matrix} = \begin{matrix} 1*2+0*2+1*2+1*3+0*2+1*1=8 \\ 1*2+1*2+0*2+1*3+1*2+0*1=9 \\ 0*2+1*2+1*2+1*3+1*2+0*1=9 \end{matrix}$$

Since cube 10 11 01 has the lowest weight, it is expanded first. The expansion of a cube is performed by changing the 0-bit in the two-bit representation of a literal to a 1; this is done from left to right for simplicity. Thus changing the 0 in the second column of 10 11 01 yields

$$11\ \ 11\ \ 01$$

This cube in addition to covering minterm $\bar{a}\bar{b}c$ from the ON-set and $\bar{a}bc$ from the DC-set covers minterms abc and $a\bar{b}c$ from the OFF-set; hence the expanded cube 11 11 01 is invalid.

Changing column 5 rather than column 2 in the cube 10 11 01 yields

10 11 11

which covers minterms $\bar{a}b\bar{c}$ and $\bar{a}\bar{b}c$ from the ON-set and minterms $\bar{a}\bar{b}\bar{c}$ and $\bar{a}bc$ from the DC-set. Thus the expanded cube 10 11 11 is valid and the original ON-set can be updated to

10 11 11
11 01 10
01 11 10

The weights of rows in the set are computed as discussed previously. The first row has weight 11, whereas both the second and third rows have weight 10. The cube corresponding to the second row (i.e., 11 01 10) is expanded first. Changing the 0 in the third column of the cube to a 1 yields

11 11 10

This cube covers minterms from the ON-set and the DC-set but not any from the OFF-set, and therefore is valid. On the other hand, changing the 0 in the sixth column of cube 11 01 10 to a 1 will result in a cube that will cover a minterm (abc) from the OFF-set. Thus only the expanded cube 11 11 10 is valid. Note that this cube also covers cube 01 11 10 in the ON-set. Thus the expanded ON-set for the original function is

10 11 11
11 11 10

3.6.2 REDUCE

The REDUCE operator decreases the size of each cube in the ON-set of a function so that any later expansion may lead to another ON-set with fewer cubes in it. The following steps are needed by the reduce operation:

Step 1. Compute the weights of the cubes in EXPAND operation.

Step 2. Select cubes one at a time in the high to low weight order (cubes with high weights cover more minterms as explained previously).

Step 3. Intersect the selected cube with the complements of the rest of the cubes in the ON-set.

Step 4. Include the resulting cube in the modified ON-set. If more than one cube is generated, they are replaced by a *supercube* containing all these cubes.

The function of the REDUCE operator is illustrated by applying it to the following function:

$$f = \bar{a}\bar{c} + \bar{c}d + bd$$

Therefore the ON-set (i.e., the cover of the function) is

$$\begin{array}{llll} 10 & 11 & 10 & 11 \quad C_1 \\ 11 & 11 & 10 & 01 \quad C_2 \\ 11 & 01 & 11 & 01 \quad C_3 \end{array}$$

which is prime.

The vector corresponding to the number of 1's in the columns of the above matrix is [32 23 31 13] and weights for cubes C_1, C_2, and C_3 are 15, 16, and 15, respectively. Cube C_2 has higher weight than cube C_1 and C_3. Therefore C_2 is reduced first. The cubes in the OFF-set derived without including C_2 in the ON-set will be

$$\begin{array}{llll} 01 & 11 & 10 & 10 \\ 11 & 10 & 01 & 01 \\ 11 & 11 & 10 & 01 \\ 11 & 11 & 01 & 10 \end{array}$$

The intersection of C_2 with the OFF-set yields the following cubes:

$$\begin{aligned} &01 \cdot 11 + 11 \cdot 11 + 10 \cdot 10 + 10 \cdot 01 = \phi \\ &11 \cdot 11 + 10 \cdot 11 + 01 \cdot 10 + 01 \cdot 01 = \phi \\ &11 \cdot 11 + 11 \cdot 11 + 10 \cdot 10 + 01 \cdot 01 = 11\ 11\ 10\ 01 \\ &11 \cdot 11 + 11 \cdot 11 + 01 \cdot 10 + 10 \cdot 01 = \phi \end{aligned}$$

As can be seen above, the intersection results in cube C_2 itself; thus there is no change in the ON-set.

Either C_1 or C_3 can be selected next because both have equal weights. The intersection of C_1 with the OFF-set (derived without including C_1 in the ON-set) is obtained in a similar manner as for C_2:

$$\begin{aligned} &01 \cdot 10 + 11 \cdot 11 + 10 \cdot 10 + 11 \cdot 11 = \phi \\ &10 \cdot 10 + 11 \cdot 11 + 10 \cdot 10 + 01 \cdot 11 = 10\ 11\ 10\ 01 \\ &11 \cdot 10 + 10 \cdot 11 + 01 \cdot 10 + 11 \cdot 01 = \phi \\ &11 \cdot 10 + 01 \cdot 11 + 01 \cdot 10 + 10 \cdot 11 = \phi \end{aligned}$$

The resulting cube 10 11 10 01 ($\bar{a}\bar{c}d$) replaces C_1 in the ON-set.

Finally, the intersection of C_3 with the OFF-set (derived without including C_3 in the ON-set) is obtained:

$$\begin{aligned} &01 \cdot 11 + 11 \cdot 01 + 10 \cdot 11 + 10 \cdot 01 = \phi \\ &11 \cdot 11 + 11 \cdot 01 + 01 \cdot 11 + 11 \cdot 01 = 11\ 01\ 01\ 01 \end{aligned}$$

Therefore cube C_3 can replaced by the new cube bcd.

Thus the original ON-set after the reduce operation becomes

10 11 10 01
11 11 10 01
11 01 01 01

Note that the cardinality of the reduced ON-set (i.e., the cover) is the same as the original cover of the function. However, the cover is not prime.

3.6.3 IRREDUNDANT

The IRREDUNDANT operator removes redundant implicants from the cover of a function to reduce the cardinality of the cover. A cover is composed of three subsets of implicants: *relatively essential prime implicants* (R_E), *partially redundant prime implicants* (R_P), and *totally redundant prime implicants* (R_T).

R_E's cover minterms of a function that are not covered by any other prime implicants in this cover (same as essential prime implicants discussed in Section 3.4). All implicants in an R_T are covered by the R_E. R_P's are those that are not included in either R_P's or R_T's. For example, in the following cover of the function $f(a,b,c) = ab + \bar{a}c + a\bar{c} + bc$

01 01 11 C_1 (ab)
10 11 01 C_2 $(\bar{a}c)$
01 11 10 C_3 $(a\bar{c})$
11 01 01 C_4 (bc)

C_2 and C_3 are relatively essential, and C_1 and C_4 are partially redundant; there are no totally redundant prime implicants.

The objective of the IRREDUNDANT operator is to determine which of the partially redundant prime implicants can be removed from the cover. The remaining partially redundant prime implicants together with relatively essential prime implicants will be the cover of the function with minimum cardinality (i.e., fewest number of cubes in the cover). It can be seen from the Karnaugh map of the function in Figure 3.34 that prime implicant C_4 can be eliminated from the cover.

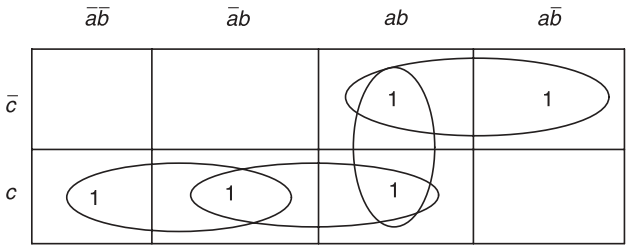


FIGURE 3.34 Karnaugh map $f(a, b, c) = ab + \bar{a}c + a\bar{c} + bc$.

In CAD-based techniques for logic minimization (e.g., ESPRESSO), a modification of the tautology check is used to determine whether an implicant α from the partially redundant subset is contained in the following set S [1], where

$$S = \{\text{relatively essential set of prime implicants}\} \cup \{\text{partially redundant set of prime implicants}\} \cup \{\text{don't care set of the function}\} - \{\alpha\}$$

A cube x is contained in a set of cubes C (i.e., C covers x) if and only if C_x (cofactor of C with respect to x) is a tautology. Thus if S_α is a tautology then α is covered by S. This is illustrated by using the cover of the function considered in the previous section for the REDUCE operator:

01	01	11	C_1	partially redundant
10	11	01	C_2	relatively essential
01	11	10	C_3	relatively essential
11	01	01	C_4	partially redundant

If C_1 is considered first, then S becomes $\{C_1, C_2, C_3, C_4\} - \{C_1\}$; note that this function does not have a don't care set. Thus S becomes

10	11	01	C_2
01	11	10	C_3
11	01	01	C_4

The corresponding logic expression for S is

$$S = \bar{a}c + a\bar{c} + bc$$

then the cofactor of S with respect to $C_1(=ab)$ is

$$S_{ab} = \bar{c} + c$$

which in PCN is

11	11	10
11	11	01

Since S_{ab} depends only on a single variable (c) and there is no column of all 0's in S_{ab}, S_{ab} is a tautology. In other words, C_1 is covered by S and can be removed from the cover of the function.

Next, cube C_4 is considered for possible elimination from the cover. Then S becomes

01	01	11	C_1
10	11	01	C_2
01	11	10	C_3

and the corresponding logic expression is

$$S = ab + \bar{a}c + a\bar{c}$$

The cofactor of S with respect to C_4 $(=bc)$ is

$$S_{bc} = \bar{a} + a$$

Thus S_{bc} in PCN is

10 11 11
01 11 11

which is also a tautology; hence C_4 can be removed from the cover of the function. Thus either cube C_1 or cube C_4 can be removed from the cover of the function; the resulting irredundant covers of the function as shown below have a cardinality of 3:

10	11	01		01	01	11
01	11	10	or	10	11	01
11	01	01		01	11	10

3.6.4 ESPRESSO

ESPRESSO performs the minimization of a Boolean function specified in terms of its ON-set, OFF-set, and DC-set. The implicants in the ON-set represent the initial unoptimized cover of the function. By successively applying REDUCE, EXPAND, and IRREDUNDANT operators in a loop, ESPRESSO finds the near-minimum cover of the function.

To illustrate, let us consider the optimization of the following function using ESPRESSO:

$$f(a, b, c, d) = \bar{b}\bar{d} + a\bar{c}d + bcd + \bar{a}c\bar{d} + bc\bar{d}$$

Figure 3.35 shows the Karnaugh map of the four-variable function.

The PCN representation of the cover of the function is

11	10	11	10	α
01	11	10	01	β
11	01	01	01	ψ
10	11	01	10	γ
11	01	01	10	δ

It is not possible to reduce cubes α, β, ψ, or γ because in each case the intersection of the cube with the complement of the cover without the cube results in two cubes. However, the intersection of cube δ with the complement of the cover without δ results

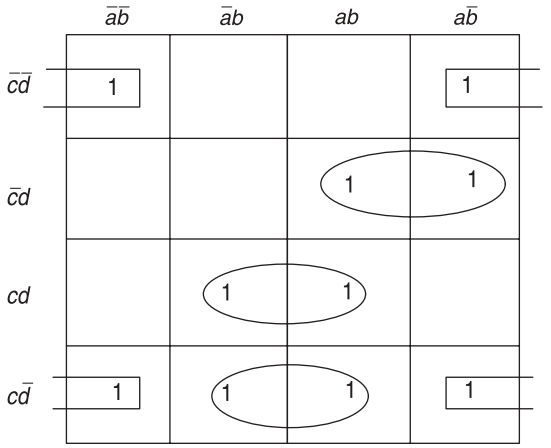


FIGURE 3.35 Karnaugh map for the original function.

in 01 01 01 10 ($abc\bar{d}$). Thus the new cover is

$$\begin{array}{cccc} 11 & 10 & 11 & 10 \quad \alpha \\ 01 & 11 & 10 & 01 \quad \beta \\ 11 & 01 & 01 & 01 \quad \psi \\ 10 & 11 & 01 & 10 \quad \gamma \\ 01 & 01 & 01 & 10 \quad \delta \end{array}$$

Figure 3.36 shows the Karnaugh map of the function after the REDUCTION step.

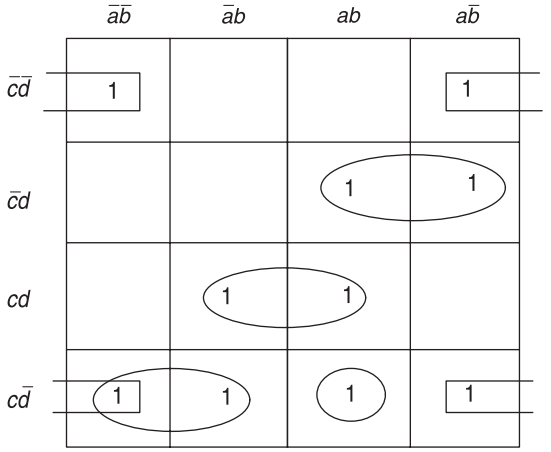


FIGURE 3.36 Karnaugh map after REDUCTION step.

Next, the EXPAND operation is used to determine whether the cubes in the cover can be expanded. The weights of the cubes are determined as discussed previously and are as follows:

$$\begin{array}{ll} \alpha & 19 \\ \beta & 15 \\ \psi & 17 \\ \gamma & 17 \\ \delta & 15 \end{array}$$

The lightest cubes are β and δ. Cube β is expanded first, but the expansion is not valid. Next, δ is expanded by raising literal a first followed by literal b; the resulting cube is 11 11 01 10 ($c\bar{d}$), which covers cube γ. Cube ψ is expanded next to (11 01 01 11) by raising literal d. Cube α cannot be raised with respect to $\bar{b}$ or $\bar{d}$; hence it cannot be expanded. The reduced and expanded cover for the function becomes

$$\begin{array}{cccc} 11 & 10 & 11 & 10 \\ 01 & 11 & 10 & 01 \\ 11 & 01 & 01 & 11 \\ 11 & 11 & 01 & 10 \end{array}$$

Figure 3.37 shows the Karnaugh map of the function after the EXPAND step.

The IRREUNDANT operation is used next. The relatively essential redundant set of implicants in the cover is

$$\{11\ \ 10\ \ 11\ \ 10,\ \ 11\ \ 01\ \ 01\ \ 11,\ \ 01\ \ 11\ \ 10\ \ 01\}$$

The implicant 11 11 01 10 is totally redundant and can be removed from the cover; there are no partially redundant implicants in the cover.

Figure 3.38 shows the Karnaugh map after the IRREDUNDANT step.

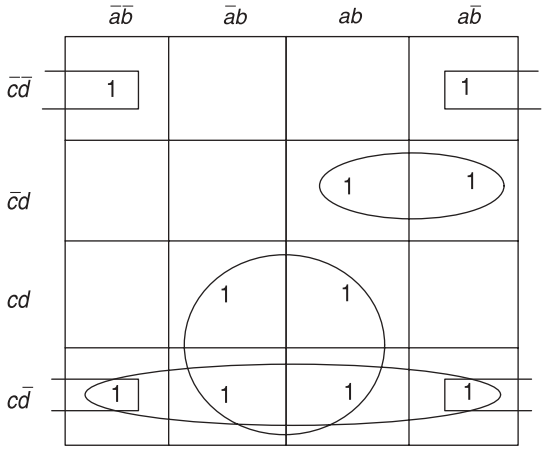


FIGURE 3.37 Karnaugh map after EXPAND step.

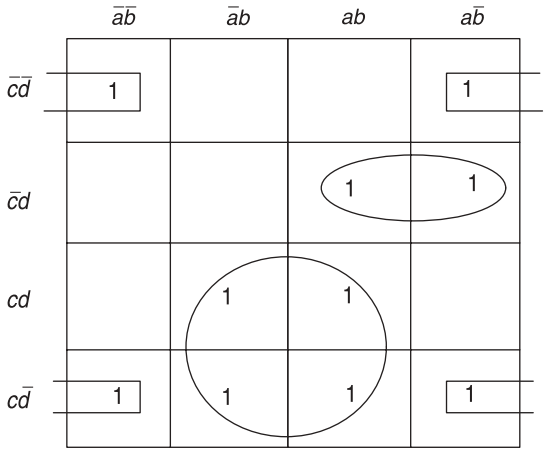


FIGURE 3.38 Karnaugh map after IRREDUNDANT step.

Further iteration of the REDUCE–EXPAND–IRREDUNDANT does not reduce the size of the cover. Thus the minimized function is

$$f(a,b,c,d) = \bar{b}\bar{d} + bc + a\bar{c}d$$

3.7 MINIMIZATION OF MULTIPLE-OUTPUT FUNCTIONS

Multiple-input/multiple-output combinational logic blocks are frequently used in complex logic systems, for example, VLSI (very large scale integrated) chips. The area occupied by such combinational logic blocks has a significant impact on the VLSI design objective of putting the maximum amount of logic in the minimum possible area.

In minimizing multiple-output functions, instead of considering individual single-output functions, the emphasis is on deriving product terms that can be shared among the functions. This results in a circuit having fewer gates than if each function is minimized independently. For example, if the following two functions are individually minimized, the resulting circuit will be as shown in Figure 3.39*a*:

$$f_1 = ab\bar{c}\bar{d} + ab\bar{c}d + abcd + a\bar{b}cd$$

$$f_2 = ab\bar{c}\bar{d} + ab\bar{c}d + abcd + a\bar{b}cd + a\bar{b}\bar{c}\bar{d} + a\bar{b}\bar{c}d$$

However, if the functions are minimized as shown in Figure 3.39*b*, one product term (i.e., *acd*) can be shared among the functions, resulting in a circuit with one fewer gate.

As is clear from this example, the determination of *shared* product terms from among many Boolean functions is an extremely complicated task. This can only be done efficiently by using a computer-aided minimization technique. For example, the two-level minimizer ESPRESSO, in general, identifies the shared terms reasonably well.

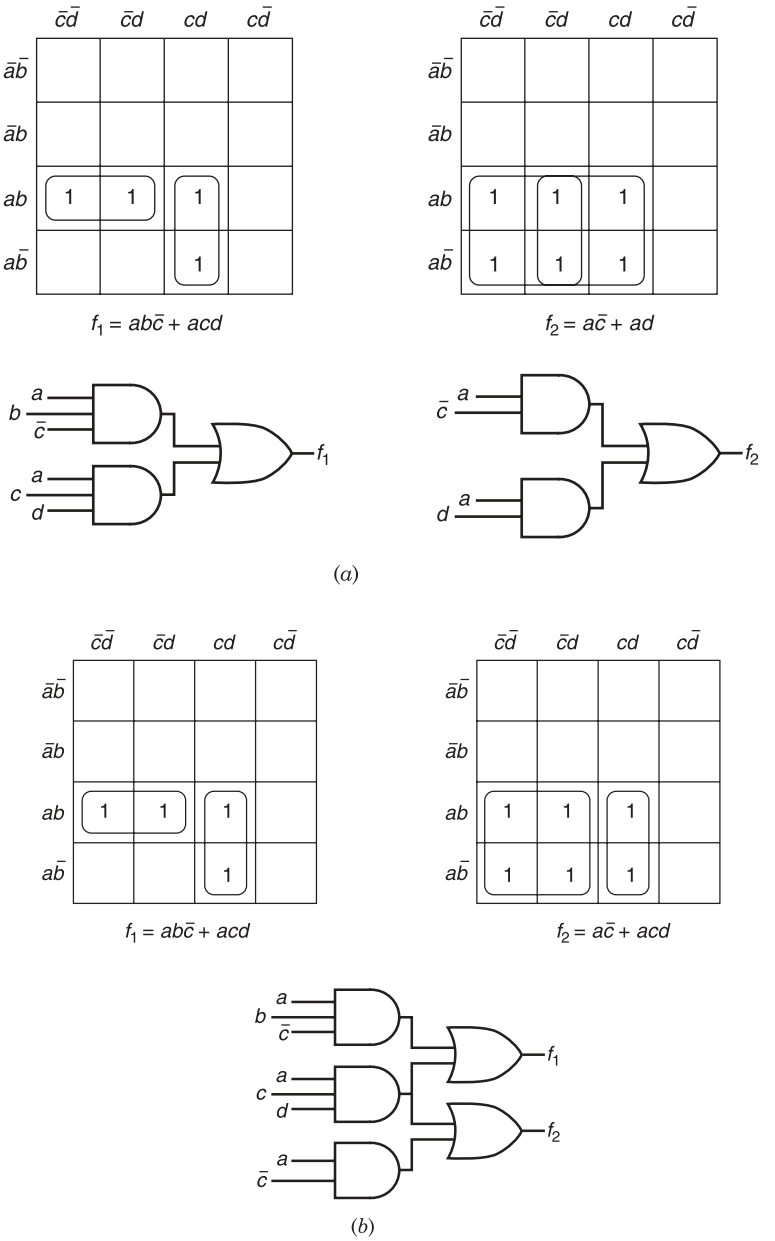


FIGURE 3.39 (*a*) Individual minimization of f_1 and f_2 and (*b*) sharing of a product term between f_1 and f_2.

Example 3.12 Let us consider the following functions:

$$f_1 = \Sigma m(0, 2, 4, 5, 9, 10, 11, 13, 15) \tag{3.1}$$

$$f_2 = \Sigma m(2, 5, 10, 11, 12, 13, 14, 15) \tag{3.2}$$

$$f_3 = \Sigma m(0, 2, 3, 4, 9, 11, 13, 14, 15) \tag{3.3}$$

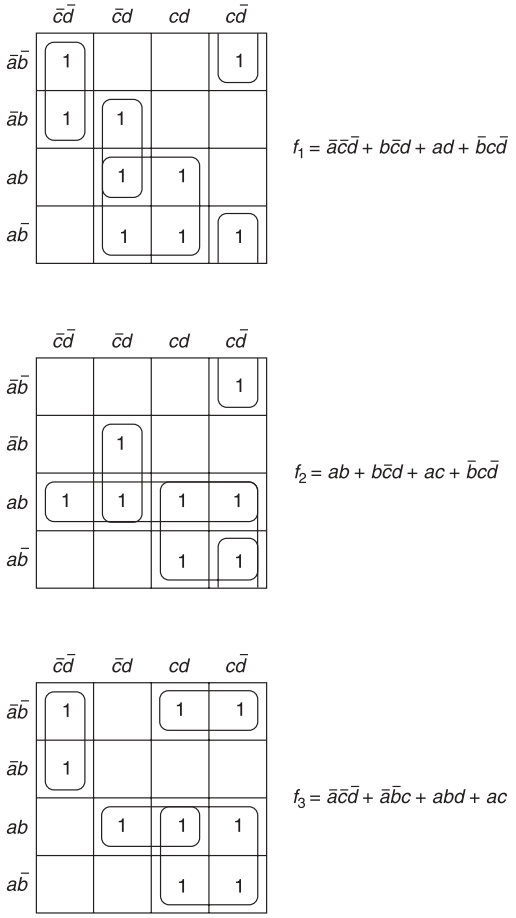


FIGURE 3.40 Karnaugh maps for functions f_1, f_2, and f_3.

The Karnaugh maps for the functions are shown in Figure 3.40. Individual minimization of the functions results in the following shared terms:

Term	Functions
$\bar{a}\bar{c}\bar{d}$	f_1, f_3
$b\bar{c}d$	f_1, f_2
ac	f_2, f_3
$\bar{b}c\bar{d}$	f_1, f_2

There are 12 product terms in the original expressions, out of which 4 can be shared. The expression can be rewritten as follows:

$$f_1 = W + X + ad + Z$$
$$f_2 = ab + X + Y + Z$$
$$f_3 = W + \bar{a}\bar{b}c + Y + abd$$

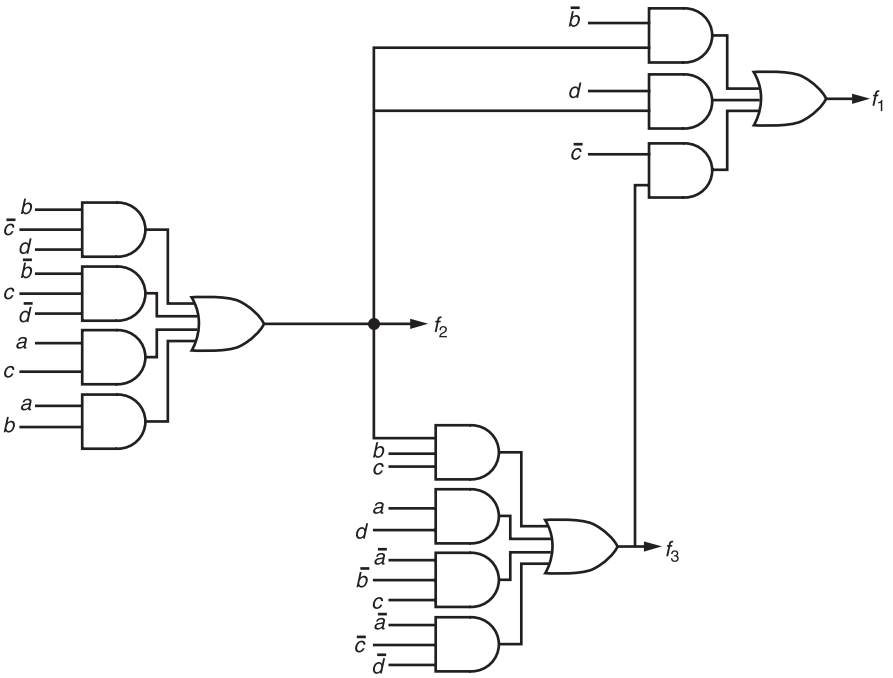


FIGURE 3.41 Multilevel implementation of the simplified expressions.

where $W = \bar{a}\bar{c}\bar{d}$, $X = b\bar{c}d$, $Y = ac$, and $Z = \bar{b}c\bar{d}$.

$$f_1 = \bar{c}f_3 + df_2 + \bar{b}f_2$$
$$f_2 = b\bar{c}d + \bar{b}c\bar{d} + ac + ab$$
$$f_3 = \bar{a}\bar{c}\bar{d} + \bar{a}\bar{b}c + f_2bc + ad$$

There are 27 literals in these equations, which can be further reduced to 23 by using factoring. The resulting circuit is shown in Figure 3.41.

3.8 NAND–NAND AND NOR–NOR LOGIC

So far, we have discussed various ways of simplifying Boolean functions. A simplified sum-of-products function can be implemented by AND–OR logic, and a simplified product-of-sums function can be implemented by OR–AND logic. This section shows that any logic function can be implemented using only one type of gate—either the NAND or the NOR gate. This has the advantage of standardizing the components required to realize a circuit.

3.8.1 NAND–NAND Logic

All Boolean functions in sum-of-products form can be realized by a two-level logic involving only NAND gates. Consider the following Boolean function in sum-of-products form:

$$f(A, B, C) = \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C + AB\bar{C}$$

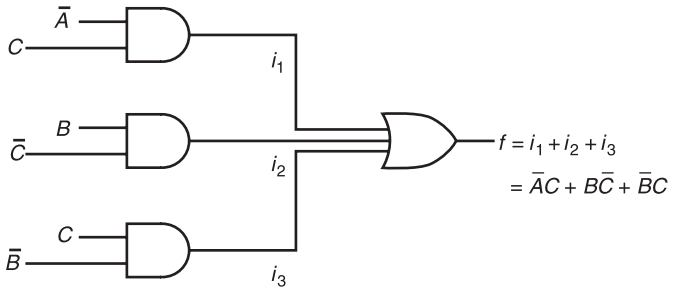


FIGURE 3.42 AND–OR logic implementation of *f*.

The first step in the design is to minimize the function. This can be done by algebraic manipulation:

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

The minimized function can be realized directly in AND–OR logic as shown in Figure 3.42. The NAND–NAND form of the function can be derived directly from the AND–OR form as illustrated in Figure 3.43. First, two inverter gates in series are inserted at each input of the OR gate (Fig. 3.43*a*). The output of the circuit remains unchanged in spite of the incorporation of the inverter gates because the output of each AND gate is inverted twice. Then the first level inverter gates are combined with the AND gates to form NAND gates. The second level inverter gates are combined with the OR gate to form a NAND gate (since by DeMorgan's theorem $i_1 + i_2 + i_3 = \overline{\bar{i}_1 + \bar{i}_2 + \bar{i}_3}$) as shown in Figure 3.43*b*.

Thus any Boolean function in sum-of-products form can be implemented by two levels of NAND gates. However, it should be noted that such an implementation is based on the assumption that double-rail inputs are available. *Double-rail inputs* to a circuit indicate that each variable and its complement can be used as inputs to the circuit. If the

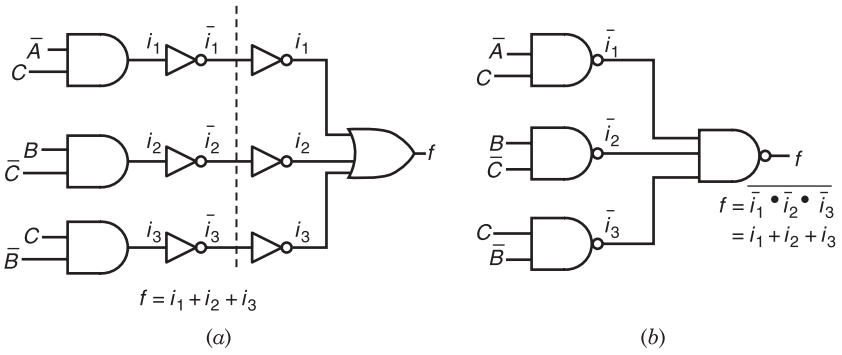


FIGURE 3.43 NAND–NAND implementation of *f*.

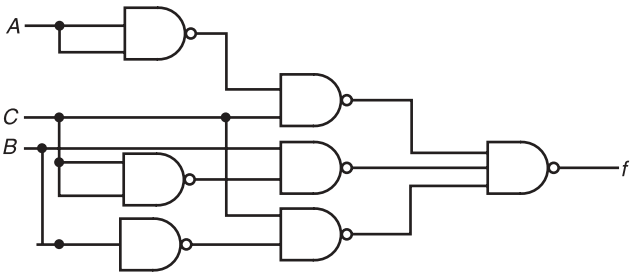


FIGURE 3.44 Three-level NAND gate representation of the circuit of Figure 3.42.

complements of the variables are not available, the circuit inputs are *single-rail*. Any Boolean function can be realized with three-level NAND gates using only single-rail inputs. For example, the circuit of Figure 3.42 can be implemented in three-level NAND gates as shown in Figure 3.44.

The two-level NAND implementation of a sum-of-products Boolean function can be obtained by using the following steps in sequence:

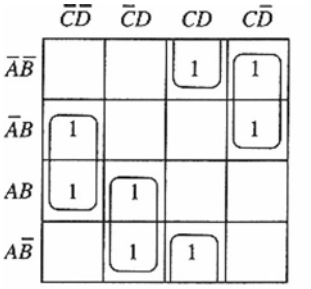
Step 1. Take the complement of the minimized expression.

Step 2. Take the complement of the complemented expression. Eliminate the OR operator from the resulting expression by applying DeMorgan's theorem.

Example 3.13 Let us implement the following function using two-level NAND logic:

$$f(A, B, C, D) = \Sigma m(2, 3, 4, 6, 9, 11, 12, 13)$$

The minimal form of the given function can be derived from its Karnaugh map:



Hence

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

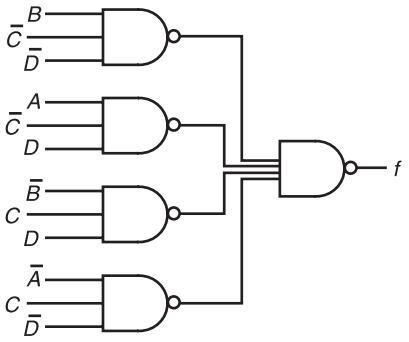


FIGURE 3.45 NAND–NAND realization of $f = \Sigma m(2, 3, 4, 6, 9, 11, 12, 13)$.

The complement of the expression for $\bar{f}$ is then derived:

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

$$\therefore \quad f = \overline{\overline{B\bar{C}\bar{D}} \cdot \overline{A\bar{C}D} \cdot \overline{\bar{B}CD} \cdot \overline{\bar{A}C\bar{D}}}$$

The NAND–NAND realization of the above expression is shown in Figure 3.45.

3.8.2 NOR–NOR Logic

All Boolean functions expressed in product-of-sums form can be implemented using two-level NOR logic. Let us assume we have the OR–AND implementation of the Boolean function $f(A, B, C) = (A + C)(A + \bar{B})(\bar{A} + C)$. Figure 3.46 shows the OR–AND implementation. The OR–AND circuit can be converted to the circuit of Figure 3.47*a* by inserting a cascade of two inverters at each input of the AND gate. Then a NOR–NOR realization of the circuit can be obtained as shown in Figure 3.47*b*.

The NOR–NOR implementation of a Boolean function expressed in sum-of-products form can be obtained by the following two steps:

Step 1. Derive the complementary sum-of-products version of the original expression.

Step 2. Take the complement of this complemented sum-of-products expression. Eliminate the AND operators from the resulting expression by using DeMorgan's theorem.

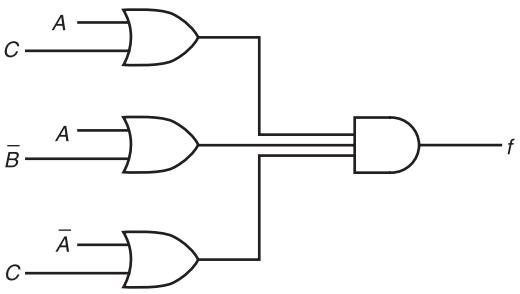


FIGURE 3.46 OR–AND logic implementation.

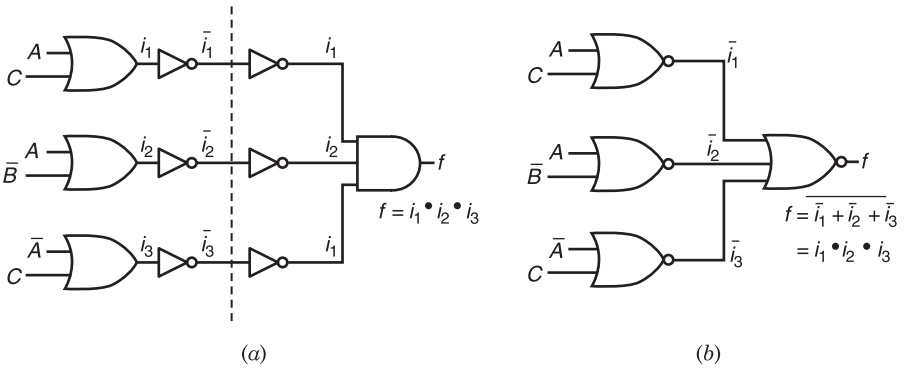


FIGURE 3.47 Conversion from OR–AND to NOR–NOR logic.

Example 3.14 Let us implement the following Boolean function in NOR–NOR logic:

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

The complementary sum-of-products expression, $\bar{f}$, can be obtained from the Karnaugh map of the function by grouping the 0's.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	0	1	1
$\bar{A}B$	1	1	1	1
AB	0	1	0	0
$A\bar{B}$	0	1	1	0

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

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

The resulting NOR–NOR logic circuit is shown in Figure 3.48.

3.9 MULTILEVEL LOGIC DESIGN

Multilevel logic, as the name implies, uses more than two levels of logic to implement a function. Two-level implementation of a function is often difficult to implement at the gate

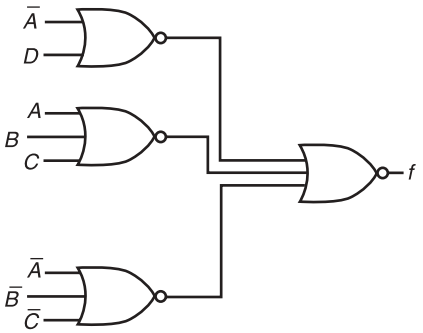


FIGURE 3.48 NOR–NOR implementation of $f(A, B, C, D) = \bar{A}B + \bar{A}C + A\bar{C}D + A\bar{B}D$.

level because of the fan-in restrictions. For example, two-level implementation of the following minimized function,

$$f(u, v, w, x, y, z) = \bar{u}\bar{v}\bar{w}\bar{y}\bar{z} + \bar{u}\bar{w}xyz + u\bar{v}wx\bar{y}z + uv\bar{w}x\bar{y} + \bar{u}\bar{v}w\bar{x}y + \bar{u}\bar{x}\bar{y}\bar{z} + \bar{v}\bar{w}\bar{x}\bar{z} \tag{3.4}$$

will require seven AND gates (one with fan-in of 6, four with fan-in of 5, and two with fan-in of 4), and one OR gate with fan-in of 7. However, by increasing the number of levels in the circuit, the fan-in of the gates can be reduced.

The starting point of the multilevel implementation of a function is the minimized two-level representation of the function. Several operations are used to manipulate this two-level representation; these include *decomposition*, *extraction*, *substitution*, *collapsing*, and *factoring*.

Decomposition is the process of representing a single expression as a collection of several subfunctions. For example, the decomposition of the function

$$f = ad + bd + \bar{a}\bar{b}c + bc\bar{d} + bce$$

will result in the following subfunctions:

$$f = dY + c\bar{Y} + X(\bar{d} + e)$$
$$X = bc$$
$$Y = a + b$$

In general, the decomposition increases the number of levels of a logic circuit while decreasing the fan-in of the gates used to implement the circuit.

The extraction operation creates some intermediate node variables for a given set of functions. These node variables together with the original variables are then used to re-express the given function. The extraction operation applied to the following functions

$$z_1 = ae + be + c$$
$$z_2 = af + bf + d$$

will yield

$$z_1 = Xe + c$$
$$z_2 = Xf + d$$
$$X = a + b$$

where X is a fan-out node.

The substitution process is used to determine whether a given function can be expressed as a function of its original inputs and other functions. As an example, let us consider the functions

$$X = ab + bd + ac + cd$$
$$Y = b + c$$

Substituting Y in X produces

$$X = Y(a + d)$$

This is in fact an example of *algebraic substitution* since $(b + c)$ is an algebraic divisor of X. If the expression is multiplied out, the resulting expression will be identical to the original form.

Another type of substitution is the *Boolean substitution*, which creates logic functions by using Boolean division. Thus the original and substituted expressions may not have the same form but they are logically equivalent. For example, algebraic substitution does not simplify the following functions:

$$X = b + ac + \bar{a}\bar{b}$$
$$Y = b + c$$

However, by using Boolean substitution X can be rewritten

$$X = (b + c)(a + b) + \bar{a}\bar{b}$$

The inverse operation of substitution is known as *collapsing* or *flattening*. For example, if

$$X = Y(c + d) + e$$
$$Y = a + b$$

then collapsing Y into X results in

$$X = ac + bc + ad + bd + e$$
$$Y = a + b$$

Thus if Y is an internal node in the circuit, it can be removed.

Factoring is the conversion of a function in the sum-of-products form to a form with parentheses and having a minimum number of literals. A straightforward approach for deriving the factored form of a function from its given two-level representation is to select a literal that is common to the maximum number of product terms. This results in partitioning of the product terms into two sets—one set contains the product terms having the literal, the other contains the rest of the product terms. If the literal is factored out from the first set, a new two-level expression results, which is then ANDed with the literal. Similarly, the second set of product terms is evaluated for a common literal. This process is repeated for the newly generated two-level expressions until they cannot be factored any further. The resulting expression is a multilevel representation of the original two-level form. By using this approach, the factored version of the two-level expression, Eg. (3.4), is derived:

$$f(u, v, w, x, y, z) = \bar{y}[\bar{u}\bar{z}(\bar{v}\bar{w} + \bar{x}) + ux(\bar{v}wz + v\bar{w})] + \bar{u}y(\bar{w}xz + \bar{v}w\bar{x}) + \bar{v}\bar{w}\bar{x}\bar{z} \ldots \quad (3.5)$$

Note that the original two-level expression has 34 literals, whereas the factored form has 25 literals. As can be seen from the factored expression, such a representation of a Boolean function automatically leads to the multilevel realization of the function.

In general, a factored-form representation of a two-level function is not unique. For example, the preceding six-variable function can also be represented in the factored form as

$$f(u, v, w, x, y, z) = \bar{z}[\bar{u}\bar{y}(\bar{w}\bar{v} + \bar{x}) + \bar{v}\bar{w}\bar{x}] + xz(\bar{u}\bar{w}y + u\bar{v}wy) + uv\bar{w}x\bar{y} + \bar{u}\bar{v}w\bar{x}y$$

which has 28 literals. Obviously, only the factored form with the fewest number of literals has to be selected in order to guarantee a minimal multilevel implementation.

3.9.1 Algebraic and Boolean Division

Let us assume two Boolean expressions f and g. If there is an operation that generates expressions h and r such that $f = gh + r$, where gh is an algebraic product (i.e., g and h have no common variable), then this operation is called an *algebraic division*. For example, if $f = wy + xy + yz$ and $g = w + x$, a polynomial division will yield

$$f = gh + r = y(w + x) + yz$$

Note that this factored-form representation is algebraically equivalent to the original sum-of-products expression. In other words, if the algebraic factor is expanded, exactly the same set of terms as in f will be obtained.

Another form of division used in factoring Boolean expressions uses the identities of Boolean algebra (e.g., $x\bar{x} = 0$, $xx = x$, and $x + \bar{x} = 1$ for variable x). Thus if in the expression $f = gh + r$, gh is a Boolean product (i.e., g and h have one or more common variable(s)), then the division of f by g is called a *Boolean division*. For example, if $f = abd + bcd + \bar{a}c + \bar{b}\bar{d}$ and $g = a + c$, the use of Boolean division will yield

$$f = gh + r = (bd + \bar{a})(a + c) + \bar{b}\bar{d}$$

whereas algebraic division will produce

$$f = gh + r = bd(a + c) + \bar{a}c + \bar{b}\bar{d}$$

3.9.2 Kernels

The quotient resulting from an algebraic division of an expression f by a cube c (i.e., f/c) is the kernel k of f, if there are at least two cubes in the quotient and the cubes do not have any common literal. The cube divisor c used to obtain the kernel is called its *cokernel*. Different cokernels may produce the same kernel, therefore the cokernel of a kernel is not unique. If a kernel has no kernels except itself, it is said to be a *level-0 kernel*. A kernel is of level n if it has at least one level-$(n-1)$ kernel but no kernel, except itself, of level n or greater.

Example 3.15 Let us consider the Boolean expression

$$f(a, b, c, d) = \bar{a}\bar{c}d + \bar{a}bc + abd + ab\bar{c} + bcd$$

The quotient of f and the cube a is

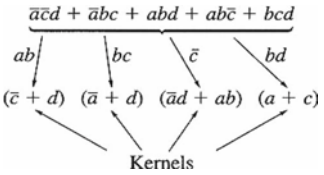
$$f/a = bd + b\bar{c}$$

but since literal b is common to both cubes, f/a is not a kernel of f.

The quotient of f and the cube $\bar{c}$ is

$$f/c = \bar{a}d + ab$$

It has two cubes and no common literal; hence it is a kernel. The cokernel of $f/\bar{c}$ is $\bar{c}$. The kernels and the corresponding cokernels of the function are represented by the tree shown below, where the leaves of the tree are kernels and the branches are the corresponding cokernels. Note that all of these kernels are of level 0.



If the original expression is rewritten as

$$f = b[a(\bar{c} + d) + c(\bar{a} + d)] + \bar{c}(\bar{a}d + ab)$$

then $b(a + c) + \bar{a}\bar{c}$ is kernel of level 1 corresponding to the cokernel d because it contains a level-0 kernel $(a + c)$.

As mentioned preciously, the cokernel of a kernel is not unique. For example,

$$f = ad + bd + ac + bc$$

has a kernel $(a + b)$ obtained by using cokernels c and d.

Kernels can be used to derive common subexpressions in two Boolean expressions. (The intersection of two kernels k_1 and k_2 is defined as the set of cubes present in both k_1 and k_2.) If there is a kernel intersection of more than one cube, then two Boolean expressions will have common subexpressions of more than one cube.

Example 3.16 Let us consider two Boolean expressions

$$f_1 = abc + a\bar{c}g + \bar{b}df + cde$$

$$f_2 = \bar{a}b\bar{d} + bc\bar{e} + \bar{b}de + \bar{b}\bar{g} + \bar{c}\bar{e}g$$

The kernels of f_1 and f_2 are shown in Table 3.2

The kernels of expression f_1 are intersected with those of expression f_2 to find terms that are identical between pairs of kernels. For example, kernel $ab + de$ in f_1 intersects with $de + \bar{g}$ in f_2 have a common terms de. Note that f_1 and f_2 have a common kernel $bc + \bar{c}g$ corresponding to different cokernels a and $\bar{e}$.

The selection of the kernel intersection (i.e., a common subexpression) that, once substituted in the given Boolean expression, will result in the minimum number of literals can be considered as a *rectangular covering problem* [2]. Let us explain the rectangular covering formulation through the above Boolean expressions. The expressions are rewritten below, with each cube being uniquely identified by an integer.

$$f_1 = \frac{abc}{1} + \frac{a\bar{c}g}{2} + \frac{\bar{b}df}{3} + \frac{cde}{4}$$

$$f_2 = \frac{\bar{a}b\bar{d}}{5} + \frac{bc\bar{e}}{6} + \frac{\bar{b}de}{7} + \frac{\bar{b}g}{8} + \frac{\bar{c}\bar{e}g}{9}$$

First we form the *cokernel cube matrix* for the set of expressions. Such a matrix shows all the kernels simultaneously and allows the detection of kernel intersections. A row in the matrix corresponds to a kernel, whose cokernel is the label for that row. Each column corresponds to a cube, which is the label for that column. The integer identifier of the cube, resulting from the product of the cokernel for row i and the cube for column j, is entered in position (i, j) of the matrix. As can be seen from Table 3.2, the unique cubes from all the kernels of the given equations are bc, $\bar{c}g$, ab, de, $\bar{b}f$, ce, $\bar{a}\bar{d}$, $c\bar{e}$, and $\bar{g}$; these are the labels of the columns of the matrix. There are six kernels; the corresponding cokernels are the labels of the rows of the matrix. Thus the cokernel matrix for the given expressions is as shown in Table 3.3.

TABLE 3.2 Kernels of f_1 and f_2

Expression	Cokernel	Kernel
f_1	a	$bc + \bar{c}g$
f_1	c	$ab + de$
f_1	d	$\bar{b}f + ce$
f_2	b	$\bar{a}\bar{d} + c\bar{e}$
f_2	$\bar{b}$	$de + \bar{g}$
f_2	$\bar{e}$	$bc + \bar{c}g$

TABLE 3.3 Co-kernel Matrix for Expressions f_1 and f_2

	1	2	3	4	5	6	7	8	9
	bc	$\bar{c}g$	ab	de	$\bar{b}f$	ce	$\bar{a}\bar{d}$	$c\bar{e}$	$\bar{g}$
1 a	1	2	0	0	0	0	0	0	0
2 c	0	0	1	4	0	0	0	0	0
3 d	0	0	0	0	3	4	0	0	0
4 b	0	0	0	0	0	0	5	6	0
5 $\bar{b}$	0	0	0	7	0	0	0	0	8
6 $\bar{e}$	6	9	0	0	0	0	0	0	0

A *rectangle* (R, C), where R and C are sets of rows and columns, respectively, is a submatrix of the cokernel cube matrix such that for each row $r_i \in R$ and each column $c_j \in C$, the entry (r_i, c_j) of the cokernel matrix is nonzero. A rectangle that has more than one row indicates a kernel intersection between the kernels corresponding to the rows in the rectangle. The columns in the rectangle identify the cubes of the kernel intersection. For example, in Table 3.3 the rectangle $\{R(1, 6), C(1, 2)\}$ indicates intersection between the kernels corresponding to rows 1 and 6. The intersection between these two kernels generates a common subexpression $(bc + \bar{c}g)$. A rectangle having more than one column will identify a kernel intersection of more than one cube.

A set of rectangles form a *rectangular cover* of a matrix B, if an integer (i.e., a nonzero entry) in B is covered by at least one rectangle from the set. Once an integer is covered, it is not necessary to cover the same integer by any other rectangle; all other appearances of the integer in the matrix can be considered as don't cares. A covering for the above cokernel cube matrix is

$$\{R(1,6), C(1,2)\}, \{R(3), C(5,6)\}, \{R(4), C(7)\}, \{R(5), C(4,9)\}$$

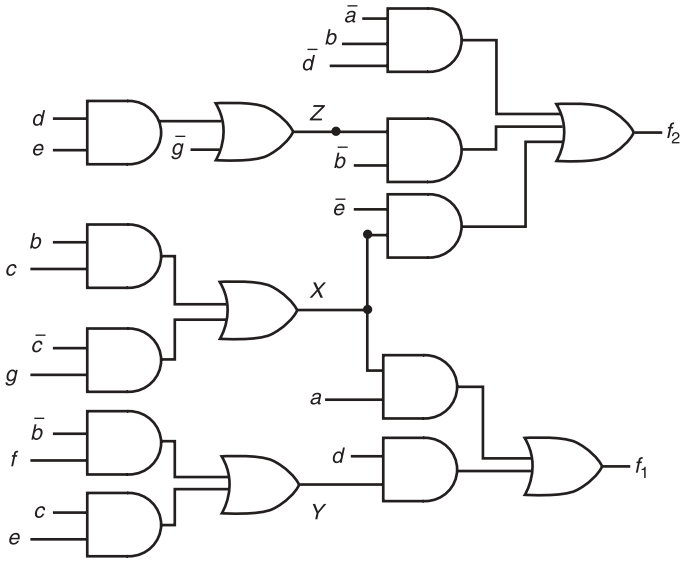


FIGURE 3.49 Multilevel implementation.

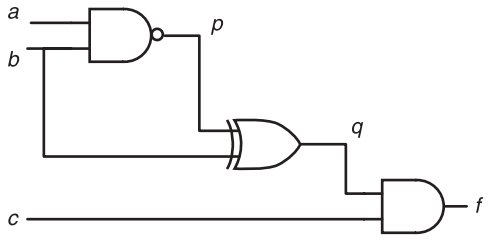
The implementation resulting from this covering is shown in Figure 3.49. The corresponding Boolean expressions are

$$f_1 = aX + dY$$
$$f_2 = \bar{e}X + \bar{b}Z + \bar{a}b\bar{d}$$
$$X = bc + \bar{c}g$$
$$Y = \bar{b}f + ce$$
$$Z = de + \bar{g}$$

This implementation has 22 literals, compared to 26 literals in the original expressions. With larger subexpressions, the reduction in the number of literals is significantly higher.

3.10 MINIMIZATION OF MULTILEVEL CIRCUITS USING DON'T CARES

Previously, we considered don't care conditions arising in two-level logic circuits and showed how these can be used to minimize logic. In multilevel logic circuits, don't care inputs (and outputs) may occur because of circuit structure [1, 2]. For example, if in a multilevel circuit a logic block is the input of another logic block, and the first block does not produce all possible output patterns, then the second block does not receive all input combinations. To illustrate, let us consider the circuit of Figure 3.50*a*. Truth tables of the first and the second logic block are shown in Figure 3.50*b* and 3.50*c* respectively. As can be seen from Figure 3.50*c*, the second block does not receive input combination 00. Thus $bp = 00$ can be considered a don't care input pattern for the second block. We shall see later in this section that such don't cares can be used for optimizing multilevel circuits.



(*a*)

a	*b*	*p*
0	0	1
0	1	1
1	0	1
1	1	0

(*b*)

p	*b*	*q*
0	1	1
1	1	0
1	0	1
(0	0	*X*)

(*c*)

FIGURE 3.50 Derivation of don't cares.

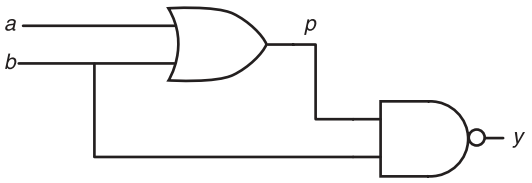


FIGURE 3.51 A circuit with an SDC.

Don't cares in multilevel circuits can in general be categorized into two types: *Satisfiabilty don't cares* (SDCs) and *observability don't cares* (ODCs).

3.10.1 Satisfiability Don't Cares

Satisfiability don't cares are input combinations that cannot *legally* occur at the inputs of an internal block driven by another block in a circuit. In other words, these are input combinations that do not *satisfy* the function of the driving block. For example, the input pattern $bp = 10$ to the second logic block in Figure 3.51 is an SDC because p must be 1 if b is 1. Formally, if an internal block of a circuit does not receive certain input patterns because they are not produced by the output of another block that is driving the internal block, these input patterns are known as SDCs. To illustrate, let us consider the circuit of Figure 3.51. Any input pattern to the first block that does not satisfy the function of the block (i.e., $p = a + b$) is an SDC. For example, it is easy to see that if $a = 1$, p cannot be 0. Similarly, p cannot be 0 if $b = 1$, and also when $a = 0$ and $b = 0$ p must be 0 not 1. Thus the SDC set for the circuit of Figure 3.51 is

a	b	p
1	0	0
1	1	0
0	1	0
0	0	1

Let us consider how SDCs can be used in further minimization of an already minimized circuit. Figure 3.52 shows a two-level circuit that implements the function

$$p = (\overline{a + b})$$

$$q = (\overline{bc})$$

$$f = p\bar{b} + \bar{p}\bar{c} + qc$$

Any input/output patterns that do not satisfy $p = (\overline{a + b})$ are SDCs generated by node p; thus the SDC for node p identified as SDC_p is

$$\text{SDC}_p = p \oplus (\overline{a + b})$$

Similarly,

$$\text{SDC}_q = q \oplus (\overline{bc})$$

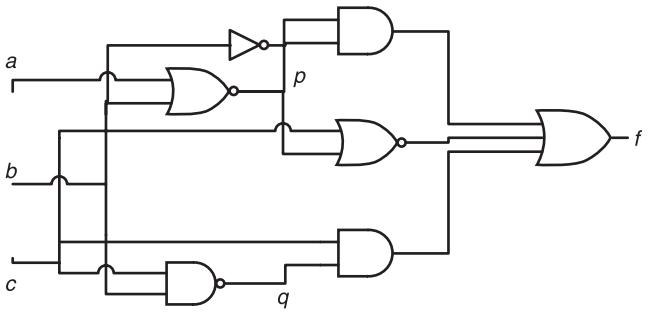


FIGURE 3.52 Circuit with SDCs.

Once the SDCs are derived for each output driving a logic block, the sum of these is *quantified* to eliminate input variable(s) not used by the driven logic block. The *universal quantification* of a function $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$ with respect to a variable x_i is

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) = f_{\bar{x}_1} \cdot f_{x_i}$$

where $f_{\bar{x}_i}$ and f_{x_i} are cofactors of the function f with respect to $\bar{x}_i$ and x_i, respectively. Thus the universal quantification of a function, also called the *consensus* of the function $Cx_i(f)$, derives the component of the function that does not depend on variable x_i. In other words, universal quantification of a function with respect to a variable removes the variable from the *support* of the function. The support of a function is the set of variables on which the function depends.

The support of the function for the second-level logic block in Figure 3.52 does not include variable *a.* Thus this variable is eliminated from the sum of SDC_p and SDC_q by deriving the universal quantification of the sum function with respect to variable *a*.

$$\begin{aligned} S = \text{SDC}_p + \text{SDC}_q &= (p \oplus (\overline{a+b})) + (q \oplus (\overline{bc})) \\ &= \bar{p}\bar{a}\bar{b} + pa + pb + \bar{q}\bar{b} + \bar{q}\bar{c} + qbc \end{aligned}$$

The universal quantification of S with respect to variable a is

$$\text{V}_a S = S_{\bar{a}} \cdot S_a = pb + \bar{q}\bar{b} + qbc + \bar{q}\bar{c}$$

Note that the terms in $\text{V}_a S$ are the SDCs for the second-level logic block and can be used to minimize the function f. The minimized function is

$$f = \bar{p}\bar{c} + q$$

The number of literals in f is reduced from 7 to 3.

The universal quantification of a function with respect to more than one variable can be derived as

$$\text{V}_{xy}(f) = \text{V}_y(\text{V}_x(f)) = f_{\bar{x}\bar{y}} \cdot f_{\bar{x}y} \cdot f_{x\bar{y}} \cdot f_{xy}$$

For example, the universal quantification of function S above with respect to variables a and b are

$$\begin{aligned} V_{ab}(S) &= V_b(V_a(S)) = f_{\bar{a}\bar{b}} \cdot f_{\bar{a}b} \cdot f_{a\bar{b}} \cdot f_{ab} \\ &= p\bar{q} + \bar{q}\bar{c} \end{aligned}$$

3.10.2 Observability Don't Cares

The observability don't cares of an internal block in a circuit are any input pattern that masks the output of the block; that is, it cannot be observed at the circuit output. In other words, the output of the circuit for this input pattern is independent of the block. For example, in the circuit of Figure 3.53 when

$$b = 1 \text{ and } c = 1, \quad f = 1$$

that is, block p has no effect on output f. Thus bc is an ODC for block p.

Similarly, when $a = 1$ and $d = 1$, the circuit output is insensitive to q. Therefore the ODC for block q is ad.

The input patterns that make an internal block output x of a circuit not observable at a circuit output f may be derived from the *Boolean difference* of f with respect to x. The Boolean difference of a function $f(x_1, x_2, \ldots, x_i, \ldots, x_n) = f(X)$ with respect to one of its inputs x_i is defined as

$$df(x_1, x_2, \ldots, x_i, \ldots, x_n)/dx_i = df(X)/dx_i = f_{\bar{x}_1}(X) \oplus f_{x_i}(X)$$

In other words, the Boolean difference of a function with respect to a variable x_i is the EX-OR of the cofactors of the function with respect to $\bar{x}_i$ and x_i. If a function is sensitive to an internal node x_i (i.e., $f_{\bar{x}_i}(X) \neq f_{x_i}(X)$) then $df(X)/dx_i = 1$. Thus the observability don't care for node x_i is

$$\text{ODC}_{x_i} = (\overline{df(X)/dx_i})$$

For example, the Boolean difference of the function $f(a, b, c) = \bar{a}c + ab$ with respect to a is

$$\begin{aligned} df/da &= f_{a=0}(\bar{a}c + ab) \oplus f_{a=1}(\bar{a}c + ab) \\ &= c \oplus b \end{aligned}$$

Therefore $\text{ODC}_a = (\overline{df/da}) = (\overline{c \oplus b}) = \bar{b}\bar{c} + bc$.

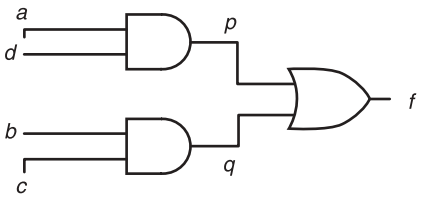


FIGURE 3.53 Circuit with observability don't cares.

In Figure 3.53 since $f = p + q$ the Boolean difference of f with respect to p is

$$\begin{aligned} df/dp &= (0 + q) \oplus (1 + q) \\ &= q \oplus 1 = \bar{q} = \overline{bc} = \bar{b} + \bar{c} \end{aligned}$$

Either $b = 0$ or $c = 0$ will allow node p to be observable at output f. Thus $(\overline{df/dp}) = (\bar{b} + \bar{c}) = bc$; that is, $b = 1$ and $c = 1$ is the pattern that will make p unobservable at output f, and thus is an ODC for p. Similarly, $(df/dq) = (p + 0) \oplus (p + 1) = \bar{p} = (\overline{ad}) = \bar{a} + \bar{d}$; thus $(\overline{df/dq}) = ad$ is an ODC for node q.

Let us consider another example to clarify the concept of observability don't cares and to illustrate how these can be used to minimize a function. Figure 3.54 shows the multilevel implementation of the circuit based on the following expressions:

$$\begin{aligned} p &= \bar{b}c + ab \\ q &= \overline{(pc)} \\ r &= \overline{(a + c)} \\ f &= q + r \end{aligned}$$

For multilevel circuits the Boolean difference of a function can be derived by taking the individual Boolean difference of each node with respect to the output of its preceding node, and then the Boolean differences are concatenated. For example, if the preceding node of node z is y, and the preceding node of y is x, then

$$(dz/dx) = (dz/dy) \cdot (dy/dx)$$

The Boolean difference of f with respect to p is

$$(df/dp) = (df/dq) \cdot (dq/dp)$$

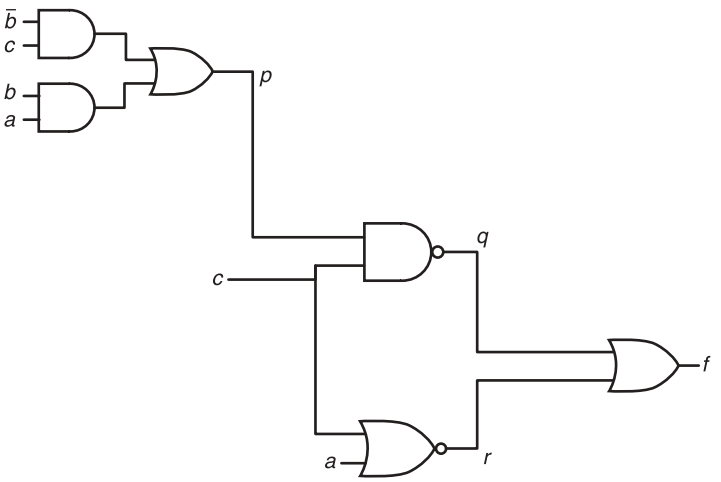


FIGURE 3.54 A four-level circuit.

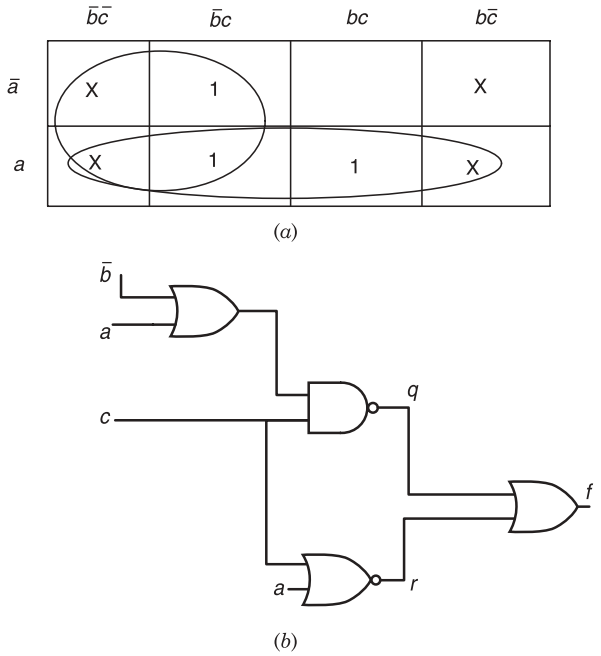


FIGURE 3.55 Minimization using (*a*) ODCs and (*b*) ODC_p.

Therefore the observability don't care for node p is

$$\begin{aligned}\text{ODC}_p &= (\overline{df/dp}) \\ &= \overline{[(df/dq)\cdot(dq/dp)]}\end{aligned}$$

Since $f = q + r$ and $q = (\overline{pc})$,

$$df/dq = \bar{r} \quad \text{and} \quad dq/dp = c$$

Therefore

$$\text{ODC}_p = \overline{(\bar{r}\cdot c)} = r + \bar{c} = \bar{a}\bar{c} + \bar{c} = \bar{c}$$

Since $\bar{c}$ is a don't care for node p, it may be used for possible minimization of the function of p as shown in the Karnaugh map of Figure 3.55*a*. Thus node p in Figure 3.54 can be replaced by $(\bar{b} + a)$ as shown in Figure 3.55*b*.

3.11 COMBINATIONAL LOGIC IMPLEMENTATION USING EX-OR AND AND GATES

An EX-OR gate can be constructed from AND, OR, and inverter gates (Chapter 2). However, the Boolean expression for the EX-OR function can be manipulated

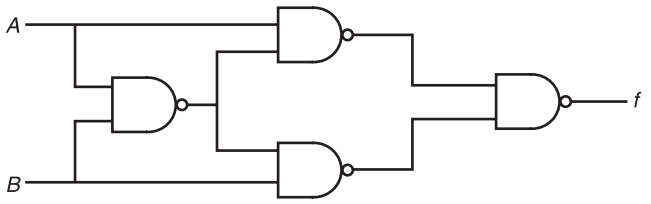


FIGURE 3.56 EX-OR gate.

algebraically into another form, which can be implemented using four two-input NAND gates:

$$\begin{aligned} f(A,B) = A \oplus B &= \bar{A}B + A\bar{B} \\ &= \bar{A}B + B\bar{B} + A\bar{B} + A\bar{A} \\ &= B(\bar{A} + \bar{B}) + A(\bar{A} + \bar{B}) \\ &= \overline{\overline{(B(\bar{A} + \bar{B}) + A(\bar{A} + \bar{B}))}} \\ &= \overline{\overline{B(\bar{A} + \bar{B})} \cdot \overline{A(\bar{A} + \bar{B})}} \\ &= \overline{\overline{B \cdot \overline{AB}} \cdot \overline{A \cdot \overline{AB}}} \end{aligned}$$

Figure 3.56 shows the implementation of the EX-OR gate.

There are several rules associated with EX-OR operation. Table 3.4 lists some of these.

EX-OR gates can be cascaded as shown in Figure 3.57 to generate the *parity* bit of an input pattern applied to the circuit. The output expression for the circuit is

$$f(A,B,C,D) = A \oplus B \oplus C \oplus D$$

TABLE 3.4 Rules for EX-OR Operation

$X \oplus X = 0$
$X \oplus \bar{X} = 1$
$1 \oplus X = \bar{X}$
$X + Y = X \oplus Y \oplus XY = X \oplus \bar{X}Y$
$X(Y \oplus Z) = XY \oplus XZ$

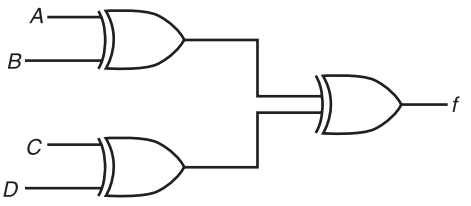
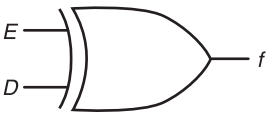


FIGURE 3.57 EX-OR circuit of four variables.



E	D	f
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 3.58 Programmable inverter.

The output of the circuit is 0 if all inputs are 0 or if the number of inputs at 1 is even; the output is 1 if the number of inputs at 1 is odd. For example, if $A = 0$, $B = 1$, $C = 0$ and $D = 1$, the output is 0 because an even number of inputs are 1. On the other hand, if $A = 1, B = 1, C = 0$ and $D = 1$, the output will be 1 because an odd number of inputs are 1.

An EX-OR gate can also be used as a *programmable inverter* as shown in Figure 3.58. If the control input E is 0, data on the D line is transferred to the output. However, if $E = 1$, the output of the gate is the inverse of the data input. This particular arrangement is widely used in field-programmable devices to control polarity of output signals.

It is also possible to realize arbitrary combinational functions using AND and EX-OR gates. However, in order to do that, it is first necessary to express the function in a canonical form using AND and EX-OR. For example, a canonical sum-of-products expression of two variables

$$f(A, B) = a_0\bar{A}\bar{B} + a_1\bar{A}B + a_2A\bar{B} + a_3AB, \quad \text{where } a_1 = 0 \text{ or } 1$$

can also be represented as

$$f(A, B) = a_0\bar{A}\bar{B} \oplus a_1\bar{A}B \oplus a_2A\bar{B} \oplus a_3AB$$

The OR operators can be replaced by EX-OR because at any time only one minterm in a sum-of-products expression can take the value of 1. The third line of Table 3.4 shows that the complemented variables can be replaced: $\bar{A} = 1 \oplus A$, $\bar{B} = 1 \oplus B$. Hence

$$\begin{aligned} f(A, B) &= a_0(1 \oplus A)(1 \oplus B) \oplus a_1(1 \oplus A)B \oplus a_2A(1 \oplus B) \oplus a_3AB \\ &= a_0(1 \oplus A \oplus B \oplus AB) \oplus a_1(B \oplus AB) \oplus a_2(A \oplus AB) \oplus a_3AB \\ &= a_0 \oplus (a_0 \oplus a_2)A \oplus (a_0 \oplus a_1)B \oplus (a_0 \oplus a_1 \oplus a_2 \oplus a_3)AB \end{aligned}$$

A combinational function expressed as the EX-OR of the products of uncomplemented variables is said to be in *Reed–Muller canonical form*. Thus the above expression represents the Reed–Muller form of two-variable functions. It can be rewritten

$$f(A, B) = c_0 \oplus c_1A \oplus c_2B \oplus c_3AB$$

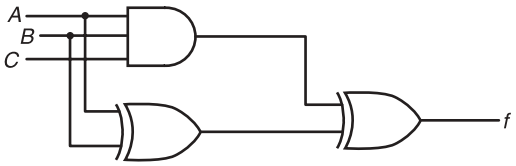


FIGURE 3.59 Reed–Muller implementation.

where $c_0 = a_0$, $c_1 = a_0 \oplus a_2$, $c_2 = a_0 \oplus a_1$, and $c_3 = a_0 \oplus a_1 \oplus a_2 \oplus a_3$. In general, any combinational function of n variables can be expressed in Reed–Muller canonical form:

$$f(x_1, x_2, \ldots, x_n) = c_0 \oplus c_1 x_1 \oplus c_2 x_2 + \cdots + c_n x_n C_{n+1}\, x_1 x_2 \\ \oplus c_{n+2}\, x_1 x_3 \oplus \cdots \oplus c_{2^n-1}\, x_1 x_2 \cdots x_n$$

As an example, let us derive the Reed–Muller form of the combinational function

$$\begin{aligned} f(A, B, C) &= A\bar{B} + B(\bar{A} + C) \\ &= A\bar{B} \oplus B(\bar{A} + C) \oplus A\bar{B}\, B(\bar{A} + C) && \text{(by line 4, Table 3.4)} \\ &= A\bar{B} \oplus B(\bar{A} + C) \oplus 0 \\ &= A\bar{B} \oplus B(\bar{A} \oplus C \oplus \bar{A}C) && \text{(by line 4)} \\ &= A(1 \oplus B) \oplus B[1 \oplus A \oplus C \oplus (1 \oplus A)C] && \text{(by line 3)} \\ &= A \oplus AB \oplus B \oplus AB \oplus BC \oplus BC \oplus ABC && \text{(by line 5)} \\ &= A \oplus B \oplus ABC \end{aligned}$$

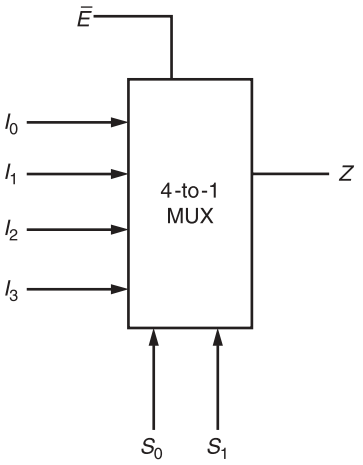
A direct implementation of the function is shown in Figure 3.59.

It has been shown that circuits realized using AND and EX-OR gates alone are easy to test.

3.12 LOGIC CIRCUIT DESIGN USING MULTIPLEXERS AND DECODERS

3.12.1 Multiplexers

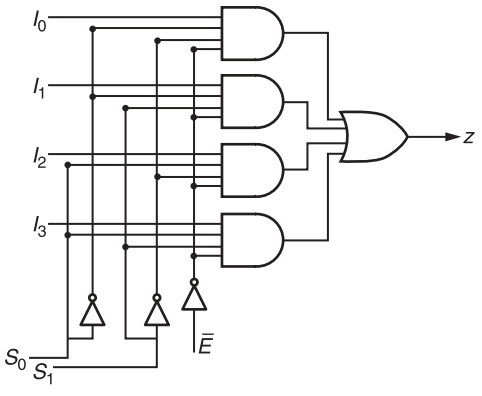
Multiplexers are typically described as data selectors and are frequently used in digital systems. In a standard multiplexing application, digital signals are connected to the multiplexer's input lines (I_0, I_1, I_2,...) and binary control signals are fed to the select lines (S_0, S_1, ...). Figure 3.60*a* shows the block diagram of a 4-to-l multiplexer (i.e., a multiplexer having four input lines—I_0, I_1, I_2, and I_3—and one output line, Z). It also has two select lines (S_0 and S_1) and an enable line $\bar{E}$. The signals on the select lines specify which of the four input lines will be gated to the output. For example, if the select lines are $S_0S_1 = 00$, the output is I_0; similarly, the select inputs 01, 10, and 11 give outputs I_1, I_2, and I_3, respectively. In order for the multiplexers to operate at all, the $\bar{E}$ line must be set to logic 0; otherwise, the multiplexer output will be 0 regardless of all other input combinations. The operation of the multiplexer can be described by the function



(*a*)

Enable	*Select*	*Input*	*Inputs*				*Output*
$\bar{E}$	S_0	S_1	I_0	I_1	I_2	I_3	Z
1	–	–	–	–	–	–	0
0	0	0	0	–	–	–	0
0	0	0	1	–	–	–	1
0	0	1	–	0	–	–	0
0	0	1	–	1	–	–	1
0	1	0	–	–	0	–	0
0	1	0	–	–	1	–	1
0	1	1	–	–	–	0	0
0	1	1	–	–	–	1	1

(*b*)



(*c*)

FIGURE 3.60 (*a*) Block diagram, (*b*) function table, and (*c*) logic diagram.

table of Figure 3.60*b*. The implementation of the multiplexer circuit using AND and OR gates and inverters is shown in Figure 3.60*c*.

A multiplexer of any size can be formed by combining several multiplexers in a tree form. Figure 3.61 shows the implementation of a 32-to-l multiplexer by interconnecting eight 4-to-l multiplexers and a single 8-to-l multiplexer. It is assumed that the enable lines of all the multiplexers are connected to ground (i.e., they are at logic 0). By replacing all 4-to-l multiplexers with their 8-to-l counterparts, the circuit of Figure 3.61 can be converted to a 64-input multiplexer circuit without adding any additional delay.

Multiplexers are often used to implement combinational logic functions. For example, a 4-to-l multiplexer can be used to generate any of the possible functions of three variables.

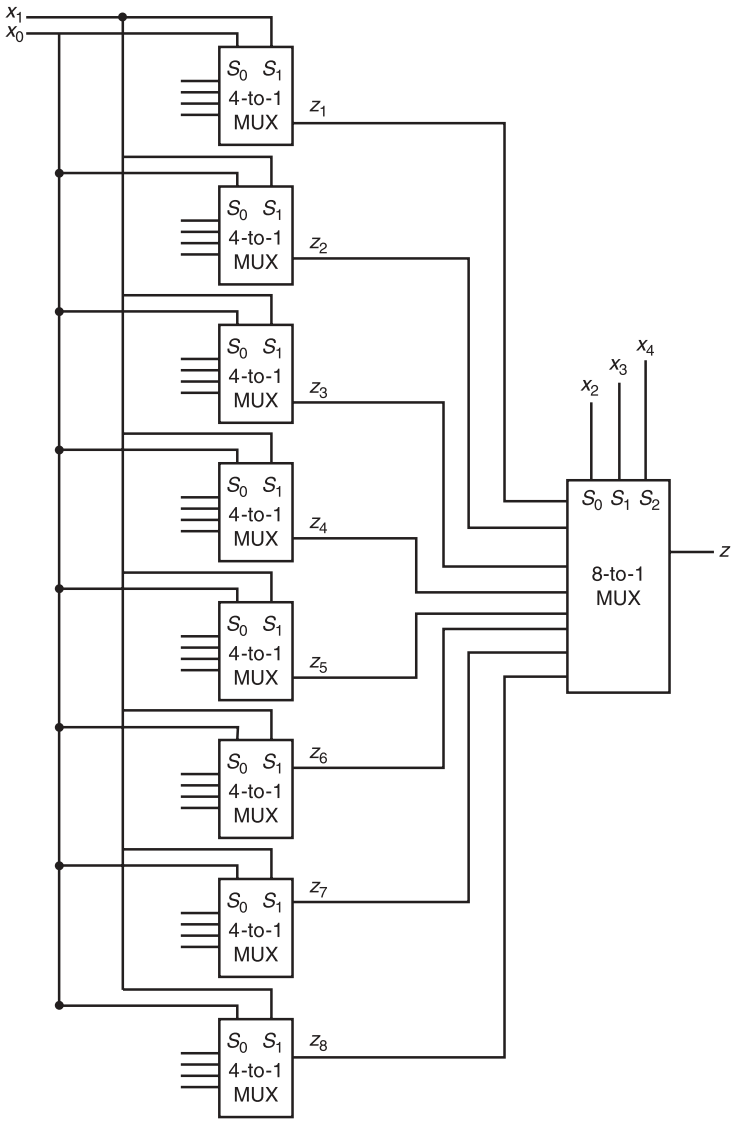


FIGURE 3.61 Configuration of 32-to-1 multiplexer.

Let us consider the following function of three variables, which is to be implemented with a 4-to-1 multiplexer:

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

The truth table for the function is first derived as shown in Figure 3.62*a*. Two of three variables (e.g., B and C) are arbitrarily chosen to feed the two select lines of the 4-to-1 multiplexer; the remaining variable A or its complement is to be connected to the input lines I_0, I_1, I_2, and I_3. The truth table shows that when B and C are 0, the output has the same value as variable A. Hence A must be connected to I_0. When $B = 0$ and $C = 1$, the truth table indicates that the output f will be 0 irrespective of the variable A. It means that I_1 must be connected to ground (i.e., to 0). For $B = 1$ and $C = 0$, the output f is the complement of A (i.e., $f = 1$ when $A = 0$ and $f = 0$ when $A = 1$). This means that $\bar{A}$ must be connected to I_2. Finally, when $B = 1$ and $C = 1$ the output f will be 1 irrespective of the variable A. Hence input line I_3 must be connected to 1 (i.e., supply line V_{CC}). Figure 3.62*b* shows the appropriate input connection to the 4-to-1 multiplexer.

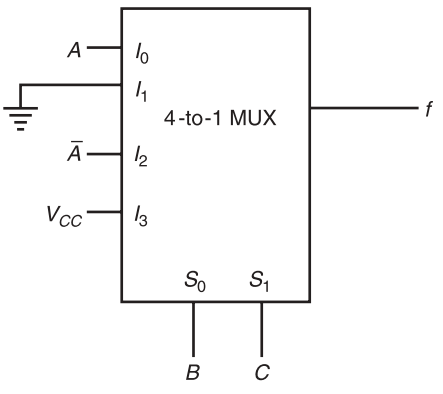
In general, any n-variable function, $f(x_1, x_2, \ldots, x_n)$, can be implemented with a multiplexer that has $(n - 1)$ select lines and 2^{n-1} input lines. This results from the fact that the function can be expanded with respect to any $(n - 1)$ of the n variables:

$$\begin{aligned} f(x_1, x_2, \ldots, x_{n-2}, x_{n-1}, x_n) = {} & \bar{x}_1\bar{x}_2 \cdots \bar{x}_{n-2}\bar{x}_{n-1} f(0, 0, \ldots, 0, 0, x_n) \\ & + \bar{x}_1\bar{x}_2 \cdots \bar{x}_{n-2}\bar{x}_{n-1} f(0, 0, \ldots, 0, 1, x_n) \\ & + \bar{x}_1\bar{x}_2 \cdots \bar{x}_{n-2}\bar{x}_{n-1} f(0, 0, \ldots, 1, 0, x_n) \\ & + \bar{x}_1\bar{x}_2 \cdots x_{n-2}x_{n-1} f(0, 0, \ldots, 1, 1, x_n) \\ & + \cdots + x_1x_2 \cdots x_{n-2}x_{n-1} f(1, 1, \ldots 1, 1, x_n) \end{aligned}$$

In this expression each of the $f(i_1, i_2, \ldots, i_{n-1}, x_n)$, where $i_j = 0$ or 1, indicates the value of f when $x_1, x_2, \ldots, x_{n-1}$ in f are substituted with $i_1, i_2 \ldots, i_{n-1}$. The equation describes a 2^{n-1}-to-1 multiplexer, where the expanding variables $x_1, x_2, \ldots, x_{n-1}$ are connected to the $(n - 1)$ select lines, and the quantity that represents the function

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

(*a*)



(*b*)

FIGURE 3.62 (*a*) Truth table and (*b*) multiplexer implementation of $f(A, B, C) = BC + \bar{A}B + A\bar{B}\bar{C}$.

$f(i_1, i_2, \ldots, i_{n-1}, x_n)$ is connected to the input line I_k (where k is the decimal equivalent of $i_1, i_2, \ldots, i_{n-1}$).

As an example, let us implement the following function of five variables:

$$f(A,B,C,D,E) = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{C}D + AB\bar{C}D + AE + \bar{B}C\bar{E} + \bar{C}E$$

Since $n = 5$, the function can be realized with a 16-to-1 multiplexer. We expand the function with respect to A, B, C, and D:

$$f(A,B,C,D,E) = \bar{A}\bar{B}\bar{C}\bar{D}f_0 + \bar{A}\bar{B}\bar{C}Df_1 + \bar{A}\bar{B}C\bar{D}f_2 + \bar{A}\bar{B}CDf_3 + \cdots + ABCDf_{15}$$

where $f_0 = f(0, 0, 0, 0, E) = 1, f_1 = f(0, 0, 0, 1, E) = 1, f_2 = f(0, 0, 1, 0, E) = \bar{E}, f_3 = \bar{E}$, $f_4 = E, f_5 = 1, f_6 = 0, f_7 = 0, f_8 = 1, f_9 = E, f_{10} = 1, f_{11} = 1, f_{12} = E, f_{13} = 1, f_{14} = E$, and $f_{15} = E$. Figure 3.63 shows the implementation of the above expression, where A, B, C, and D are control inputs and E is the multiplexed variable. The information obtained by expanding the function about A, B, C, and D can also be derived by representing it in a tabular form as shown in Table 3.5.

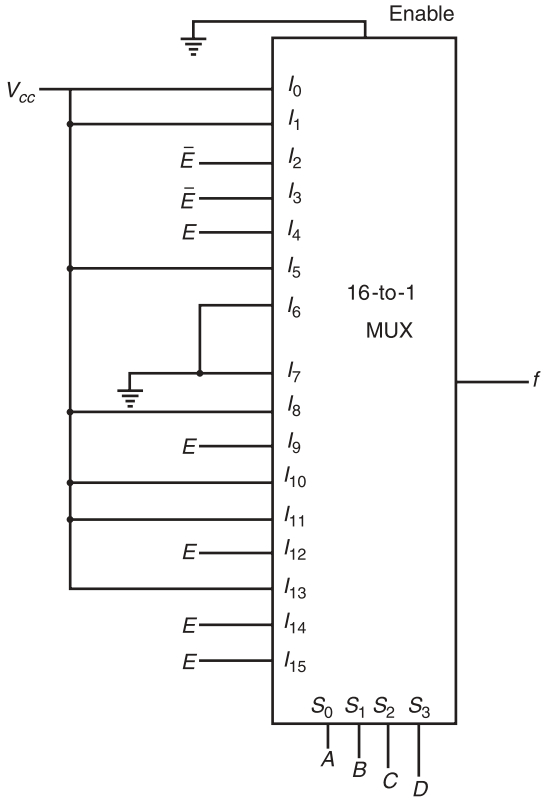


FIGURE 3.63 Multiplexer implementation of the function $f(A, B, C, D, E) = \bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C}\bar{D} + \bar{A}\bar{C}D + AB\bar{C}D + AE + \bar{B}C\bar{E} + \bar{C}E$.

TABLE 3.5 Selection of Control Inputs and the Multiplexed Variable

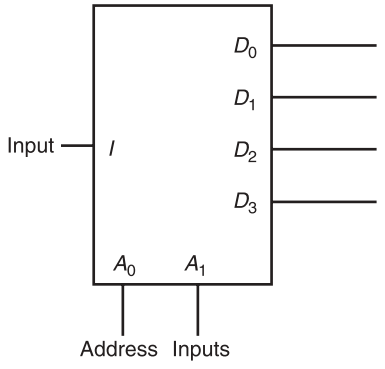
L	$\bar{A}\bar{B}\bar{C}\bar{D}$	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}\bar{B}CD$	$\bar{A}B\bar{C}\bar{D}$	$\bar{A}B\bar{C}D$	$\bar{A}BC\bar{D}$	$\bar{A}BCD$	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}\bar{C}D$	$A\bar{B}C\bar{D}$	$A\bar{B}CD$	$AB\bar{C}\bar{D}$	$AB\bar{C}D$	$ABC\bar{D}$	$ABCD$
0	1	1	1	1	0	1	0	0	1	0	1	1	0	1	0	0
1	1	1	0	0	1	1	0	0	1	1	1	1	1	1	1	1
$f(i_1 = A,$ $i_2 = B,$ $i_3 = C,$ $i_4 = D, E)$	1	1	$\bar{E}$	$\bar{E}$	E	1	0	0	1	E	1	1	E	1	E	E

3.12.2 Demultiplexers and Decoders

A *demultiplexer* performs the function opposite to that of a multiplexer. It is used to route data on a single input to one of several outputs, which is determined by the choice of signals on the address lines. Figure 3.64*a* shows the block diagram of a demultiplexer with four output lines (D_0, D_1, D_2, and D_3), one input line (I), and two address lines (A_0, A_1). The operation of the demultiplexer can be described by the function table shown in Figure 3.64*b*. The implementation of the demultiplexer using NAND gates and inverters is shown in Figure 3.64*c*.

A *decoder* produces a unique output corresponding to each input pattern. If the input line in Figure 3.64 is set to logic 0 (i.e., $I = 0$), the 1-to-4 demultiplexer will act as a decoder. For example, $A_0 = 0$, $A_1 = 0$ will give an output of 0 on line D_0; lines D_1, D_2, and D_3 will be at logic 1. Similarly, $A_0 = 0$, $A_1 = 1$ will give an output of 0 on line D_1; $A_0 = 1$, $A_1 = 0$ will give an output of 0 on line D_2; and $A_0 = 1$, $A_1 = 1$ will give an output of 0 on line D_3. Thus the 1-to-4 demultiplexer can function as a 2-to-4 decoder, allowing each of the four possible combinations of the input signals A_0 and A_1 to appear on the selected output line.

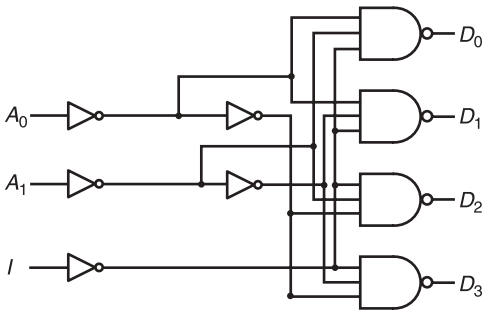
A decoder of any size can be made up by interconnecting several small decoders. For example, a 6-to-64 decoder can be constructed from four 4-to-16 decoders and a 2-to-4



(*a*)

Address Inputs		Input	Output			
A_0	A_1	I	D_0	D_1	D_2	D_3
0	0	0	0	1	1	1
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	0
1	1	1	1	1	1	1

(*b*)

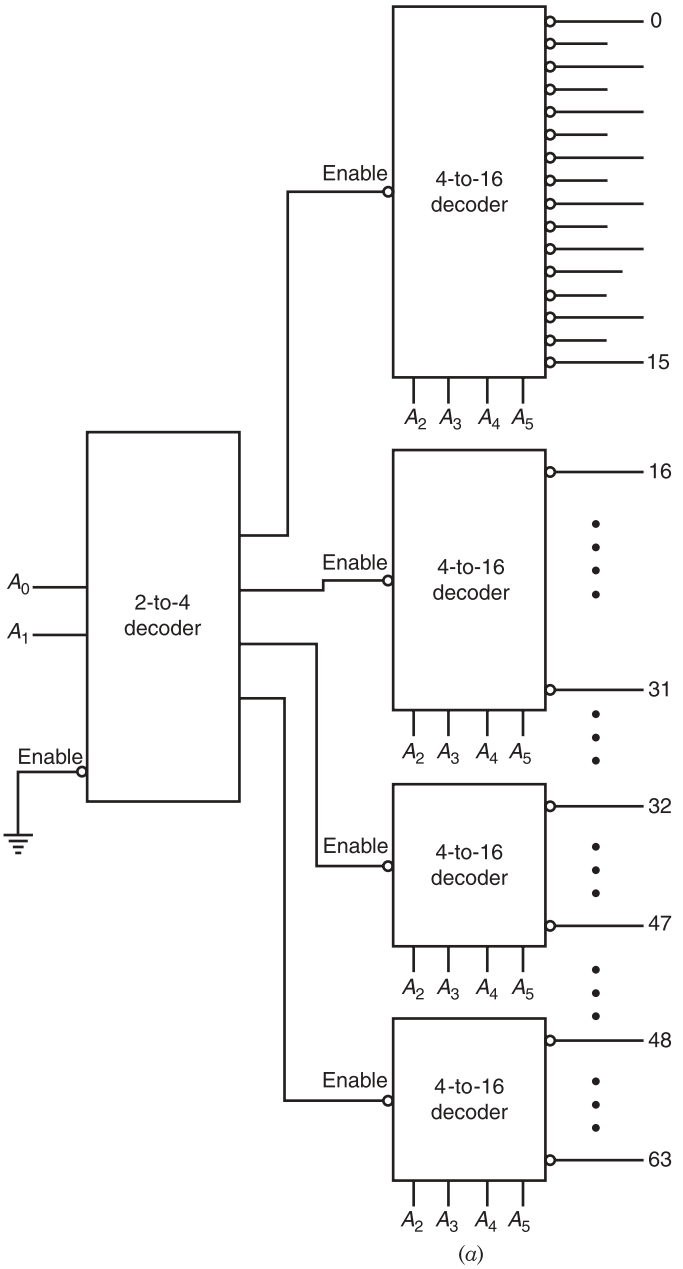


(*c*)

FIGURE 3.64 Demultiplexer: (*a*) block diagram, (*b*) function table, and (*c*) circuit diagram.

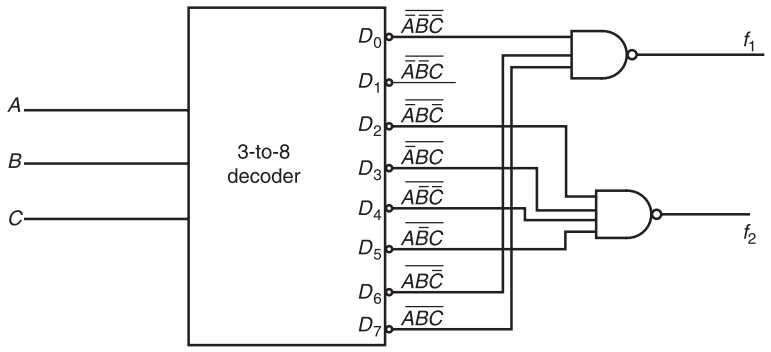
decoder as shown in Figure 3.65. The 2-to-4 decoder enables one of the four 4-to-16 decoders, depending on the address bits A_0 and A_1. Address bits A_2, A_3, A_4, and A_5 determine which output of the enabled 4-to-16 decoder goes low. We thus have one of the 64 outputs going low, as selected by the 6-bit address.

Like a multiplexer, a decoder can also be used to implement any arbitrary Boolean functions. As indicated in the previous section, a decoder generates all the product



(*a*)

FIGURE 3.65 (*a*) A 6-to-64 decoding circuit. (*b*) Implementation of $f_1 = AB + \bar{A}\bar{B}\bar{C}$ and $f_2 = \bar{A}B + A\bar{B}$ using a 3-to-8 decoder and NAND gates.



(*b*)

FIGURE 3.65 (*Continued*).

terms of its input variables. Thus by connecting the outputs of the decoder corresponding to the canonical sum-of-products expression to an output NAND gate, any Boolean function can be realized.

Let us implement the following functions using a 3-to-8 decoder:

$$f_1(A, B, C) = AB + \bar{A}\bar{B}\bar{C}$$
$$f_2(A, B, C) = \bar{A}B + A\bar{B}$$

First of all, the functions must be expressed in canonical form,

$$f_1(A, B, C) = \underbrace{ABC}_{7} + \underbrace{AB\bar{C}}_{6} + \underbrace{\bar{A}\bar{B}\bar{C}}_{0}$$
$$f_2(A, B, C) = \underbrace{\bar{A}BC}_{3} + \underbrace{\bar{A}B\bar{C}}_{2} + \underbrace{A\bar{B}\bar{C}}_{4} + \underbrace{A\bar{B}C}_{5}$$

The input variables A, B, and C are connected to address inputs of the decoder; the outputs of the decoder corresponding to the minterms of the given functions are then fed to the inputs of the two NAND gates. The resulting circuit is shown in Figure 3.65*b*.

3.13 ARITHMETIC CIRCUITS

Arithmetic operations are frequently performed in digital computers and other digital systems. In this section we deal with the design of adders, subtractors, and multipliers.

3.13.1 Half-Adders

A half-adder is a combinational logic circuit that accepts two binary digits and generates a sum bit and a carry-out bit. Table 3.6 shows the truth table of the half-adder circuit. Columns a_i and b_i correspond to the sum and the carry-out bit, respectively. The Boolean expressions

TABLE 3.6 Truth Table for Half-Adder

a_i	b_i	s_i	c_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

for the sum and the carry-out can be derived directly from the truth table and are as follows:

$$\begin{aligned} s_i &= \bar{a}_i b_i \oplus a_i \bar{b}_i \\ &= a_i \oplus b_i \\ c_i &= a_i b_i \end{aligned}$$

The NAND–NAND implementation of the sum and the carry-out is shown in Figure 3.66.

3.13.2 Full Adders

The limitation of a half-adder is that it cannot accept a carry-in bit; the carry-in bit represents the carry-out of the previous low-order bit position. Thus a half-adder can be used only for the two least significant digits when adding two multibit binary numbers, since there can be no possibility of a propagated carry to this stage. In multibit addition, a carry bit from a previous stage must be taken into account, which gives rise to the necessity for designing a full adder. A full adder can accept two operands bits, a_i and b_i, and a carry-in bit c_i from previous stage; it produces a sum bit s_i and a carry-out bit c_0. Table 3.7 shows the truth table for a full adder circuit. As can be seen from the truth table, sum bit s_i is 1 if there is an odd number of 1's at the inputs of the full adder, whereas the carry-out c_0 is 1 if there are two or more 1's at the inputs. The sum and carry out bits will be 0 otherwise. The Boolean expressions for s_i and c_0 obtained from the truth table are as follows:

$$s_i = \bar{a}_i \bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i \bar{b}_i \bar{c}_i + a_i b_i c_i$$

$$c_0 = \bar{a}_i b_i c_i + a_i \bar{b}_i c_i + a_i b_i \bar{c}_i + a_i b_i c_i$$

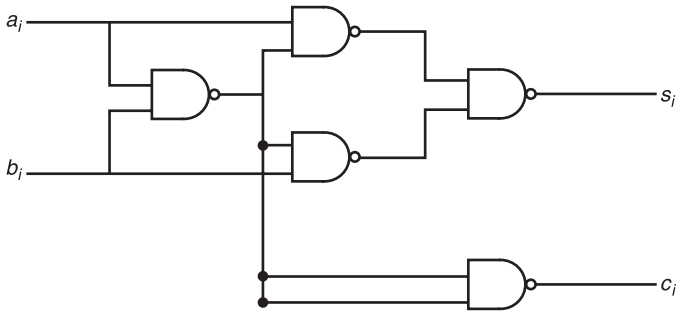


FIGURE 3.66 Half-adder circuit.

TABLE 3.7 Truth Table for a Full Adder

a_i	b_i	c_i	s_i	c_0
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

These expressions are plotted on the Karnaugh maps shown in Figure 3.67. The expression for s_i cannot be reduced. The expression for c_0 reduces to

$$c_0 = a_i b_i + b_i c_i + a_i c_i$$

The expressions for s_i and c_0 can be rewritten as follows:

$$\begin{aligned} s_i &= \overline{\bar{a}_i \bar{b}_i \bar{c}_i + \bar{a}_i b_i c_i + a_i b_i \bar{c}_i + a_i \bar{b}_i c_i} \\ &= \overline{(\overline{a_i + b_i + c_i}) + (\overline{a_i + \bar{b}_i + \bar{c}_i}) + (\overline{\bar{a}_i + \bar{b}_i + c_i}) + (\overline{\bar{a}_i + b_i + \bar{c}_i})} \\ c_0 &= \overline{\bar{a}_i \bar{c}_i + \bar{b}_i \bar{c}_i + \bar{a}_i \bar{b}_i} \\ &= \overline{\overline{(a_i + c_i)} + \overline{(b_i + c_i)} + \overline{(a_i + b_i)}} \end{aligned}$$

The implementation of the expression for s_i and c_0 using NOR gates is shown in Figure 3.68. It is also possible to implement the full adder by combining two half-adders with some NAND gates as shown in Figure 3.69.

A multibit adder can be constructed by cascading full adders such that the carry-out from the ith full adder is connected to the carry-in of the $(i + 1)$th adder. The number of adders required is equal to the bit length of the binary numbers to be added. Figure 3.70 shows a 4-bit adder. Since the least significant adder FA_0, cannot have a carry-in, it can be replaced by a half-adder if desired, although in practice a full adder

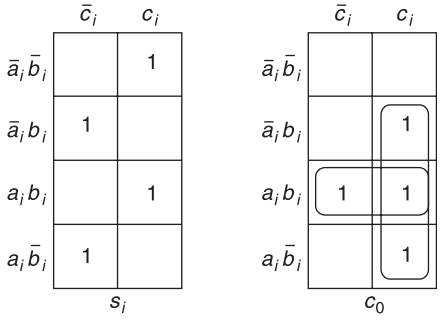


FIGURE 3.67 Karnaugh maps for s_i and c_0.

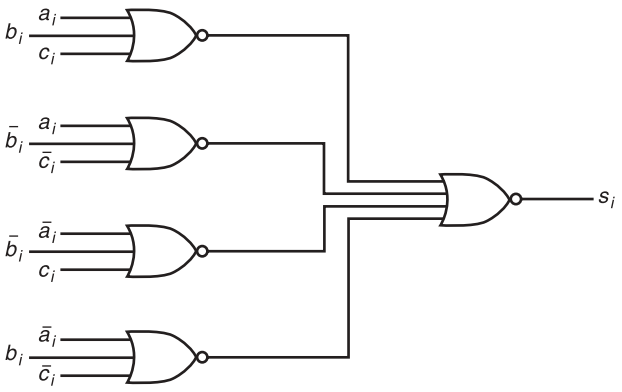


FIGURE 3.68 Implementation of the full adder using NOR gates.

with the carry-in connected to the ground is used. It can be seen from Figure 3.70 that the sum bit S_3 can have a steady value only when its carry-in signal C_2 has a steady value; similarly, S_2 has to wait for C_1 and S_1 has to wait for C_0. In other words, the carry signals must ripple through all the full adders before the outputs stabilize to the correct values; hence such an adder is often called a *ripple* adder. For example, if the following addition is to be performed the carry-out generated from the least significant stage of the

$$\begin{array}{cccc} a_3 & a_2 & a_1 & a_0 \\ b_3 & b_2 & b_1 & b_0 \end{array} \quad \begin{array}{cccc} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{array}$$

$$\begin{array}{cccc} 1 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \end{array} \longleftarrow \text{Carry-in}$$

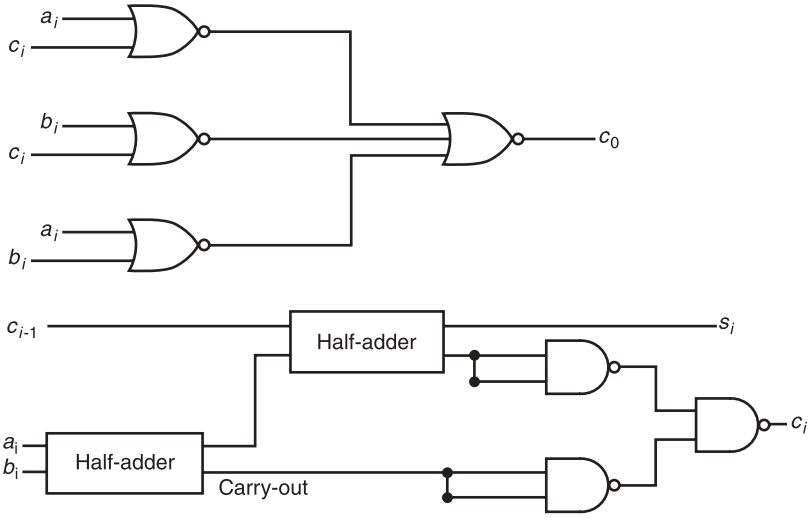


FIGURE 3.69 Full adder constructed from half-adders.

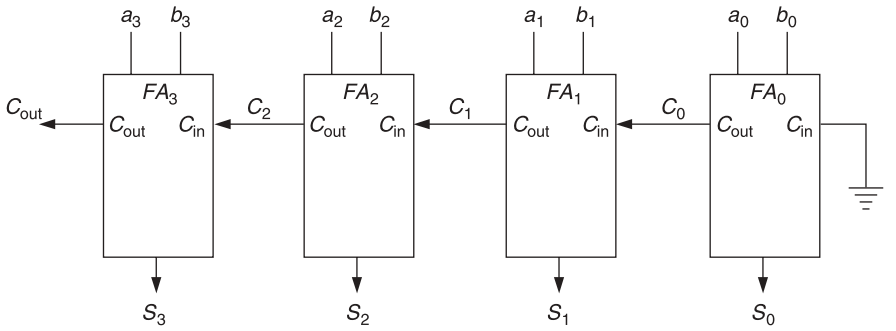


FIGURE 3.70 A 4-bit ripple carry adder.

adder propagates through the successive stages and produces a carry-in into the most significant stage of the adder. The time required to perform addition in a ripple adder depends on the time needed for the propagation of carry signals through the individual stages of the adder. Thus ripple carry addition is not instantaneous. The greater the number of stages in a ripple carry adder the longer is the carry propagation time, and consequently the slower the adder.

3.13.3 Carry-Lookahead Adders

The long carry propagation time of the ripple adder can be overcome by an alternative implementation of the carry generation circuit known as *carry-lookahead*.

Let us consider the 4-bit adder of Figure 3.70 to understand the principle of carry-lookahead. We can write the following equations for the carry-outs:

$$c_0 = a_0 b_0 \tag{3.6}$$

$$c_1 = a_1 b_1 + (a_1 \oplus b_1) c_0 \tag{3.7}$$

Substituting Eq. (3.6) into Eq. (3.7), we get

$$c_1 = a_1 b_1 + (a_1 \oplus b_1) a_0 b_0 \tag{3.8}$$

In a similar manner, we can write

$$c_2 = a_2 b_2 + (a_2 \oplus b_2) c_1 \tag{3.9}$$

By utilizing Eqs. (3.8, 3.9) becomes

$$\begin{aligned} c_2 &= a_2 b_2 + (a_2 \oplus b_2)[a_1 b_1 + (a_1 \oplus b_1) a_0 b_0] \\ &= a_2 b_2 + (a_2 \oplus b_2) a_1 b_1 + (a_2 \oplus b_2)(a_1 \oplus b_1) a_0 b_0 \end{aligned} \tag{3.10}$$

Finally, we can write

$$c_{\text{out}} = a_3 b_3 + (a_3 \oplus b_3) c_2 \tag{3.11}$$

which with Eq. (3.10) becomes

$$\begin{aligned} c_{\text{out}} &= a_3b_3 + (a_3 \oplus b_3)[a_2b_2 + (a_2 \oplus b_2)a_1b_1 + (a_2 \oplus b_2)(a_1 \oplus b_1)a_0b_0] \\ &= a_3b_3 + (a_3 \oplus b_3)a_2b_2 + (a_3 \oplus b_3)(a_2 \oplus b_2)a_1b_1 \\ &\quad + (a_3 \oplus b_3)(a_2 \oplus b_2)(a_1 \oplus b_1)a_0b_0 \end{aligned} \tag{3.12}$$

Next, we define P_i and G_i as the *carry-propagate* and *carry-generate* signals for the ith stage of the adder, where

$$P_i = a_i \oplus b_i \tag{3.13}$$

$$G_i = a_ib_i \tag{3.14}$$

P_i indicates that if $a_i = 0$, $b_i = 1$ or $a_i = 1$, $b_i = 0$, then the carry-in to the ith stage will be propagated to the next stage, G_i indicates that a carry-out will be generated from the ith stage when both a_i and b_i are 1, regardless of the carry-input to this stage.

Substituting Eqs. (3.13) and (3.14) in Eqs. (3.6), (3.8), (3.10) and (3.12), we get

$$\begin{aligned} c_0 &= G_0 \\ c_1 &= G_1 + G_0P_1 \\ c_2 &= G_2 + G_1P_2 + G_0P_2P_1 \\ c_{\text{out}} &= G_3 + G_2P_3 + G_1P_3P_2 + G_0P_3P_2P_1 \end{aligned}$$

Figure 3.71 shows the implementation of a 4-bit carry-lookahead adder.

The propagation delay of the carry in the circuit of Figure 3.71 is independent of the number of bit pairs to be added and equal to the propagation delay of the two-level carry-lookahead circuit. In principle, the circuit of Figure 3.71 can be extended to a large number of bit pairs; however, the complexity of the carry-generation equations for large number of stages makes it impractical.

3.13.4 Carry-Select Adder

The carry-select adder is one of the faster type of adders. Figure 3.72 shows the block diagram of a 4-bit carry-select adder. The adder consists of two independent units. Each unit implements the addition operation in parallel. The first unit implements the addition assuming a carry-in of "0," generating the sum and carry-out bit. The second unit performs the same operation assuming "1" as the carry-in. The carry-out generated from each full adder is used as the new carry-in for the successive bit. The resultant sum is multiplexed by the "actual carry-in" coming in from the previous state. If the actual carry-in is "0" then the sum multiplexed from the first unit is selected; alternatively, if the carry-in is "1" then the sum from the second unit is selected. The main difference between a carry-select adder and a ripple carry adder is that in ripple carry adder the carry has to ripple through four full adders, but in the case of a carry-select adder the carry has to pass through a single multiplexer.

3.13.5 Carry-Save Addition

In a ripple carry or carry-lookahead adder, $m - 1$ additions are required to add m numbers; each addition except the first uses the accumulated sum and a new number. Thus the total time for addition is $(m - 1)t_d$, where t_d is the time required to do each addition. In a carry-save addition the carry-out of a full adder is not connected to the carry-in of the

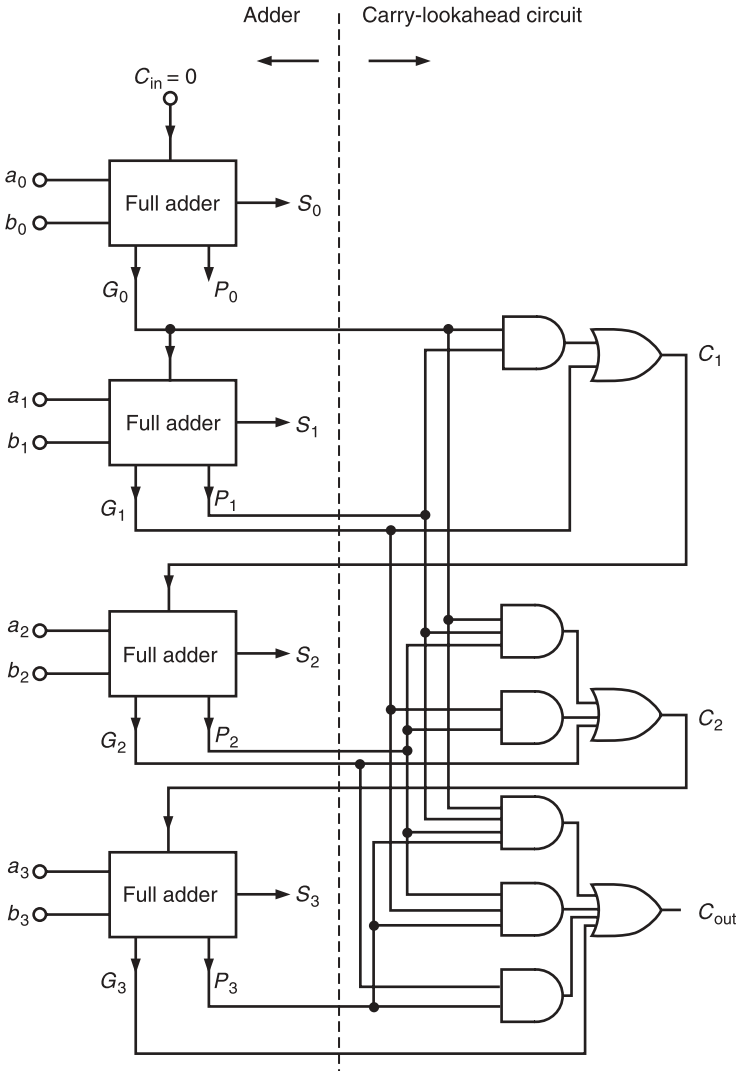


FIGURE 3.71 A 4-bit adder circuit with carry-lookahead.

more significant adder; instead, the sum outputs and the carry-outs of the full adders are stored as sum tuple S and carry tuple C, respectively. The final sum is obtained by adding S and C using a carry-propagate adder. Let us illustrate the carry-save addition by adding three 4-bit numbers:

A		1	1	0	0	$= 12$
B		0	1	0	1	$= 5$
C		1	0	1	1	$= 11$
S		0	0	1	0	
C	1	1	0	1		
Sum	1	1	1	0	0	28

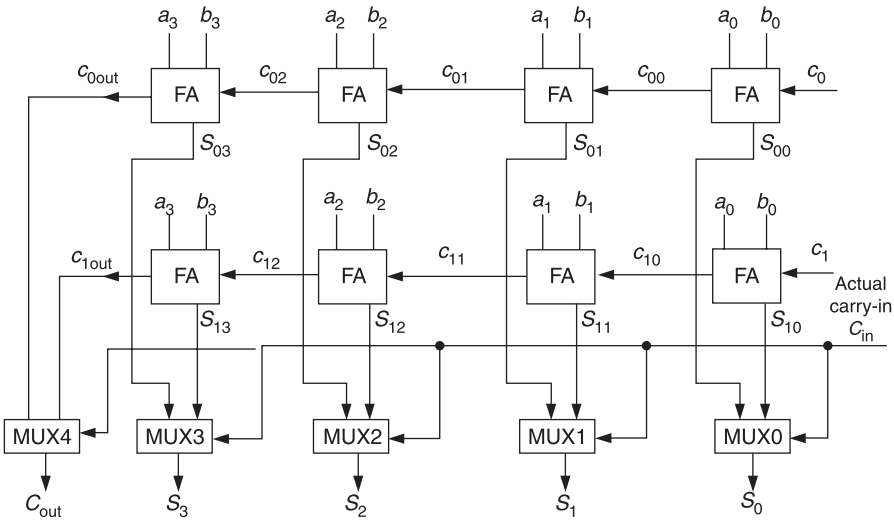


FIGURE 3.72 Carry-select adder.

Note that the carry bits are shifted left to correspond to normal carry-propagation in conventional adders. Figure 3.73*a* shows the circuit for generating *S* and *C* for the above addition. Thus a carry-save adder reduces three input tuples into two tuples, which constitute a set of sum bits and a set of carry bits. Figure 3.73*b* shows the symbol for a carry-save full adder.

3.13.6 BCD Adders

It is often more convenient to perform arithmetic operations directly with decimal numbers, especially if the results of the operations are to be displayed directly in

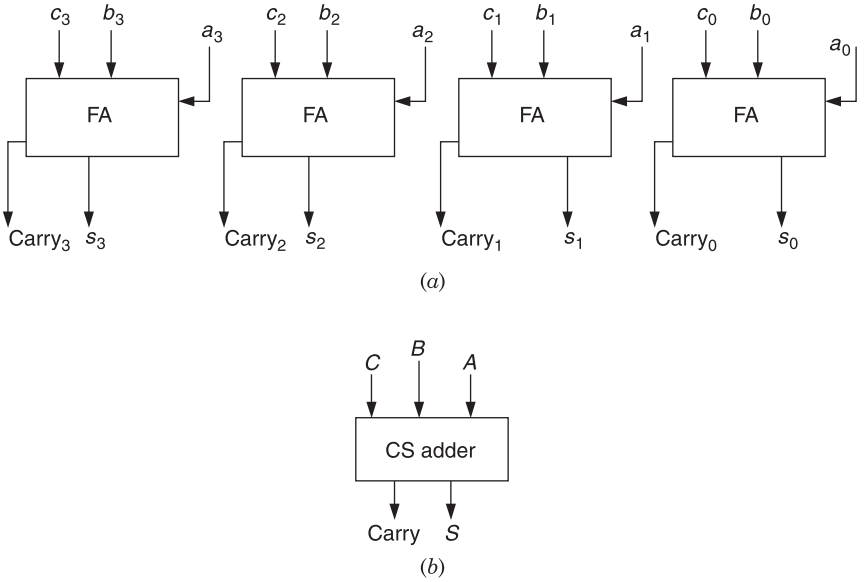


FIGURE 3.73 (*a*) Carry and some generations, (*b*) symbol.

decimal form. Each decimal digit is usually represented by 4-bit 8-4-2-1 BCD code; thus six combinations of the 4-bit code are not valid. When two BCD digits are added, the sum has a value in the range of 0 to 18. If the sum exceeds 9, an adjustment has to be made to the resulting invalid combination. This adjustment is made by adding decimal 6 (i.e., 0110_2) to the result, which generates a valid sum as well as a carry-in to the next-higher-order digit.

Table 3.8 shows the 20 possible sum digits that may result from the addition of two BCD digits and a carry-in. Whenever the sum digit is greater than 9 or the carry bit b_c is 1 for the unadjusted sum, the sum digit is adjust by adding 6 to it. Consequently, a logic circuit that detects the condition for the adjustment and produces a carry-out C must be used. Such a circuit can be expressed by the following Boolean function:

$$C = b_c + b_8b_4 + b_8b_2$$

When $C = 1$, it is necessary to add the correction 0110 to the sum bits $b_8b_4b_2b_1$ and to generate a carry for the next stage. The implementation of one stage of a BCD adder is shown is Figure 3.74.

3.13.7 Half-Subtractors

The half-subtractor circuit is used to implement a 1-bit binary subtraction. Figure 3.75*a* shows the truth table of a half-subtractor used to subtract Y (subtrahend) from X

TABLE 3.8 Derivation of BCD Sum Digit

	BCD Sum (Without Adjustment)					BCD Sum (with Adjustment)				
Decimal	b_c	b_8	b_4	b_2	b_1	C	S_8	S_4	S_2	S_1
0	0	0	0	0	0					
1	0	0	0	0	1					
2	0	0	0	1	0					
3	0	0	0	1	1					
4	0	0	1	0	0	adjustment not necessary				
5	0	0	1	0	1					
6	0	0	1	1	0					
7	0	0	1	1	1					
8	0	1	0	0	0					
9	0	1	0	0	1					
10	0	1	0	1	0	1	0	0	0	0
11	0	1	0	1	1	1	0	0	0	1
12	0	1	1	0	0	1	0	0	1	0
13	0	1	1	0	1	1	0	0	1	1
14	0	1	1	1	0	1	0	1	0	0
15	0	1	1	1	1	1	0	1	0	1
16	1	0	0	0	0	1	0	1	1	0
17	1	0	0	0	1	1	0	1	1	1
18	1	0	0	1	0	1	1	0	0	0
19	1	0	0	1	1	1	1	0	0	1

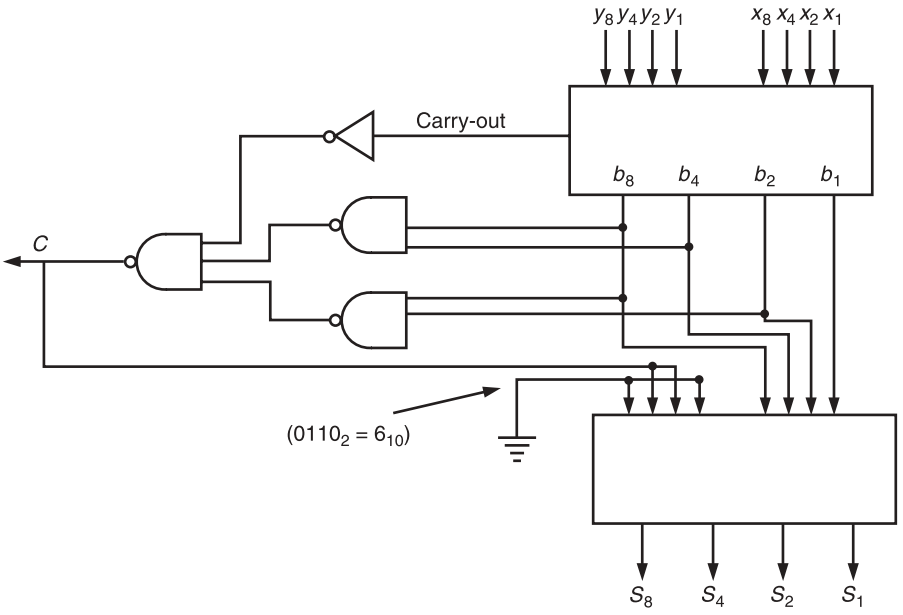
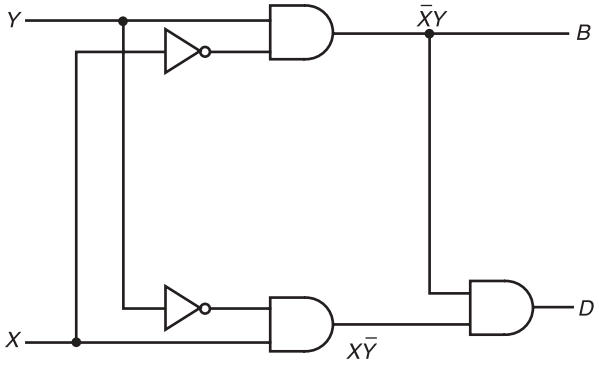


FIGURE 3.74 A single-stage 4-bit BCD adder.

X	Y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

(*a*)



(*b*)

FIGURE 3.75 (*a*) Truth table for half-subtractor and (*b*) half-subtractor circuit.

(minuend) and generate difference bit D and the borrow bit B. The Boolean expressions for the D and B outputs are derived from the truth table and are given by

$$\begin{aligned} D &= \bar{X}Y + X\bar{Y} \\ &= X \oplus Y \\ B &= \bar{X}Y \end{aligned}$$

The implementation of the above expressions are shown in Figure 3.75*b*.

3.13.8 Full Subtractors

A full subtractor has three inputs X (minuend), Y (subtrahend), and Z (the previous borrow). The outputs of the full subtractor are the difference bit D and the output borrow B. The truth table of a full subtractor is shown in Figure 3.76*a*. The output bit D is obtained from the subtraction, $a_1 - (b_i + c_i)$. The output bit B is 0 if $a_i \geq b_i$ provided $c_i = 0$. If $c_i = 1$, output bit B is 1 of and only if $a_i \leq b_i$. The simplified Boolean expressions for outputs B and D are derived from their Karnaugh map plots as shown in Figure 3.76*b*.

The simplified expressions are

$$\begin{aligned} D_i &= \bar{a}_i\bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i \bar{b}_i \bar{c}_i + a_i b_i c_i \\ B_i &= \bar{a}_i c_i + \bar{a}_i b_i + b_i c_i \end{aligned}$$

The implementations of the expressions for D and B are shown in Figure 3.76*c*. Note that the expression for the output D_i is identical to the expression for S_i in the full adder circuit. Furthermore, the expression for B_i is similar to the carry-out expression C_0 in the full adder, except that the input variable a_i is complemented. Thus a full adder can be used to perform the function of subtraction as well, by applying the complement of input a_i to the circuit that generates the carry output.

3.13.9 Two's Complement Subtractors

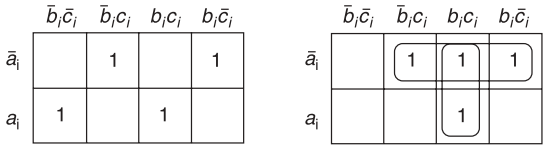
All modern digital systems use 2's complement number systems. Subtraction in 2's complement is performed by 2's complementing the subtrahend and adding it to the minuend; any carry-out is ignored. If the sign bit of the resulting number is 0, the numerical part of the number is expressed in magnitude form. However, if the sign bit of the resulting number is 1, the numerical part of the number must be changed to 2's complement in order to get the correct magnitude.

A circuit of a 4-bit 2's complement subtractor is shown in Figure 3.77. The inverters are used to produce the 1's complement of the subtrahend, which is then added to the minuend. The carry-in of the low-order full adder is held at logic 1, which adds 1 in order to implement 2's complementation.

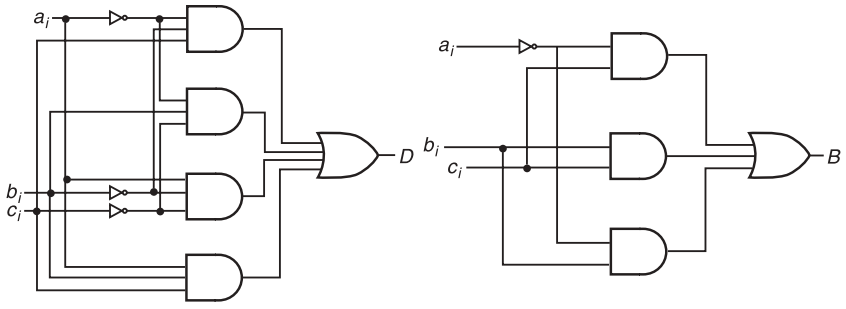
Let us use the circuit shown in Figure 3.77 to subtract $Y = +9(01001_2)$ from $X = +7(00111_2)$. The outputs of the inverter will be 10110. Then 00111 is added to

a_i	b_i	c_i	D_i	B_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(*a*)



(*b*)



(*c*)

FIGURE 3.76 (*a*) Truth table for a full subtractor, (*b*) Karnaugh maps for outputs B and D, and (*c*) logic implementation for a full subtractor.

01001 along with carry-in $= 1$:

$$\begin{array}{r} 00111 \\ 10110 \\ \qquad 1 \\ \hline 11110 \end{array}$$

The 2's complement of the magnitude part gives the difference

$$0001 + 1 = 0010\,(2_{10})$$

Thus the correct result is -2.

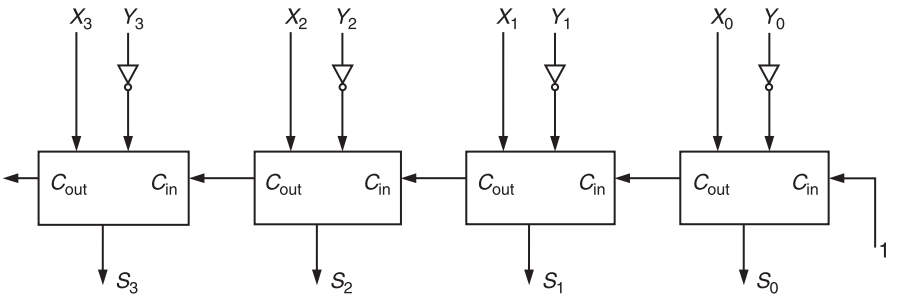


FIGURE 3.77 Two's complement subtractor circuit.

As another example, let us subtract $Y = -6(11010_2)$ from $X = +3(00011_2)$ using Figure 3.77. The output of the inverter (i.e., 00101) is added to 00011 assuming the carry-in input is at 1:

$$\begin{array}{r} 00011 \\ 00101 \\ \quad\quad 1 \\ \hline 01001 \end{array}$$

Since the result is positive, it is not necessary to take the 2's complement of the magnitude part. Thus, the result of the subtraction is $+9$.

3.13.10 BCD Substractors

Substraction of two decimal digits can be carried out by using the BCD adder. The 9's complement of the subtrahend is added to the minuend to find the difference. Thus in addition to the BCD adder, a small amount of circuitry is required in BCD substraction. The 9's complement of a BCD digit can be obtained by subtracting the digit from 9. Table 3.9 shows the 9's complement representation for the BCD digits. It can be

TABLE 3.9 The 9's Complement of BCD Digits

	BCD				9's Complement			
Decimal Number	b_8	b_4	b_2	b_1	f_8	f_4	f_2	f_1
0	0	0	0	0	1	0	0	1
1	0	0	0	1	1	0	0	0
2	0	0	1	0	0	1	1	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	0	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	0	1	0
8	1	0	0	0	0	0	0	1
9	1	0	0	1	0	0	0	0

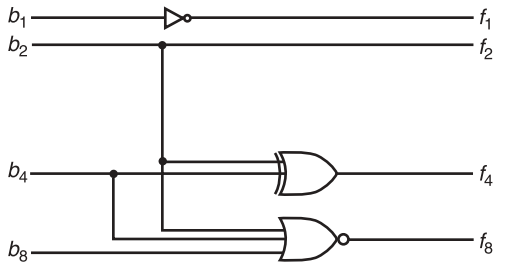


FIGURE 3.78 Circuit for a 9's complementer.

seen from Table 3.9 that a combinational circuit defined by the following Boolean expressions is required to derive the 9's complement of a BCD digit.

$$f_1 = \bar{b}_1$$
$$f_2 = b_2$$
$$f_4 = b_2 \oplus b_4$$
$$f_8 = \bar{b}_2\bar{b}_4\bar{b}_8$$

The logic diagram of the 9's complement circuit is shown in Figure 3.78. In practice, the BCD subtraction can also be used as a BCD adder by incorporating a mode control imput M to the 9's complement circuit such that

$M = 0$ Add operation
$M = 1$ Substract operation

The expression for the 9's complementer circuit are then modified as follows:

$$f_1 = M \oplus b_1$$
$$f_2 = b_2$$
$$f_4 = \overline{M}b_4 + M(b_2 \oplus b_4)$$
$$f_8 = \overline{M}b_8 + M \cdot \bar{b}_2\bar{b}_4\bar{b}_8$$

Figure 3.79 shows one stage of the BCD adder/subtractor circuit.

3.13.11 Multiplication

Multiplication schemed used in digital systems are quite similar to pencil-and-paper multiplication. An array of partial products is found first, and these are then added to generate the product. Figure 3.80 shows a simple numerical example of the multiplication. As shown in the diagram, the first partial product is formed by multiplying 1110 by 0, the second partial product is formed by multiplying 1110 by 1, and so on. The multiplication of two bits produces a 1 if both bits are 1; otherwise it produces a 0. The summation of the partial products is accomplished by using full adders. In general, the multiplication of an

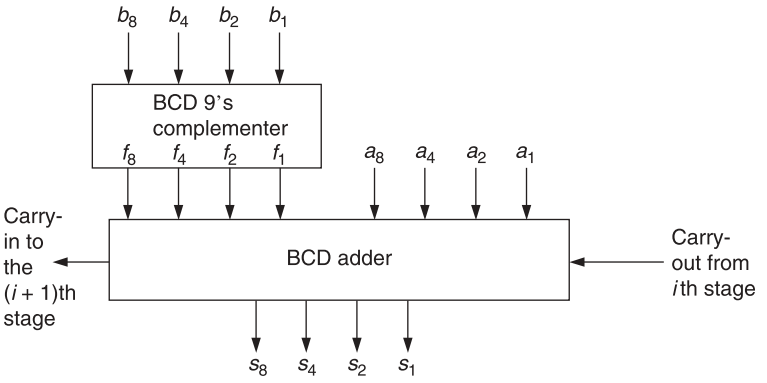
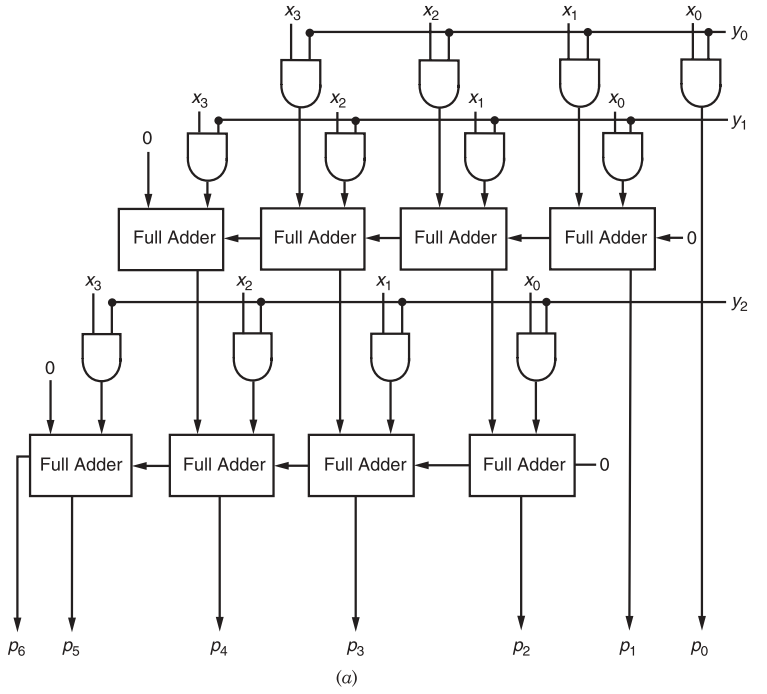


FIGURE 3.79 One stage of a BCD adder/subtractor unit.

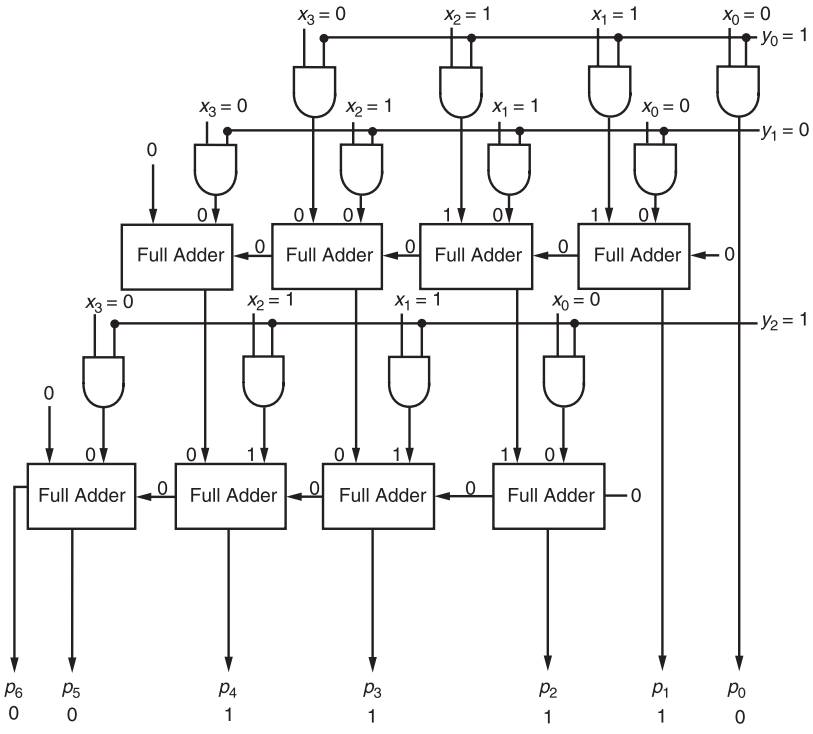
Multiplicand =	1 1 1 0
Multiplier =	1 0 1 0
	0 0 0 0
Partial product bits	1 1 1 0
	0 0 0 0
	1 1 1 0
Product p =	1 0 0 0 1 1 0 0

FIGURE 3.80 Binary multiplication example.



(*a*)

FIGURE 3.81 (*a*) A 4-bit by 3-bit multiplier circuit. (*b*) Multiplication of 6 by 5.



(*b*)

FIGURE 3.81 (*Continued*)

m-bit multiplicand $X(=x_{m-1} \ldots x_1\ x_0)$ by an *n*-bit multiplier $Y(=y_{n-1} \ldots y_1\ y_0)$ results in an $(m+n)$-bit product. Each of the *mn* 1-bit products $x_i y_j$ may be generated by a 2-input AND gate; these products are then added by an array of full address. Figure 3.81*a* shows a 4-bit by 3-bit multiplier circuit.

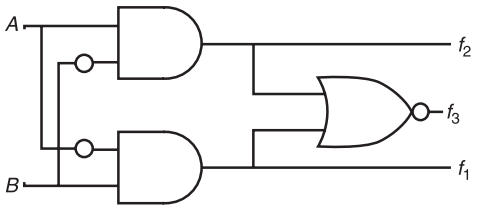
Let us multiply $x_3x_2x_1x_0 = 0110(6_{10})$ by $y_2y_1y_0 = 101(5_{10})$ using the multiplier circuit of Figure 3.81*a*. The outputs of the AND gates and the full adders in the multiplier circuit corresponding to the applied input values are recorded in Figure 3.81*b*. Note that the outputs of the AND gates form the partial products and the outputs of the full adders form the partial sum during the multiplication process. The final output pattern is $p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0 = 0011110$ (i.e., 30_{10}), which is the expected result. One of the problems in performing multiplication using this scheme is that when many partial products are to be added, it becomes difficult to handle the carries generated during the summation of partial products.

3.13.12 Comparator

A comparator is used to determine the relative magnitudes of two binary numbers. It compares two *n*-bit binary numbers and produces three possible results at the outputs. Figure 3.82*a* shows the result of comparing two 1-bit numbers *A* and *B*. It is seen from

A	B	f_1 $A > B$	f_2 $A < B$	f_3 $A = B$
0	0	0	0	1
1	0	1	0	0
0	1	0	1	0
1	1	0	0	1

(*a*)



(*b*)

FIGURE 3.82 Comparator.

the truth table that the specified conditions are satisfied by the following Boolean expressions:

$$f_1 = A\bar{B}$$

$$f_2 = \bar{A}B$$

$$f_3 = \bar{A}\bar{B} + AB$$

These expressions can be realized using NAND gates as shown in Figure 3.82*b*. The Boolean expressions for the 1-bit comparator can be expanded to *n*-bit operands.

3.14 COMBINATIONAL CIRCUIT DESIGN USING PLDs

Programmable logic devices, popularly known as PLDs, have a general architecture that a user can customize by inducing physical changes in select parts. Thus these devices can be configured to be application specific by utilizing their user programmable features. The actual programming of such devices can be done by a piece of equipment called a PLD programmer. The process takes only a few seconds.

Most PLDs consist of an AND array followed by an OR array. The inputs to a PLD enter the AND array through the buffers, which generate both the true and the complement of the input signals. Each gate in the AND array generates a minterm of the input variables; a minterm is known as a product line in the programmable logic device nomenclature. The device outputs are produced by summing the product terms in the array of OR gates.

Figure 3.83 shown the generic structure of a PLD with two inputs and two outputs. As can be seen from the diagram, there are programmable connections between the input lines and the product lines, as well as between product lines and sum lines; such connections are known as *crosspoints*.

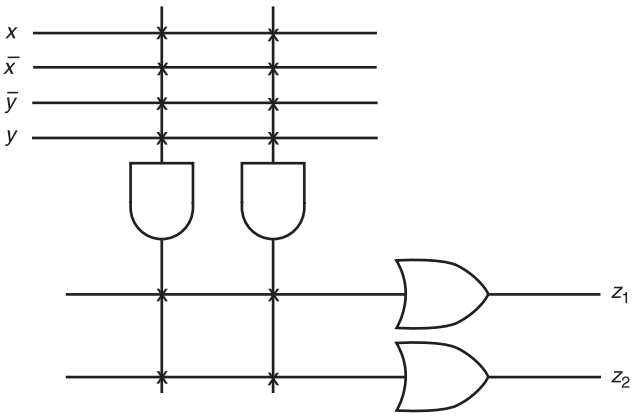


FIGURE 3.83 Generic PLD structure.

In PLDs manufactured using bipolar technology, the crosspoints are implemented using *fuses*, which break open when current flowing through then exceeds a certain limit. A major drawback of PLDs based on bipolar technology is that the programming of the fuses is irreversible (i.e., a device cannot be reprogrammed). On the other hand, PLDs manufactured using CMOS technology use memory cells as crosspoints. These cells are reprogrammable; hence the function of an already programmed device can be altered by merely changing the contents of the memory cells.

The AND–OR structured programmable logic devices can be grouped into three basic types:

Programmable read-only memory (PROM)
Programmable logic array (PLA)
Programmable array logic (PAL)

In a PROM, the AND array is fixed and the OR array is programmable. In a PLA, both arrays are programmable. A PAL device is the mirror image of the PROM its AND array is programmable, while the OR array is fixed.

3.14.1 PROM

Traditionally, PROM devices have been used to store software in micorprocessor-based systems. However, they can also be used to implement logic functions. The major advantage of PROM-based logic functions is that there is no need to employ any of the conventional minimization techniques. The input variables in a combinational function are used as inputs to a PROM with the required output values being stored in the location corresponding to the input combination. It will be obvious that a PROM can be used for realizing multioutput combinational circuits, each bit in the PROM output corresponding to a particular output value of the combinational circuit.

To illustrate, we implement a three-output combinational function of four variables (w, x, y, z). A 1 programmed in the PROM represents the presence of minterm in the function, and a 0 its absence. Thus the input combination 1001 applied to the PROM wil be interpreted as the minterm $w\bar{x}\bar{y}z$; in the output of the OR gate driven by the minterm,

the 1's correspond to functions containing that minterm and 0's corresponding to functions that do not.

$$f_1(w,x,y,z) = \bar{w}x\bar{y}z + \bar{w}\bar{x}yz + wxyz + \bar{w}\bar{x}\bar{y}\bar{z}$$
$$f_2(w,x,y,z) = w\bar{x}y\bar{z} + wxyz + \bar{w}x\bar{y}z$$
$$f_3(w,x,y,z) = \bar{w}xy\bar{z} + w\bar{x}\bar{y}z$$

These functions can be implemented by programming a PROM as shown:

w	x	y	z	f_1	f_2	f_3
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	0
0	1	0	1	1	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	1	1	0

Note that in employing PROM devices to implement logic function, the minterms are programmed directly from the truth table for each of the functions; the minimization process is bypassed, since this does not result in any savings. In fact, if an expression is already reduced it must be expanded to its canomical form in order to property specify the PROM program.

Let us implement a multiplier of two 2-bit numbers a_1a_0 and b_1b_0 using a PROM. Since the multiplier uses two 2-bit numbers, a PROM with 4 inputs is needed. The product of two such numbers will also contain $4(=2+2)$ bits, so the PROM will require 4 output lines. Each address to the PROM will consist of 4 bits, and the content corresponding to the address will be a 4-bit number that is the product of the first and the second halves of the address bits. The program table for the PROM is

b_1	b_0	a_1	a_0	o_8	o_4	o_2	o_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1

b_1	b_0	a_1	a_0	o_8	o_4	o_2	o_1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

3.14.2 PLA

The PLA structure offers a high level of flexibility because both the AND array ad the OR array are programmable. Since the AND array is programmable, PLAs do not suffer from the limitation of PROM devices that the AND array must provide all possible input combinations. Since both the arrays are user programmable, it is possible for an OR gate to access any number of product terms. Moreover, all OR gates can access the same product term(s) simultaneously. In other words, product sharing does not require any additional resources in a PLA architecture.

In PLAs every connection is made through a fuse at every intersection point the undesired connection can be removed later by blowing the fuses. Alternatively the desired connections can be made during the chip fabrication according to a particular interconnection pattern; these types of PLAs are mostly embedded in VLSI chips.

PLAs are configured by the number of inputs(I), outputs(O), and product terms(P): $I \times P \times O$. Figure 3.84 shows the logic diagram of a $4 \times 8 \times 4$ PLA. The inputs are each internally buffered and also inverted so as to provide true or complemented values. Product terms are formed by appropriate ANDing of any combination of input variables and their complements. Each of the OR gates can provide a sum of any or all of the product terms. In addition, each output function can be individually programmed true or complement. This is accomplished by an EX-OR gate on each output, one input of which is connected to the ground via a fuse so that the sum-of-products function is not inverted (Fig. 3.85). If the fuse is blown, this input is forced to be at logic 1, thus providing the complement of the sum-of-products function. The programmable inversion has the added advantage in that if the complement of a sum-of-products function is simpler to realize than the original output function, then the complemented function may be programmed into the AND array and an inversion of it obtained at the output.

As an example, let us consider the implementation of the following Boolean expressions using a PLA:

$$f_1(w,x,y,z) = \bar{w}xyz + w\bar{y}\bar{z} + \bar{x}yz$$
$$f_2(w,x,y,z) = w + x + y$$
$$f_3(w,x,y,z) = \bar{w}x + w\bar{x} + \bar{y}z + y\bar{z}$$

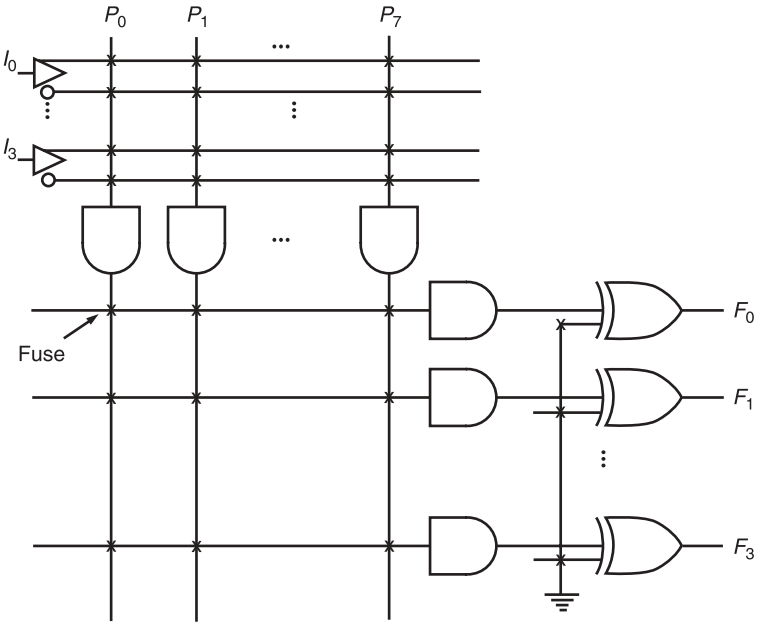


FIGURE 3.84 Logic diagram of a 4 × 8 × 4 PLA.

Expression f_2 can be rewritten

$$f_2(w, x, y, z) = \overline{\overline{w}\overline{x}\overline{y}}$$

which required only a single product term. Figure 3.86 shows the appropriately programmed logic diagram for the PLA.

A major problem in the PLA implementation of a combinational logic circuit is to ensure that the number of product terms in the Boolean expressions for the circuit will not exceed the number of product terms available in the PLA. Therefore minimization of multioutput Boolean functions is extremely important in the PLA implementation of combinational circuits.

To illustrate, let us consider the truth table of a circuit shown in Table 3.10. This circuit has 5 inputs and outputs. By using the two-level minimizer ESPRESSO, the following

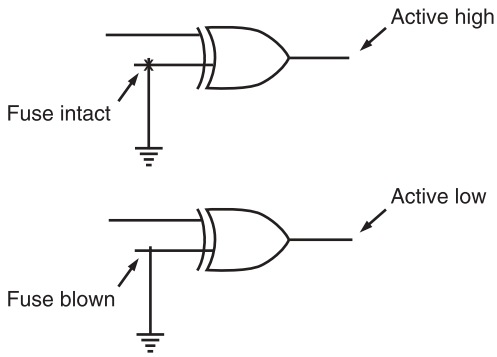


FIGURE 3.85 Programmable output polarity.

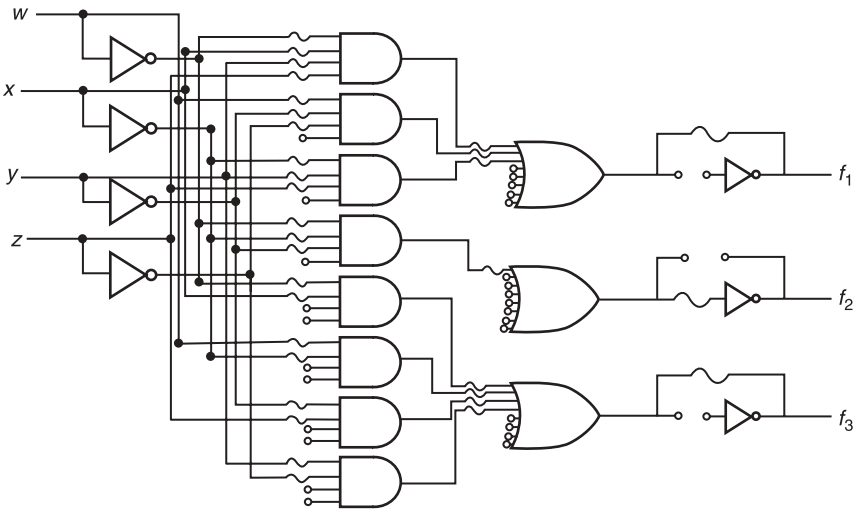


FIGURE 3.86 Logic diagram of the programmed PLA.

equations are obtained:

$$W = abde + abcd + bcde + acde + abce$$

$$X = a\bar{b}\bar{c}\bar{d}\bar{e} + \bar{a}\bar{b}\bar{c}\bar{d}e + \bar{a}b\bar{c}\bar{d}\bar{e} + \bar{a}\bar{b}c\bar{d}\bar{e} + \bar{a}\bar{b}\bar{c}d\bar{e} + a\bar{b}c\bar{d}e + ab\bar{c}d\bar{e} + \bar{a}b\bar{c}de$$
$$+ a\bar{b}cd\bar{e} + \bar{a}bc\bar{d}e + \bar{a}bcd\bar{e} + abcde + ab\bar{c}\bar{d}e + a\bar{b}\bar{c}de + abc\bar{d}\bar{e} + \bar{a}\bar{b}cde$$

$$Y = a\bar{b}\bar{d}e + ab\bar{c}\bar{d}e + a\bar{b}\bar{c}de + ab\bar{c}\bar{e} + \bar{a}\bar{c}de + a\bar{b}d\bar{e} + a\bar{b}c\bar{e} + \bar{a}c\bar{d}e + \bar{a}b\bar{d}e$$
$$+ abc\bar{d}\bar{e} + \bar{a}\bar{b}cde + \bar{a}cd\bar{e} + \bar{a}bd\bar{e} + \bar{a}bc\bar{e}$$

Output expressions X and Y have four shared product terms, $ab\bar{c}\bar{d}e$, $a\bar{b}\bar{c}de$, $abc\bar{d}\bar{e}$, and $\bar{a}\bar{b}cde$. Thus the total number of unique product terms needed to implement the circuit is 31. Note that multioutput minimization often leads to sharing of product terms, which may not obvious from the truth table. In general, determining whether a design will *fit* in a particular PLD required sophisticated software capability.

3.14.3 PAL

The basic PAL architecture is exactly opposite to that of a PROM. It is comprised of a programmable AND array and a fixed OR array. The programmability in the AND array removes one of the main deficiencies in PROM devices—that the AND plane must be large enough to produce product terms corresponding to all possible input combinations. Thus, as in PLAs, only the desired input combinations have to be programmed. Moreover, logic minimization techniques can be employed to further reduce the required number of product terms. However, since the OR array is not programmable, only a fixed number of product terms, typically eight, can drive a specific OR gate.

TABLE 3.10 A Truth Table for Circuit rd53, an MCNC (Microelectronics Center of North Carolina) Benchmark Circuit

Input					Output		
a	*b*	*c*	*d*	*e*	*W*	*X*	*Y*
1	–	1	1	1	1	–	–
1	1	–	1	1	1	–	–
1	1	1	1	–	1	–	–
1	1	1	–	1	1	–	–
–	1	1	1	1	1	–	–
0	1	–	0	1	–	–	1
–	0	1	1	0	–	–	1
0	0	1	–	1	–	–	1
1	–	0	0	1	–	–	1
1	–	1	0	0	–	–	1
1	1	0	–	0	–	–	1
0	1	1	–	0	–	–	1
1	0	0	1	–	–	–	1
0	–	0	1	1	–	–	1
–	1	0	1	0	–	–	1
–	0	1	0	1	–	–	1
0	1	1	1	0	–	1	–
0	0	0	1	0	–	1	–
0	1	0	0	0	–	1	–
1	1	1	1	1	–	1	–
0	0	1	0	0	–	1	–
0	0	1	1	1	–	1	–
1	1	1	0	0	–	1	–
1	1	0	1	0	–	1	–
0	1	1	0	1	–	1	–
0	1	0	1	1	–	1	–
1	0	1	1	0	–	1	–
1	0	0	0	0	–	1	–
1	1	0	0	1	–	1	–
0	0	0	0	1	–	1	–
1	0	1	0	1	–	1	–
1	0	0	1	1	–	1	–

PAL devices with active high outputs can implement AND–OR logic, and the devices with active-low outputs can implement AND–NOR logic. Figure 3.87*a* and 3.87*b* show the logic for a cell from an active-high and an active-low PAL, respectively. Any combinational function represented by NAND–NAND, OR–NAND, or NOR–OR logic can be replaced by a PAL with active-high output. Similarly, a function in NAND–AND. OR–AND, or NOR–NOR form can be realized by a PAL with active-low outputs.

Let us illustrate the application of combinational PAL devices in logic design by using such a device to implement a circuit with four inputs (a, b, c, d) and two outputs (X, Y) [3]. One output (X) is high when the majority of the inputs are high and low at other times. The remaining output (Y) is high only during a tie (i.e., when two inputs are high and two are low). Table 3.11 shows the truth table for the circuit.

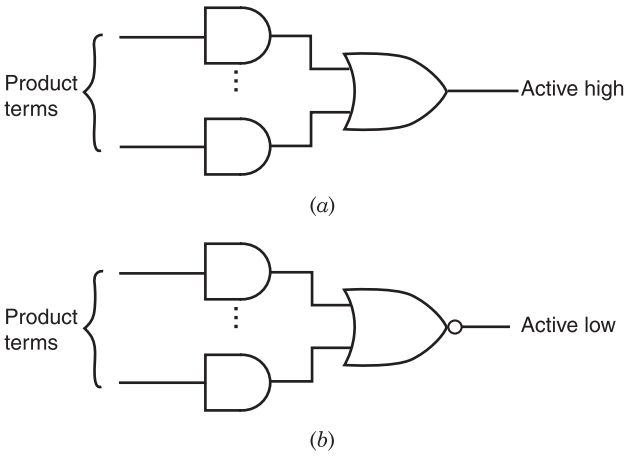


FIGURE 3.87 Structure of active-low/active-high output PAL devices.

The output expression for X and Y can be minimized using Karnaugh maps as shown in Figure 3.88. From the Karnaugh map for X we get

$$X = abd + abc + bcd + acd \tag{3.15}$$

The expression for X can also be represented as

$$X = \overline{\bar{c}\bar{d} + \bar{a}\bar{b} + \bar{b}\bar{c}d + \bar{a}c\bar{d} + \bar{b}c\bar{d} + \bar{a}\bar{c}d} \tag{3.16}$$

TABLE 3.11 Truth Table for 4-Input and 3-Output Circuit

Input				Output	
a	b	c	d	X	Y
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	1	1	0
1	1	1	0	1	0
1	1	1	1	1	0

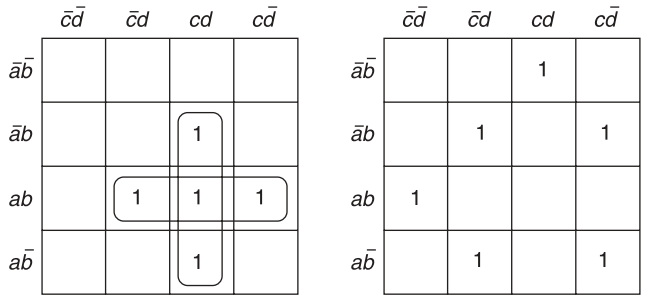
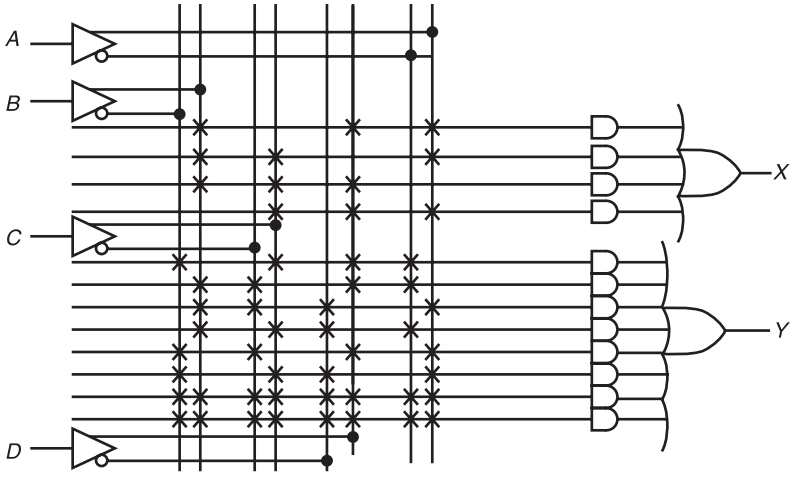
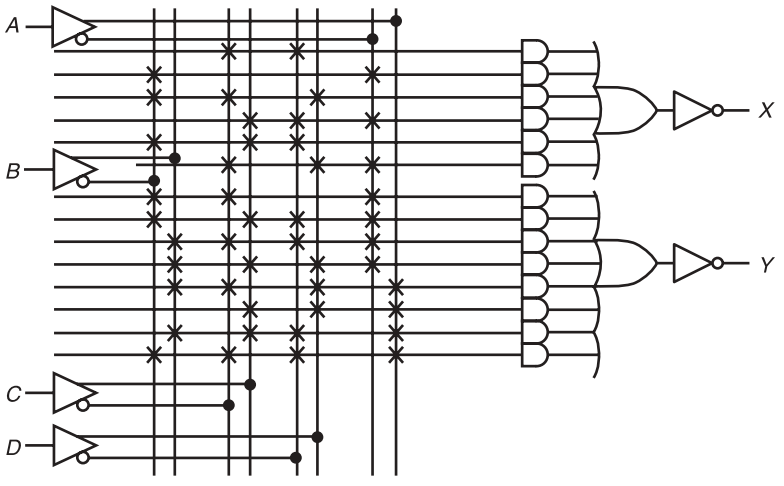


FIGURE 3.88 Karnaugh maps for output expressions X and Y.



(*a*)



(*b*)

FIGURE 3.89 (*a*) Active-high PAL and (*b*) active-low PAL.

Similarly,

$$Y = \bar{a}\bar{b}cd + \bar{a}b\bar{c}d + \bar{a}bc\bar{d} + ab\bar{c}\bar{d} + a\bar{b}\bar{c}d + a\bar{b}c\bar{d} \tag{3.17}$$

and also

$$Y = \overline{\bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}c\bar{d} + \bar{a}b\bar{c}\bar{d} + \bar{a}bcd + ab\bar{c}d + acd + abc\bar{d} + a\bar{b}\bar{c}\bar{d}} \tag{3.18}$$

Figure 3.89*a* and 3.89*b* shows the implementation of the expressions using an active-high and active-low device, respectively.

EXERCISES

1. Derive the truth tables for the following functions:

a. $f(a, b, c) = ac + ab + bc$

b. $f(a, b, c, d) = (a\bar{c} + b\bar{d})(\bar{a}b\bar{c} + a\bar{d})$

c. $f(a, b, c, d) = a \oplus b \oplus c + \bar{a}\bar{c}\bar{d}$

2. Derive the minterm and the maxterm list form of the Boolean functions specified by the following truth tables:

a.

a	b	c	$f(a, b, c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

b.

a	b	c	d	$f(a, b, c, d)$
0	0	0	0	1
0	0	0	1	1
0	0	1	1	1
0	1	1	1	1
1	0	0	0	1
1	1	0	0	1
1	0	1	1	1
All other combinations				0

3. Derive the product-of-sums form of the following sum-or-products Boolean expressions:

a. $f(a, b, c) = \bar{a}\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + \bar{a}bc$

b. $f(a, b, c, d) = ac + ad + ab + c\bar{d}$

c. $f(a, b, c, d) = \bar{a}\bar{b}\bar{c} + a\bar{b}cd + a\bar{c}$

4. Derive the canonical product-of-sums form of the following functions:

a. $f(a, b, c) = (a + \bar{b}) + (\bar{a} + \bar{c})(b + c)$

b. $f(a, b, c, d) = (a + \bar{b} + d)(b + \bar{c} + \bar{d})(\bar{a} + c + \bar{d})(b + c + d)$

c. $f(a, b, c, d, e) = (a + \bar{b} + c + \bar{e})(b + \bar{c} + d + \bar{e})(\bar{a} + c + \bar{d} + e)$

5. Derive the canonical sum-of-products form of the following functions:

a. $f(a, b, c, d) = ac\bar{d} + a\bar{b}c + \bar{b}cd + \bar{a}d + ab$

b. $f(a, b, c, d) = \bar{a}b + a\bar{c} + ad$

c. $f(a, b, c, d, e) = \bar{a}bc\bar{e} + b\bar{d}e + abd\bar{e} + \bar{a}b\bar{c}e + \bar{a}de$

6. Minimize the following functions using Karnaugh maps:

a. $f(a, b, c, d) = \bar{a}\bar{b}\bar{c}\bar{d} + ab\bar{c}\bar{d} + a\bar{b}\bar{c}\bar{d} + abcd + \bar{a}\bar{b}cd + a\bar{b}cd + \bar{a}bcd$

b. $f(a, b, c, d) = \Sigma m(1, 3, 4, 6, 9, 11, 13, 15)$

c. $f(a, b, c, d) = \prod M(3, 6, 7, 9, 11, 12, 13, 14, 15)$

d. $f(a, b, c, d, e) = \Sigma m(4, 8, 10, 15, 17, 20, 22, 26) + d(2, 3, 12, 21, 27)$

7. Derive the set of prime implicants for the following functions using the Quine–McCluskey method. In each case identify the essential prime implicants if there are any.

a. $f(a, b, c, d) = \Sigma m(0, 1, 2, 3, 7, 9, 12, 13, 14, 15, 22, 23, 29, 31)$

b. $f(a, b, c, d) = \Sigma m(5, 6, 7, 10, 14, 15) + d(9, 11)$

c. $f(a, b, c, d) = \Sigma m(1, 7, 9, 11, 13, 21, 24, 25, 30, 31) + d(0, 2, 6, 8, 15, 17, 22, 28, 29)$

8. The prime implicant chart for two Boolean functions are shown in (a) and (b). Obtain the minimal sum-of-products expression for each case.

a.

	0	*4*	*5*	*6*	*11*	*13*	*15*
$PI_1 = \bar{a}\bar{c}$	X	X	X				
$PI_2 = \bar{c}d$			X			X	
$PI_3 = \bar{b}d$					X		
$PI_4 = ad$					X	X	X
$PI_5 = \bar{a}b\bar{d}$		X		X			

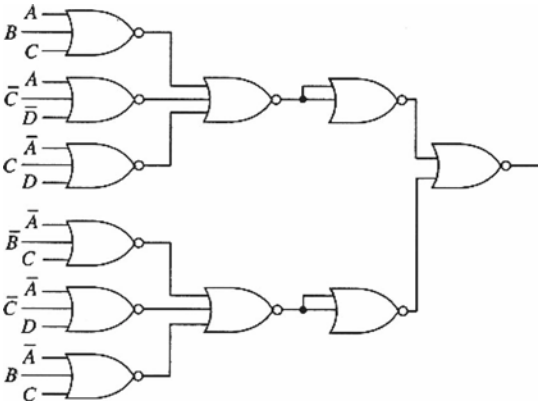
b.

	0	*1*	*4*	*5*	*6*	*7*	*9*	*11*	*15*
$PI_1 = \bar{a}\bar{c}$	X	X	X	X					
$PI_2 = \bar{a}b$			X	X	X	X			
$PI_3 = ac$								X	X
$PI_4 = bc$					X	X			X
$PI_5 = a\bar{b}d$							X	X	

9. Implement the function $f(w, x, y, z) = (w + \bar{x})(w + \bar{x} + \bar{y})(y + \bar{z})$ using NOR–NOR logic.

10. Design a combinational circuit to generate the parity bit for digits coded in BCD code. The circuit should also have an additional output that produces an error signal if a non-BCD digit is input to the circuit. Realize the circuit using NAND–NAND logic.

11. Determine the Boolean function for the circuit shown. Obtain an equivalent circuit with fewer NOR gates. (Assume only 2-input or 3-input NOR gates.)



12. Assuming you have any number of 8-to-1 multiplexers and a single 4-to-1 multiplexer, design a 32-to-1 multiplexer.

13. Show how two 2-to-1 multiplexers can be used to implement a half-adder.

14. Implement each of the following functions using an appropriate multiplexer:

a. $f(w, x, y, z) = \Sigma m(3, 5, 7, 9, 10, 11, 13)$

b. $f(w, x, y, z) = w\bar{y}\bar{z} + \bar{x}z$

c. $f(t, w, x, y, z) = \bar{w} + txy + w\bar{x}z + \bar{t}\bar{y}\bar{z}$

15. Design a multioutput circuit whose input is BCD data and whose outputs

w: detects numbers that are divisible by 3

and

x: detects numbers greater than or equal to 4

Implement the circuit using an appropriate decoder.

16. A combinational circuit has six inputs x_i $(i = 1, \ldots, 6)$ and six outputs z_i $(i = 1, \ldots, 6)$. An output z_j is to be 1 if and only if x_j is 1 and each $x_i = 0$ for all $i < j$. Implement the circuit using decoder(s) and the minimum number of gates.

17. A combinational circuit is to be designed to control a seven-segment display of decimal digits. The inputs to the circuit are BCD codes. The seven outputs correspond to the segments that are activated to display a given decimal digit.

a. Develop a truth table for the circuit.

b. Derive simplified sum-of-products and product-of-sums expressions.

c. Implement the expressions using either NAND or NOR gates as appropriate.

18. Given the following Boolean expression,

$$f(A, B, C) = AB\bar{C} + \bar{A}BC + A\bar{B}C + \bar{A}\bar{B}C$$

a. Develop an equivalent expression using NAND functions only, and draw the logic diagram.

b. Develop an equivalent expression using NOR functions only, and draw the logic diagram.

19. A certain "democratic" country is ruled by a family of four members (A, B, C, and D). A has 35 votes, B has 40 votes, C has 15 votes, and D has 10 votes. Any decision taken by the family is based on its receiving at least 60% of the total number of votes. Design a circuit that will produce an output of 1 if a certain motion is approved by the family.

20. In a digital system a circuit is required to compare two 3-bit binary numbers, $X = x_2x_1x_0$ and $Y = y_2y_1y_0$, and generate separate outputs corresponding to the conditions $X = Y$, $X > Y$, and $X < Y$. Implement the circuit using NAND gates only.

21. A 6-to-64 decoder is to be implemented using 3-to-8 decoders only. Show the block diagram of the 6-to-64 decoder.

22. Design a combinational circuit to generate a parity (even) bit for digits coded in 5421 code. Provide an error output if the input to the circuit is not a 5421 code.

23. Implement the following functions using NAND gates having a maximum fan-in of three.

a. $f(A, B, C, D) = \Sigma m(0, 1, 3, 7, 8, 12) + d(5, 10, 13, 14)$

b. $f(A, B, C, D) = AB(C + D) + CD(A + B)$

24. Implement the following function using NOR gates having a maximum fan-in of three.

a. $f(A, B, C, D, E) = \bar{A}(\bar{B} + C + DE)(\bar{B} + CD + \bar{A}E)$

b. $f(A, B, C, D) = \bar{A}B + \bar{B}CD + A\bar{B}\bar{D}$

25. Find the reduced cover for the following function using *EXPAND*, *REDUCE*, and *IRREDUNDANT* operations.

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

26. Derive the cofactors of the function in Exercise 25 with respect A, $\bar{B}$, and $B\bar{C}$.

27. Prove the following:

a. $A \oplus B = A \oplus B \oplus AB$

b. $A \oplus (\overline{B \oplus C}) = (\overline{A \oplus B \oplus C})$

28. It is required to design lighting for a room such that the lights may be switched on or off from any one of three switch points. Implement the circuit using the minimum number of EX-OR gates.

29. Derive the kernels and cokernels of the following function:

$$F(a,b,c,d,e,f,g) = adf + aef + bdf + bef + cdf + cef + g$$

30. Using the *rectangular covering* approach discussed in the text, extract all common cubes from the following expressions, and also determine how theses cubes can be utilized to minimize the number of literals in the expressions.

a. $W = be + ef + ab + df + ad$

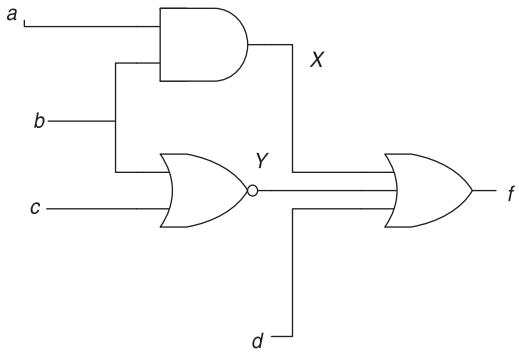
b. $X = ac + af + ef + ab$

c. $Y = aef + ac + bde + cde$

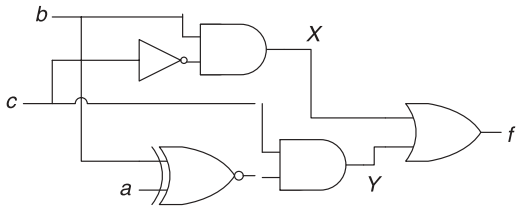
31. Implement the logic circuit represented by the following Boolean functions in a multi-level form*:

$$f_1(a,b,c,d,e,t,g) = ac + ade + bc + bde$$
$$f_2(a,b,c,d,e,t,g) = afg + bfg + efg$$

32. Find all the *satisfiablity don't cares* in the following circuit:



33. Find the *observability don't cares* for nodes X and Y in the following circuit:



34. A combinational circuit that generates the square of all the combinations of a 3-bit binary number is to be implemented using a PLA. Show the program table for implementing the circuit using the format shown in the text.

35. Implement the following Boolean functions using a PROM:

a. $f_1(w, x, y, z) = wxy + w\bar{x}\bar{y}$

b. $f_2(w, x, y, z) = \bar{w} + \bar{x} + y + z$

c. $f_3(w, x, y, z) = w + \bar{x} + \bar{y}z + wz$

*MCNC (Microelectronics Center of North Carolina), Technical Report TR87-15.

d. $f_4(w, x, y, z) = wyz + w\bar{y}\bar{z} + \bar{x}yz + \bar{w}xz$

36. Design an 8-bit ripple carry adder using the 1's complement form to represent negative numbers.

37. Design a circuit capable of adding two 4-bit numbers such that its output is the mod-5 sum of the inputs.

38. Using the principle of carry-save addition, multiply the following pairs:

a. 11011 and 10010.

b. 1010111 and 1100110.

39. Design a 24-bit adder that uses six 4-bit adder circuits, carry-lookahead circuits, and additional logic to generate C_8 and C_{16} from carry-in, carry-generate, and carry-propagate variables.

40. Prove that if two 2's complement numbers are added, the overflow bit is the EX-OR of the carry-in and carry-out of the most significant bit.

41. Show how a 4-bit adder can be used to convert 5-bit BCD representation of decimal numbers 0 to 19 to 5-bit binary numbers.

42. Design a circuit, using full adders only, to multiply a 4-bit number by decimal 10.

43. Show how an 8-bit number ($X_7 \cdots X_0$) can be multiplied by a 4-bit number ($X_3 \cdots X_0$) using four 2-bit multipliers and 6-bit adders.

REFERENCES

1. Giovanni De Michelli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1995.
2. Gary Hachtel and Fabio Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Publishers, Norwell, MA, 1996.
3. F. Cave and D. Terrel, *Digital Technology with Microprocessors*, Reston Publishers, 1981.

4 Fundamentals of Synchronous Sequential Circuits

4.1 INTRODUCTION

Combinational logic refers to circuits whose output is strictly dependent on the present value of the inputs. Once the input values are changed, the information regarding the previous inputs is lost; in other words, combinational logic circuits have no memory. In many applications, information regarding input values at a certain instant of time is needed at some future time. Circuits whose output depends not only on the present values of the input but also on the past values of the inputs are known as sequential logic circuits. The mathematical model of a sequential circuit is usually referred to as a *sequential machine* or a *finite state machine*.

A general model of a sequential circuit is shown in Figure 4.1. As can be seen in the diagram, sequential circuits are basically combinational circuits with the additional property of memory (to remember past inputs). The combinational part of the circuit receives two sets of input signals: *primary* (coming from the circuit environment) and *secondary* (coming from the memory). The particular combination of secondary input variables at a given time is called the *present state* of the circuit; the secondary input variables are also known as state variables. If there are m secondary input variables in a sequential circuit, then the circuit can be in anyone of 2^m different present states.

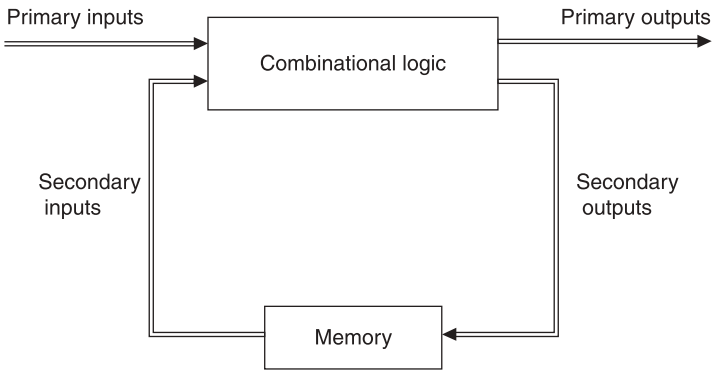


FIGURE 4.1 General model of a sequential logic circuit.

Principles of Modern Digital Design, by Parag K. Lala

The outputs of the combinational part of the circuit are divided into two sets. The primary outputs are available to control operations in the circuit environment, whereas the secondary outputs are used to specify the *next state* to be assumed by the memory. The number of secondary output variables, often called the *excitation variables*, depends on the type of memory element used.

4.2 SYNCHRONOUS AND ASYNCHRONOUS OPERATION

Sequential logic circuits can be categorized into two classes: *synchronous* and *asynchronous*. In synchronous circuits internal states change at discrete instants of time under the control of a synchronizing pulse, called the *clock*. The clock is generally some form of square wave as illustrated in Figure 4.2. The *on-time* is defined as the time the wave is in the 1 state; the *off-time* is defined as the time the wave is in the 0 state. The *duty cycle* is defined as

$$\text{Duty cycle} = \frac{\text{(On-time)}}{\text{(Period)}} \quad \text{(expressed as a percentage)}$$

State transitions in synchronous sequential circuits are made to take place at times when the clock is making a transition from 0 to l or from 1 to 0. The 0-to-1 transition is called the *positive edge* or the *rising edge* of the clock signal, whereas the 1-to-0 transition is called the *negative edge* or the *falling edge* of the clock signal (as shown in Fig. 4.2). Between successive clock pulses there is no change in the information stored in memory. Synchronous sequential circuits are also known as *clocked sequential circuits*.

In asynchronous sequential circuits the transition from one state to another is initiated by the change in the primary inputs; there is no external synchronization. Since state transitions do not have to occur at specific instants of time, asynchronous circuits can operate at their own speed. The memory portion of asynchronous circuits is usually implemented by feedback among logic gates. Thus asynchronous circuits can be regarded as combinational circuits with feedback. Because of the difference in the delays through various signal paths, such circuits can give rise to transient conditions during the change of

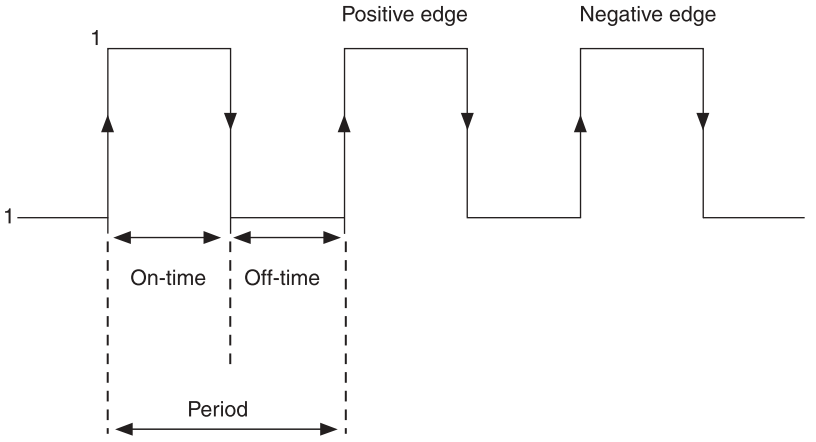


FIGURE 4.2 Clock signal.

inputs or state variables. Hence such circuits have to be designed in a special way in order to ensure that their operations are not affected by transient conditions.

4.3 LATCHES

As can be seen from Figure 4.1, the memory unit is an essential part of a sequential circuit. This unit usually consists of a series of *latches*. A latch is a logic circuit with two outputs, which are the complement of each other. A basic latch can be constructed by cross-coupling two NOR gates as shown in Figure 4.3. The two inputs are labeled *Set* and *Reset*. The outputs Q and $\bar{Q}$ are always the complement of each other during normal operation. Let us assume both inputs are at 0 initially, output Q is at 1, and output $\bar{Q}$ is at 0. Thus the output of gate G_1 is at 1. Since the output of gate G_1 is fed back to the input of gate G_2, the output of G_2 will be 0. The circuit is therefore stable with Q at 1 and $\bar{Q}$ at 0, as was originally assumed.

If the Reset input is now taken to 1, the output of G_1 will change to 0. Both the inputs of G_2 are at 0, so its output will change to 1. The circuit is now stable with $Q = 0$ and $\bar{Q} = 1$. The circuit remains in this stable state even if the Reset input is changed back to 0. If the Set input is now taken to 1, the output of G_2 (i.e., $\bar{Q}$) will be at 0. Since both inputs of G_1 are now at 0, its output (i.e., Q) will be at 1. The circuit remains stable with $\bar{Q} = 1$ and $Q = 0$ even when the Set input returns to 0.

The input combination Set = 1 and Reset = 1 is not allowed because both Q and $\bar{Q}$ go to 0 in this case, which violates the condition that $\bar{Q}$ and Q should be complements of each other. Furthermore, when the Set and Reset inputs are returned to 0, an ambiguous situation arises. For example, if the propagation delay of G_1 is lower than that of G_2, then the output of G_1 (i.e., Q) will change to 1 first. This in turn will make the output of G_2 (i.e., $\bar{Q}$) = 0. On the other hand, if the output of G_2 changes to 1 first, then the output of G_1 will be forced to 0. In other words, it is impossible to predict the output. Therefore the Set = 1 and Reset = 1 combination is avoided during operation of this type of latch. The behavior of the latch circuit can be represented by the truth table of Figure 4.4*a*. The cross-coupled NOR latch is generally known as an *SR* (Set–Reset) latch. The logic symbol used to represent the *SR* latch is shown in Figure 4.4*b*.

An *SR* latch can also be constructed by cross-coupling NAND gates as shown in Figure 4.5*a*. The circuit operates in the same manner as the NOR latch, but it does have a few subtle differences. Unlike the NOR latch, the NAND latch inputs are normally 1 and must be changed to 0 to change the output. An ambiguous output results when both the Set and the Reset inputs are at 0. Figure 4.5*b* shows the truth table of the NAND latch. The logic symbol for the NAND latch is shown in Figure 4.5*c*; the circles denote that the latch responds to 0 on its inputs (i.e., it has active-low inputs).

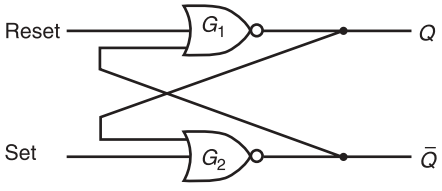
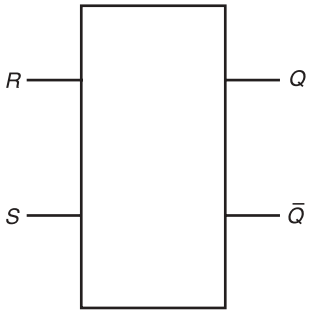


FIGURE 4.3 Cross-coupled NOR flip-flop.

Set	Reset	Q	$\bar{Q}$
0	0	No change	
0	1	0	1
1	0	1	0
1	1	Ambiguous	

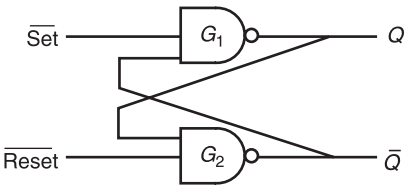
(*a*)



(*b*)

FIGURE 4.4 (*a*) Truth tabe for NOR latch and (*b*) logic symbol.

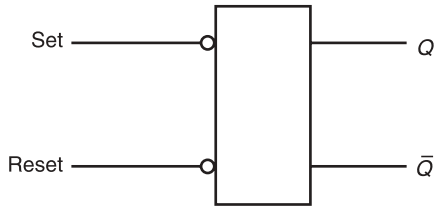
In many applications a latch has to be set or reset in synchronization with a control signal. Figure 4.6*a* shows how the NAND latch of Figure 4.5*a* has been modified to incorporate a control input, which is usually driven by a clock. The resulting circuit is known as a *gated latch* or *clocked latch*. As long as the control is at 0 in Figure 4.6*a*, the outputs of gates G_3 and G_4 will be 1 and the latch will not change state. When the enable input changes to 1, the set and reset inputs affect the latch. Thus if Set = 1 and Reset = 0, the output of G_3 is 0 and that of G_4 is 1, Q goes to 1, and $\bar{Q}$ goes to 0. If Set = 0 and Reset = 1, Q goes to 0. The truth table for the latch is constructed as shown in Figure 4.6*b*. Note that Q_t is the present state of the latch, and Q_{t+1} is the next state. The next state of the latch depends on the present state of the latch and the present value of the input. These types of latches are often called *transparent* because the output changes (after the latch propagation delay) as the inputs change, if the enable



(*a*)

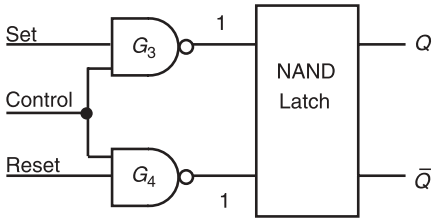
$\overline{\text{Set}}$	$\overline{\text{Reset}}$	Q	$\bar{Q}$
0	0	Ambiguous	
0	1	1	0
1	0	0	1
1	1	No change	

(*b*)



(*c*)

FIGURE 4.5 (*a*) NAND latch, (*b*) truth table, and (*c*) logic symbol for NAND latch.

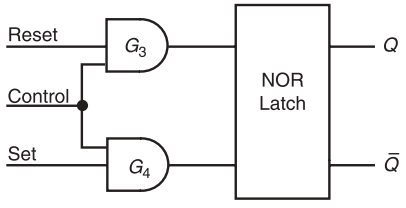


(*a*)

Control	Set	Reset	G_3	G_4	Q_{t+1}
0	—	—	1	1	Q_t
1	0	0	1	1	Q_t
1	0	1	1	0	0
1	1	0	0	1	1
1	1	1	0	0	Ambiguous

(*b*)

FIGURE 4.6 (*a*) Gated NAND latch with enable input and (*b*) truth table.



(*a*)

Control	Set	Reset	G_3	G_4	Q_{t+1}
0	—	—	0	0	Q_t
1	0	0	0	0	Q_t
1	0	1	1	0	0
1	1	0	0	1	1
1	1	1	1	1	Ambiguous

(*b*)

FIGURE 4.7 (*a*) Gated NOR latch with enable input and (*b*) truth table.

input is high. The gated NOR latch and its truth table are shown in Figure 4.7*a* and 4.7*b*, respectively.

It is possible for *SR* latches to have more than two inputs. Figure 4.8 represents an *SR* latch constructed from two 3-input NOR gates. The Q output will be 1 if any of the Set inputs is at 1 and the Reset inputs are at 0. The Q output goes to 0 if any of the Reset inputs is at 1 and all the Set inputs are at 0. As before, Set and Reset inputs are not simultaneously allowed to be at 1. A normal *SR* latch can be converted to another type of latch, known as a *D* latch, by generating Reset $= \overline{\text{Set}}$ (using an additional inverter).

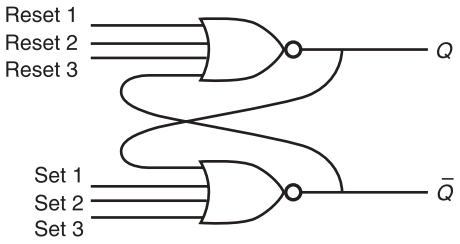
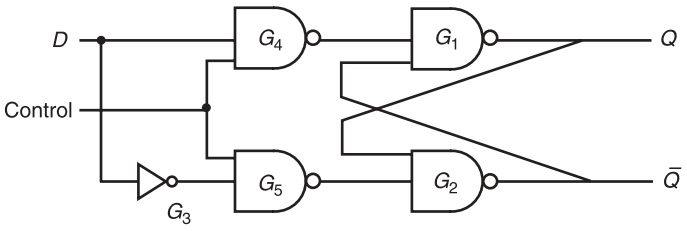


FIGURE 4.8 Three-input NOR *SR* latch.



(*a*)

D	Control	G_3	G_4	G_5	*Q*	$\bar{Q}$
0	1	1	1	0	0	1
1	1	0	0	1	1	0
—	0	—	1	1	No change	

(*b*)

FIGURE 4.9 (*a*) D latch and (*b*) truth table.

Figure 4.9*a* shows a *D* latch constructed from an *SR* NAND latch. As can be seen from the truth table of the latch (Fig. 4.9*b*), the output *Q* follows the *D* input if the control line is at 1. With the control line at 0, *Q* holds the value of *D* prior to the l-to-0 transition of the control line. The *D* latch has the advantage over the *SR* latch that no input combination produces ambiguous output, so no input has to be avoided.

4.4 FLIP-FLOPS

A flip-flop, like a gated latch, possesses two stable states, but unlike a gated latch, transitions in a flip-flop are activated by the edge rather than the level of the clock pulse on the control input. Figure 4.10*a* shows timing diagrams for a *D* latch and a *D* flip-flop. A flip-flop is often called a *bistable* element because when its *Q* output is at logic 1, the $\bar{Q}$ output is at logic 0 or vice versa. However, it is also possible for a flip-flop to be in a *metastable state*. Metastability implies that the output of a flip-flop is undefined (i.e., neither 0 nor 1) for a certain period of time. This phenomenon occurs if the data input to a flip-flop does not satisfy the specified *setup time* and *hold time* with respect to the clock.

The setup time is the period of time the data must be stable at the input of a flip-flop before the flip-flop is triggered by the clock edge. The period of time the data must remain stable after the flip-flop has been triggered is known as the hold time. Figure 4.10*b* illustrates the meaning of setup time and hold time. In order to guarantee that valid data is produced at the output of a flip-flop after a maximum clock-to-output delay time (i.e., the time from the rising edge of the clock to the time valid data is available on the output), the input data must not violate the specified setup and hold times. Otherwise, the output will be in a metastable state for a time greater than the maximum clock-to-output delay time. The indeterminate logic value produced by a flip-flop while it is in a metastable state can result in unpredictable circuit behavior.

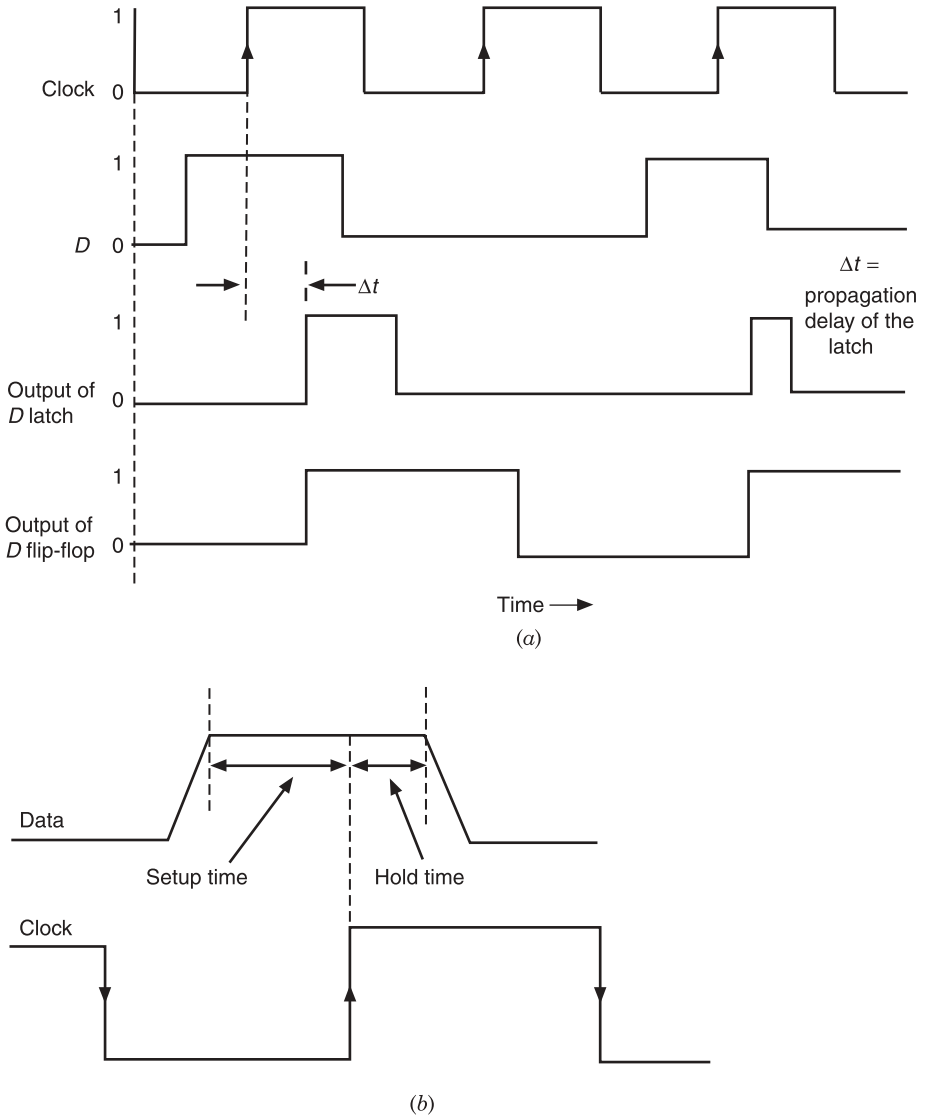
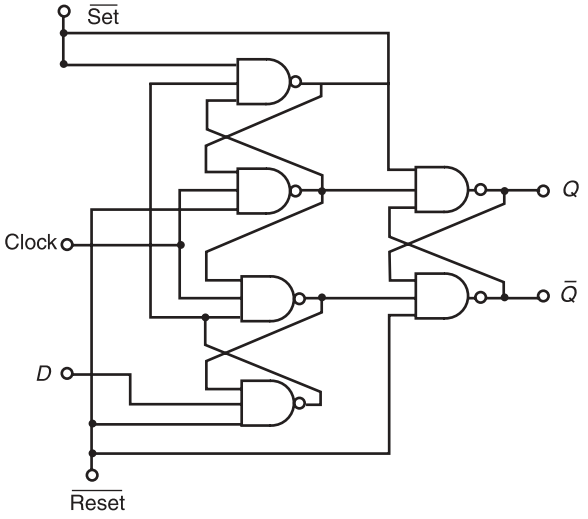


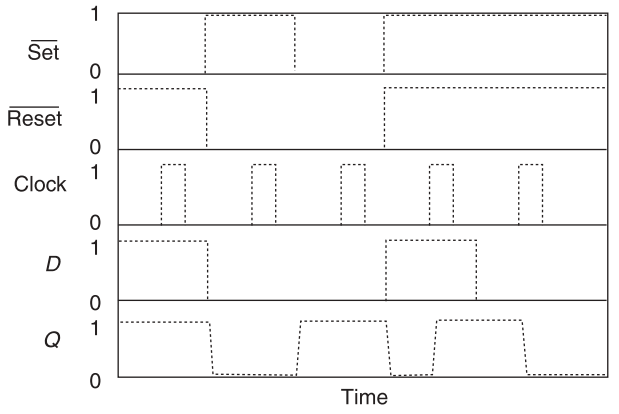
FIGURE 4.10 (*a*) Timing diagram for *D* latch and *D* flip-flop and (*b*) setup and bold times for *D* flip-flops.

4.4.1 *D* Flip-Flop

The input data to a *D* flip-flop is transferred to the output and held there on a $0 \rightarrow 1$ transition of the clock pulse (see Fig. 4.10*a*). In other words, the *D* flip-flop is triggered on the positive edge of the clock pulse. A positive edge-triggered *D* flip-flop can be implemented by modifying the NAND latch configuration as shown in Figure 4.11*a*. In addition to the *D* input, the flip-flop has a pair of direct clock-independent (asynchronous) inputs, $\overline{\text{Reset}}$ and $\overline{\text{Set}}$. When the $\overline{\text{Reset}}$ input is at logic 0, the *Q* output of the flip-flop goes to 0. On the other hand, when the $\overline{\text{Set}}$ input is at logic 0, the *Q* output goes to 1. The $\overline{\text{Set}}$ and $\overline{\text{Reset}}$ inputs are not allowed to be at 0 simultaneously because this will make both *Q* and $\bar{Q}$ outputs of the



(*a*)



(*b*)

FIGURE 4.11 (*a*) *D* flip-flop and (*b*) timing diagram of *D* flip-flop.

flip-flop go to 1; in other words, the condition that Q must always be the complement of $\bar{Q}$ in a flip-flop is violated.

Figure 4.11*b* shows the timing diagram of the *D* flip-flop. As can be seen from the diagram, the *D* flip-flop transfers the input value at *D* to the Q output on the positive edge of the clock pulse only when both $\overline{\text{Set}}$ and $\overline{\text{Reset}}$ inputs are at 1. Table 4.1 summarizes the operation of the *D* flip-flop. The logic symbol for the *D* flip-flop is shown in Figure 4.12*a*. The small circles on the Set and Reset inputs mean that when the Set input is driven to 0 the flip-flop is preset to 1, and when the Reset input is at 0 the flip-flop is reset to 0. The characteristics of the *D* flip-flop can be represented by the following equation (Fig. 4.12*b*).

$$Q_{t+1} = D_t$$

In other words, the next state of a *D* flip-flop corresponds to the data input applied at time t and is independent of the present state of the flip-flop.

TABLE 4.1 Function Table of the *D* Flip-Flop

$\overline{\text{Set}}$	$\overline{\text{Reset}}$	D	Clock	Q	$\bar{Q}$
0	1	–	–	1	0
1	0	–	–	0	1
0	0	–	–	1	1
1	1	0	$0 \rightarrow 1$	0	1
1	1	1	$0 \rightarrow 1$	1	0

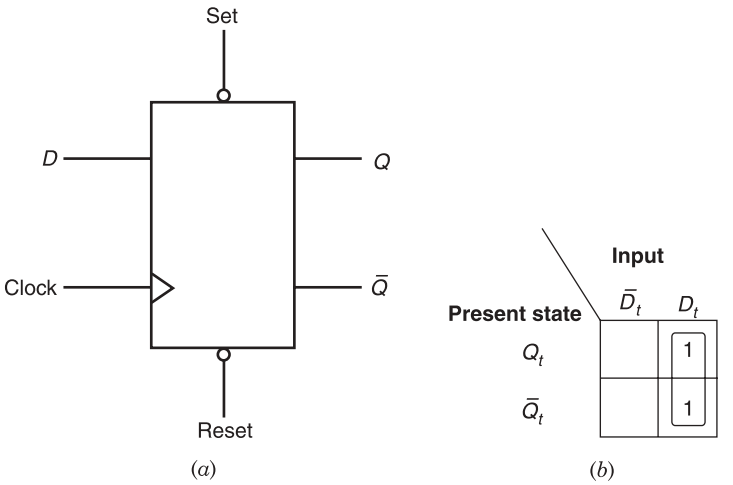


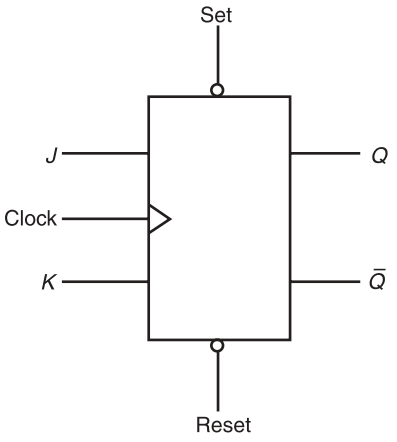
FIGURE 4.12 (*a*) Logic symbol for *D* flip-flop and (*b*) Karnaugh map for deriving characteristic equation.

4.4.2 *JK* Flip-Flop

The *JK* flip-flop has similar functions to an *SR* latch, with *J* equivalent to Set and *K* equivalent to Reset. In addition, if both *J* and *K* are set to 1, the outputs complement when the flip-flop is clocked. Thus a *JK* flip-flop does not have any invalid input combinations. Figure 4.13*a* shows the implementation of a positive-edge-triggered *JK* flip-flop. The analysis of the circuit operation is similar to that of the *D* flip-flop. The logic symbol and the function table for the flip-flop are shown in Figure 4.l3*a* and 4.13*b*, respectively. The characteristic equation for the *JK* flip-flop can be derived from its truth table (see Fig. 4.l3*c*) and is given by

$$Q_{t+l} = J_t\bar{Q}_t + \bar{K}_tQ_t$$

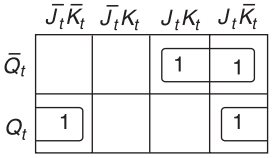
The characteristic equation indicates that a *JK* flip-flop can also be implemented using a *D* flip-flop. This is shown in Figure 4.14. The *JK* flip-flops are often used in place of *SR* latches, as these devices are not commonly available as commercial parts. Figure 4.15*a* shows the proper connections of a *JK* flip-flop in order to operate as an *SR* latch.



(*a*)

Present State	Inputs			Next State
Q_t	J_t	K_t	Clock	Q_{t+1}
0	0	0	0 → 1	0
0	0	1	0 → 1	0
0	1	0	0 → 1	1
0	1	1	0 → 1	1
1	0	0	0 → 1	1
1	0	1	0 → 1	0
1	1	0	0 → 1	1
1	1	1	0 → 1	0

(*b*)



(*c*)

FIGURE 4.13 (*a*) Logic symbol for *JK* flip-flop, (*b*) truth table, and (*c*) Karnaugh map for deriving characteristic equation.

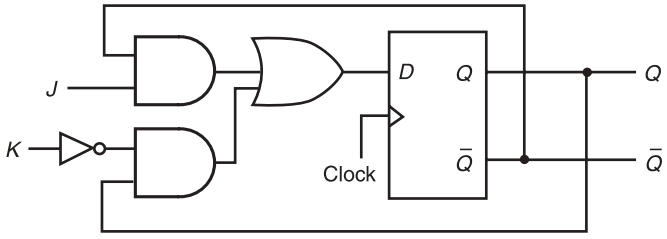


FIGURE 4.14 A *JK* flip-flop constructed from a *D* flip-flop.

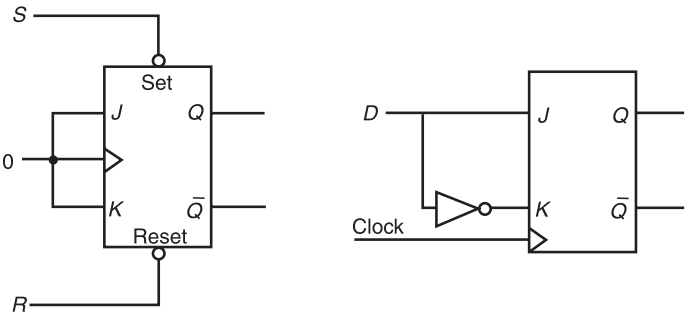


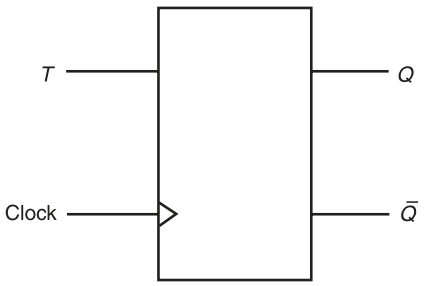
FIGURE 4.15 (*a*) *JK* flip-flop as an *SR* latch and (*b*) *D* flip-flop.

A *JK* flip-flop can also operate as a *D* flip-flop; the configuration used to achieve this is shown in Figure 4.15*b*. In this case $J = D$ and $K = \bar{D}$; thus from the *JK* flip-flop characteristic equation,

$$Q_{t+1} = D_t\bar{Q}_t + \bar{D}_tQ_t = D_t$$

4.4.3 *T* Flip-Flop

The *T* flip-flop, known as a *toggle* or *trigger* flip-flop, has a single input line. The symbol for the *T* flip-flop is shown in Figure 4.16*a*. If $T = 1$ when the clock pulse changes from 0 to 1,



Present State	Inputs		Next State
Q_t	T	Clock	Q_{t+1}
0	0	0→1	0
0	1	0→1	1
1	0	0→1	1
1	1	0→1	0

(*b*)

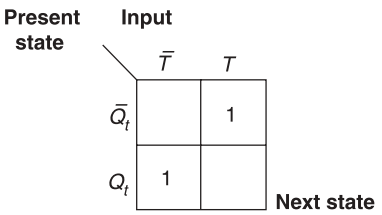


FIGURE 4.16 (*a*) *T* flip-flop, (*b*) truth table, and (*c*) Karnaugh map for deriving characteristic equation.

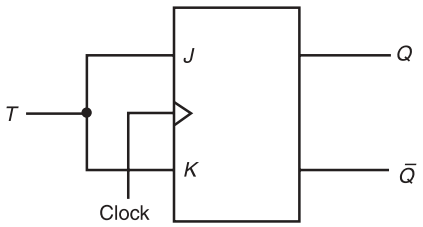


FIGURE 4.17 A *T* flip-flop constructed from a *JK* flip-flop.

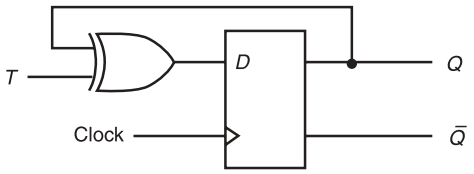


FIGURE 4.18 A *T* flip-flop constructed from a *D* flip-flop.

the flip-flop assumes the complement of its present state; if $T = 0$, the flip-flop does not change state. The function table of the flip-flop is shown in Figure 4.16*b*. The characteristic equation for the flip-flop can be derived from its function table, as shown in Figure 4.16*c*.

$$Q_{t+1} = Q_t\bar{T} + \bar{Q}_t T$$

Thus a *T* flip-flop can be considered as a single-input version of a *JK* flip-flop. A *JK* flip-flop can be configured as shown in Figure 4.17 to operate as a *T* flip-flop. Alternatively, a *T* flip-flop can also be derived from a *D* flip-flop (Fig. 4.18). The *D* input is driven by an exclusive-OR gate, which in turn is fed by the *Q* output of the flip-flop and the *T* input line, as dictated by the characteristic equation of the *T* flip-flop.

4.5 TIMING IN SYNCHRONOUS SEQUENTIAL CIRCUITS

As mentioned previously, in a synchronous sequential circuit the state transition (i.e., the change in the outputs of the flip-flops) occurs in synchronization with a pulse. In positive-edge-triggered flip-flops, the delay between the positive-going transition on the clock input and the changes on the outputs is specified as propagation delay, t_{pd}. Usually the delays are different for the two possible directions of output change and are specified as:

t_{pLH}: The delay between the transition midpoint of the clock signal and the transition midpoint of the output, where the output is changing from low to high. This delay is also referred to as *rise delay*.

t_{pHL}: The delay between the transition midpoint of the clock signal and the transition mid-point of the output, where the output is changing from high to low. This delay is also referred to as *fall delay*.

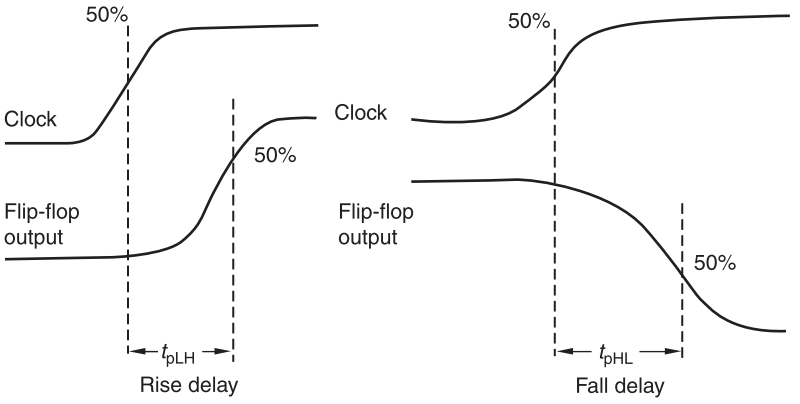


FIGURE 4.19 Flip-flop propagation delay.

Figure 4.19 illustrates these propagation delays. The values of t_{pLH} and t_{pHL} are generally not the same. The manufacturers' data sheets usually specify typical and maximum values; the minimum value must obviously be nonzero.

The clock frequency f_{max}, which determines the maximum speed at which a synchronous sequential circuit can reliably operate, is related to the minimum clock period T_{min} by

$$f_{max} = \frac{1}{T_{min}}$$

where

$$\begin{aligned} T_{min} = {} & \text{minimum setup time for flip-flops} \\ & + \text{minimum hold time for flip-flops} \\ & + \text{maximum gate propagation delay} \\ & + \text{maximum flip-flop propagation delay} \end{aligned}$$

Figure 4.20 shows a very simple synchronous sequential circuit, designed using two-level NAND gates and two flip-flops. Assuming the gates have propagation delays of 4 ns, the propagation delay of the flip-flops is 10 ns, their setup time is 3 ns, and the hold time is 1 ns, we first determine T_{min}:

$$T_{min} = 3 + 1 + 2 \times 4 + 10 = 22 \text{ ns}$$

Thus the maximum clock frequency is

$$f_{max} = \frac{1}{22 \times 10^{-9}} = 45.5 \text{ MHz}$$

One additional consideration that has to be taken into account in sequential circuits is the *clock skew*. So far, we have assumed that all the flip-flops in the circuit are triggered

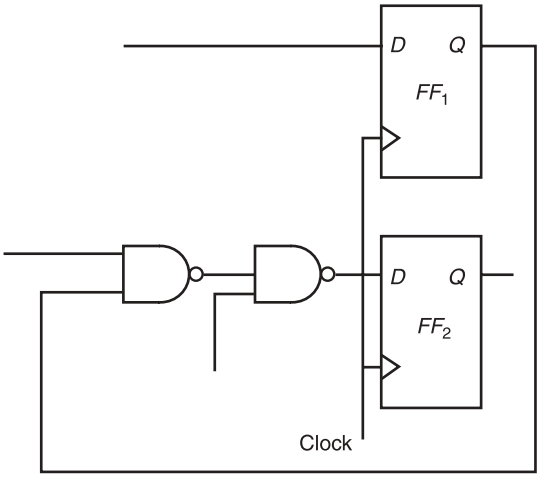


FIGURE 4.20 A sequential circuit.

simultaneously. However, in a large circuit this assumption is rarely true, so it is more realistic to assume that the clock signal appears at the clock inputs of the various flip-flops at different times. This is due to the delays in the conducting paths between the clock generator and the flip-flops as well as the delay variations between different clock buffers. Hence the flip-flops are triggered at different times, resulting in incorrect circuit operation. For example, in the circuit of Figure 4.20, FF_1 drives FF_2 through two gates with delay t_g (= 8 ns by previous assumption). If the clock signal arrives at FF_2 later than FF_1, this delay must be less than

$$\begin{aligned}\Delta t_{\max} &= t_{\text{pd}}(FF_1 + t_g + t_s(FF_2)\\ &= 10 + 8 + 3 = 21 \text{ ns}\end{aligned}$$

for the circuit to operate properly. If the clock pulse arrives later than $\Delta t_{\max}$, then the new state of FF_1 will be clocked into FF_2.

4.6 STATE TABLES AND STATE DIAGRAMS

In Section 4.1 we examined a general model for sequential circuits. In this model the effect of all previous inputs on the outputs is represented by a state of the circuit. Thus the output of the circuit at any time depends on its current state and the input; these also determine the next state of the circuit. The relationship that exists among primary input variables, present state variables, next state variables, and output variables can be specified by either the *state table* or the *state diagram*. In the state table representation of a sequential circuit, the columns of the table correspond to the primary inputs and the rows correspond to the present state of the circuit. The entries in the table are the next state and the output associated with each combination of inputs and present states. As an example, consider the sequential circuit of Figure 4.21. It has one input x, one output z, and two state variables

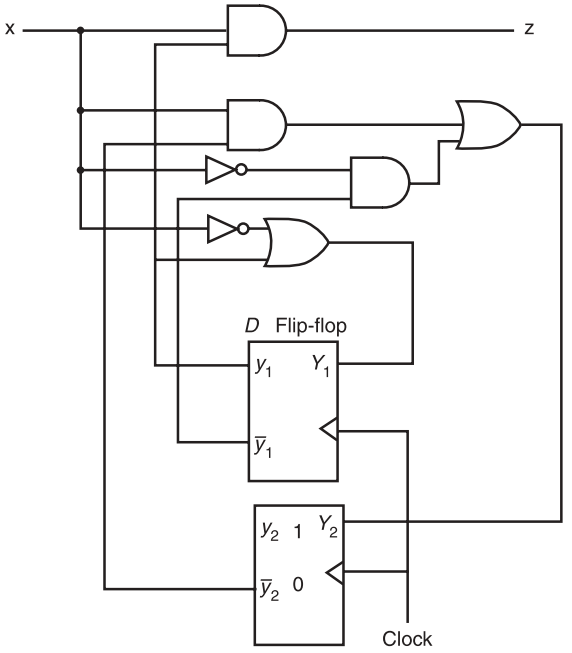


FIGURE 4.21 A state table.

y_1y_2 (thus having four possible present states 00, 01, 10, 11). The behavior of the circuit is determined by the following equations:

$$Z = xy_1$$
$$Y_1 = \bar{x} + y_1$$
$$Y_2 = x\bar{y}_2 + \bar{x}\bar{y}_1$$

These equations can be used to form the state table.

Suppose the present state (i.e., y_1y_2) = 00 and input $x = 0$. Under these conditions, we get $Z = 0$, $Y_1 = 1$, and $Y_2 = 1$. Thus the next state of the circuit $Y_1Y_2 = 11$, and this will be the present state after the clock pulse has been applied. The output of the circuit corresponding to the present state $y_1y_2 = 00$ and $x = 1$ is $Z = 0$. This data is entered into the state table as shown in Figure 4.22*a*. Normally each combination of present state variables is replaced by a letter; Figure 4.22*b* is derived from Figure 4.22*a* by replacing states 00, 01, 11, and 10 by *A*, *B*, *C*, and *D*, respectively. The output and the next state entries corresponding to other present states and the input are derived in a similar manner. In general, an *m*-input, *n*-state machine will have *n* rows and one column for each of the 2^m combinations of inputs. The next state and output corresponding to a present state and an input combination are entered at their intersection in the table.

A sequential circuit can also be represented by a *state diagram* (also known as a *state transition graph*). A state diagram is a directed graph with each node corresponding to a state in the circuit and each edge representing a state transition. The input that causes a state transition and the output that corresponds to the present state/input combination are given beside each edge. A slash separates the input from the output. The state diagram for the sequential circuit of Figure 4.22 is shown in Figure 4.23; states 00, 01,

Present State	Input	
y_1y_2	$x = 0$	$x = 1$
0 0	11, 0	01, 0
0 1	11, 0	00, 0
1 1	10, 0	10, 0
1 0	10, 0	11, 1
	Next state, Output	

(*a*)

	Input	
Present State	$x = 0$	x = 1
A	*C*,0	*B*,0
B	*C*,0	*A*,0
C	*D*,0	*D*,1
D	*D*,0	*C*,1

(*b*)

FIGURE 4.22 (*a*) Binary representation of states and (*b*) state table.

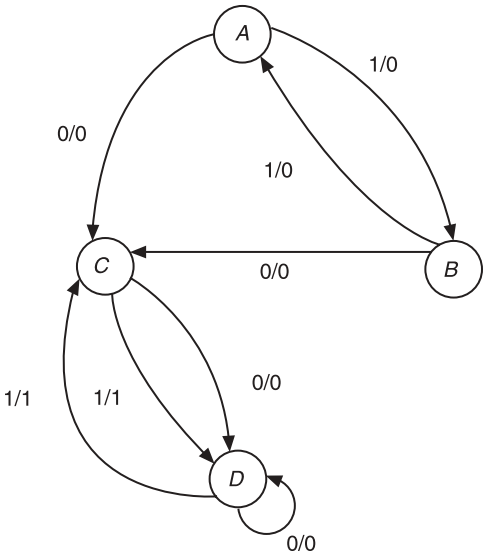


FIGURE 4.23 State diagram.

11, and 10 are denoted by the letters *A*, *B*, *C*, and *D*, respectively. For example, the edge from *B* to *A* and the associated label $1/0$ indicate that if *B* is the present state and the input is 1, then the next state is *A* and the output is 0.

Both state tables and state diagrams can be used to define the operation of sequential circuits; they provide exactly the same information. However, in general, state diagrams are used to represent the overall circuit operation, whereas the state table is employed for the actual circuit design.

4.7 MEALY AND MOORE MODELS

So far, we have considered sequential circuits in which the output at any time depends on the present state and the input, and these also determine the next state. This particular

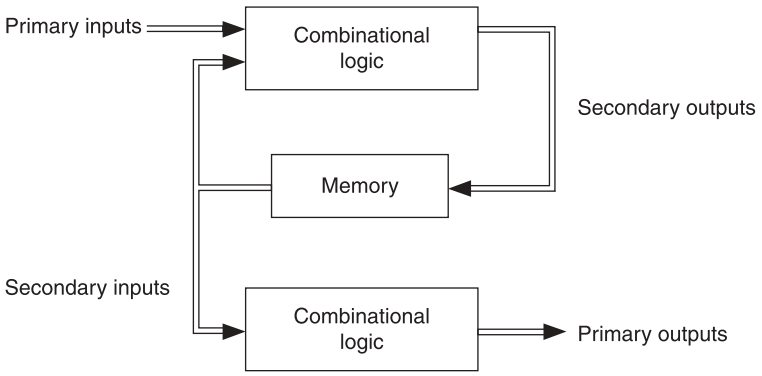


FIGURE 4.24 Moore model.

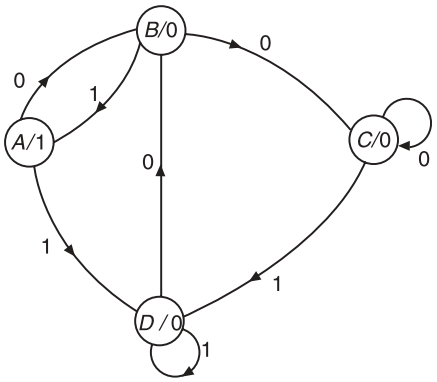
model of sequential circuits is known as the *Mealy model* (Fig. 4.1). In an alternative model, called the *Moore model*, the next state depends on the present state and the input, but the output depends only on the present state. Figure 4.24 illustrates the Moore model of the sequential circuit.

In the state table representation of Moore-type sequential circuits, the rows of the table correspond to present states and the columns to input combinations as in Mealy-type circuits. The entries in the table for the input/present state combinations are the next states associated with each combination of inputs and present states; there is a separate output column with the entry corresponding to each row (i.e., present state) in the table. An example of such a table is shown in Figure 4.25*a*. The state diagram for the circuit is shown in Figure 4.25*b*. Since each state has a unique output, the output can be associated with the state. Thus the state and the output are labeled within the node, separated by a slash. An edge corresponds to a state transition, and the input causing a transition is given beside the edge.

A sequential circuit can be represented either by a Moore model or by a Mealy model, and conversion from one model to the other is always possible. Let us first consider the conversion of the Mealy model of a sequential circuit to an equivalent Moore model [1]. If in the Mealy-type circuit, a next state entry S is always associated with the same

Present State	Input		Output
	$x = 0$	$x = 1$	
A	*B*	*D*	1
B	*C*	*A*	0
C	*C*	*D*	0
D	*B*	*D*	0
	Next state		

(*a*)



(*b*)

FIGURE 4.25 (*a*) State table and (*b*) state diagram.

	Input	
Present State	$x = 0$	$x = 1$
A	B,0	A,1
B	D,0	C,1
C	B,0	C,0
D	E,1	E,1
E	A,0	B,0

FIGURE 4.26 A Mealy-type sequential circuit.

output Z, then S will be associated with the output Z in the state table of the equivalent Moore-type circuit.

As an example, let us consider the state table of a Mealy-type sequential circuit shown in Figure 4.26. The next state entries B, D, and E are always associated with outputs 0, 0, and 1, respectively. Hence in the equivalent Moore-type circuit shown in Figure 4.27, B is associated with an output of 0, D with 0, and E with 1.

If a state S is the next state entry for several states in the Mealy-type circuit and is associated with n different outputs, then in the state table of the equivalent Moore-type circuit, state S is replaced by n different states. The next state entries corresponding to each of these new states are identical to the next state entries for S in the Mealy-type circuit.

For instance, A as a next state entry is associated with both outputs 0 and 1 in Figure 4.26. Hence A is replaced by the two states A,0 and A,1 in Figure 4.27—A,0 being associated with output 0 and A,1 being associated with output 1. The next state entries for both A,0 and A,1 are B when $x = 0$, and A,1 when $x = 1$. The reason for A,1 being the next state entry when $x = 1$ is because in Figure 4.26, the next state for A when $x = 1$ is A and the associated output entry is 1. Similarly, state C in Figure 4.26 is replaced by two states C,0 and C,1 in Figure 4.27.

Let us now illustrate the conversion of a Moore-type circuit to an equivalent Mealy-type circuit. The state table of a Moore-type circuit is shown in Figure 4.28. In converting a Moore-type circuit to a Mealy-type circuit, if a state S is associated with an output Z, then the output associated with the next state S in the state table of the Mealy-type circuit will be Z. The state table of a Mealy-type circuit is derived from Figure 4.28 as shown in Figure 4.29.

	Input		
Present State	$x = 0$	$x = 1$	Output
A,0	B	A,1	0
A,1	B	A,1	1
B	D	C,1	0
C,0	B	C,0	0
C,1	B	C,0	1
D	E	E	0
E	A,0	B	1

FIGURE 4.27 State table of equivalent Moore-type circuit.

	Input		
Present State	$x = 0$	$x = 1$	Output
A	*B*	*C*	0
B	*A*	*C*	0
C	*D*	*E*	1
D	*C*	*B*	0
E	*A*	*F*	1
F	*A*	*E*	1
	Next state		

FIGURE 4.28 A Moore-type circuit.

	Input	
Present State	$x = 0$	$x = 1$
A	*B*,0	*C*,1
B	*A*,0	*C*,1
C	*D*,0	*E*,1
D	*C*,1	*B*,0
E	*A*,0	*F*,1
F	*A*,0	*E*,1
	Next state, Output	

FIGURE 4.29 State table of the equivalent Mealy-type circuit.

4.8 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUITS

In the previous section, we discussed several models for sequential circuits. Such circuits are used to perform many different functions. This section covers the analysis of these circuits. As will be seen, the analysis of the behavior of such circuits identifies the processes that are required to synthesize them. We shall consider the sequential circuit shown in Figure 4.30. The circuit has one input x and one output z. Two *JK* flip-flops are used as memory elements that define the four possible states of the circuit, $y_1y_2 = 00$, 01, 10, or 11. The equations that describe the circuit's operation, known as the *control equations*, can be derived directly from Figure 4.30.

$$J_1 = x + y_2 \qquad J_2 = xy_1$$
$$K_1 = x + \bar{y}_2 \qquad K_2 = \bar{x}$$
$$z = y_1 \cdot y_2$$

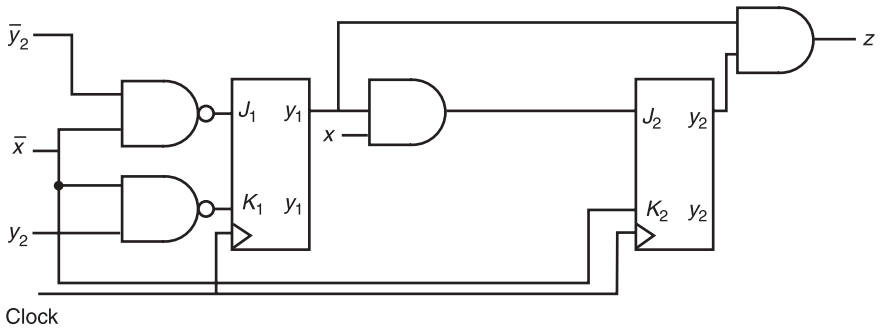


FIGURE 4.30 A sequential circuit.

The characteristic equation for *JK* flip-flops was derived in Section 4.4. It is repeated here:

$$Q_{t+1} = J\bar{Q}_t + \bar{K}Q_t$$

where Q_t and Q_{t+1} are, respectively, the present state and the next state of a flip-flop. By substituting J_1, K_1 and J_2, K_2 in this equation, the next state functions of the two flip-flops are obtained:

$$\begin{aligned}(y_1)_{t+1} &= (x + y_2)\bar{y}_1 + (\overline{x + \bar{y}_2})y_1 \\ &= x\bar{y}_1 + \bar{y}_1 y_2 + \bar{x}y_1 y_2 \\ &= x\bar{y}_1 + \bar{x}y_2\end{aligned}$$

$$\begin{aligned}(y_2)_{t+1} &= xy_1\bar{y}_2 + \bar{\bar{x}}y_2 \\ &= xy_1\bar{y}_2 + xy_2 \\ &= xy_1 + xy_2\end{aligned}$$

It is now possible to construct a table (Table 4.2) that gives the next state values of the flip-flops for given present state values and for a given input. This form of the state table is known as a *transition table*. The two output columns of the table result from the interpretation of the output equation $z = y_1 \cdot y_2$. Thus the output of the circuit is 1 when the present state of the circuit is $y_1y_2 = 11$, irrespective of the input value. Replacing $y_1y_2 = 00$, 01, 10, 11 by *A*, *B*, *C*, and *D*, respectively, we can derive the state table (Table 4.3) of the circuit from its transition table.

The state diagram of the circuit can be derived from its state table and is shown in Figure 4.31.

TABLE 4.2 Transition Table for the Circuit of Figure 4.30

Present State		Input		Output	
y_1	y_2	$x = 0$	$x = 1$	z (when $x = 0$)	z (when $x = 1$)
0	0	00	10	0	0
0	1	10	11	0	0
1	0	00	01	0	0
1	1	10	01	1	1
		Next state			

TABLE 4.3 State Table of the Circuit of Figure 4.30

	Input	
Present State	$x = 0$	$x = 1$
A	*A*,0	*C*,0
B	*C*,0	*D*,0
C	*A*,0	*B*,0
D	*C*,1	*B*,1

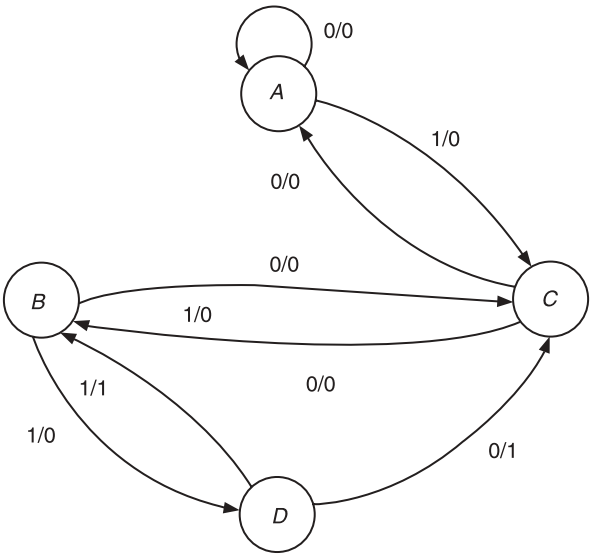
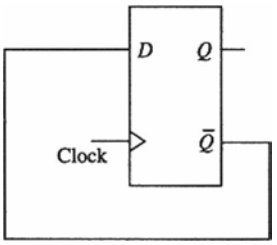


FIGURE 4.31 State diagram of the circuit of Figure 4.30.

EXERCISES

1. A *D* flip-flop is connected as shown. Determine the output of the flip-flop for ten clock pulses assuming it has initially been reset. What function does this configuration perform?



2. Modify a *JK* flip-flop such that when both *J* and *K* inputs are at logic 0, the flip-flop is reset.

3. Modify a *T* flip-flop such that it functions as a *JK* flip-flop.

4. Assume a *D* flip-flop with separate set (*S*) and reset (*R*) inputs. How can this flip-flop be configured such that its output will be set to logic 1, when both *S* and *R* inputs are high simultaneously?

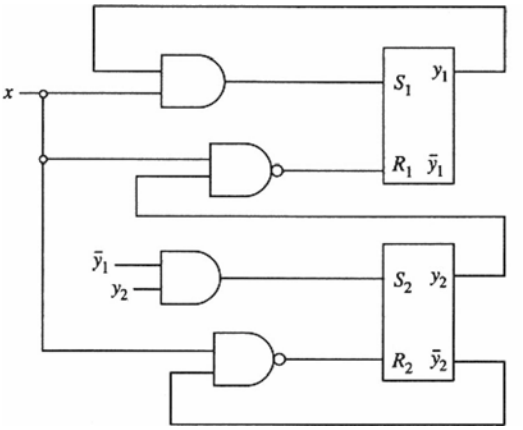
5. Modify a *T* flip-flop such that it functions as a *D* flip-flop.

6. A sequential circuit uses two *D* flip-flops as memory elements. The behavior of the circuit is described by the following equations:

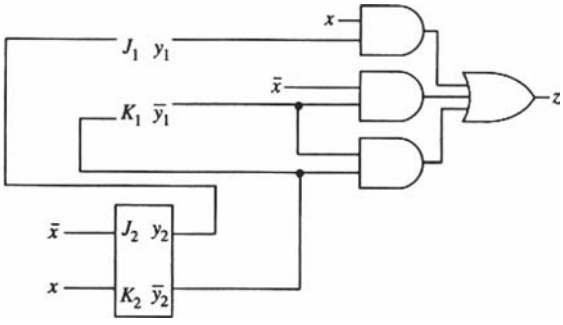
$$Y_1 = y_1 + \bar{x}y_2$$
$$Y_2 = x\bar{y}_1 + \bar{x}y_2$$
$$z = \bar{x}y_1y_2 + x\bar{y}_1\bar{y}_2$$

Draw the state diagram of the circuit.

7. Derive the transition table for the following circuit:

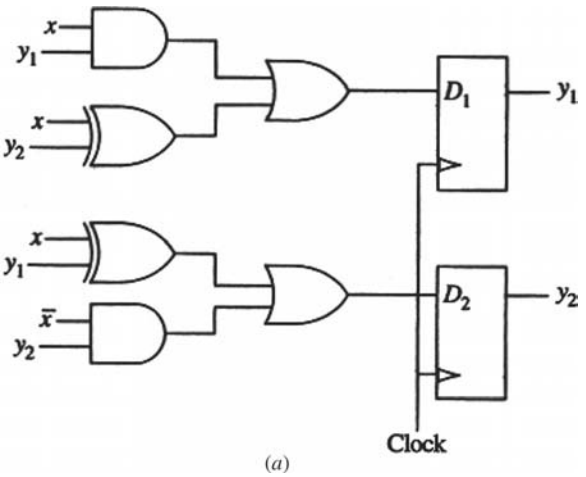


8. For the circuit shown, fill in the values for the *J* and *K* inputs, and the output values in the table.

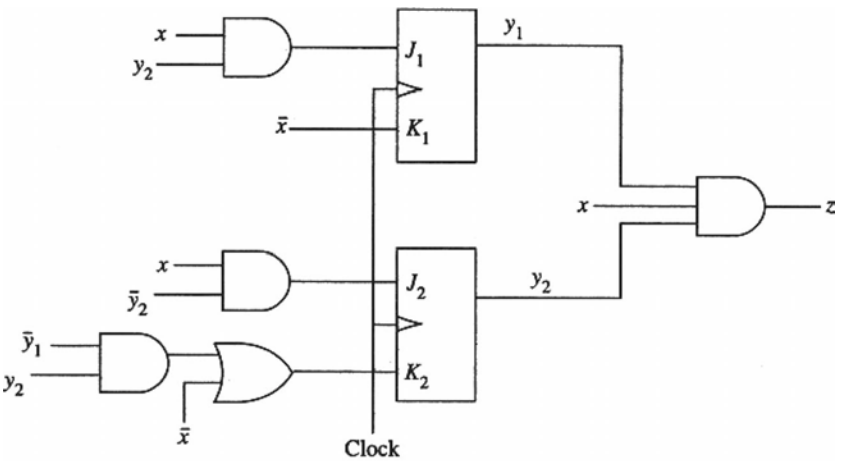


	$x = 0$		$x = 1$		z	
y_1, y_2	J_1, K_1	J_2, K_2	J_1, K_1	J_2, K_2	$x = 0,$	$y = 1$
00						
01						
11						
10						

9. Derive the state tables for the following circuits:



(*a*)



(*b*)

10. Convert the following state tables for Mealy-type sequential circuits into those of Moore-type circuits.

	$x = 0$	$x = 1$
A	B,0	A,0
B	B,0	C,0
C	D,0	A,0
D	B,0	C,1

	$x = 0$	$x = 1$
A	B,0	C,1
B	A,0	E,0
C	E,1	D,0
D	B,1	A,1
E	C,0	D,1

11. Convert the following state tables for Moore-type sequential circuits into those of Mealy-type circuits.

	$x = 0$	$x = 1$	z
A	B	A	1
B	B	C	0
C	D	A	1
D	B	C	0

	$x = 0$	$x = 1$	z
A	B	A	0
B	D	C	1
C	B	C	1
D	A	E	0
E	E	D	0

REFERENCE

1. C. L. Sheng, *Introduction to Switching Logic*, International Text Book Co., Canada, 1972.

5 VHDL in Digital Design

5.1 INTRODUCTION

The dramatic increase in the logic density of silicon chips has made it possible to implement digital systems with multimillion gates on a single chip. The complexity of such systems makes it impractical to use traditional design descriptions (e.g., logic schematics) to provide a complete and accurate description of a design. Currently, all complex digital designs are expressed using a *hardware description language* (HDL). An HDL, unlike traditional programming languages such as C or C++, can describe functions that are inherently parallel. A major advantage of an HDL is that it provides a better and more concise documentation of a design than gate-level schematics. Two very popular HDLs are VHDL and VERILOG.

In this text we use VHDL. VHDL is an acronym for VHSIC hardware description language; VHSIC in turn is an acronym for very high speed integrated circuit. The development of VHDL was funded by the U.S. Department of Defense (DoD) in the early 1980s. The syntax of VHDL is similar to that of programming language ADA; however, it has some significant differences from ADA. We present the important concepts of VHDL, especially the ones that are used in digital circuit design.

VHDL can provide unambiguous representation of a design at different levels of abstraction as shown in Figure 5.1. Modern CAD (computer-aided design) tools can generate gate-level implementation of a design from its VHDL description.

A behavioral VHDL description of a circuit describes the function of the circuit in terms of its inputs using the types of statements used in a high-level programming language. The objective is to describe the correct operation of a circuit to be designed without being concerned with redundant details. This description does not specify how the function is actually implemented; thus the same description may result in several implementations of a circuit.

The register transfer level (RTL) description of a circuit specifies the flow of data from an input or a register to another register or the output of the circuit through combinational logic blocks. The RTL description is also known as *data flow description*.

The structural level description specifies what components a circuit is composed of and how these components are interconnected. This is similar to the logic schematic diagram of a circuit.

The architectural descriptions are discussed in more detail later in the chapter.

Principles of Modern Digital Design, by Parag K. Lala

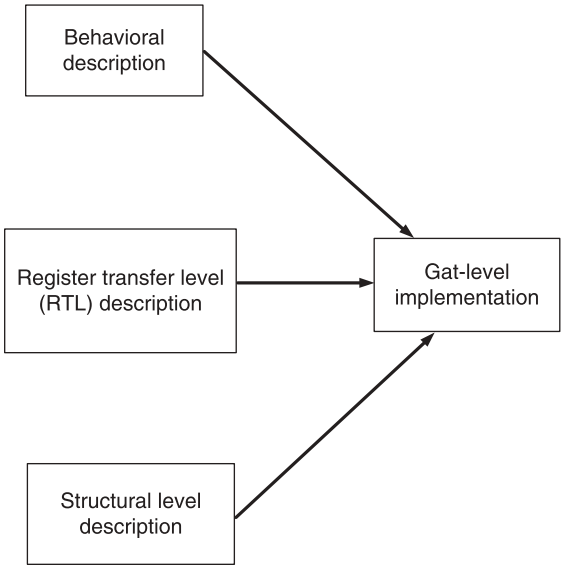


FIGURE 5.1 Different VHDL abstractions.

5.2 ENTITY AND ARCHITECTURE

A circuit block in VHDL is considered a "black box." An entity declaration identifies the external view of the black box, while its inside is unknown. A description of the implementation of the content of the black box is called its architecture. Thus in VHDL an architecture is always associated with an entity, as shown in Figure 5.2.

5.2.1 Entity

As indicated earlier, an entity defines the inputs and outputs of a VHDL module. For example, if the module is a 2-input NAND gate, the entity declaration identifies the inputs and the output of the module. The architecture describes the internal operation of the module.

The relation between the NAND gate and its entity is shown in Figure 5.3.

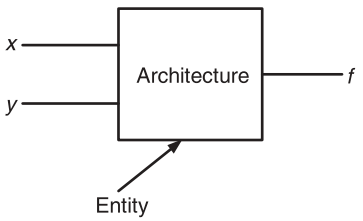
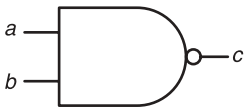


FIGURE 5.2 Entity–architecture pair.




```
entity  my_nand_gate  is

   port ( a:  in std_logic;

          b: in std_logic;

          c: out std_logic);

end my_nand gate;
```

FIGURE 5.3 Two-input NAND gate and its entity.

The name of an **entity** is selected by the user. An input or an output signal in an entity declaration is known as a **port**. Each port must have a mode that can be one of the following:

In	An input to the entity
Out	An output of the entity
Buffer	An output of the entity that can also used for internal feedback and has limited fanouts
Inout	Can be used as both an input and an output (i.e., can operate in tristate mode)

Figure 5.4 illustrates the operation of the buffer and the inout ports. An inout signal can be used internally by the entity, whereas the signal on the out port cannot be.

All signals in VHDL must have a *type*. The most commonly used types are `std_logic` for single-bit signals, and `std_logic_vector` for multibit signals. The term `std_logic` indicates that an input or output signal can take several different values in addition to 0 or 1. We discuss `std_logic` later in this chapter.

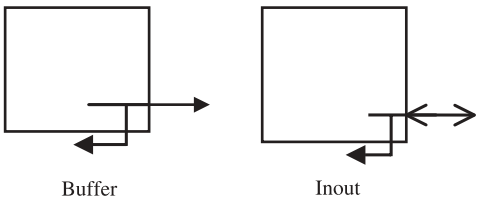


FIGURE 5.4 Inout and buffer ports.

```
architecture example of my_nand_gate  is

 begin

    if( a = '1' and b ='1' ) then

    c = '0';

    else c = '1';

 end example;
```

FIGURE 5.5 Architecture of the entity `my_nand_gate`.

5.2.2 Architecture

An **architecture** describes how the design entity outputs are related to the design entity inputs. An *architecture* can be given any name (without violating VHDL syntax) but the name after *of* must match the entity name. An entity can have several optional architectures. Figure 5.5 shows an architecture for the entity of Figure 5.3. The primitive logic operators such as **and**, **or**, and so on are part of a library (*library IEEE*), which is declared before the entity in all VHDL code descriptions of circuits.

It should be obvious that describing the operation of a complex circuit is significantly more difficult than describing its entity. Let us illustrate different ways of describing the architecture of a circuit by using a 2-to-1 multiplexer shown in Figure 5.6 as an entity.

The entity for the multiplexer is

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer is
port (a,b: in std_logic;
s: in std_logic;
f: out std_logic);
end multiplexer;
```

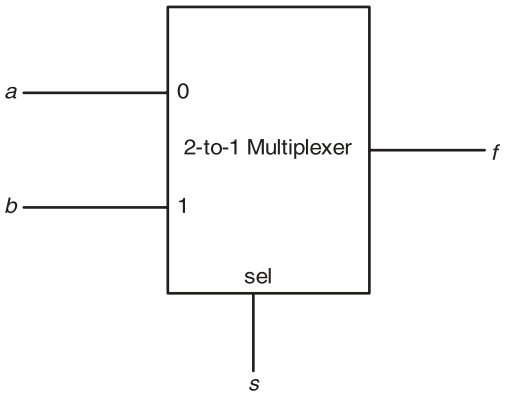


FIGURE 5.6 A 2-to-1 multiplexer.

One way of describing the architecture is based on the *behavioral* model of the multiplexer:

```
architecture behavior of multiplexer is
begin
process (a,b,s)
begin
if s='0' then
f<=a;
else
f<=b;
end if;
end process;
end behavior;
```

The **process** block is used only in the behavioral description and is discussed in Section 5.6. The second approach is to use the RTL (register transfer language) description as given below. The RTL description defines the Boolean expression corresponding to an entity function using only the input signals at the input ports of the entity.

```
architecture rtl of multiplexer is
begin
f<=(not s and a) or (s and b);
end rtl;
```

Finally, the architectural description based on the structural model of the multiplexer is

```
architecture structural of multiplexer is
signal t1,t2: std_logic;
begin
t1<=not s and a;
t2<=s and b;
f<=t1 or t2;
end structural;
```

It should be mentioned here that the structural description specifies the function of each gate in the multiplexer separately. The outputs of the AND gates are defined as internal signals `t1` and `t2`, which are then used as inputs by the OR gate to produce circuit output `f`.

Note that the structural description is based on an implementation of the Boolean expression for the multiplexer whereas in the RTL description the Boolean expression itself was specified without consideration of its implementation.

5.3 LEXICAL ELEMENTS IN VHDL

VHDL like most programming languages has several types of lexical elements:

Numbers
Characters and strings

Identifiers
Comments
Operators
Reserved words

Numbers A wide range of number representation can be used in VHDL; the decimal number system is the default. If a base different from 10 is used to represent a number, then the base number is written first followed by the hash (#) sign, the actual number, and another # sign. For example, 2#0111# indicates 0111(base 2), which is equal to 7 in base 10. The hexadecimal representation of 7 is 16#7#. The readability of a large number can be improved by inserting underscores in the number provided there is no underscore at the beginning or at the end of the number. For example, the binary representation of decimal number 863 can be written 2#1101_0111_11#.

Characters and Strings A character is enclosed in single quotation marks, for example, '0', '1', and 'a'. A string of characters is enclosed in double quotation marks, for example, "VLSI," "10010110" (indicates a binary string), and X "96" (X indicates a hexadecimal string).

Identifiers Identifiers are user-defined words that are used as *variables, constants*, and *signal* names and also as the names of ports, design entities, architectures, or similar objects. Basic identifiers are chosen according to the following rules:

- An identifier is case-insensitive. For example, the exclusive-OR operation can be identified as EX-OR or ex-or.
- The first character can only be a letter not a number or an underscore (_). For example, Clk1 is a valid identifier but 1_clk and _clk1 are not.
- The last letter cannot be an underscore. Thus clk_ is not a valid identifier.
- Two consecutive underscores cannot be included in an identifier. For example clk_ _1 is an invalid identifier.

Comments All statements in VHDL code are terminated with a semicolon (;). Comments are indicated with a "– –" (two consecutive dashes); the carriage return terminates a comment

```
c <= a and b; - - this is an example of an AND operation
```

Note that VHDL is indifferent to spaces and tabs in the source code.

Data Objects Any item that stores data is known as an *object* in VHDL. The VHDL objects include signals, variables, and constants. A signal like a wire in a circuit is used to connect different design units. It cannot be assigned any initial value. The signal assignment operator is "<=".

```
signal a: std_logic bit;
```

A constant, on the other hand, is assigned a value that cannot be changed. The constant assignment operator is ":=". For example,

```
constant k: integer:=12;
constant bitcount: std_logic: ='0';
```

The first line indicates *k* is a constant and is assigned a value of 12. In the second line `bitcount` is a constant and is set to logic 0. Note that a single bit must be within single quotes; multiple bits, on the other hand, are considered as a string and must be within double quotes as shown in the following declaration:

```
constant bytecount: std_logic:="0101";
```

This indicates `bytecount` is a constant and is set to 0101.

A variable in VHDL is similar to variables used in conventional programming languages. It is assigned a temporary value when it is declared and is replaced with other values during the execution of the code. Variables are assigned values using the := operator as in constants. For example,

```
variable x: integer:=0;
```

indicates that `x` is a variable and is initialized to `0`. Variables must always be declared within a *process* block; we discuss it in more detail in Chapter 9.

This assignment does not clarify the number of bits to be used to represent variable *x*. As a result, the VHDL compiler will produce an inefficient realization of the circuit from the code containing the assignment statement. However, if the statement is changed to

```
variable x: integer range 0 to 15;
```

the VHDL compiler will know that `x` can have any integer value in the range of 0 to 15 and thus is to be represented by 4 bits.

5.4 DATA TYPES

Each VHDL object must be assigned a specific data type; it indicates the common characteristic of the set of values the object holds. A variety of predefined types are available in VHDL, for example, *bit, boolean, and std_logic*. The bit type takes values '0' and '1' whereas the boolean type can have values true and false. They are defined as follows:

```
signal a: bit:='1';
variable a: boolean:= true;
```

Note that type bit is not the same as type boolean. The following example illustrates the difference:

```
if (a) then ----- implies a is boolean type
if (a='1') then ----- indicates a is bit type
```

A separate notation, `bit_vector`, is used to indicate whether a signal has multiple component lines (e.g., a bus in digital circuits). A `bit_vector` is an array with each element being of type bit. For example,

```
signal inbus: bit_vector (0 to 7);
```

The `std_logic` (standard logic) type was introduced in IEEE Standard 1164. It allows a signal to have other values in addition to 0 and 1 as shown below:

'U' Uninitialized
'X' Unknown
'Z' High impedance
'W' Weak unknown
'L' Weak zero
'H' Weak one
'–' Don't care

This simplifies evaluation of the VHDL model of a circuit for unintended behavior when it receives inputs other than 0 or 1.

The vector for `std_logic` type is `std_logic_vector`. The component elements in a `std_logic_vector` are specified by the keywords to and downto as illustrated below:

```
w: in std_logic_vector (0 to 15);
w: in std_logic_vector (15 downto 0);
y: out std_logic_vector (0 to 15);
y: out std_logic_vector (15 downto 0);
```

The first line indicates that `w` is an input bus with 16 signals and the signals are identified in an ascending order: `w(0), w(1), ..., w(15)`. In the second line the bus signals are identified in a descending order: `(w15), w(14), ..., w(0)`.

In order to use `std_logic` type signals the following two statements must be inserted before the entity declaration of a VHDL model:

```
library ieee;
use ieee_std_logic_1164. all;
```

`The ieee_std_logic_1164` is a package that is compiled into a library called ieee. The integer type is used to represent a signed integer using 32 bits; the most significant bit is used for indicating the sign of the integer. Thus the integers in VHDL have a default range of $(-2^{31}+1)$ to $(+2^{31}-1)$. However, it is possible to constrain the range of an integer type object. For example,

```
variable x: integer range -63 to 63;
```

VHDL also allows generation of a unique *enumerated* type by explicitly listing all its possible values. These values are listed in an ascending order and each value in the list has

associated with it a positive position number starting with 0 for the leftmost value. For example,

```
type color is (red, green, yellow, blue);
```

In this example, the position numbers for values red, green, yellow, and blue are 0, 1, 2, and 3, respectively, and an expression like yellow $<$ blue is valid. An enumerated type is usually used to represent the various states of a state machine while describing it in VHDL. For example, a 5-state machine can use an enumerated type as shown below:

```
type state is (s1, s2, s3, s4, s5);
```

5.5 OPERATORS

The VHDL language supports several classes of operators: logical, relational, shift, unary, multiplying, and miscellaneous. Operators are chained to define complex expressions.

Logic Operators The logical operators are used to define logical operations on bit-type signals and variables. Parentheses are normally required to correctly represent Boolean expressions when logic operators NAND and NOR are used because they do not obey the associative law of Boolean algebra. For example, the Boolean expression $f = ab + cd$ if written

```
f<=a nand b nand c nand d
```

will not be correct although it is a valid expression; it will result in the Boolean expression

$$f = \bar{a}c + \bar{b}c + \bar{d}$$

The correct representation is

```
f<=(a nand b) nand (c nand d)
```

Thus it is desirable that parentheses be used when in doubt to avoid changing the meaning of the code.

Relational Operator The relational operators compare values of similar type objects and produce a true (logic 1) or a false (logic 0) output. They are in general used with if–then–else statements. As an example, let us assume `X`, `Y`, `Z` are register type and `W` is a counter-type object. Suppose `X`, `Y`, and `W` have been assigned values 31, 63, and 31, respectively; then

```
If X = Y then Z = 1 else Z = 0; results in Z = FALSE
If (X /= Y) then Z =1 else Z = 0; results in Z = TRUE
```

TABLE 5.1 Shift and Rotate Operators

Operator	Function
ssl	Shift left logical
ssr	Shift right logical
sla	Shift left arithmetic
sra	Shift right arithmetic
rol	Rotate left
ror	Rotate right

Similarly for

```
X<Y  Z = TRUE
X<=Y  Z = TRUE
X>Y  Z = FALSE
X>=Y  Z = FALSE
W = X ILLEGAL (because W and X are not the same base type)
```

Shift Operators Shifting is performed on bit vectors and can be one of the following types: logical shift, arithmetic shift, and rotate as shown in Table 5.1. In logical left (right) shift, zeros are fed into the right (left) end of the operand and the shifted operand bits on the left (right) are lost. For example, the shift operation

```
variable test: standard logic bit_vector:="10011001";
test ssl 2;
```

will result in the number "01100100". The shift operation

```
variable test: standard logic bit_vector:="10011001";
test ssr 2;
```

will result in the number "00100110".

In an arithmetic left shift, zeros are fed in from the right and all shifted operand bits on the left are lost. As an example, the arithmetic left shift operation

```
variable test: standard logic bit_vector:="10011001";
test sla 2;
```

will result in the number "01100100".

In an arithmetic right shift the empty bit positions are filled with copies of the most significant (sign) bit and the shifted bits on the right are lost. This is needed for sign extension when working with signed numbers. Figure 5.7 illustrates arithmetic right shift. As an example, let us consider the arithmetic right shift by 3 bits on the data 10011100:

```
variable test: standard logic bit_vector:="10011100"
test sra 3;
```

This shift results in the number "11110011".

In the rotate left (right) operation, the left most (right most) bit of the operand is fed to the right (left) end of the operand. For example, the operation

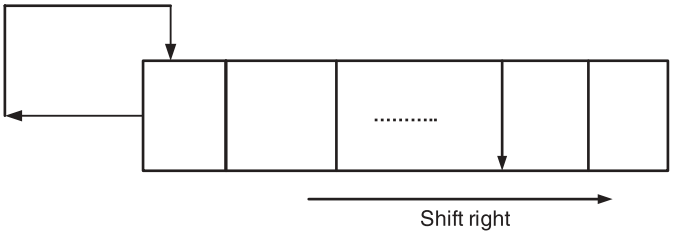


FIGURE 5.7 Arithmetic right shift.

```
variable test: standard logic bit_vector:="10011001";
test ror 2;
```

will result in the number "01100110" and

```
variable test: standard logic bit_vector:="10011001";
test rol 2;
```

will result in the number "01100110".

Addition Operators The arithmetic and concatenation operators are collectively known as addition operators. The arithmetic operators are defined for integer type operands:

+ Addition of two integers
− Subtraction of an integer from another

The following example shows the addition and subtraction operation on two integers:

```
signal w, x, y, z: integer range 0 to 2;
y<=w+x;
z<=w-x;
```

The concatenation operator (&) produces one-dimensional arrays by combining other operands of types `bit`, `bit_vector`, or `string` as well as constant bit(s). The length of the new array is equal to the sum of the lengths of component operands. The following example illustrates the concatenation operation:

```
Signal w: standard_logic vector (3 down to 0);
Signal x: standard_logic vector (6 down to 0);
Signal y: standard_logic vector (2 down to 0);
Signal a, b, c, d: standard_logic bit;
x<='1' & y & w;
y<=a & b & c;
w<="00" & y(2);
```

Multiplying Operators The * (multiplication) and / (division) operators are valid for all numeric type operands. The operators *mod* (modulus) and *rem* (remainder) are valid only for integers. Both give the remainder on division. The only difference is X *rem* Y has the

sign of X, whereas X *mod* Y has the sign of Y as given below:

$$\begin{aligned} X\,rem\,Y &= X - int\,(X/Y)^{*}Y \quad (int \text{ is the integer corresponding to the quotient}) \\ X\,mod\,Y &= X - int\,(X/Y)^{*}Y \quad \text{if X and Y have the same sign} \\ &= X - \lceil(|X/Y|)\rceil^{*}Y \quad \text{otherwise} \end{aligned}$$

For example, if $X = -11$ and $Y = 3$ then (-11) rem $3 = -2$ and (-11) mod$3 = 1$. *Division, mod* and *rem* operators are usually not supported by current logic synthesis tools. The following example shows the multiplication of a vector by another vector and the multiplication of a vector by an integer.

```
signal x, y: unsigned (0 to 2);
signal w: unsigned (0 to 5);
w<=x * y;
w<=y * 2;
```

Miscellaneous Operators These include operators *abs*, *not*, and exponentiation (**). The abs and exponentiation operators are applicable to any integer operands . The *not* operator can be applied only to a *bit* or a *boolean* type operand. The following examples illustrate the application of the operators:

```
variable x,y,z: integer:=4;
signal m, n, p: standard_logic bit;
y:=2**x; --y gets the value 16.
z:=abs ((y)*(-3)); --z gets the value 12.
m<=n or not p;
```

Reserved Words Certain words in VHDL have special meaning and cannot be used as identifiers in the VHDL code. These words are known as keywords or reserved words and are listed below:

abs access after alias all and architecture array assert attribute
begin block body buffer bus
case component configuration constant
disconnect downto
else elsif end entity exit
file for function
generate generic guarded
if in inout is
label library linkage loop

5.6 CONCURRENT AND SEQUENTIAL STATEMENTS

As in a conventional programming language, a VHDL statement within an architecture body represents a certain action. However, unlike in a conventional programming

language where all statements in a source code are executed sequentially (i.e., one at a time in the order they appear to complete the specified task), signal assignment statements within an architectural body are *concurrent.* A concurrent statement is executed if there is a change in the value of any of signals on its right side; thus the order of appearance of these statements has no relevance to their execution. Concurrent statements are used mainly to describe combinational logic circuits.

The operation of the logic circuit of Figure 5.8 can be represented using the following VHDL code; the architecture uses three concurrent statements (9, 10, and 11) to define the outputs of the circuit:

```
1. library ieee;
2. use ieee.std_logic_1164.all;
3. entity combcircuit is
4. port (w, x, y, z : in std_logic;
5. f1, f2, f3 : out std_logic);
6. end combcircuit;
7. architecture logicfunc of combcircuit is
8. begin
9. f1<=(w and x and y) or (not w and not x and y);
10. f2<=not w or not x or y or z;
11. f3<=w or not x or (not y and z) or (w and z);
12. end logicfunc;
```

A statement with the symbol " <= " is used to specify a Boolean operation and is known as the *signal assignment* statement. The signal on the left of the < = symbol is determined by logical (or arithmetic) operation on the right side of the symbol. The three concurrent statements used in the VHDL code for Figure 5.8 perform Boolean operations on the input

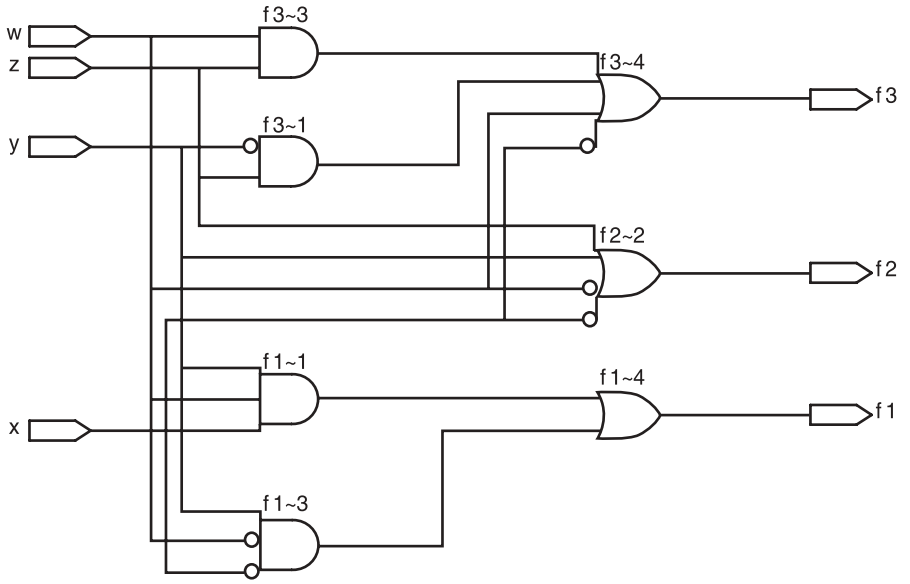


FIGURE 5.8 An example circuit.

signals at the input ports of the entity and transfer the result to the signals on the left. There is no significance to the order in which the three assignments have been made. This is a unique feature of VHDL and is not available in traditional programming languages. The concurrent statements imitate the simultaneous computation of outputs of the hardware components used to implement the VHDL entity.

All statements in a VHDL description are considered to be concurrent unless they are part of a process block. All statements inside a process block are sequential. Sequential statements are executed one after the other in the order they appear within a process; thus the order of statements is critical. Sequential statements can be used to describe both combinational and sequential logic circuits. A VHDL architecture can have multiple process blocks, each one of which is considered to be a single concurrent statement. The process block is discussed in detail in Chapter 6.

5.7 ARCHITECTURE DESCRIPTION

An architecture body as indicated previously describes the function of an entity. One may consider the entity part as a black box—the inputs and outputs of the box are specified but its internal details are unknown. The entity is split from the architecture in order to separate the input/output specification from the functional details of the architecture. So as a design becomes more detailed, an engineer can keep the same entities but upgrade and/or substitute the more detailed architecture version.

An entity may have more than one architecture with varied levels of detail in functional description as shown in Figure 5.9.

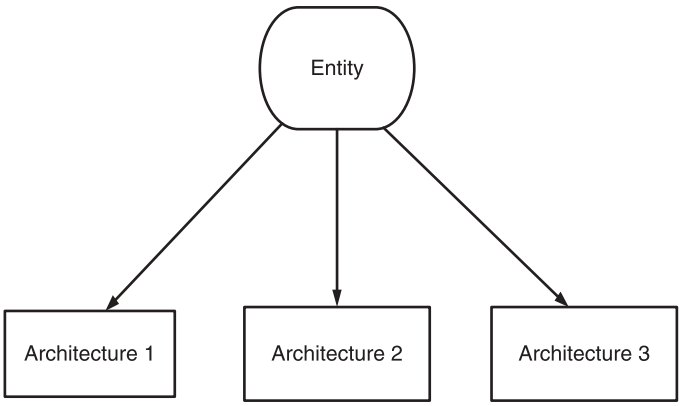


FIGURE 5.9 Entity with multiple architectures.

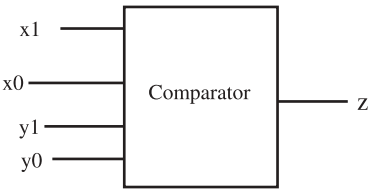


FIGURE 5.10 Block diagram of a 2-bit comparator.

To illustrate, let us consider the architectural description of a 2-bit comparator. The block diagram of the comparator is shown in Figure 5.10. The input pairs to the comparator block are (x1, x0) and (y1, y0); the output z is 1 when the input pairs match, otherwise it is 0. The VHDL code of the comparator circuit is given below. Since each input is 2 bits rather than a single bit, the input data is represented as vectors in the entity declaration:

```
library ieee;
use ieee.std_logic_1164. all;

entity comparator is
port(x: in std_logic_vector (1 downto 0);
y: in std_logic_vector (1 downto 0);
z: out std_logic);
end comparator;
architecture behavior of comparator is
begin
z<='1' when (x=y) else '0';
end behavior;
```

The architectural body is based on the behavioral model in this example.

An alternative architectural description of the entity 2-bit comparator is

```
library ieee
use ieee.std_logic_1164. all;

entity comparator is
port(x: in std_logic_vector (1 downto 0);
     y: in std_logic_vector (1 downto 0);
     z: out std_logic);
end comparator;

architecture structure of comparator is
signal t1, t0: std_logic;
begin
t1<=x(1) xnor y(1);
t0<=x(0) xnor y(0);
z<=t1 and t0;
end structure;
```

In this description two intermediate nodes t1 and t0 are created using a signal statement. The outputs of the nodes are connected to the inputs of an AND gate to generate output z. The signal statement is placed in an architecture body between the architecture and the begin statement. Each concurrent statement in an architecture body is mapped into a logic block during the synthesis process. The three concurrent statements used in the architectural description of the 2-bit comparator result in the schematic shown in Figure 5.11.

Note that the first architecture description was based on the functional behavior of the comparator. The second architecture description, on the other hand, used purely the circuit structure of the comparator. In the following sections, VHDL architecture descriptions for

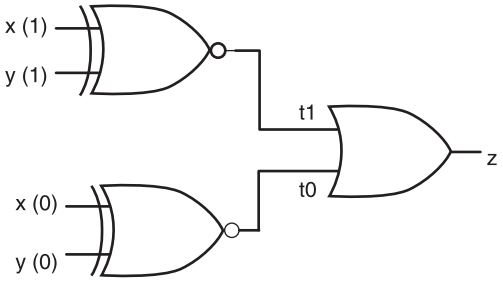


FIGURE 5.11 Two-bit comparator circuit.

circuits at the structural level, the behavioral level, and the register transfer level are discussed in more detail.

5.8 STRUCTURAL DESCRIPTION

The structural description of a system specifies how the building blocks of the system, referred to as *components*, are connected. The system behavior or functionality can only be determined from the behavior of the individual components of which the system is composed. Components are declared before the *begin* statement of the architecture. As an example, let us consider the majority voter circuit shown in Figure 5.12; in a majority voter circuit the output is 1 if at least two of the inputs are 1. Each component of the same type has been uniquely labeled in the circuit; the three 2-input AND gates in the circuit are labeled as AND_2 and the 3-input OR gate is labeled as OR_3. This allows one to uniquely identify multiple copies of the same component in a circuit. The internal signals (i.e., the outputs of the AND gates) are also uniquely labeled as t1, t2 and t3. As can be seen in Figure 5.12 the outputs of the AND gates are connected to signals t1, t2, and t3, and the inputs of the OR gate are also connected to t1, t2, and t3.

The VHDL structural description of the majority voter circuit is shown below. There is a component declaration statement identified by keyword *component*, corresponding to

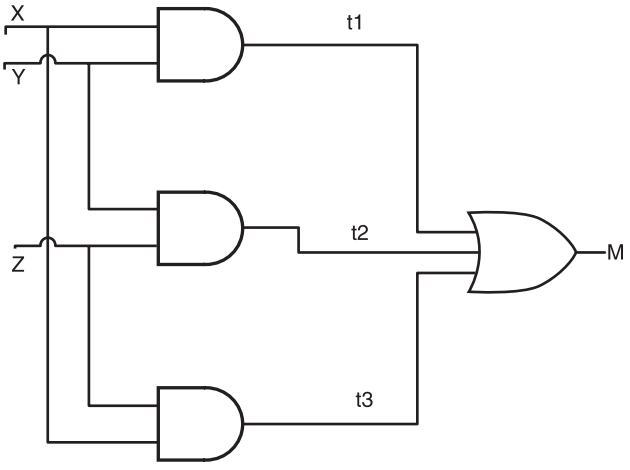


FIGURE 5.12 A majority voter circuit.

each component type used in the circuit. The component declarations are done before the *begin* keyword of the architecture body. Each component declaration statement contains a name of the component and its input and output ports; a component declaration statement is terminated by *end component*.

The components used in an architecture may be those that are part of a library or previously defined as part of a design and stored in the same directory as the VHDL compiler. Once a component has been declared, it can be used as many times as required. In the VHDL description of the majority voter there are two component statements corresponding to the two types of components used (e.g., AND_2 and OR_3). The VHDL codes for these components are stored in the same directory as Altera Quartus II software so that they can be used as components in other designs.

The component declarations are followed by a signal statement. The signal statement identifies all the signals that are used for interconnecting the declared components. The signal statement in the VHDL description declares three signals t1, t2, and t3. Note that the components and signals are declared before their interconnections.

A VHDL description of the majority voter circuit is

```
library ieee;
use ieee.std_logic_1164.all;

entity majority_voter is
port (X, Y, Z: in std_logic;
  M: out std_logic);
end majority_voter;

architecture structure of majority_voter is
component AND_2
port(x,y: in std_logic;
f:out std_logic);
end component;
 component OR_3
port(w,x,y:in std_logic;
f:out std_logic);
end component;

signal t1, t2, t3: std_logic;
begin
C1: AND_2 port map (X, Y, t1);
C2: AND_2 port map (X, Z, t2);
C3: AND_2 port map (Y, Z, t3);
C4: OR_3 port map (t1, t2, t3, M);
end structure;
```

Component declarations

Signal declaration

The VHDL codes for the AND_2 and OR_3 components are also stored in the same directory as the code for the majority voter. The codes are as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity AND_2 is
port (x, y: in std_logic;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity OR_3 is
port (w,x,y: in std_logic;
```

```
    f: out std_logic);
end AND_2;
architecture behavior of
AND_2 is
begin
f<=x and y;
end behavior;
```

```
    f: out std_logic);
end OR_3;
architecture behavior of
OR_3 is
begin
f<=(w or x or y);
end behavior;
```

The declared components in the architecture of the majority voter are then instantiated after the keyword *begin* in the architecture body. The VHDL compiler looks for the instantiated components by their names in the same directory where the source code of the file that used the components is stored . A *component instantiation* (placement) statement contains a unique label that identifies a particular component in the circuit followed by the actual component name and the keyword *port map*. The port map describes how input and output ports of a component instance are associated with input and output ports of the component. In the above VHDL code the component instances C1, C2, and C3 correspond directly with the AND_2 gates and the component instance C4 corresponds with the OR_3 gate. The first port in instance C1 is connected to X, the second port is connected to Y, and the third port is connected to signal t1. A comparison with the AND_2 component declaration shows that the first and second ports are input ports and are called x and y, respectively, whereas the third port is an output port called f. The component instantiation statement indicates the connection of input port x of the AND_2 gate to X, the connection of input port y to Y, and the connection of output port f to signal t1. Thus the ports of an instantiation statement are associated with the corresponding ports of the component; this method of association is known as *positional association*.

The drawback of the positional association in port mapping is that the ports in a component instantiation statement must be specified in the same order as in the component declaration. For example, if the order of two ports in instance C1 are inadvertently reversed as

```
C1: AND_2 port map (X,t1,Y);
```

then signal t1 will be connected to the input y of the AND_2 gate and Y will be the output of the gate!

An alternate method of port mapping called the *named association* allows explicit association of the port maps of a component instantiation with the port names of the component declaration statement, and also their orders are not important. For example, the component instantiation statement C1 can be written

```
C1: AND_2 port map (x =>X,f =>t1,y =>Y);
```

In port mappings based on the named association, the port names of a component are written on the left side of the => operator, and the port names of a component instance are written on the right side of the operator. Note that the mapping of the ports are not in any particular order.

5.9 BEHAVIORAL DESCRIPTION

As indicated earlier, a behavioral description specifies the function of a system in terms of operations on its inputs without any reference to the actual implementation of the function. It is usually used at the early stage of the design process when emphasis is more on understanding how a circuit functions rather than on its structural implementation. The advantage of the behavioral description over its structural counterpart is that in general it leads to several implementations of the behavior, thus providing designers with a number of options to choose from based on other design constraints.

The structural description of the majority voter shown in Figure 5.12 is composed of four primitive components and their interconnections. The behavioral module, on the other hand, is concerned only with describing the actual function of the circuit as shown below:

```
library ieee;
use ieee.std_logic_1164.all;

entity majority_voter is
port (X, Y, Z: in std_logic;
  M: out std_logic);
end majority_voter;

architecture behavior of majority_voter is
begin
process(x, y, z)
begin
if (X ='1' and Y = '1') then
M<='1';
elsif (X ='0' and Y:='0') then
M<='0';
elsif Z = '1' then
M<='1';
else M<='0';
end if;
end process;
end behavior;
```

As can be seen, the entity part is identical to that in the structural description. The architecture part, however, uses a *process* statement that includes *if–else* statements as in a procedural language, to describe the majority voter function. Figure 5.13 shows the function simulation of the VHDL code.

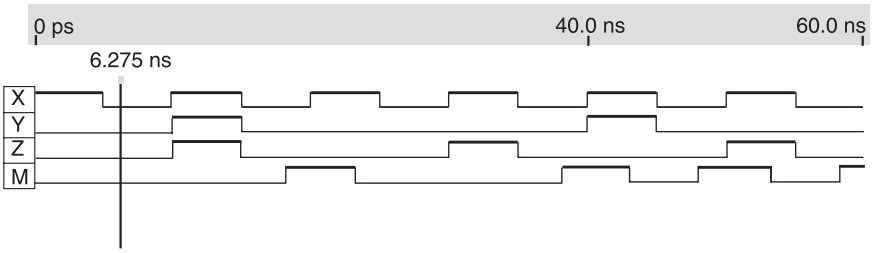


FIGURE 5.13 Functional simulation of the behavioral description.

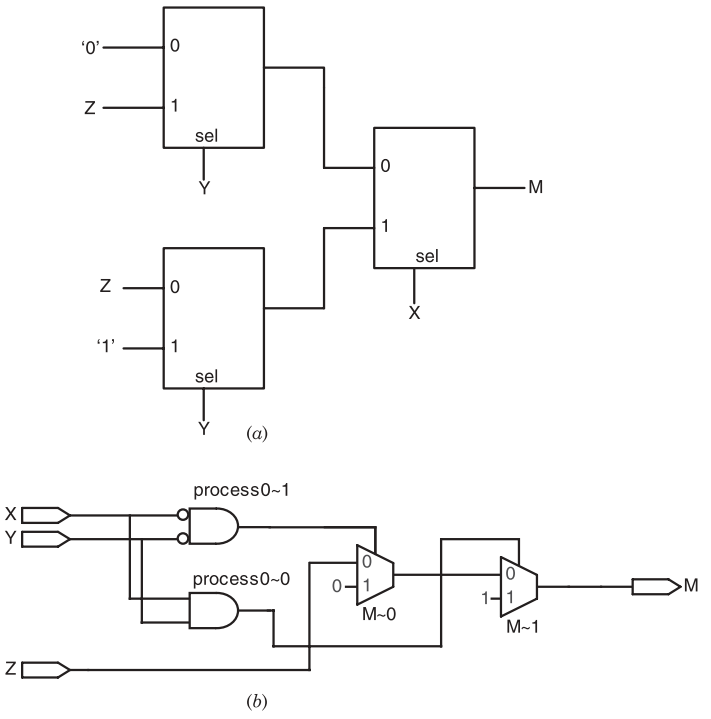


FIGURE 5.14 Alternative implementations of the majority voter circuit.

The behavioral description of the majority voter function is not unique and also it does not provide any glimpse of the structure of the circuit. For example, this behavioral description can be implemented using three AND gates and an OR gate as shown in Figure 5.12. It can also result in several alternate implementations as shown in Figure 5.14. The circuit in Figure 5.14*b* was generated by Altera's Quartus II software.

5.10 RTL DESCRIPTION

A system is described at the RTL level in terms of the transfer of information between memory elements in the system; the behavior of combinational logic blocks driving the memory elements are specified by Boolean functions. Figure 5.15 illustrates the concept

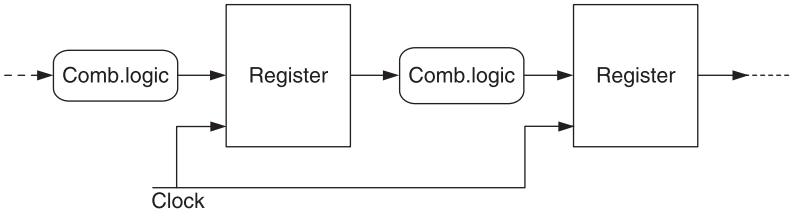


FIGURE 5.15 Concept of RTL description.

of the register transfer model. It is based on the principle of synchronous sequential circuit operation; combinational logic blocks are used as next state logic that drives the registers (memory elements) and also for generating the circuit output(s). The application of a clock pulse results in the modification of the data content of a register and the transfer of the modified data to another register. Thus the operation of a sequential circuit is synchronized with a clock and is explicitly defined at each clock cycle. (Chapters 4 and 7 deal with concepts and design of sequential circuits, respectively.)

In RTL descriptions the modeling of registers is done at the functional level. Thus RTL descriptions may be considered a form of behavioral description. On the other hand, the combinational logic blocks are described at the structural level; hence RTL descriptions are implicitly structural. The Boolean expressions for the combinational blocks use only input signals at the ports of the circuit entity and are represented by concurrent signal assignments.

As an example, the VHDL code for the majority voter circuit of Figure 5.12 at the RTL level is

```
library ieee;
use ieee.std_logic_1164.all;
entity majority_voter is
port (X, Y, Z: in std_logic;
      M: out std_logic;
end majority_voter;

architecture regtransfer of majority_voter is
begin
    M<=(X and Y) or (X and Z) or (Y and Z);
end regtransfer;
```

Another example of the VHDL description at the RTL level is given below for a circuit with three inputs a, b, and c, and three outputs w, x, and y. Output w is 1 if any one of the inputs is 0, x is 1 if any two inputs are 0, and y is 1 if all three inputs are 0.

```
library ieee;
use ieee.std_logic_1164.all;
entity zerocount is
port (a,b,c: in std_logic;
      w,x,y: out std_logic);
end zerocount;

architecture rtl of zerocount is
begin
w<=not (a xor b xor c) and (a or b or c);
x<=(a xor b xor c) and (not (a and b and c));
y<=not (a or b or c);
end rtl;
```

Note that the VHDL description uses three concurrent statements in the architecture. The simulation results of the circuit are shown in Figure 5.16. The circuit implementation from the code is generated by the Quartus II software and is shown in Figure 5.17.

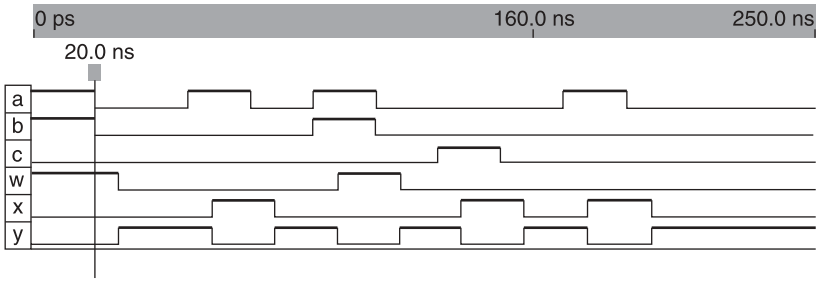


FIGURE 5.16 Simulation results from the RTL description of zero counting circuit.

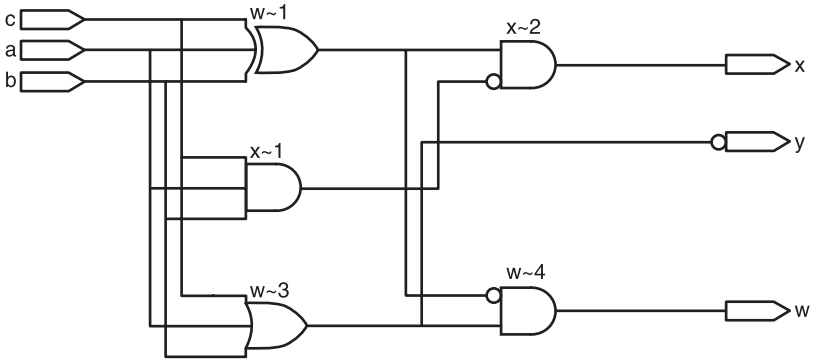
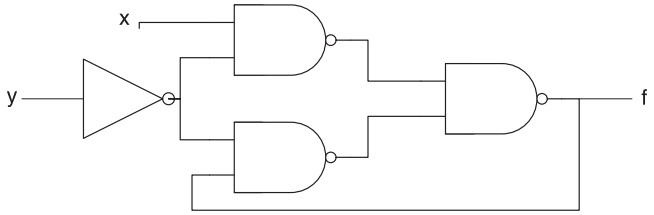


FIGURE 5.17 Circuit for counting the number of 0's in the inputs.

EXERCISES

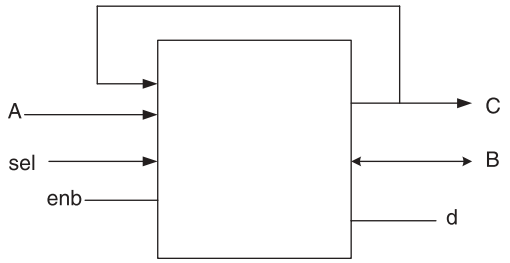
1. A circuit to compare two 4-bit numbers (A_{3-0} and B_{3-0}) is to be designed. The status of the comparison is available on the outputs *Equal, Not_equal, Less_than*, and *Greater_than*. Write the entity of the circuit.

2. Write the entity of the circuit shown below.



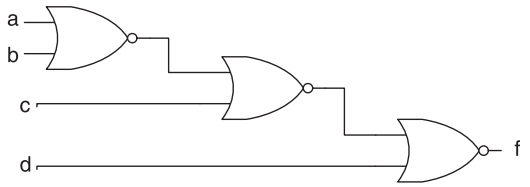
3. Describe the architecture of the comparator circuit of Figure 5.1 at the behavioral level.

4. Write the entity of a circuit represented as a black box:

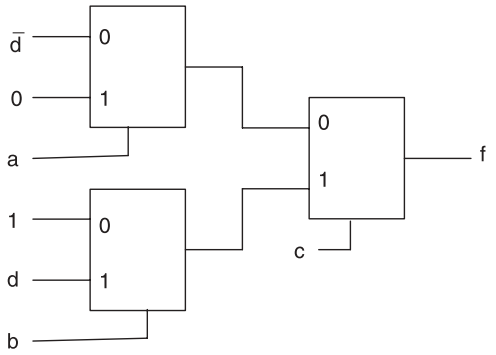


A is 8-bit input bus
enb is the output enable signal
sel is a 2-bit select input
B is an 8-bit bidirectional bus
C is a 9-bit output bus that also feeds back to the input of the circuit
d is a tristate output

5. Describe the architecture of the following circuit in VHDL at the structural level:



6. Write the entity and the architecture for the following circuit; each box in the circuit is a 2-to-1 multiplexer.

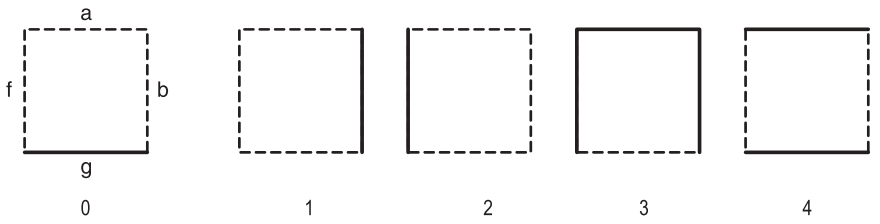


7. Write the VHDL code to specify a circuit that generates the square of a 3-bit number.

8. Write the VHDL code for a full adder.

9. Use the full adder of Exercise 3 to describe a 2-bit adder.

10. A circuit receives binary numbers corresponding to 0 and 4 inclusive and displays the following patterns:



Describe the circuit in VHDL assuming NAND–NAND logic is to be used to implement the circuit.

11. A combinational circuit is to be designed to generate a parity bit for input digits in BCD code. The circuit also has an additional output that produces an error signal if a non-BCD digit is input to the circuit. The circuit is to be realized using NAND–NAND logic. Write the VHDL code for the circuit at the structural level.

12. A computing system consists of four processors, P1, P2, P3, and P4, and four blocks of memories, M1, M2, M3, and M4. Processor P1 is allowed to use memory blocks M1 and M4 only. Memory block M2 can be used only by P2 and P3. Processor P3 can use all blocks of memories. All processors can use block M4. The circuit produces an active high output only if a processor uses an appropriate memory block. Write the entity and the architecture for the circuit.

6 Combinational Logic Design Using VHDL

6.1 INTRODUCTION

Combinational logic circuits are described in VHDL using concurrent signal assignment statements or process statements. All concurrent signal statements used to describe a circuit are executed simultaneously. The following example shows the VHDL code for a circuit function:

```
library ieee;
use ieee.std_logic_1164.all;

entity example1 is
port (x,y: in std_logic;
      f1,f2: out std_logic);
end example1;

architecture simple of example1 is
begin
f1 <= not x and not y;
f2 <= x or y;
end simple;
```

The architecture part of the code has two concurrent signal assignment statements `f1` and `f2`. Both outputs `f1` and `f2` will change simultaneously as soon as either x or y signal on the right side of the assignment operator "<= " changes value; thus the order of the assignment statements has no significance. Note that the signal assignment operator is not represented by "=" as in a conventional programming language.

When the VHDL description composed of concurrent assignment statements is translated by a logic synthesis program, the hardware implementation may not necessarily be in the form of gates defined in the concurrent statements; it is determined based on the target device architecture.

Principles of Modern Digital Design, by Parag K. Lala

6.2 CONCURRENT ASSIGNMENT STATEMENTS

Concurrent assignment statements can be in one of four categories:

- Instantiations
- Direct assignment
- Conditional assignments
- Selected assignments

Instantiations were discussed previously in Chapter 5.

6.2.1 Direct Signal Assignment

The concurrent signal assignments used in the VHDL code of the above example are of the direct type. They are used to specify Boolean expressions. A second example of direct concurrent signal assignment is shown by writing the VHDL code for the Boolean expression

$$f(a, b, c, d) = ab\bar{c} + \bar{b}c\bar{d} + ad$$

The VHDL code is

```
library ieee;
use ieee.std_logic_1164.all;

entity example2 is
port (a,b,c,d: in std_logic;
f: out std_logic);
end example2;

architecture simple of example2 is
signal s1,s2,s3: std_logic;
begin
 s1 <= a and b and not c;
 s2 <= not b and c and not d;
 s3 <= a and d;
 f <= s1 or s2 or s3;
end simple;
```

Three internal *signals* of type `std_logic` called `s1`, `s2`, and `s3` are declared. Note that signal statements are similar to port statements in the VHDL entity part, except that there is no mode declaration for internal signals. They are like wires in electronic circuits and are used to connect different components of a design. As in an electronic circuit, signals can be single bits (e.g., a clock or a reset) or they can be buses of a specified width. All signals must be declared with both a name and a data type.

In this example `s1`, `s2`, and `s3` are ''wires'' that connect AND gates corresponding to product terms $ab\bar{c}$, $\bar{b}c\bar{d}$, and ad, respectively, to the OR gate that drives output f as shown in Figure 6.1. All four concurrent assignment statements are executed simultaneously.

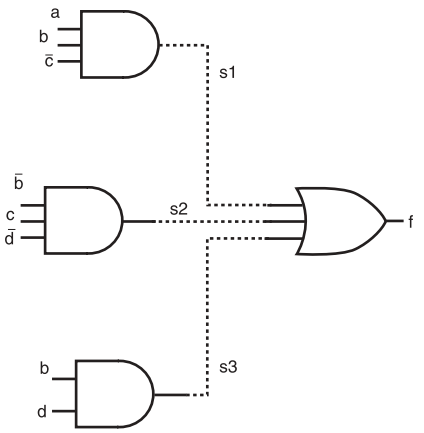


FIGURE 6.1 Example of signals as wires.

It should be mentioned that the inclusion of signal statements is a matter of choice, not a definite requirement. The functionally equivalent code for the architecture part of example2 in RTL coding style is

```
architecture simple of example2 is
begin
  f <= (a and b and not c) or (not b and c and not d) or  (b and d);
end simple;
```

6.2.2 Conditional Signal Assignment

A conditional signal assignment statement, also known as a *when–else* statement, has more than one expression associated with it, each of which corresponds to a condition. The syntax of a *when–else* statement is as follows:

```
signal_name <= expression_w when condition1 else
               expression_x when condition2 else
               expression_y when condition3 else
               expression_z;
```

However, there is only one signal assignment operator (<=) in a conditional signal assignment. When a conditional signal assignment statement is executed, each condition is tested in order as it appears in the statement, that is, `condition1` first, then `condition2`, and so on. The first condition that is satisfied has the evaluated value of the associated expression assigned to the target signal on the left of the signal assignment operator. For example, if `condition2` is satisfied `expression_x` is evaluated and its value assigned to the target signal. If none of the conditions are satisfied the value of the final expression is assigned to the target signal. Alternatively, if two or more conditions are satisfied only the first condition is assigned to the target signal; the rest are ignored. Note that instead of an expression a predetermined bit or vector can be associated with

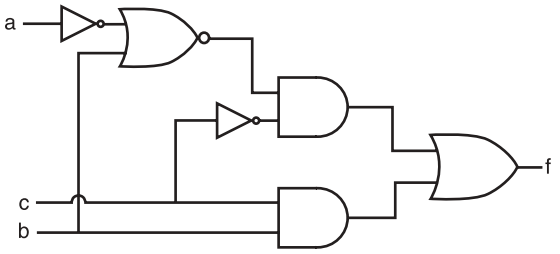


FIGURE 6.2 Circuit corresponding to VHDL code of `example3`.

a particular condition. The following example illustrates the use of a conditional assignment statement expression:

```
library ieee;
use ieee.std_logic_1164.all;

entity example3 is
port (a,b,c: in std_logic;
f: out std_logic);
end example3;

architecture simple of example3 is
begin
f <= (b xnor c) when a = '1' else
     (b and c);
end simple;
```

In the above example, if a is at logic 1 expression $(\overline{b \oplus c})$ is evaluated and assigned to f, otherwise f is set to b **and** c. The corresponding circuit is shown Figure 6.2, as illustrated by this example, conditional signal assignment statements are always synthesized into combinational logic.

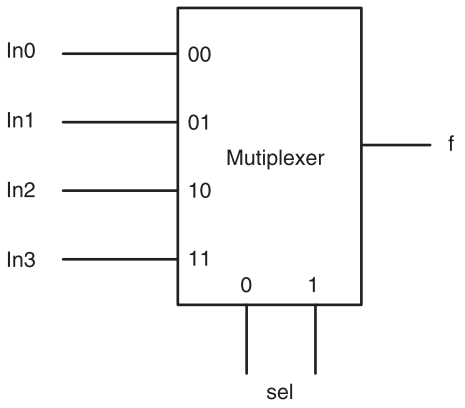


FIGURE 6.3 A 4-to-1 multiplexer.

Next, we consider more conventional use of conditional signal statements, for example, specifying operations of multiplexers and decoders. The VHDL code for a 4-to-1 multiplexer shown in Figure 6.3 is

```
1  library ieee;
2  use ieee.std_logic_1164.all;

3  entity multiplexer is
4  port (sel:in std_logic_vector(0 to 1);
5  In0,In1,In2,In3: in std_logic;
6  f: out std_logic);
7  end multiplexer;

8  architecture simple of multiplexer is
9  begin
10 f <= In0 when sel = "00" else
11 In1 when sel = "01" else
12 In2 when sel = "10" else
13 In3 when sel = "11" else
14 '0';
15 end simple;
```

Any time the values of the `sel` bits change the condition statement is executed. Line '`0`' in code indicates that for all other possible conditions of `sel` bits the output will be 0. Note that each *standard_logic* bit can have 9 values, thus for two bits there are 81 ($=9 \times 9$) combinations. A functionally equivalent code could be obtained by replacing lines 13 and 14 by just `In3`. However, the code given above is more informative.

The final `else` (line 13) clause followed by the last statement containing '`0`' (line 14) is included to avoid implied memory at the output circuit synthesized from the code. Recall that each bit defined as *standard_logic* type can have nine possible values including 0 and 1. Thus if the selected control bits in the multiplexer code are neither of the specified four conditions, the output of the multiplexer will retain its old value (*implied memory*), thus inadvertently resulting in the creation of a latch at the output of the multiplexer as shown in Figure 6.4. This behavior can sometimes also be used to create a latch when one is needed.

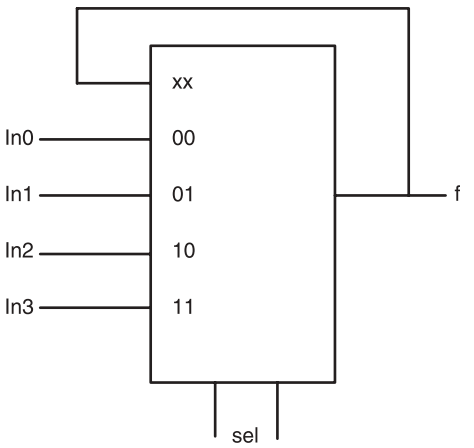


FIGURE 6.4 Implied memory at the multiplexer output.

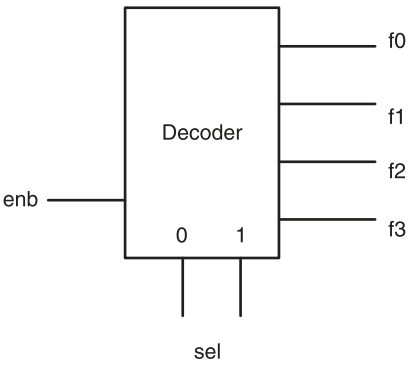


FIGURE 6.5 A 2-out of-4 decoder.

As another example of the use of conditional signal assignment in VHDL, let us specify the operation of a 2-out-of-4 decoder shown in Figure 6.5.

The VHDL code for the 2-out-of-4 decoder of Figure 6.5 is

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
port (sel: in std_logic_vector(0 to 1);
enb: in std_logic;
f: out std_logic_vector (3 downto 0));
end decoder;

architecture simple of decoder is
begin
f <= (others => '0')when enb = '0' else
 "0001" when sel = "00" else
 "0010" when sel = "01" else
 "0100" when sel = "10" else
 "1000" when sel = "11";
end simple;
```

The first statement in the architecture part of the VHDL code is used to set the outputs of the decoder to all 0's when `enb` input is at logic 0. As was shown previously in Chapter 3, the `enb` input of a decoder is used to interconnect smaller size decoders in order to form a larger decoder.

The conditions specified in *when–else* statements have implied priority. This should be clear from the VHDL code for a 4-to-2 bit priority encoder shown below in which the order of priority is emphasized by the order of the *when–else* statements; the condition `"11"` in first statement has the highest priority and `"00"` in the last one has the lowest priority.

```
library ieee;
use ieee.std_logic_1164.all;

entity priority_encoder is
port (w: in std_logic_vector(3 downto 0);
```

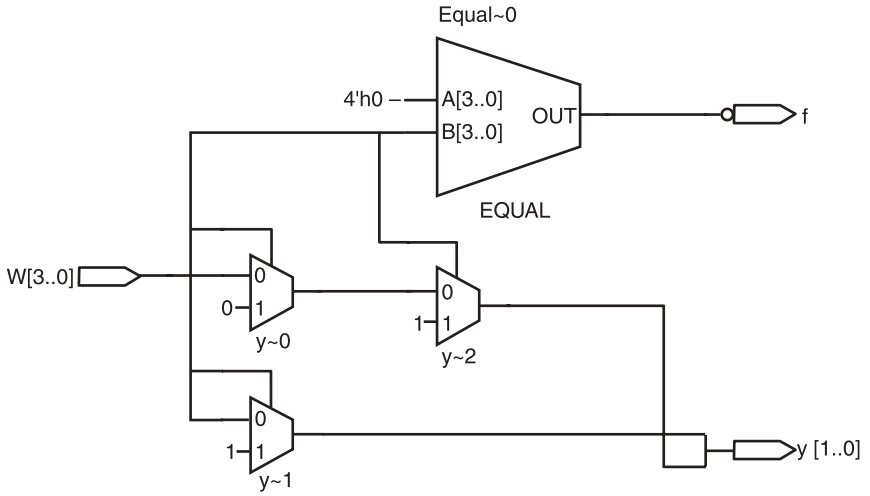


FIGURE 6.6 A 4-to-2 priority encoder generated from the VHDL code using *when–else* statements.

```
y: out std_logic_vector(1 downto 0);
f: out std_logic);
end priority_encoder;

architecture behavior of priority_encoder is
begin
y <= "11" when w(3)='1' else
    "10" when w(2)= '1' else
    "01" when w(1)='1' else
    "00";
f <= '0' when w = "0000" else '1';
end behavior;
```

Thus *when–else* statements allow specifying conditions that have certain priority in an appropriate order. Figure 6.6 shows the generation of the priority encoder circuit from the VHDL code by the Quartus II software.

6.2.3 Selected Conditional Signal Assignment

Selected conditional signal assignment statements in *with–select–when* form are similar to conditional signal assignments except that no priority is implied by the order of the statements. The syntax of a *with–select* statement is

```
with condition select
 signal_name <= expression1 when value1;
             <= expression2 when value2;
             <= expression3 when value3;
             <= expression4 when others;
```

The *with . . . select* part of a selected conditional signal assignment statement evaluates the condition and compares the result with each choice value. All possible values of the condition must be covered by the set of *when* clauses. The *when* clause with the matching choice value has its associated expression assigned to the target signal on the left of the assignment operator (<=).

Like conditional signal assignment statements, select signal assignments are also converted into combinational logic by a synthesis program.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux8to1 is
port (In0,In1,In2,In3,In4,In5,In6,In7: in std_logic;
      s:in std_logic_vector(2 downto 0);
      f:out std_logic);
end mux8to1;

architecture behavior of mux8to1 is
begin
with s select
f <=  In0 when "000",
            In1 when "001",
            In2 when "010",
            In3 when "011",
            In4 when "100",
            In5 when "101",
            In6 when "110",
            In7 when others;
end behavior
```

This code for an 8-to-1 multiplexer is in general similar to the code that used *when–else* statements. However, the *when others* clause must be used to cover all unwanted combinations of `s` bits. The simulation result of the VHDL code of the multiplexer is shown Figure 6.7.

The input conditions to a multiplexer have no implied priority; thus either conditional or select signal assignment statements can be used to describe a multiplexer. However, if the conditions have priorities then the VHDL code using *with–select–when* statements will

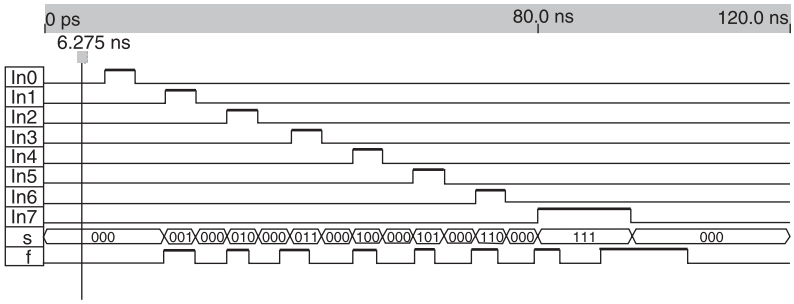


FIGURE 6.7 Simulation results of 8-to-1 multiplexer.

be less efficient than the code using *when–else* statements. For example, the VHDL code for the priority encoder using a *with–select–when* statement is

```
library ieee;
use ieee.std_logic_1164.all;

entity priority_encoder1 is
port (w: in std_logic_vector(3 downto 0);
y: out std_logic_vector(1 downto 0);
f: out std_logic);
end priority_encoder1;

architecture behavior of priority_encoder1 is
begin
with w select
 y <= "11" when "1000",
     "11" when "1001",
     "11" when "1010",
     "11" when "1011",
     "11" when "1100",
     "11" when "1101",
     "11" when "1110",
     "11" when "1111",
     "10" when "0100",
```

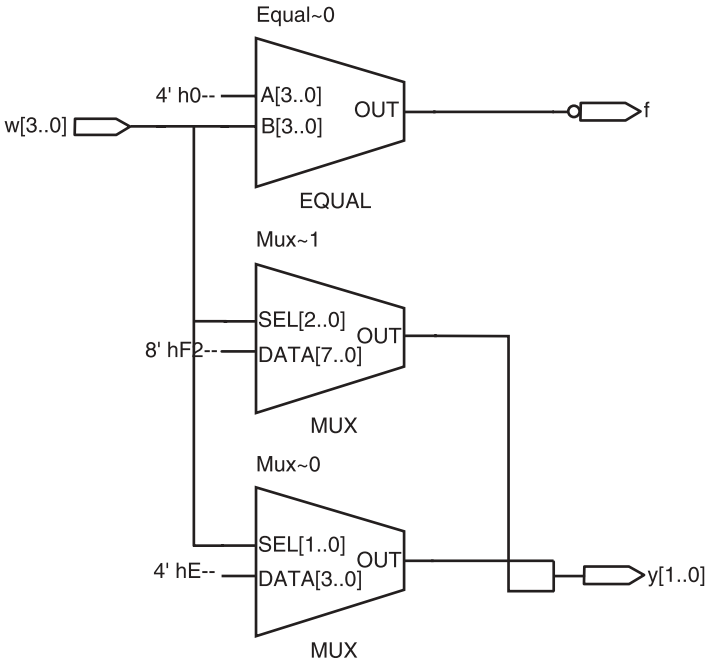


FIGURE 6.8 A 4-to-2 priority encoder generated from the VHDL code using *with–select–when* statements.

```
    "10" when "0101",
    "10" when "0110",
    "10" when "0111",
    "01" when "0010",
    "01" when "0011",
    "00" when others;
 f <= '0' when w = "0000" else '1';
end behavior;
```

Note that all possible conditions have been specified to avoid implied priority. As a result, this version produces less efficient code and in general requires more logic for implementing the priority encoder. The encoding circuit in this case requires a 4-bit comparator and two 4-to-1 multiplexers, whereas the circuit in Figure 6.8 used a 4-bit comparator and three 2-to-1 multiplexers.

6.3 SEQUENTIAL ASSIGNMENT STATEMENTS

Concurrent assignment statements used in VHDL code for describing combinational logic circuits are executed in parallel: that is, each statement operates independently of all other statements. In fact, this is the way the circuit represented by the VHDL code works. It is also possible to specify combinational logic functions using sequential statements; this resembles coding using a conventional programming language. The VHDL code (of a circuit) composed of sequential statements provides behavioral information of the circuit in terms of its inputs and outputs; the actual hardware structure of the circuit is determined by the logic synthesis tool that uses the VHDL code.

6.3.1 Process

All sequential statements must be grouped inside a *process*. They are executed in sequence; any change in the order in which they appear in the process affects the intended function of the code. A signal within a process may be assigned different values during the execution of the process; however, the last value assigned to the signal before the end of the process is the one it retains.

A process itself is considered a concurrent statement even though all its component statements are sequential in nature. The architecture of a VHDL program may have several processes as well as other concurrent statements, all of which are executed in parallel. The syntax of a process is

```
[label] process (sensitivity list);
        declaration statements
        begin
          sequential statement 1
          sequential statement 2
                .
                .
                .
          sequential statement n
        end process;
```

The label to identify a process is optional. If several processes are used in a VHDL program, the process name can identify the operation performed by a specific process. The list of signals in parentheses immediately following the keyword *process* in the process definition is known as the *sensitivity list.* A process is *activated* (i.e., the statements within the process are executed) if any signal in its sensitivity list changes value; otherwise the process remains *suspended*. It should be mentioned here that a process can also be defined without a sensitivity list; in that case the process must include a `wait` statement before other sequential statements to prevent continuous execution of the process. Later in the section we discuss processes without sensitivity lists.

As an example, let us specify the Boolean expression $f(a, b, c) = ab \oplus c$ using the following process:

```
library ieee;
use ieee.std_logic_1164.all;

entity example is
port (a,b,c: in std_logic;
f:out std_logic);
end example;

architecture behavior of example is
signal x: std_logic;
begin

process (a,b,c)
begin
x <= a and b;
 f <= x xor c;
end process;
end behavior;
```

Input signals `a`, `b`, and `c` are included in the sensitivity list of the process because the output may get affected if one of these changes value. A signal assignment statement is used to declare `x` as a signal bit. This process can be defined using a concurrent signal assignment statement:

```
f <= (a and b) xor c;
```

This statement can indeed be considered as a concise description of the above process. If any of the signals on the right-hand side of the expression assignment operator changes, the statement will be executed just as the process statement is activated if a signal in its sensitivity list changes its value.

A VHDL process can also use variable(s) as in conventional programming languages. A variable assignment statement must be declared inside a process just before the keyword *begin* in the process statement. A signal assignment statement, on the other hand, is declared inside the architecture before the keyword *begin*. The variable declaration statement is the same as its signal declaration counterpart except the keyword *variable* is used

instead of *signal*. The above VHDL code is repeated below with appropriate changes; the entity part is skipped because there are no changes to be made in it.

```
architecture behavior of example is
signal x:std_logic;
begin
process (a,b,c)
variable y: std_logic;
begin
y:= a and b;
y:= y xor c;
x <= y;
end process;
f <= x;
end behavior;
```

Both a signal assignment statement and a variable assignment statement are used. The signal and the variable declared are `x` and `y`, respectively. Variable `y` is first assigned the value of `a and b`. Then the `xor` of `y` and `c` is computed, and `y` is updated with the new value. The value of a variable is valid only inside a process; therefore `y` is transferred to signal `x` before exiting the process. The value of signal `x` is then assigned to output `f`. Note that VHDL statements outside a process are considered concurrent, not sequential.

A VHDL architecture may have several processes active at the same time. These processes communicate with each other through internal signals. An internal signal generated by one processor is included in the sensitivity list of another one, thus activating that process when the signal changes value.

There are four sequential statements that, like concurrent assignment statements, transfer values to target signals on the left side of the assignment operator. These are

1. *If*
2. *Case*
3. *Loop*
4. *For generate*

Only these statements can be used inside a process; concurrent assignment statements *when* and *with* cannot be used inside a process.

6.3.2 *If–Then* Statement

An *if–then* statement evaluates each condition in a sequence of conditions in the order in which they are presented, until one of the conditions is satisfied or until they are exhausted. A condition must of course evaluate to true or false. The statement(s) associated with the satisfied condition is executed and the rest of the conditions are ignored.

The syntax of an if statement is

```
If condition 1 then
Statement(s) executed if condition 1 is true
```

```
else
Statement(s) executed if condition 1 is false
end if;
                .
                .
                .
                .
If condition n then
Statement(s) executed if condition n is true
else
Statement(s) executed if condition n is false
end if;
```

To illustrate, the architecture part of the VHDL code for a 2-to-1 multiplexer (Fig. 6.9) using the *if–then* statement is as follows:

```
if (s='1') then
    f <= In0;

else
    f <= In1;

end if;
```

If condition `s=1` is satisfied `In0` is transferred to output `f` of the multiplexer, and the second condition `s=0` is ignored. On the other hand, if `s=0`, the second condition not the first is satisfied and `f` gets the value of `In1`.

The *else* clause in an *if* statement is optional. However, the absence of the clause will result in the creation of a latch at the output of the multiplexer as discussed previously.

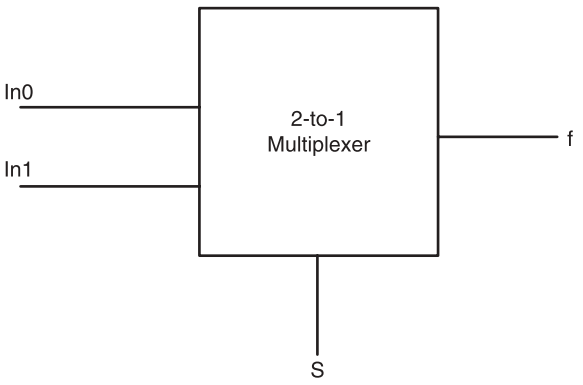


FIGURE 6.9 Block diagram of 2-to-1 multiplexer.

As another example, let us specify the VHDL code of an *n*-bit comparator:

```
library ieee;
use ieee.std_logic_1164.all;

entity comparator is
generic(n: natural:=4);

port(  X: in std_logic_vector(n-1 downto 0);
       Y: in std_logic_vector(n-1 downto 0);
       less: out std_logic;
       equal: out std_logic;
       greater: out std_logic);
end comparator;

architecture behavior of comparator is
  begin
  process(X,Y)
  begin
  if (X>Y) then
   less <= '0';
   equal <= '0';
   greater <= '1';
  elsif (X=Y) then
   less <= '0';
   equal <= '1';
   greater <= '0';
  else
   less <= '1';
   equal <= '0';
   greater <= '0';
  end if;
  end process;
  end behavior;
```

Any number of *if* statements can be combined to form *nested if*. In *nested if* the *else* part of one statement is combined with the *if* part of the following *if* statement to form the *elsif* clause; note that *elsif* is one word (not *else if*). The syntax of *nested if* is

```
If condition 1 then
Statement(s) executed if condition1 is true
elsif condition 2 then
Statement(s) executed if condition1 is true
                .
                .
                .
elsif condition n then
Statement(s) executed if condition n is true

Statement(s) executed if none of conditions are true
end if;
```

To illustrate, the 4-to-1 multiplexer shown in Figure 6.3 is described using nested *if* statements:

```
library ieee;
use ieee.std_logic_1164.all;

entity multiplexer is
port (in0,in1,in2,in3: in std_logic;
s0,s1: in std_logic;
f: out std_logic);
end multiplexer;

architecture behavior of multiplexer is
begin
process (s0,s1,in0,in1,in2,in3)
begin
  if ((not s1 and not s0)= '1') then
  f <= in0;
  elsif ((not s1 and s0)= '1') then
  f <= in1;
  elsif ((s1 and not s0)= '1') then
  f <= in2;
  else
  f <= In3;
end if;
end process;
end behavior;
```

The circuit structure resulting from nested *if* statements can have long signal paths from the input to the output of the circuit, thereby increasing the signal propagation delay of the circuit. The logic equations for the multiplexer circuit derived by the Quartus II system from the above VHDL code are

$$L2 = s0 \cdot s1 + \bar{s}0 \cdot (s1 \cdot in2 + \bar{s}1 \cdot in0)$$
$$L3 = s0 \cdot (L2 \cdot in3 + L\bar{2} \cdot in1 + s\bar{0} \cdot L2)$$
$$f = L3$$

The resulting circuit will have multiple logic levels. Note that the VHDL code does not result in a single standard sum-of-products expression for the multiplexer; that is,

$$f = s1 \cdot s0 \cdot in3 + s1 \cdot \bar{s}0 \cdot in2 + \bar{s}1 \cdot s0 \cdot in1 + \bar{s}0 \cdot \bar{s}1 \cdot in0$$

From the syntax of an *if* statement it should be clear that conditions in an *if* statement have implied priority. Condition 1 has the highest priority because it is evaluated first, and if it is true all the remaining conditions are skipped. Condition 2 is evaluated only if condition 1 is false, and condition n is evaluated only if all previous $n - 1$ conditions are false. Thus condition n has the lowest priority. This property of *if* statements can be utilized in

designing priority encoder circuits. The VHDL code for a 4-to-2-priority encoder circuit using an *if* statement is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity priorityencoder is
port (w: in std_logic_vector(3 downto 0);
y: out std_logic_vector(1 downto 0);
z: out std_logic);
end priorityencoder;

architecture behavior of priorityencoder is
begin
 process(w)
 begin
 if w(3)='1'
 then y <= "11";
   elsif w(2)='1'
 then y <= "10";
   elsif w(1) ='1'
 then y <= "01";
   else y <= "00";
 end if;
 end process;
 z <= '0' when w = "0000" else '1';
end behavior;
```

6.3.3 *Case* Statement

A *case* statement is an alternative to an *if* statement. However, unlike in an *if* statement a *case* statement does not have an implied priority of conditions. It is in fact equivalent to a selected conditional signal assignment (i.e., *with–select* statement). The only difference is that a *with–select* statement is a concurrent statement whereas a *case* is a sequential statement and therefore must be inside a process. The syntax of a case statement is

```
case expression is
when choice1 => sequential statements;
when choice2 => sequential statements;
             .
             .
             .
when others  => sequential statements;
end case;
```

The expression associated with the *case* part is evaluated to a value that is either an integer, a standard logic vector, or an enumerated type. This value is compared with each choice of value; all possible choices of value must be covered by the set of *when* clauses.

The statements associated with the *when* clause that has the matching choice value are then executed. The *when others* statement must be included if all possible choices are not covered by the *when* clauses. All choices should be unique so that only one of them matches the value evaluated from the expression.

To illustrate, let us write the VHDL code for the following Boolean expression:

$$f(a, b, c) = \Sigma m\ (1, 2, 4, 7)$$

The code is

```
library ieee;
use ieee.std_logic_1164.all;

entity circuit1 is
port (a,b,c: in std_logic;
f: out std_logic);
end circuit1;
architecture comb1 of circuit1 is
begin
process(a,b,c)
begin
case std_logic_vector'(a,b,c) is
when "001" =>  f <= '1';
when "010" =>  f <= '1';
when "100" =>  f <= '1';
when "111" =>  f <= '1';
when others =>  f <= '0';
end case;
end process;
end comb1;
```

In this code inputs `a`, `b`, and `c` are converted into a standard logic vector (`a`, `b`, `c`); this vector represents the *expression* part in the *case* statement. The choice in each of the four *when* clauses corresponds to four minterms in the on-set of the given function. The *when others* statement covers all possible combination of inputs `a`, `b`, and `c` that generate `0` at output `f`. The Boolean expression of the function derived by the Quartus II system is:

```
p2 = b · c̄
p3 = b̄ · c;
f = a xor (p2 + p3)
```

The *case* statement in the above VHDL code can be written in a more concise form as

```
case std_logic_vector (a,b,c)is
when "001"|"010"|"100"|"111" => f <= '1';
when others => f <= '0';
end case;
```

Another example of the VHDL code for a 4-to-1 multiplexer using a *case* statement is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity mux1 is
port (in0,in1,in2,in3: in std_logic;
sel:in std_logic_vector (1 downto 0);
 f: out std_logic);
end mux1;
architecture comb1 of mux1 is
begin
process(sel,in0,in1,in2,in3)
begin
case sel is
when "00" => f <= in0;
when "01" => f <= in1;
when "10" => f <= in2;
when "11" => f <= in3;
end case;
end process;
end comb1;
```

This code does not use the *when others* statement because all possible choices of `sel` value have been used.

It should be clear from the above that a *case* statement requires an alternative choice for every value of the expression. In certain cases, however, no action is required for a particular value; a *null* statement is used to specify no action. To illustrate, let us consider the following code that describes a circuit in which vector `temp` is incremented or decremented if the `sel` bits are 01 or 10, respectively; in all other cases no operation is performed. The *case* statement in the code uses a null statement to represent this:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity test is
       port (reg: out std_logic_vector (2 downto 0);
             sel:in std_logic_vector (1 downto 0);
            temp: in std_logic_vector (2 downto 0));
end test;

architecture behavior of test is
begin
process(sel,temp)
begin
case sel is
```

```
when "01" => reg <= temp +1;
when "10" => reg <= temp -1;
when others => null;
end case;
end process;
end behavior;
```

6.3.4 *If* Versus *Case* Statements

As explained earlier, the key difference between an *if* and a *case* statement is that the former can contain more than one condition whereas in the later a single expression is evaluated against multiple mutually exclusive conditions. In general, the VHDL description of a circuit using a *case* statement results in more efficient hardware than an *if*-based description. This is illustrated by specifying the function of a 4-bit ALU in VHDL; the modes of operation of the ALU are as follows:

Mode	*Function*
00	Add
01	Subtract (2's complement addition)
10	AND
11	OR

The VHDL code for the ALU using *if* statements is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ALU1 is
generic(n: natural:=4);
port(  A,B : in std_logic_vector((n-1)downto 0);
       mode: in std_logic_vector(1 downto 0);
       Cout: out std_logic;
       C   : out std_logic_vector((n-1) downto 0));
end ALU1;

architecture behavior of ALU1 is
signal D: std_logic_vector(n downto 0);
begin
  process(A,B,mode)
  begin
        D <=(others => '0');
        If mode = "00" then
        D <= '0'&A+ B;
        end if;
        if mode ="01" then
```

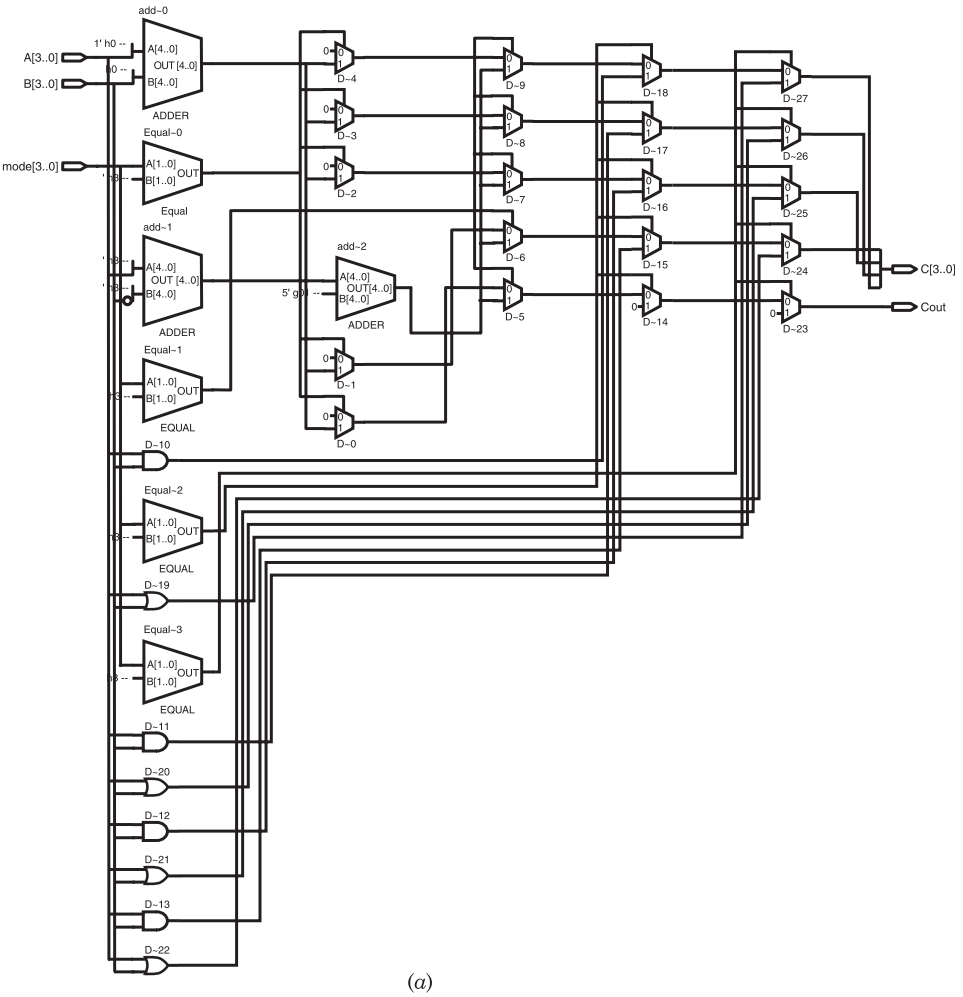
```
        D <= '0'&A + not B+1;
        end if;
        if mode ="10" then
        D <= '0'&A and B;
        end if;
        if mode ="11" then
        D <= '0'&A or B;
        end if;
 end process;
        C <= D(n-1 downto 0);
        Cout <= D(n);
 end behavior;
```

The result of each operation is transferred to an n-bit vector `D` declared as a signal; this is needed so that `D(n)` can hold the carry-out bit generated during the addition of two $(n-1)$-bit vectors. A '0' is concatenated with vector `A` to make it n-bit. After the process is executed, the least significant $(n-1)$ bits of `D` are transferred to `C`, which provides the final result. The most significant bit of `D` is transferred to `Cout`, which will be 1 if a carry-out of 1 is generated during the addition or the subtraction operation. The logic implementation of the ALU obtained from the VHDL description by the Quartus II system is shown in Figure 6.10*a*.

The ALU can also be specified using a *case* statement as below; since the entity part is the same as in the previous case, only the architecture part is shown:

```
architecture behavior of ALU is
signal D: std_logic_vector (n downto 0);
begin
  process(A,B,mode)
 begin
  D<= (others => '0');
  case mode is
  when "00" =>
  D<='0'&A+B;
  when "01" =>
  D<='0'&A + not B + 1;
  when "10" =>
  D<='0'&A and B;
  when others =>
  D<='0'&A or B;
  end case;
end process;
  C<= D(n-1 downto 0);
  Cout<= D(n);
end behavior;
```

Figure 6.10*b* shows the Quartus-generated logic implementation of the ALU. Note that the *if*-based ALU circuit is significantly more complex than the *case*-based implementation and will be slower.



(*a*)

FIGURE 6.10 (*a*) Implementation of the *if*-based ALU code and (*b*) implementation of the *case*-based ALU code.

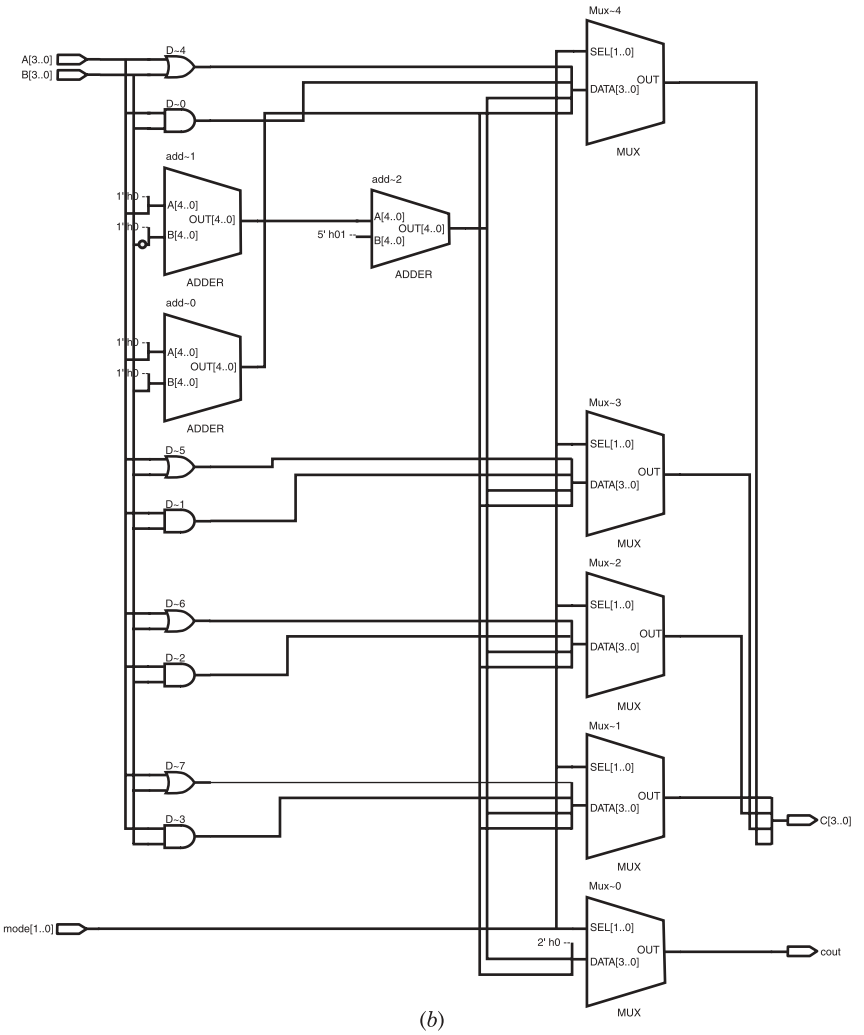
6.4 LOOPS

Loops are used to specify repeated execution of one or a sequence of statements; the statements must be inside a process. Two variants of loops in VHDL are the *for loop* and the *while loop*.

6.4.1 *For Loop*

The *for loop* allows a fixed number of iterations of a set of statements. The syntax is as follows:

```
label: for identifier in range loop
 statement(s) to be repeated
end loop;
```



(*b*)

FIGURE 6.10 (*Continued*).

The label for a loop is optional. The *in* range part of the *for loop* defines the number of times the statement(s) in the loop are to be repeated. It is defined in the same format as a logic vector. The identifier is a variable that is automatically created and is assigned a value of the range each time the statements in the loop are executed.

To illustrate, let us write the VHDL code for a circuit that counts the number of 1's in an n-bit register and stores the binary equivalent of the number in an m-bit register, where

$$m = \lceil \log_2 n \rceil$$

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity circuit3 is
generic (n:integer:=7; m:integer:=3);
port (a: in std_logic_vector (n-1 downto 0);
      b: out std_logic_vector(m-1 downto 0));
end circuit3;

architecture comb1 of circuit3 is
signal s: integer;
begin

process(a)
variable p: integer;
begin
p:= 0;
for i in (n-1)downto 0 loop
if a(i)= '0' then
p:= p + 1;
end if;
end loop;
s <= p;
end process;
b <= conv_std_logic_vector(s,m);
end comb1;
```

The process in the code uses a local variable (p) that is initialized to 0 when the process starts. When entering the loop, i is initialized to the first value in the range (6). In the first iteration of the loop, a(6) is checked for a 0, and the result is added to p, and i is decremented. In the second iteration, a(5) is checked for a 0 again and the result, once more, is added to p and i is decremented again. This sequence repeats until i=0. After the last iteration, p will hold an integer value that equals the number of 0's in vector a. For example, if a holds the vector 1000101, the codes resulting from each iteration of the loop will be

```
i=6   a(6)=1  p:=0
i=5   a(5)=0  p:=1
i=4   a(4)=0  p:=2
i=3   a(3)=0  p:=3
i=2   a(2)=1  p:=3
i=1   a(1)=0  p:=4
i=0   a(0)=1  p:=4
```

The value of a variable is not valid outside a process, therefore p is transferred to s,which was declared as an integer type signal before the process started. Note that a signal is

declared in an architecture body just before the keyword `begin`, whereas a variable is declared inside a process also just prior to `begin`. The binary equivalent of integer `p` is obtained by converting it into a vector of length `m`; this is done by using the function `conv_std_logic_vector(s,m)`. This function converts integer `s` into an `m`-bit vector. The VHDL compiler will accept the function as a valid statement only if the `use.ieee_std_arith.all` package is specified before the entity declaration.

As an example of the use of a *for loop* statement, the VHDL code for a 4×4 multiplier is given below. It also shows the use of variables inside a process. The code uses the shift-add algorithm for multiplication.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity multiply is
port(multiplicand, multiplier:in std_logic_vector(3 downto 0);
result: out std_logic_vector(7 downto 0));
end multiply;

architecture behavior of multiply is
signal zeros: std_logic_vector (3 downto 0);
begin
zeros <= (others => '0');

process(multiplicand,multiplier)
variable P: std_logic_vector(7 downto 0);
variable multiplicand_reg: std_logic_vector (7 downto 0);
begin
multiplicand_reg:= zeros & multiplicand;
P:= (others => '0');
for i in 1 to 4 loop
if multiplier(i-1)='1' then
P:= P + multiplicand_reg;
end if;
multiplicand_reg (7 downto 0):= multiplicand_reg(6 downto 0)&'0';
end loop;
result <= P;
end process;
end behavior;
```

Both the multiplier and the multiplicand are declared as 4-bit vectors and the `result` is transferred to an 8-bit vector. The `signal zeros` is a 4-bit vector; it is initialized with all 0's. The process has the multiplier and the multiplicand in its sensitivity list. Two variables `P` and `multiplicand_reg` are used in the process statement. `P` is initialized with all 0's. The first 4 bits of `multiplicand_reg` are initialized with 0's and the next 4 bits with the value of the multiplicand. The *for-loop* statement has the range of 1 to 4. During each iteration, a bit of the multiplier starting from the least significant bit is

checked to determine whether it is 0 or 1. If it is a 1, the `multiplicand_reg` is added to `P` and the old value of `P` is replaced with this. Bits 6–0 of variable `multiplicand_reg` are concatenated with a `0` and the result replaces the old value of `multiplicand_reg`; note this is equivalent to shifting the value of `multiplicand_reg` to the left. If a bit of the multiplier is 0 during an iteration, `multiplicand_reg` is not added to P but `multiplicand_reg` is shifted to the left. When the loop is completed, variable `P` contains the multiplication result and it is transferred to output vector `result`. Note that since `P` is a variable its value must be transferred to the output or to a signal before the end of the process; the value of a variable is not available outside the process.

Figure 6.11 shows the simulation results of the 4×4 multiplier code for different combinations of multiplier and multiplicand values.

6.4.2 *While Loop*

The *while loop* unlike the *for loop* does not specify the number of iterations but rather the condition in the form of a Boolean expression which if evaluated to be true allows repeated execution of the statement(s) in the loop, otherwise the loop terminates. The condition is tested every time before the start of an iteration; the iteration is skipped if the condition is false. The label is optional. The syntax of *while loop* is

```
label: while condition loop
statement(s)
end loop label;
```

As an example, let us consider the following VHDL code, which generates the EX-OR of two 8-bit binary numbers:

```
library ieee;
use ieee.std_logic_1164.all;

entity circuit4 is
port (a: in std_logic_vector (7 downto 0);
      b: in std_logic_vector(7 downto 0);
      c: out std_logic_vector(7 downto 0));
end circuit4;
architecture comb1 of circuit4 is
```

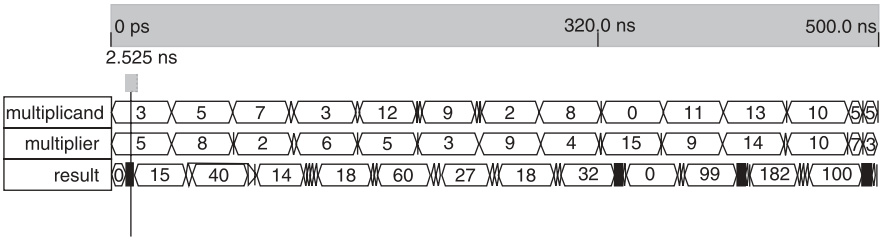


FIGURE 6.11 Simulation results for the multiplier.

```
begin
 process(a,b)
  variable p: std_logic;
  variable i:integer;
  begin
  i:=0;

 while i<8 loop
  p:= a(i) xor b(i);
  c(i) <= p;
  i:=i+1;
  end loop;
 end process;
end comb1;
```

In general, a *while loop* cannot be synthesized in hardware and is used only in *testbenches*. A testbench is a VHDL source file that contains a group of input vectors to exercise a particular VHDL code and check its correct operation.

6.5 *FOR–GENERATE* STATEMENT

In certain cases it is possible to implement a design as a regular structure of identical components. In the VHDL code of such a design the component to be used is defined once, and then it is repeatedly instantiated. The replication of component instantiation statements over a specified range is done by using the *for–generate* statement. The *for–generate* statement can in fact be used for the replication of any concurrent statement(s). The syntax of *for–generate* statement is as follows:

```
label: for variable in range generate
       begin
         concurrent statement(s)
       end generate;
```

The label must be used with a *for–generate* statement.

We illustrate the use of the *for–generate* statement by describing the VHDL code for a 6-to-64 decoder implemented using 3-to-8 decoders only. The 3-to-8 decoders are constructed from 2-to-4 decoders. The 2-to-4 decoder is specified first, which is then used as a component for the 3-to-8 decoder, the 3-to-8 decoder in turn is used as a component in specifying the 6-to-64 decoder.

The VHDL code for the 2-to-4 decoder is

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder2_4 is
port (w: in std_logic_vector(1 downto 0);
```

```
  en      : in std_logic;
  y       : out std_logic_vector(3 downto 0));
end decoder2_4;
architecture behavior of decoder2_4 is
begin
process (w,en)
begin
 if en='0' then
 y <= (others => '0');
 elsif w="11" then
  y  <= "1000";
elsif w="10" then
  y  <= "0100";
elsif w="01" then
y  <= "0010";
else
y  <= "0001";
end if;
end process;
end behavior;
```

The decoder is then used as a component in the code for the 3-to-8 decoder:

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder3_8 is
port (w: in std_logic_vector(2 downto 0);
      en: in std_logic;
       y: out std_logic_vector(7 downto 0));
end decoder3_8;

architecture structure of decoder3_8 is
signal en1,en2: std_logic;
 component decoder2_4
  port (w: in std_logic_vector(1 downto 0);
        en: in std_logic;
         y: out std_logic_vector(3 downto 0));
        end component;
begin
en1 <= not w(2) and en;
en2 <= w(2) and en;
decoder1: decoder2_4 port map (w(1 downto 0), en1, y (3 downto 0));
decoder2: decoder2_4 port map (w(1 downto 0), en2, y (7 downto 4));
end structure;
```

The circuit structure of the 3-to-8 decoder as derived from the VHDL description by the Quartus software is shown in Figure 6.12.

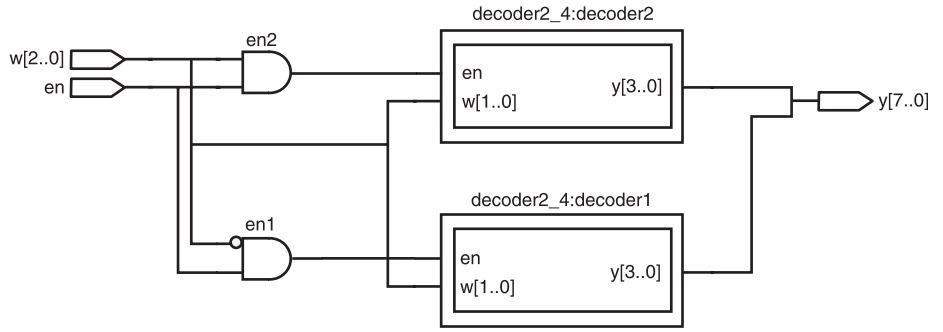


FIGURE 6.12 A 3-to-8 decoder generated from the VHDL code.

Next, the 3-to-8 decoder is used as a component to specify the VHDL code of the 6-to-64 decoder:

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder6_64 is
port (w: in std_logic_vector(5 downto 0);
 en    : in std_logic;
 y     : out std_logic_vector(63 downto 0));
end decoder6_64;
architecture structure of decoder6_64 is
signal m: std_logic_vector(7 downto 0);
component decoder3_8
port (w: in std_logic_vector(2 downto 0);
 en    : in std_logic;
 y     : out std_logic_vector(7 downto 0));
end component;

begin
G1: for i in 0 to 7 generate
dec_right: decoder3_8 port map (w(2 downto 0),m(i),y((8*i+7) downto 8*i));
end generate;
dec_left: decoder3_8 port map (w(5 downto 3), en, m);
end structure;
```

Note that the *for–generate* statement in the above code is in fact a compact representation of the following group of statements, each of which instantiates a 3-to-8 decoder:

```
decoder3_8 port map (w(2 downto 0),m(0),y(7 downto 0));
decoder3_8 port map (w(2 downto 0),m(1),y(15 downto 8));
decoder3_8 port map (w(2 downto 0),m(2),y(23 downto 16));
decoder3_8 port map (w(2 downto 0),m(3),y(31 downto 24));
decoder3_8 port map (w(2 downto 0),m(4),y(39 downto 32));
decoder3_8 port map (w(2 downto 0),m(5),y(47 downto 40));
```

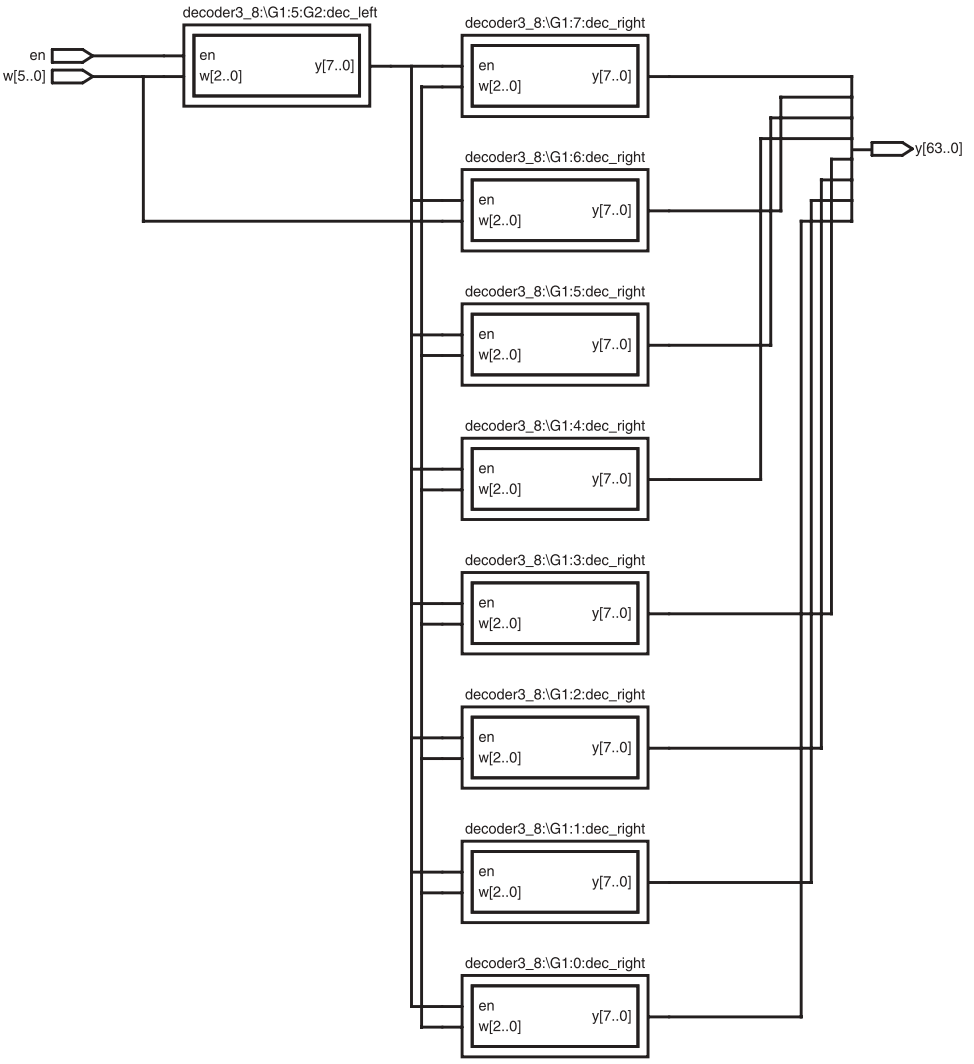


FIGURE 6.13 A 6-to-64 decoder generated from the VHDL code.

```
decoder3_8 port map (w(2 downto 0),m(6),y(55 downto 48));
decoder3_8 port map (w(2 downto 0),m(7),y(63 downto 56));
```

The circuit configuration of the 6-to-64 decoder is produced from the VHDL code by the Quartus software and is shown in Figure 6.13. There are eight decoders on the right side of the diagram created by the *for–generate* statement; the separate port map statement in the code generated the decoder on the left.

EXERCISES

1. Implement a circuit that will compare two 3-bit binary numbers, $X = x_2x_1x_0$ and $Y = y_2y_1y_0$, and generate separate outputs corresponding to the conditions $X = Y$, $X < Y$,

$X > Y$. The circuit is implemented using NAND gates only. Write the VHDL code for the circuit using a *for–generate* statement.

2. Write the VHDL code for an 8-bit two's complement adder/subtractor circuit, which functions as an adder when a select input x is set at logic 0, and as a subtrator when x is set at logic 1.

3. Write the VHDL code for the following function using a *selected signal assignment* statement:

$$f(a, b, c, d) = \sum m(0, 5, 9, 11, 13, 15)$$

4. Write the VHDL code for a BCD to seven-segment decoder circuit using a *selected signal assignment* statement.

5. Write the VHDL code for a 2-bit subtractor.

6. Using the subtractor designed in Exercise 5 as a component, write the VHDL code for a 4-bit subtractor employing a *for–generate* statement.

7. Write the VHDL description for the 5421 code to BCD encoder using a *conditional signal assignment* statement.

8. Repeat Exercise 7 using *if–then–else* statements.

9. The circuit for a digital combinational lock is to be designed. There are four inputs to the clock—A, B, C, and D. The desired combinations that open the lock are:

(i) Inputs A and D are 0, and B and C are 1.
(ii) Inputs A, C, and D are 1, and B is 0.
(iii) Inputs A, B, and C are 1, and D is 0.
(iv) Inputs A, B, C, and D are all 1's.

Write the VHDL code to describe the circuit for the lock.

10. A combinational circuit is to be designed to generate the quotient from the division of any number from 49 to 63 by 7. Write the VHDL code for the circuit.

11. Write the VHDL code to describe a circuit that converts BCD digits to 2-out-of-5 code. (A 2-out-of-5 code is a weighted code in which any 2 bits out of the 5 bits are 1's; the remaining bits are 0's.)

12. A circuit is to be designed to derive the *greatest common divisor* (GCD) of two numbers, each of which is less than or equal to 63. Write the VHDL code to describe the circuit.

13. Write the VHDL code for an 8-bit carry-select adder at the behavioral level.

14. A combinational logic circuit is to be designed to verify the account number a customer enters into an automated teller machine (ATM) with the customer's account number stored in the bank system. It receives two inputs—*compa_en* (a signal that enables the logic) and *account_entered* (a 20-bit vector). The output *ac* of the circuit is asserted if the account number is verified to be correct. Write the VHDL code for the circuit.

7 Synchronous Sequential Circuit Design

7.1 INTRODUCTION

The analysis of a synchronous sequential circuit in Section 4.8 identified the major steps required for synthesizing such circuits; these steps have to be executed in sequence as shown in Figure 7.1. This chapter discusses each step individually and demonstrates that collectively these steps constitute a design procedure for implementing arbitrary synchronous sequential circuits. Henceforth, unless indicated otherwise, a sequential circuit will mean a synchronous sequential circuit.

The purpose of the first step in Figure 7.1 is to provide a precise definition of the intended behavior of the circuit to be designed. This definition should not constrain the means by which the circuit achieves the desired behavior, but it should completely define the external characteristics of the circuit such that it is possible to verify the circuit's behavior from its specification. In other words, the specification should define a black box, whose behavior is known but whose internal construction is unknown.

In the second step of the design process, the specification of the circuit is expressed in terms of the states of the circuit. No formal procedure is available that can be used to derive state diagrams or tables. In fact, this step may be considered as the most difficult part of the design process, and only with experience can a logic designer acquire the skill to describe the state-to-state behavior of a sequential circuit. The third step is generally known as *state minimization* and consists of removing the equivalent states (if there are any) from the state table derived in the second step. This results in a state table with fewer states, which often leads to the simplification of the logic needed to realize the state table.

In the fourth step a unique binary code is assigned to each state; this is known as *state assignment*. The problem is to assign codes to different states such that more economic logic realization than that obtained by arbitrary assignment can be achieved. In the fifth step, the type of flip-flops to be used is decided and the Boolean logic expressions (known as *excitation* or *next state expressions*) are derived for the flip-flops from the transition table. This table is also used to derive the output logic expressions. Finally, the logic diagram of the sequential circuit is drawn using the chosen flip-flops and the logic expressions derived in the fifth step. In the following sections we shall examine each of these steps in detail.

Principles of Modern Digital Design, by Parag K. Lala

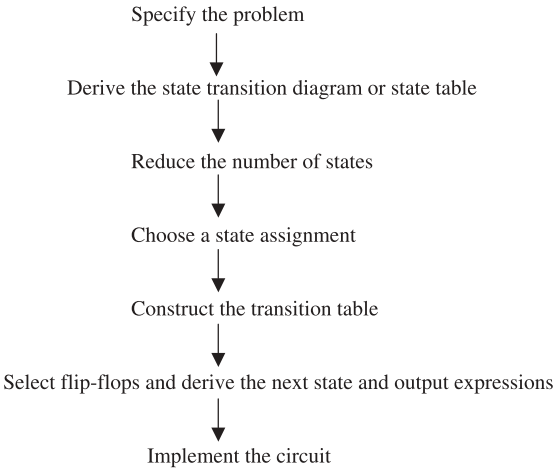


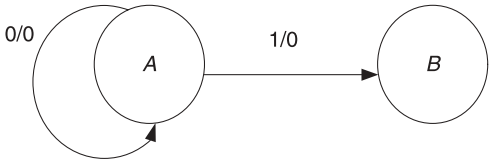
FIGURE 7.1 Design procedure for sequential circuits.

7.2 PROBLEM SPECIFICATION

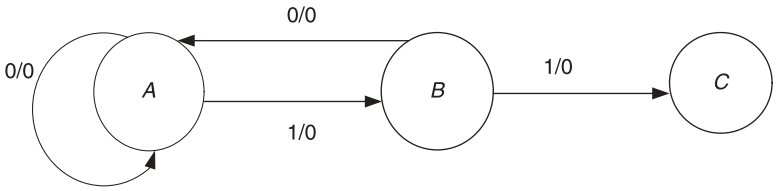
As mentioned before, the development of state diagrams/state tables from the original specification of a circuit is mainly an intuitive process and is heavily dependent on past experience. Either a Moore model or a Mealy model can be used to represent a sequential circuit; however, in practice the Mealy model is often preferred because it is more general.

As an example, let us derive the state diagram for a synchronous sequential circuit required to recognize the 4-bit sequence 1101 and to produce an output 1 whenever the sequence occurs in a continuous serial input. For example, if the input sequence is 010110110101, the output sequence is 000000100100. We assume an initial state *A*, where the circuit waits to receive the first input symbol.

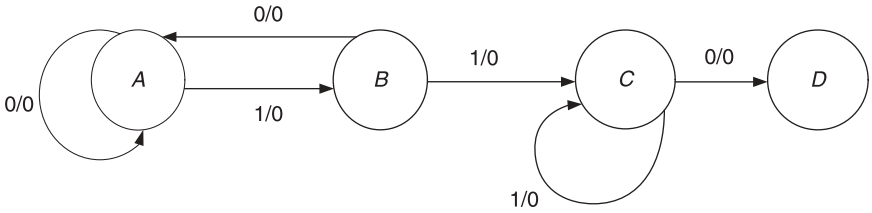
At this state the circuit can receive either a 1 or 0. There is no change in state if 0 is received (indicated by a self-loop). If a 1 is applied, the circuit goes to a new state, *B*, with an output 0:



If a 1 is received when the circuit is in state *B* (i.e., the sequence 11), the circuit changes to state *C*; on the other hand, a 0 input takes the circuit back to state *A*:



When in state *C*, if a 1 is received the circuit remains in the same state. The circuit moves to a new state, *D*, if a 0 is applied (i.e., the sequence is 110).



The next input symbol will be the fourth bit of the 4-bit sequence; therefore the circuit must decide whether or not the sequence is the one to be recognized. If a 1 is applied, the sequence is correct and the circuit changes to *B*, giving the required output. However, if a 0 is received when the circuit is in state *D*, it returns to state *A* to await the start of another sequence. This completes the derivation of the state diagram for the sequential circuit. Figure 7.2 shows the state diagram.

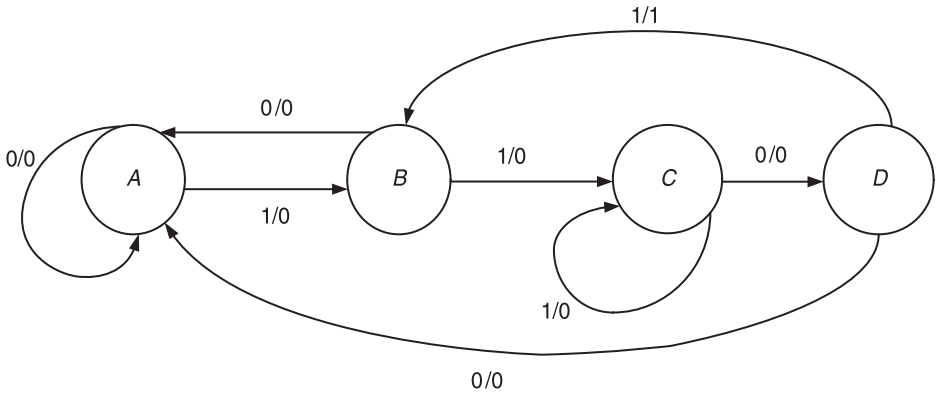


FIGURE 7.2 State diagram for 1101 sequence detector.

It is now possible to construct a state table with the aid of this state diagram. Table 7.1 shows the resulting state table.

We next derive the state diagram and the state table for a sequential circuit that has a single input x and a single output z. It examines incoming serial data in consecutive sequences of 4 bits. The output of the circuit is 1 if and only if an input sequence is a 2-out-of-4 code word (i.e., there are exactly two l's in a 4-bit sequence).

TABLE 7.1 State Table for the 1101 Sequence Detector

	Input	
Present State	$x = 0$	$x = 1$
A	*A*,0	*B*,0
B	*A*,0	*C*,0
C	*D*,0	*C*,0
D	*A*,0	*B*,1
	Next state, Output	

Assume the initial state is *A*. Since the circuit has only a single serial input, each state in the state diagram will have two transition edges, one corresponding to input 0 and the other corresponding to input 1. Besides, an input sequence consists of 4 bits, so we must go back to the initial state after 4 bits have been examined. Figure 7.3 shows the complete state diagram for the desired circuit. The information in the diagram is transferred to the state table (Table 7.2).

As can be seen from Figure 7.3, each combination of 4 bits has been taken into consideration while deriving the state diagram. This has produced quite a few *redundant states* in the state diagram/state table. A state is redundant if its function can be served by another state in the circuit. In the following section, various techniques available for determining redundant states are considered.

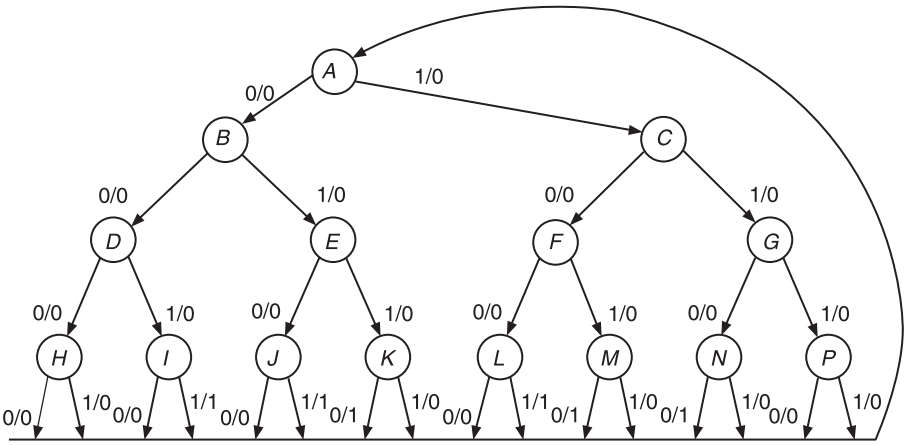


FIGURE 7.3 State diagram for a 2-out-of-4 code detector.

TABLE 7.2 State Table for 2-out-of-4 Detector

	Input	
Present State	$x = 0$	$x = 1$
A	*B*,0	*C*,0
B	*D*,0	*E*,0
C	*F*,0	*G*,0
D	*H*,0	*I*,0
E	*J*,0	*K*,0
F	*L*,0	*M*,0
G	*N*,0	*P*,0
H	*A*,0	*A*,0
I	*A*,0	*A*,1
J	*A*,0	*A*,1
K	*A*,1	*A*,0
L	*A*,0	*A*,1
M	*A*,1	*A*,0
N	*A*,1	*A*,0
P	*A*,0	*A*,0

7.3 STATE MINIMIZATION

The number of states in a sequential circuit has a significant impact on its physical implementation. It is therefore desirable to know when two or more states play identical roles (i.e., are *equivalent* in all respects). *State minimization* eliminates the equivalent states from the state transition graph of a sequential circuit and transforms it into another one with no redundant states; the function of the original sequential circuit remains unchanged. This process corresponds to the minimization of logic functions in combinational circuit design.

Let us first consider an intuitive approach for state minimization. Table 7.3 shows the state table of an arbitrary sequential circuit. It can be seen from the table that present states B and D both have the same next states, A (when $x = 0$) and E (when $x = 1$). They also produce the same outputs 0 (when $x = 0$) and 1 (when $x = 1$). It can be reasoned that if the next states are the same, the outputs produced to any subsequent inputs will also be the same. Thus one of the states, B or D, can be removed from the state table. For example, if we remove row D from Table 7.3 and replace all D's by B's in the columns, the state table is modified as in Table 7.4. It is apparent from Table 7.4 that states A and E are equivalent. Replacing E's by A's results in the reduced table shown in Table 7.5. The removal of the equivalent states has reduced the number of states in the circuit from five to three. Note that in the original state table (Table 7.3) states A and E are not equivalent, because the next states for A and E when $x = 1$ were different. Thus two states are equivalent even if their next states are not the same provided the next states are equivalent. Two states are defined as *equivalent* if and only if for every input sequence the circuit produces the same output sequence irrespective of which one of the two states is the starting state, and their next states are also equivalent. State equivalence is a mathematical equivalence relationship. Thus if states A and B are equivalent and states B and C are equivalent, then A is equivalent to C; the three states form a set of equivalent states. If no two states in a circuit are equivalent, then the circuit is reduced.

7.3.1 Partitioning Approach

The equivalent sets of states in a sequential circuit can be determined by using a procedure based on partitioning. The first step is to partition the set of states of a circuit into a number of blocks so that all states in a block have identical output for each possible input. Let us consider, for example, Table 7.6. The output produced for each of the states A, C, and E is

TABLE 7.3 A State Table

	Input	
Present State	$x = 0$	$x = 1$
A	A,0	B,0
B	A,0	E,1
C	D,1	C,1
D	A,0	E,1
E	A,0	D,0

TABLE 7.4 State *D* Removed

	Input	
Present State	$x = 0$	$x = 1$
A	*A*,0	*B*,0
B	*A*,0	*E*,1
C	*B*,1	*C*,1
E	*A*,0	*B*,0

TABLE 7.5 State *E* Removed

	Input	
Present State	$x = 0$	$x = 1$
A	*A*,0	*B*,0
B	*A*,0	*A*,1
C	*B*,1	*C*,1

0 for both $x = 0$ and $x = 1$. The outputs associated with the inputs 0 and 1 are, respectively, 1 and 0 for each of the states *B*, *D*, and *F*. Hence the first partition P_1 for the circuit is $P_1 = (ACE)(BDF)$.

The next step of the procedure is to derive a partition P_2 by placing two states in the same block if for each input value their next states lie in a common block of P_1. In the example of Table 7.6 the next states for *A*, *C*, and *E* (i.e., states in the first block of P_1) corresponding to $x = 0$ are *B*, *A*, and *B*, respectively. Since *A* and *B* are in different blocks of P_1, partition P_2 must separate *C* from *A* and *E*. For $x = 1$ the next states *A*, *C*, and *E* lie in the same block. In the second block of P_1, the next states for *B*, *D*, and *F* with $x = 0$ belong to the same block of P_1. However, for $x = 1$ the next state of *F* lies in a different block of P_1 than the next states of *B* and *D*. Hence the block (*BDF*) is split into blocks (*F*)(*BD*). Thus partition P_2 is

$$P_2 = (C)(AE)(F)(BD)$$

TABLE 7.6 A State Table

	Input	
Present State	$x = 0$	$x = 1$
A	*B*,0	*A*,0
B	*D*,1	*D*,0
C	*A*,0	*C*,0
D	*B*,1	*F*,0
E	*B*,0	*E*,0
F	*D*,1	*E*,0
	Next state, Output	

TABLE 7.7 Minimized State Table

	Input	
Present State	$x = 0$	$x = 1$
A	B,0	A,0
B	D,1	D,0
C	A,0	C,0
D	B,1	F,0
F	D,1	A,0
	Next state, Output	

Partition P_3 can be formed in a similar manner. The next states for A and E lie in the same blocks of P_2 for both $x = 0$ and $x = 1$, so block (AE) cannot be separated. However, the next states for B and D with $x = 1$ lie in different blocks of P_2, so block (BD) must be split into blocks $(B)(D)$. Therefore

$$P_3 = (C)(AE)(F)(B)(D)$$

The next partition, P_4, is derived from P_3 in the same way and is given by

$$P_4 = (C)(AE)(F)(B)(D)$$

Since P_3 and P_4 are identical, all subsequent partitions $P_5, P_7, \ldots$ will also be identical to P_3. Therefore if a partition P_{k+1} is identical to its predecessor partition P_k, the partitioning process is terminated, and partition P_k is said to be an *equivalence partition*. All states belonging to a block in the equivalence partition are equivalent. For the example under consideration, P_3 is the equivalence partition and states A and E are equivalent. The original state table (Table 7.6) can be reduced by eliminating row E and replacing each E by an A (Table 7.7).

As a second example let us consider the state table of a circuit shown in Table 7.8. Since outputs for states A, F, and G are 0 and 1 for $x = 0$ and $x = 1$, respectively, these states are grouped in one block of the first partition P_1. The outputs are 1 irrespective of the input

TABLE 7.8 A State Table

	Input	
Present State	$x = 0$	$x = 1$
A	F,0	D,1
B	C,1	F,1
C	F,1	B,1
D	E,1	G,1
E	A,1	D,1
F	G,0	B,1
G	A,0	D,1
	Next state, Output	

value when the states are B, C, D, and E; therefore they are included in the other block of P_1. Thus the first partition P_1 is

$$P_1 = (AFG)(BCDE)$$

The next states for A, F, and G belong to the same block in P_1 for both inputs; therefore A, F, and G cannot be split. On the other hand, the next states for B, C, D, and E belong to different blocks; hence they need to be separated. The next states for B and D belong to identical blocks for both inputs; this is also true for C and E. Thus partition P_2 is

$$P_2 = (AFG)(BD)(CE)$$

The third partition P_3 is derived in a similar manner and is identical to P_2:

$$P_3 = (AFG)(BD)(CE)$$

Thus no further partitioning is possible and P_3 is the equivalence partition. Assuming $(AFG) = X$, $(BD) = Y$, and $(CE) = Z$, the minimized state table is as shown in Table 7.9.

7.3.2 Implication Table

A more formal method for finding equivalent states in a sequential circuit is based on deriving an *implication table* that shows the necessary conditions or *implications* that exist between all possible equivalent pairs of states. We shall consider the state table of Table 7.6 to explain the method.

The first step is to form a table with the rows consisting of all but the first state and the columns consisting of all states except the last. The resulting table has as many cells as there are permissible state pairs. Figure 7.4*a* shows the implication table for our example. Next, we consider whether a state pair in the implication table is equivalent or not; a state pair cannot possibly be equivalent if the states have different outputs. A cross (X) is placed in a cell of the implication table if the corresponding state pair has differing outputs (Fig. 7.4*b*). The nonequivalent state pairs are called *incompatibles*. The vacant cells must now be completed.

Each vacant cell is filled with the required state pairs whose equivalence implies the equivalence of the state pair that defines the vacant cell. For example, consider the cell corresponding to the state pair AC. We enter into AC the state pair AB, which must be equivalent in order for A and C to be equivalent (Fig. 7.4*c*). A check ($\surd$) is inserted in a cell if the corresponding state pair is equivalent. For example, in Figure 7.4*c* the cell defined by the state pair AE has a check, indicating that the states A and E are equivalent. When the table is completed it is examined column by column, starting from the extreme right-hand column, to determine whether any other cells should be crossed out.

TABLE 7.9 Minimized State Table

	Input	
Present State	$x = 0$	$x = 1$
X	X,0	Y,1
Y	Z,1	X,1
Z	X,1	Y,1
	Next state, Output	

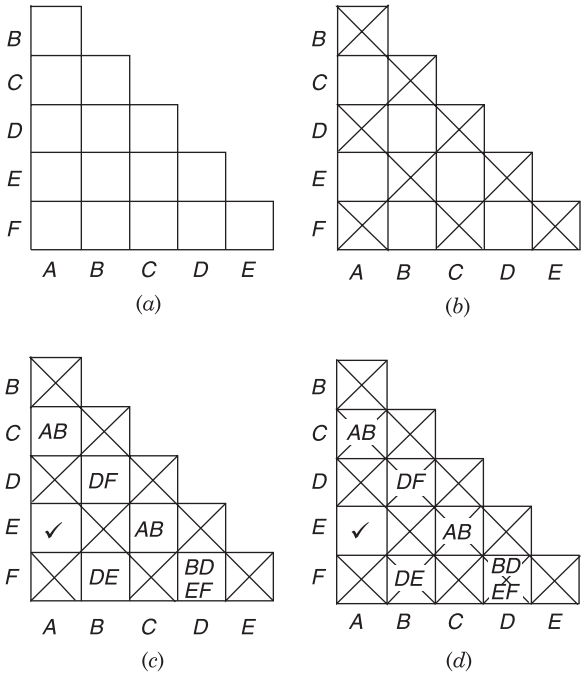


FIGURE 7.4 (*a*) Implication table, (*b*) identification of state pairs with different outputs, (*c*) possible equivalent state pairs, and (*d*) equivalent state pairs.

In Figure 7.4*c* the first cell to be considered is the one defined by D and F; it contains the pair BD and EF. Since the cell defined by E and F was already crossed out, it follows that any state pair whose equivalence is implied by the equivalence of E and F must also be crossed out. Hence the cell corresponding to D and F is crossed out (Fig. 7.4*d*). The procedure is repeated until no further cells can be crossed out. The state pairs corresponding to the cells that have not been crossed out are the equivalent states. The only equivalent state pair in Figure 7.4*d* is AE. Thus the equivalence partition P is

$$P = (AE)(B)(C)(D)(F)$$

Note that this equivalence partition is identical to the one derived earlier by partitioning.

As another example, let us consider the application of the implication table in deriving the equivalence partition for the state table shown in Figure 7.5*a*. The corresponding implication table is shown in Figure 7.5*b*. As can be seen from Figure 7.5*b*, the equivalence partition is

$$P = (CD)(EF)(EG)(FG)(A)(B)(H)$$

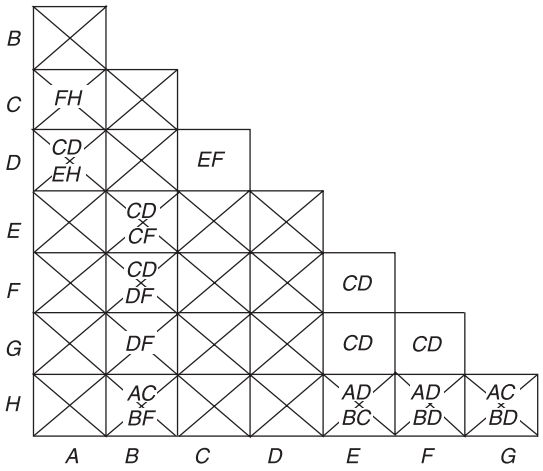
By using the transitivity relationship, the state pairs (EF), (EG), and (FG) can be grouped into a set of states (EFG). Thus

$$P = (CD)(EFG)(A)(B)(H)$$

Assigning $(CD) = \alpha$, $(EFG) = \beta$, $(A) = \gamma$, $(B) = \delta$, and $(H) = \omega$, the reduced state table can be derived as shown in Figure 7.5*c*.

Present State	Input $x = 0$	Input $x = 1$
A	D,0	H,1
B	F,1	C,1
C	D,0	F,1
D	C,0	E,1
E	C,1	D,1
F	D,1	D,1
G	D,1	C,1
H	B,1	A,1
	Next state, output	

(a)



(b)

Present State	Input $x = 0$	Input $x = 1$
α	α,0	β,1
β	α,1	α,1
γ	α,0	ω,1
δ	β,1	α,1
ω	δ,1	γ,1
	Next state, output	

(c)

FIGURE 7.5 (a) A state table, (b) implication table, and (c) minimized state table.

7.4 MINIMIZATION OF INCOMPLETELY SPECIFIED SEQUENTIAL CIRCUITS

State tables of completely specified sequential circuits do not contain don't cares. In other words, all next state and/or output entries are specified. However, in practice it is very likely that some input combinations will not be applied to a sequential circuit, so next

Present State	Input $x = 0$	Input $x = 1$
A	A,1	C,0
B	B,–	A,0
C	C,1	–,0
	Next state, output	

(a)

Present State	Input $x = 0$	Input $x = 1$
A	A,1	C,0
B	B,0	A,0
C	C,1	–,0
	Next state, output	

(b)

Present State	Input $x = 0$	Input $x = 1$
A	A,1	C,0
B	B,1	A,0
C	C,1	–,0
	Next state, output	

(c)

FIGURE 7.6 (a) An incompletely specified state table, (b) don't care output replaced with 0, and (c) don't care output replaced with 1.

states and outputs corresponding to these inputs are of no consequence. *A* sequential circuit is *incompletely specified* if the next state and/or the output of the circuit for an input and a current state are not specified; hence the corresponding entries in the state table of the circuit are don't cares. *A* don't care entry in an incompletely specified state table is usually denoted by a dash (–). As in completely specified sequential circuits, reduction of states in an incompletely specified state table is needed for efficient implementation of the corresponding sequential circuit.

One approach to reducing the number of states in an incompletely specified sequential circuit will be to specify entries for the unspecified next states and outputs such that some of the states become equivalent. This is illustrated using the state table of Figure 7.6*a*.

In the first column the don't care output can be set either to a 0 or to a 1. Thus two separate incompletely specified state tables result if the don't care output is replaced with a fixed value; these state tables are shown in Figure 7.6*b* and 7.6*c*.

The don't care state in the second column of both state tables in Figure 7.6*b* and 7.6*c* can be one of the three states in the circuit—that is, A, B, or C. Thus three fully specified state tables can be generated from each of the state tables of Figure 7.6*b* and 7.6*c*, respectively. The resulting tables are shown in Figure 7.7.

Using the partitioning approach discussed in the previous section, the first partition P_1 for each of the state tables of Figure 7.7 is

$$
\begin{aligned}
s_1 \quad & P_1 = (AC)(B) \\
s_2 \quad & P_1 = (AB)(C) \\
s_3 \quad & P_1 = (AC)(B) \\
s_4 \quad & P_1 = (B)(AC) \\
s_5 \quad & P_1 = (B)(AC) \\
s_6 \quad & P_1 = (A)(BC)
\end{aligned}
$$

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,0	A,0
C	C,1	A,0

(*a*) s_1

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,1	A,0
C	C,1	B,0

(*b*) s_2

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,0	A,0
C	C,1	C,0

(*c*) s_3

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,1	A,0
C	C,1	A,0

(*d*) s_4

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,0	A,0
C	C,1	B,0

(*e*) s_5

Present State	Input $x=0$	Input $x=1$
A	A,1	C,0
B	B,1	A,0
C	C,1	C,0

(*f*) s_6

FIGURE 7.7 Completely specified state tables derived from Figure 7.6*a*.

The second partition P_2 derived from P_1 shown below indicates that only in state table s_1, s_3 and s_4 states are A and C equivalent. Thus in s_1, s_3, and s_4 the number of states can be reduced from three to two.

$$
\begin{aligned}
s_1 \quad & P_2 = (AC)(B) \\
s_2 \quad & P_1 = (A)(B)(C) \\
s_3 \quad & P_1 = (AC)(B) \\
s_4 \quad & P_1 = (AC)(B) \\
s_5 \quad & P_1 = (A)(B)(C) \\
s_6 \quad & P_1 = (A)(B)(C)
\end{aligned}
$$

It should be clear from the above example that considerable trial and error is required to determine the best way to fill in the unspecified entries. This will be unfeasible if the number of don't cares is large.

We shall consider a systematic procedure in this section to minimize the number of states in an incompletely specified state table [1]. The procedure is based on the concept of compatibility. Two states S_i and S_j are said to be *compatible* if in response to an *applicable input sequence*, the same output sequence is produced irrespective of whether S_i or S_j is the initial state; if S_i and S_j are not compatible, then they are said to be *incompatible*. An applicable input sequence always leads to a specified next state for each bit of the sequence. For example, in the incompletely specified table (Table 7.10),

TABLE 7.10 An Incompletely Specified State Table

	Input	
Present State	$x = 0$	$x = 1$
A	A,–	—
B	C,1	B,0
C	D,0	–,1
D	—	B,–
E	A,0	C,1
	Next state, Output	

the output sequences produced in response to the input sequence 1100 for starting states B and D are the same when both are specified:

Input	1	1	0	0	
State	$B \rightarrow$	$B \rightarrow$	$B \rightarrow$	$C \rightarrow$	D
Output	0	0	1	0	
State	$D \rightarrow$	$B \rightarrow$	$B \rightarrow$	$C \rightarrow$	D
Output	–	0	1	0	

A set of states for which every pair of states is compatible is called a *compatibility class*. A maximal compatible is a compatibility class that is not a subset of any other compatibility class. For Table 7.10, {AD} is not a maximal compatible because it is a proper subset of {ABD}, whereas {ABD} is a maximal compatible. The implication table (discussed in Section 7.3) can be used to determine all pairs of compatible states in an incompletely specified state table. Figure 7.8 shows the implication table for the state table shown in Table 7.10. From the implication table the compatible pairs are

$$(AB)(AC)(AD)(AE)(BD)(CD)(CE)$$

The maximal compatibles are found by combining these state pairs into larger groups using the following procedure [1]:

Step 1. List the pairwise compatibles, if any, for the rightmost column of the implication table. If the rightmost column does not have a compatible pair, then move to the next

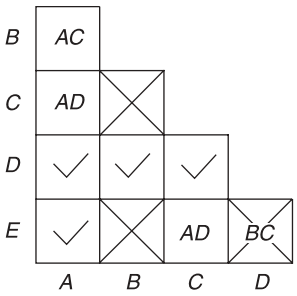


FIGURE 7.8 Implication table.

column on the left and check whether it has a compatible pair. Continue this process until a column containing a compatible pair is found.

Step 2. Proceed to the next column on the left. If the state to which this column corresponds is compatible with all members of a previously determined compatible class, then add this state to the class, thereby forming a larger class. If the state to which this column corresponds is compatible only with a subset of the compatible class, then form a new class consisting of the subset and this state. Finally, list all the compatible pairs that are not included in an already formed compatible class.

Step 3. Repeat step 2 until all columns in the implication table have been considered. The compatibility classes remaining are the set of maximal compatibles.

Applying this procedure to the implication table of Figure 7.8 yields the following sequence of compatibility classes:

Column D	( )
Column C	$(CD)(CE)$
Column B	$(CD)(CE)(BD)$
Column D	$(CD)(CE)(BD)(AB)(AC)(AD)(AE)$

Thus there are three maximal compatibles— $C_1' = \{ABD\}$, $C_2' = \{ACD\}$, and $C_3' = \{\text{ACE}\}$. Note that the sets of maximal compatibles are similar to the blocks of an equivalence partition of a completely specified state table. However, the blocks of an equivalence partition are disjoint, whereas the maximal compatibles are not necessarily so, because they can have common states. In order to determine the reduced table for an incompletely specified machine we must select a set of maximal compatibles that satisfy the *covering* and *closure* conditions:

1. A set of maximal compatibles *covers* an incompletely specified sequential circuit if each state of the circuit is contained in at least one of the maximal compatibles.
2. A set of maximal compatibles is *closed* if for every compatible contained in the set, the next states corresponding to the states in the compatible for all possible input combinations are also contained in a maximal compatible of the set.

The maximal compatibles derived here cover all the states of Table 7.10. Maximum compatible C_1' covers states A, B, and D, and C_3' covers states A, C, and E. The resulting set $\{C_1'\ C_3'\}$ covers all the states and also satisfies the closure conditions as shown in Table 7.11. The incompletely specified state table can therefore be reduced to a table with two states corresponding to C_1' and C_3'. Denoting C_1' and C_3' by α and β, respectively, the reduced state table is shown in Table 7.12.

TABLE 7.11 Verification of the Closure Condition

	Input	
Present State	$x = 0$	$x = 1$
$C_1' = \{ABD\}$	C_3',1	C_1',0
$C_3' = \{ACE\}$	C_1',0	C_3',1
	Next state, Output	

TABLE 7.12 Minimized State Table

	Input	
Present State	$x = 0$	$x = 1$
α	β,1	α, 0
β	α,0	β,1
	Next state, Output	

7.5 DERIVATION OF FLIP-FLOP NEXT STATE EXPRESSIONS

Once the reduced state table has been obtained, the next step in the design process is to encode the states in binary form. This is known as *state assignment*. A state assignment must allocate a unique binary combination to each state. In order to obtain a distinct binary combination for each state of an n-state circuit, we need s secondary input variables such that

$$s = \lceil \log_2 n \rceil \quad \text{(i.e., } s \geq \log_2 n\text{)}$$

Each secondary variable is generated by a flip-flop. Thus the number of flip-flops required to implement an n-state sequential circuit is $\lceil \log_2 n \rceil$.

The flip-flops in a sequential circuit are *excited* to take on the various states in proper sequence as required by the state table of the circuit. Suppose the current content (the *present state*) of a D flip-flop is 0. To change the content of the flip-flop from 0 to 1, its input must be set to 1. In other words, the D flip-flop must be excited to 1 in order to make a transition from the present state 0 to the *next state* 1. The present-state to next-state transition input requirement of any flip-flop can be derived from its excitation table (Table 7.13). For example, if the present state of a JK flip-flop is 0 and it has to be changed to 1, then the J input should be set to 1, while the K input can be either 0 or 1, that is, a don't care (–), because it does not affect the next state of the flip-flop. As we shall see shortly, the information contained in the excitation table is necessary to obtain the next state expression for each flip-flop used in a circuit.

To illustrate, let us derive the next state expressions for the sequence detector circuit of Section 7.2; the state table for the circuit is repeated in Table 7.14. First, we select the state assignment. The manner in which the binary combinations are assigned to the states of a circuit has a considerable impact on the complexity of the combinational logic necessary to implement the next state expressions. We will consider some general rules for finding

TABLE 7.13 Excitation Table for Flip-Flops

Present State	Next State	D Flip-Flop	JK Flip-Flop		T Flip-Flop
Q_t	Q_{t+1}	D	J	K	T
0	0	0	0	–	0
0	1	1	1	–	1
1	0	0	–	1	1
1	1	1	–	0	0

TABLE 7.14 Reduced State Table

	Input	
Present State	$x = 0$	$x = 1$
A	A,0	B,0
B	A,0	C,0
C	D,0	C,0
D	A,0	B,1
	Next state, Output	

reasonably good state assignments. The state assignment for the sequence detector circuit is arbitrarily chosen as follows:

$$A = 00$$
$$B = 01$$
$$C = 10$$
$$D = 11$$

Since there are four states A, B, C, and D, a minimum of two state variables is required to represent them; consequently, two flip-flops are needed.

We may choose any type of flip-flop to implement the memory portion of a sequential circuit. Let us use D flip-flops for this example. Before we can derive the excitation equations for the D flip-flops, the *transition table* corresponding to the state assignment has to be derived from the state table. The entries in the transition table (Table 7.15) represent the next states of the D flip-flops for each combination of present state and input value. Since the next state value of a D flip-flop is the same as the excitation input, the transition table entries in effect specify the required excitation of the D flip-flops. Karnaugh maps can now be plotted for each of the flip-flop excitation inputs. These are shown in Figure 7.9*a* and 7.9*b*; the positions of rows 10 and 11 are swapped in these maps in order to satisfy the requirements of a Karnaugh map. The Karnaugh map for output Z is shown in Figure 7.9*c*. Hence the next state and the output expressions needed to implement the sequence detector circuit are as follows:

$$D_1 = y_1\bar{y}_2 + x\bar{y}_1y_2$$
$$D_2 = \bar{x}y_1\bar{y}_2 + xy_1y_2 + x\bar{y}_1\bar{y}_2$$
$$Z = xy_1y_2$$

The logic diagram for the complete design is shown in Figure 7.10.

TABLE 7.15 Transition Table Derived from the Reduced State Table of Table 7.14

Present State	Input	
y_1y_2	$x = 0$	$x = 1$
$A \rightarrow 00$	00,0	01,0
$B \rightarrow 01$	00,0	10,0
$C \rightarrow 10$	11,0	10,0
$D \rightarrow 11$	00,0	01,1

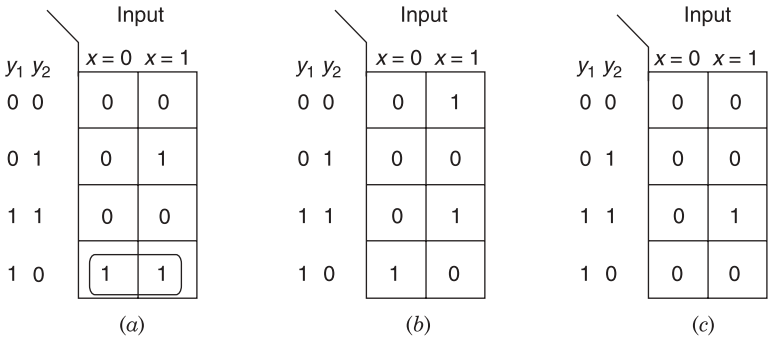


FIGURE 7.9 (*a*) Karnaugh map for D_1, (*b*) Karnaugh map for D_2, and (*c*) Karnaugh map for output Z.

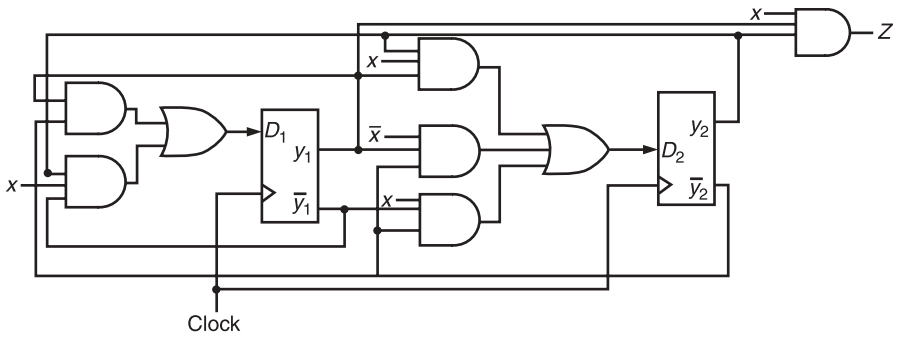


FIGURE 7.10 Logic diagram of the sequence detector circuit.

As another example of the derivation of next state and output expressions for sequential circuits, let us consider the sequential circuit specified in Table 7.16. We will use *JK* flip-flops to realize the circuit. By choosing the state assignment

$$A = 00$$
$$B = 01$$
$$C = 10$$
$$D = 11$$

the transition table shown in Table 7.17 is obtained.

TABLE 7.16 A State Table

	Input x_1 x_2			
Present State	00	01	l0	11
A	*C*,0	*D*,0	*A*,1	*D*,0
B	*C*,1	*D*,0	*A*,1	*D*,1
C	*A*,0	*B*,0	*A*,1	*B*,0
D	*A*,1	*B*,0	*A*,1	*B*,1
	Next state, Output			

TABLE 7.17 Transition Table

	Input $x_1\ x_2$			
Present State	00	01	l0	11
00	10,0	11,0	00,1	11,0
01	10,1	11,0	00,1	11,1
10	00,0	01,0	00,1	01,0
11	00,1	01,0	00,1	01,1

Next, we derive the next state expressions for the two flip-flops from the Karnaugh maps of their J and K inputs (Fig. 7.11); the Karnaugh maps are formed from the transition table of the circuit by using the excitation table of JK flip-flops. The Karnaugh map for the output function is shown in Figure 7.12.

The next state expressions for the flip-flops and the output expressions for the sequential circuit are derived from the Karnaugh maps of Figure 7.11 and Figure 7.12, respectively:

$$J_1 = \bar{x}_1 + x_2 \quad K_1 = 1$$
$$J_2 = x_2 \quad K_2 = \bar{x}_2$$
$$Z = x_1 y_2 + x_1 \bar{x}_2 + \bar{x}_2 y_2$$

The actual implementation of the circuit is shown in Figure 7.13.

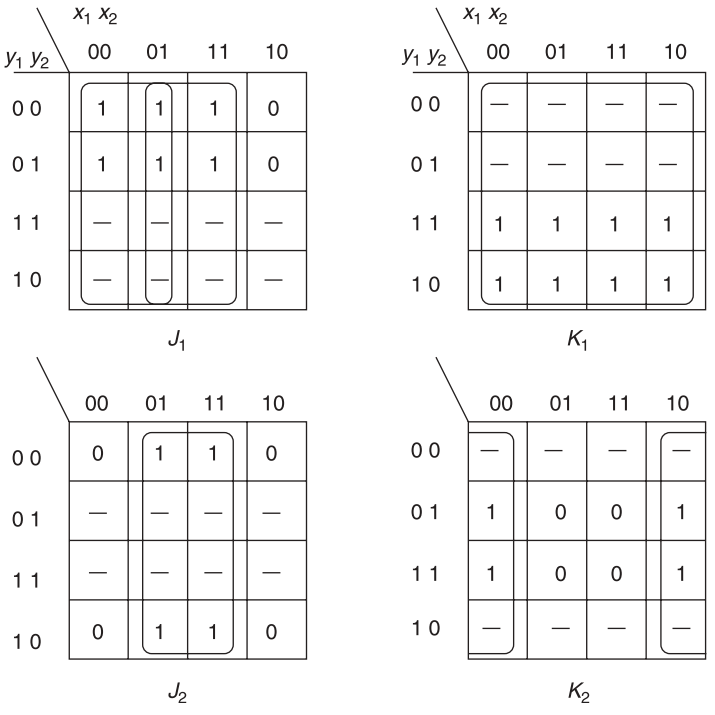


FIGURE 7.11 Karnaugh maps for *JK* flip-flop realization.

$y_1\ y_2$ \ $x_1\ x_2$	00	01	11	10
0 0	0	0	0	1
0 1	1	0	1	1
1 1	1	0	1	1
1 0	0	0	0	1

FIGURE 7.12 Karnaugh map for output *Z*.

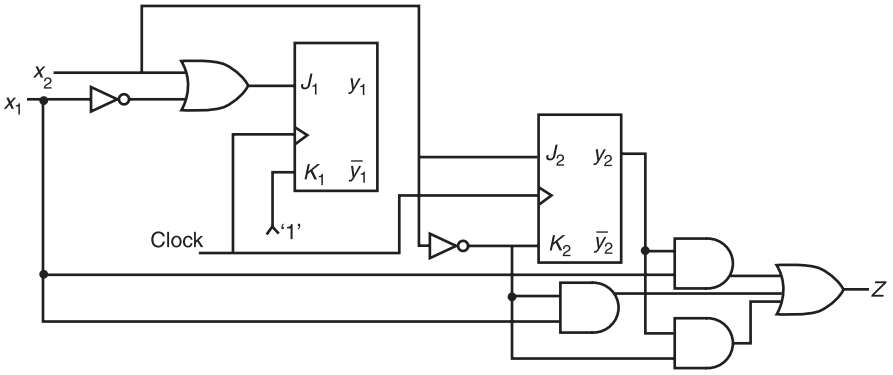


FIGURE 7.13 Realization of the sequential circuit specified in Table 7.16.

The Karnaugh maps for *T* flip-flop realization of this sequential circuit can be derived from Table 7.17; these are shown in Figure 7.14. The output map remains the same as shown in Figure 7.12. The next state and the output expressions are as follows:

$$T_1 = \bar{x}_1 + x_2 + y_1$$
$$T_2 = \bar{x}_2 y_2 + x_2 \bar{y}_2$$
$$Z = x_1 y_2 + x_1 \bar{x}_2 + \bar{x}_2 y_2$$

The corresponding circuit is shown in Figure 7.15.

As we saw in the previous examples, a sequential circuit with n states requires $\lceil \log_2 n \rceil$ flip-flops. However, there are occasions when the number of states in a sequential circuit is fewer than the maximum number that can be specified with $\lceil \log_2 n \rceil$. For example, a sequential circuit with five states requires three flip-flops, but three flip-flops can specify up to eight states, so there are three unused or *invalid* states in the circuit. Normally, when power is turned on, the flip-flops in a sequential circuit can settle in any state, including one of the invalid states. In that case it is necessary to ensure that the circuit goes to a valid or specified state with the fewest number of clock pulses (a circuit changes state only after the application of a clock pulse). Once the circuit goes to a valid state, it can continue to operate as required.

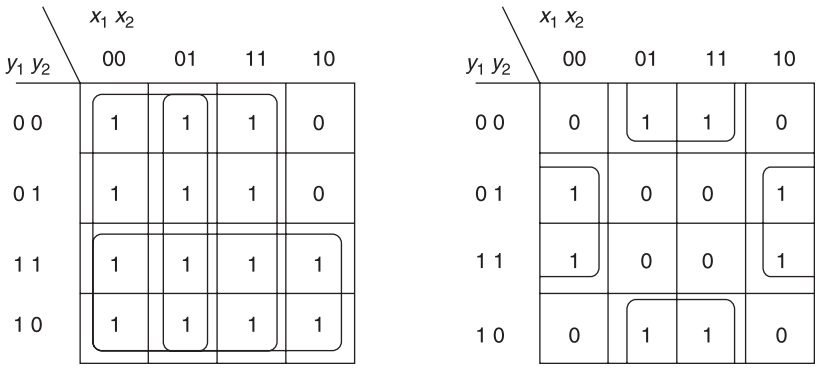


FIGURE 7.14 Karnaugh maps for *T* flip-flop realization.

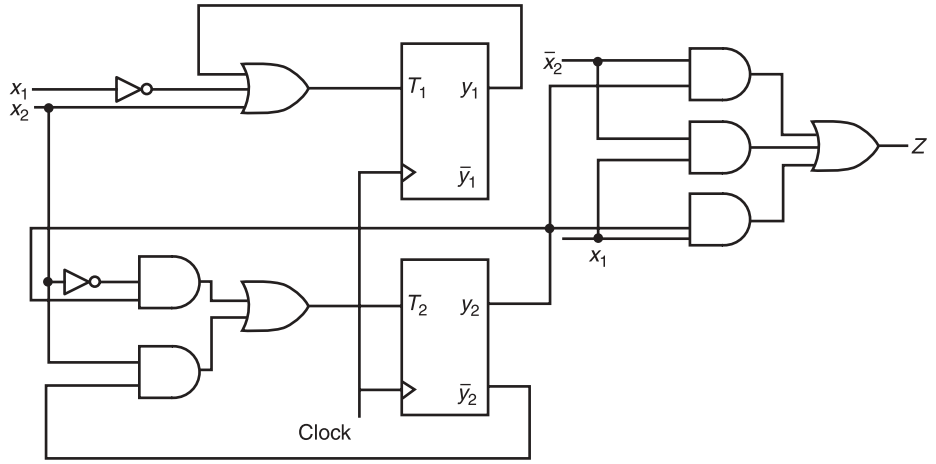


FIGURE 7.15 *T* flip-flop implementation.

To illustrate, let us design the sequential circuit specified by Table 7.18. The circuit has six states, so three flip-flops will be needed to implement the circuit. However, three flip-flops can specify eight states, so there are two invalid states in the circuit. In order to make sure the circuit is transferred to a valid state (i.e., A, B, . . . , F) from an invalid state with

TABLE 7.18 State Table of a Sequential Circuit

	Input	
Present State	$x = 0$	$x = 1$
A	*A*,0	*B*,0
B	*B*,0	*C*,0
C	*C*,0	*D*,0
D	*D*,0	*E*,0
E	*E*,0	*F*,0
F	*F*,0	*A*,1

TABLE 7.19 Augmented State Table

Present State		Input $x = 0$	Input $x = 1$
Valid states	A	A,0	B,0
	B	B,0	C,0
	C	C,0	D,0
	D	D,0	E,0
	E	E,0	F,0
	F	F,0	A,1
Invalid states	G	E,–	B,–
	H	D,–	A,–

one clock pulse, it will be necessary to augment the state table as shown in Table 7.19. The choice of next states for the invalid states G and H are arbitrary in this case; in practice, the next states are selected such that a minimal increase in the circuitry is needed to implement the augmented state table as compared to the original state table. The state assignment for the circuit is arbitrarily chosen as follows:

$$A = 000$$
$$B = 001$$
$$C = 010$$
$$D = 011$$
$$E = 100$$
$$F = 101$$
$$G = 110$$
$$H = 111$$

The resulting transition table is shown in Table 7.20. The Karnaugh maps for a *JK* flip-flop realization of the circuit are shown in Figure 7.16. The output map is

TABLE 7.20 Transition Table

Present State $y_1y_2y_3$	Input $x = 0$	Input $x = 1$
000	000,0	001,0
001	001,0	010,0
010	010,0	011,0
011	011,0	100,0
100	100,0	101,0
101	101,0	000,1
110	100,–	001,–
111	011,–	000,–

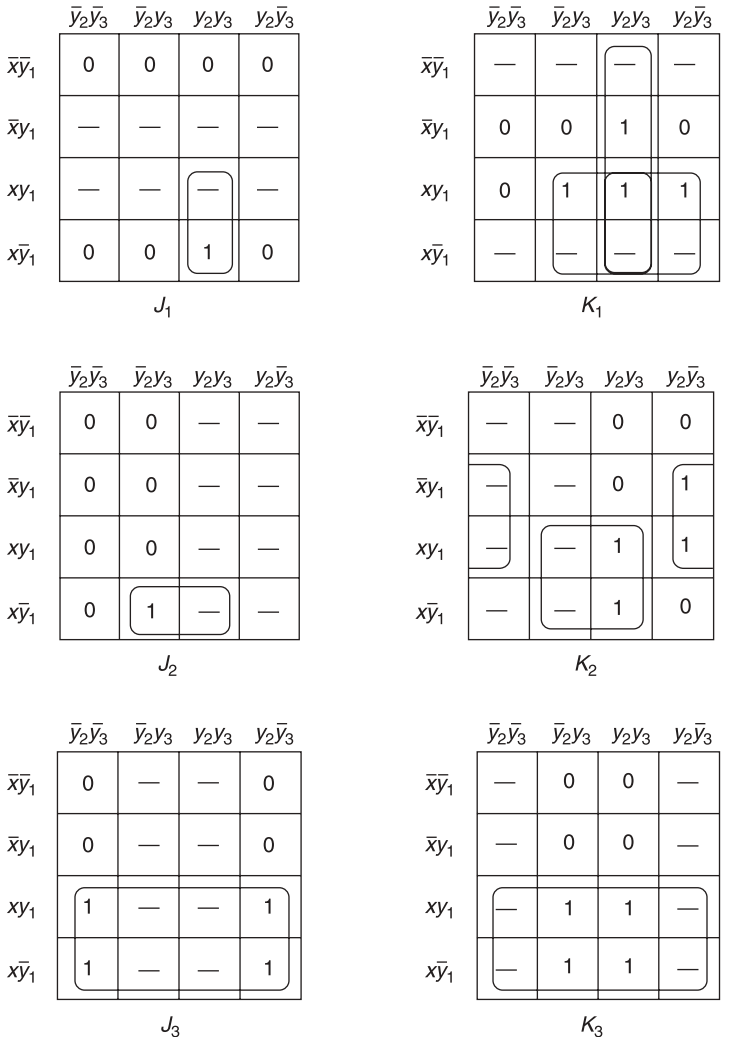


FIGURE 7.16 Karnaugh maps for *JK* flip-flop realization.

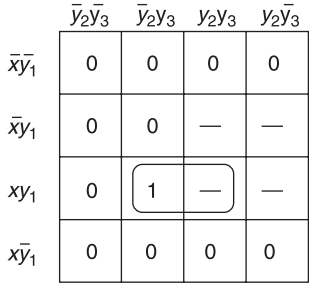


FIGURE 7.17 Karnaugh map for output *Z*.

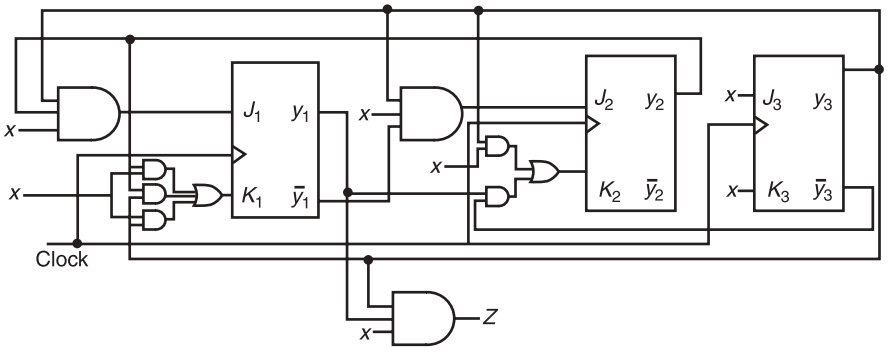


FIGURE 7.18 Implementation of Table 7.18.

shown in Figure 7.17. The next state and output expressions are obtained from Figures 7.16 and 7.17, respectively, and are

$$J_1 = xy_2y_3 \quad K_1 = xy_2 + xy_3 + y_2y_3$$
$$J_2 = x\bar{y}_1y_3 \quad K_2 = xy_3 + y_1\bar{y}_3$$
$$J_3 = x \qquad K_3 = x$$
$$Z = xy_1y_3$$

The circuit implementation is shown in Figure 7.18.

7.6 STATE ASSIGNMENT

So far in all the design problems we have considered, an arbitrary state assignment has been adopted. For example, in the 1101 sequence detector circuit designed in Figure 7.10 the state assignment selected was

$$A = 00, \quad B = 01, \quad C = 10, \quad D = 11$$

However, a different state assignment may be chosen, and this will lead to a different set of design equations. For example, if we choose the assignment

$$A = 00, \quad B = 11, \quad C = 01, \quad D = 10$$

then the design equations can be derived as shown in Figure 7.19 and are given by

$$D_1 = x\bar{y}_2 + \bar{x}\bar{y}_1y_2$$
$$D_2 = x$$
$$Z = xy_1\bar{y}_2$$

Alternatively, the state assignment

$$A = 00, \quad B = 10, \quad C = 11, \quad D = 01$$

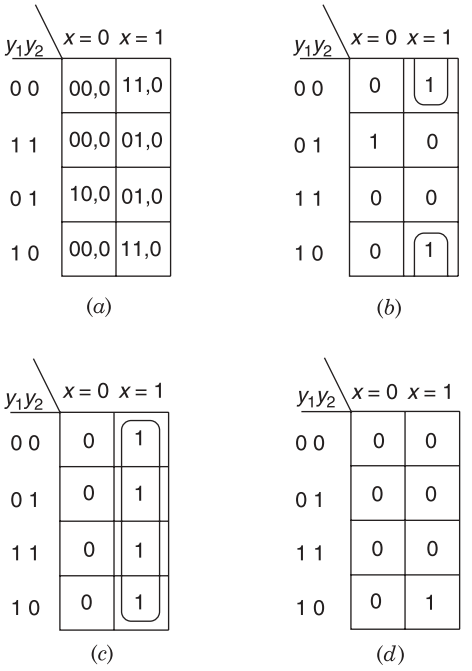


FIGURE 7.19 (*a*) Transition table, (*b*) Karnaugh map for D_1, (*c*) Karnaugh map for D_2, and (*d*) Karnaugh map for Z.

will result in the following next state and output expressions (derived as shown in Fig. 7.20)

$$D_1 = x$$
$$D_2 = y_1y_2 + xy_1$$
$$Z = x\bar{y}_1y_2$$

Either of these assignments will lead to a simpler circuit for the 1101 sequence detector circuit than that obtained by choosing the arbitrary assignment that led to the circuit of Figure 7.10. This can be seen from Table 7.21, which shows a comparison of the number of gates required to implement the circuit for each of the three assignments. In fact, the third assignment turns out to be the best; as we shall see later, this is not just good luck!

As we saw in the preceding example, the criterion for a good state assignment is that it should result in simpler next state and output expressions. The problem associated with state assignment, therefore, is to select the state variables such that the complexity of the combinational logic required to control the memory elements of the sequential circuit is minimized. However, the number of possible state assignment increases very rapidly with the number of states of the sequential circuit. If a circuit has n states, $s = \lceil \log_2 n \rceil$ state variables are needed for an assignment; thus 2^s combinations of state variables are available. The first state of the circuit can be allocated any one of the 2^s combinations, the second state can be allocated any one of the remaining $2^s - 1$ combinations, and so on.

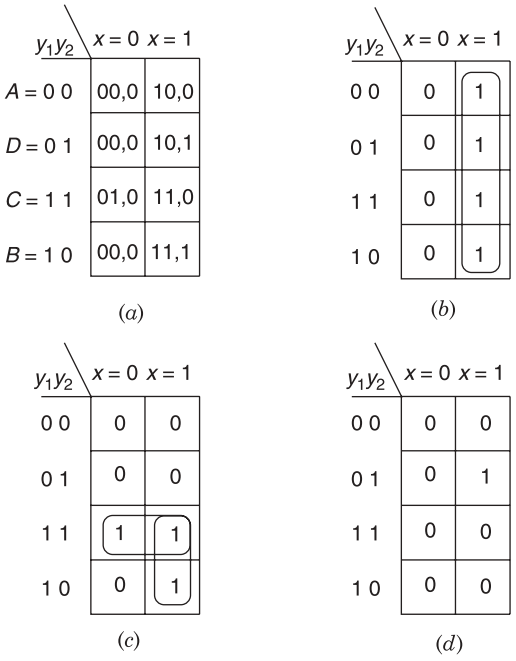


FIGURE 7.20 (*a*) Transition table, (*b*) Karnaugh map for D_1, (*c*) Karnaugh map for D_2, and (*d*) Karnaugh map for *Z*.

Hence the *n*th state of the circuit can be assigned any one of the $2^s - n + 1$ combinations of state variables. Thus there are

$$2^s \cdot (2^s - 1) \cdot \cdots \cdot (2^s - n + 1) = 2^s!/(2^s - n)!$$

ways of assigning 2^s combinations of state variables to the *n* states. The state variables can be permuted in *s*! ways. In addition, each state variable can be complemented, so the set of state variables *s* can be complemented in 2^s ways. Therefore the number of unique state assignments is

$$(2^s!/(2^s - n)!) \cdot (1/s!\ 2^s) = (2^s - 1)!/(2^s - n)!s!$$

Table 7.22 shows the number of state assignments for different values of *n*. Thus even for a circuit with six states, the number of possible state assignments to be considered is 420, and it rapidly rises to more than ten million for a circuit with nine states! Since the number of possible state assignments grows profusely with the number of internal states, it is almost impossible to try all possible assignments in order to select the one that leads to the simplest design equations. However, rather than using exhaustive evaluation, one may follow two simple rules that often result in good state assignments:

Rule 1. Assign adjacent codes (i.e., differing in one bit) to states with the same next state in a column. Assign adjacent codes to states that are the next states of the same present state.

TABLE 7.21 Gate Comparison for Three State Assignments

Number	State Assignment	Number of 2-Input ANDs	Number of 3-Input ANDs	Number of Inverters	Number of 2-Input ORs	Number of 3-Input ORs	Total Number of Gates
1	$A = 00$, $B = 01$, $C = 10$, $D = 11$	1	4	1	1	1	8
2	$A = 00$, $B = 11$, $C = 01$, $D = 10$	1	2	1	1	0	5
3	$A = 00$, $B = 10$, $C = 11$, $D = 01$	2	1	0	1	0	4

TABLE 7.22 Number of State Assignments

n	s	Number of State Assignments
2	1	1
3	2	3
4	2	3
5	3	140
7	3	420
8	3	840
9	4	10,810,800
10	4	75,775,700

Rule 2. Assign adjacent codes to states that are the next states of the same present state. If there is any conflict in the adjacencies obtained by using these rules, the adjacencies obtained from the first rule take precedence.

Let us apply the rules for state assignment to the four-state sequential circuit specified by Table 7.14. Using the first rule, states A and B should be given adjacent assignments because both of them go to state A for $x = 0$. Similarly, state pairs (B,D), (A,D), and (B,C) should be given adjacent assignments; the pair (A,D) appears twice. The application of the second rule shows that A and B should be given adjacent assignments because they are the next states of the present state A. For similar reasons state pairs (A,C) and (C,D) should be adjacent, with (A,B) appearing twice.

The plotting of the three state assignments for the sequential circuit (Table 7.14) on two-variable Karnaugh maps is illustrated in Figure 7.21. It can be seen from the Karnaugh map for assignment III that it satisfies most of the adjacencies and hence produces a better result than the other assignments. It should be noted that although assignment II satisfies the same number of adjacencies as assignment III, it does not fulfill the adjacency requirement for the state pair (A,B) as determined by rule 1.

7.6.1 State Assignment Based on Decomposition

An alternative way of obtaining good state assignments for sequential circuits is to decompose a circuit into smaller subcircuits so that each subcircuit is a function of a small subset

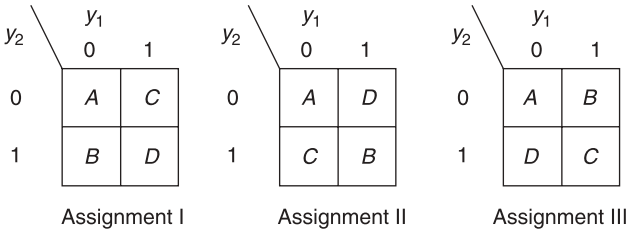


FIGURE 7.21 Comparison of state assignments.

TABLE 7.23 A State Table

	Input	
Present State	$x = 0$	$x = 1$
A	C,0	A,1
B	D,1	F,1
C	A,0	B,0
D	B,1	F,0
E	F,0	B,0
F	E,0	F,1
	Next state, Output	

of the present state variables. This can be done by partitioning the states of a sequential circuit such that each next state variable depends on as few present state variables as possible (*reduced dependency*), thus considerably simplifying the next state equations for the flip-flops in the subcircuits.

A sequential circuit can have a state assignment with reduced dependency if there exists a partition with substitution property on the states of the circuit [1]. As defined in Chapter 1, a partition P on a set of elements S is a collection of disjoint subsets of S such that their set union is S. The subsets of P are called the blocks of S. A partition P on a set of states S of a sequential circuit is said to have the *substitution property* (SP) if any two states belonging to a block of P, under the same input combinations, move to next states that again belong to a common block of P. This common block may or may not be the same block containing the two original states.

For example, the following partitions on the states of the sequential circuit described by Table 7.23 have substitution properties

$$P_1 = (ABF)(CDE)$$
$$P_2 = (AF)(CE)(BD)$$

The partitions with substitution properties for a given sequential circuit can be determined as follows:

1. Identify any two distinct states S_1 and S_2.
2. Identify the pairs of states S_{1K} and S_{2K} to which S_1 and S_2 move if we apply the Kth input, $K = 1, 2, \ldots, m$.
3. To this set of states, add those pairs that can be identified by the transitive law: that is, if S_i and S_j are identified and S_j and S_k are also identified, then we have to identify S_i and S_k.
4. Repeat the process, looking up the identifications induced by the new pairs.

If after x steps, the $(x + 1)$st step does not yield any new identifications, a partition P with the substitution property is obtained on the set of states of the circuit. If a nontrivial partition with the substitution property does not exist, then the process stops after identifying all states of the circuit. For a machine with n states, it is necessary to try $n(n - 1)/2$ distinct pairs of states before deciding whether or not a partition with the substitution property exists.

TABLE 7.24 A State Table

	Input	
Present State	$x = 0$	$x = 1$
A	B,0	D,1
B	B,1	E,0
C	B,0	D,0
D	C,1	A,0
E	C,0	A,1
	Next state, Output	

Let us apply the procedure for obtaining all partitions with the substitution property to the sequential circuit specified by Table 7.24. For this circuit we must consider $5(5-1)/2 = 10$ distinct pairs of states in order to determine all partitions with the substitution property. We start with the state pair (A,B). From Table 7.24 we see that when $x = 0$, both states A and B go to state B. The $x = 1$ column entries show that when $x = 1$, A goes to D and B goes to E. Since the pairs of states (A,B) and (D,E) are disjoint, it is not necessary to add any new state pairs because of the transitive law. This step may be represented as

$$(A,B) \rightarrow (B,B)(D,E)$$

Here the arrow signifies "implies" or "requires." Note that requirements such as (B,B) (i.e., B must be in the same block as B) are always satisfied and need not require further consideration.

Since the pairs of states (A,B) and (D,E) are disjoint, it is not necessary to add new state pairs because of the transitive law. States D and E have to be identified next. From the state table we see that D and E go to C when $x = 0$ and go to A when $x = 1$:

$$(A,B) \rightarrow (D,E) \rightarrow (A,A)(C,C)$$

Since there are no more state pairs to be identified, the process is complete and we get the following partition:

$$(A,B) \rightarrow (D,E) \rightarrow (A,A)(C,C) \equiv\equiv (A,B)(D,E)(C) \equiv\equiv P_1$$

Continuing in the same manner for the other state pairs, we obtain the following partitions:

$$(A,C) \rightarrow (B,B)(D,D) \equiv\equiv (A,C)(B)(D)(E) \equiv\equiv P_2$$
$$(A,D) \rightarrow (B,C) \rightarrow (D,E) \equiv\equiv (A,D,E)(B,C) \equiv\equiv P_3$$
$$(A,E) \rightarrow (B,C)(A,D) \rightarrow (D,E) \equiv\equiv (A,D,E)(B,C) \equiv\equiv P_4$$
$$(B,C) \rightarrow (D,E) \rightarrow (C,C)(A,A) \equiv\equiv (A)(B,C)(D,E) \equiv\equiv P_5$$
$$(B,D) \rightarrow (B,C)(A,E) \rightarrow (D,E)(A,D) \equiv\equiv (A,B,C,D,E) \equiv\equiv P(I)$$
$$(B,E) \rightarrow (B,C)(A,E) \rightarrow (D,E)(A,D) \equiv\equiv (A,B,C,D,E) \equiv\equiv P(I)$$
$$(C,D) \rightarrow (B,C)(A,D) \rightarrow (D,E) \equiv\equiv (A,B,C,D,E) \equiv\equiv P(I)$$
$$(C,D) \rightarrow (B,C)(A,D) \rightarrow (D,E) \equiv\equiv (A,B,C,D,E) \equiv\equiv P(I)$$
$$(D,E) \rightarrow (C,C)(A,A) \equiv\equiv (A)(C)(D,E)(B) \equiv\equiv P_7$$

The sum of P_1 and P_2 yield a new partition P_7 with substitution property

$$P_1 + P_2 = (A,B,C)(D,E) \equiv\equiv P_7$$

A sequential circuit with n states and a binary variable assignment of length s ($=\lceil \log_2 n \rceil$) can be split into two parts such that the first k variables $1 \leq k \leq s$, and the last $(s - k)$ variables can be computed independently, if and only if there exist two nontrivial partitions P_a and P_b with the substitution property that satisfies the following conditions:

(i) $P_a \cdot P_b = P(0)$
(ii) $\lceil \log_2 \#(P_a) \rceil + \lceil \log_2 \#(P_b) \rceil = s$

where $\#P_i$ denotes the number of blocks or subsets in P_i. The sequential circuit under consideration has five states, so we need 3 bits to represent these states. Partitions P_1 and P_2 (and also P_2 and P_3) satisfy the first condition. However, only the partition pair P_2 and P_3 satisfies the second condition,

$$\begin{aligned} s &= \lceil \log_2 \#(P_2) \rceil + \lceil \log_2 \#(P_3) \rceil \\ &= \lceil \log_2 4 \rceil + \lceil \log_2 2 \rceil \\ &= 2 + 1 = 3 \end{aligned}$$

Therefore the assignment is made such that

(a) The secondary variables y_1 and y_2 distinguish the blocks of P_2.
(b) The secondary variable y_3 distinguishes the blocks of P_3.

P_2

Blocks	y_1y_2
(A,C)	00
(B)	01
(D)	10
(E)	11

P_3

Blocks	y_3
(A,D,E)	0
(B,C)	1

The transition table for the composite sequential circuit is shown in Table 7.25. By utilizing the don't care conditions resulting from the three unused binary combinations

TABLE 7.25 Transition Table

Present State	Input	
$y_1\ y_2\ y_3$	$x = 0$	$x = 1$
$A \rightarrow 000$	011,0	100,1
$B \rightarrow 011$	011,1	110,0
$C \rightarrow 001$	011,0	100,0
$D \rightarrow 100$	001,1	000,0
$E \rightarrow 110$	001,0	000,1
	Next state, Output	

010, 101, and 111, the simplified design equations for the *D* flip-flop implementation of the circuit are as follows:

$$D_1 = x\bar{y}_1$$
$$D_2 = \bar{x}\bar{y}_1 + \bar{y}_1 y_2$$
$$D_3 = x$$
$$Z = \bar{x}\bar{y}_1 y_2 + \bar{x} y_1 \bar{y}_2 + x y_1 y_2 + x\bar{y}_1\bar{y}_3$$

It can be seen that the dependence of the next state variables on present state variables is reduced because of the choice of partition pairs. D_1 is dependent on y_1 alone, D_2 is dependent on both y_1 and y_2, and D_3 is not dependent on any of the present state variables.

7.6.2 Fan-Out and Fan-In Oriented State Assignment Techniques

Over the years several techniques have been developed to automate the state encoding process. Two such techniques, MUSTANG and JEDI, are utilized extensively in current computer-aided logic synthesis tools. These techniques lead to a multilevel logic implementation; the number of literals in the next state and output logic expressions after logic optimization is considered as the measure of the quality of the design. Both techniques compute *weight* for each state in a state diagram. The weight for a pair of states estimates the affinity of the states to each other and indicates the *adjacency* of the binary codes that can be assigned to these states (the smaller the *Hamming distance* between two binary codes the more adjacent they are).

Both MUSTANG and JEDI use two distinct algorithms to assign weights to a state pair: *fan-out oriented* algorithm and *fan-in oriented* algorithm. These algorithms assign codes with minimum Hamming distance to a pair of states that have strong *attraction* between them. The attraction between a pair of states is calculated quantitatively from the attraction graph of a sequential circuit. The attraction graph is a weighted undirected graph that is derived from the state transition graph of a sequential circuit. The attraction between a pair of states is computed differently in fan-out and fan-in oriented algorithms.

Fan-Out Oriented Algorithm In this algorithm the *state transition matrix* is derived from a given state transition graph. The rows in the matrix correspond to the present states and the columns to the next states. Each entry in the matrix is the total number of edges from a present state (row) to a next state (column). This is illustrated by applying it to the state transition graph of Figure 7.22. The state transition matrix for the state transition graph is shown in Table 7.26.

For example, there is one transition from state *A* to itself (i.e., self-loop); thus the entry in row *A* and column *A* is 1 in Table 7.26. Similarly, there is only one transition from state *A* to state *B* in Figure 7.22; hence the entry in row *A* and column *B* is also 1. However, there are no transitions from *A* to states *C* or *D*; thus entries in row *A* and columns *C* and *D* are 0's. The entries in other rows and columns are derived in a similar manner. Each state is represented by its corresponding row vector: for example *A*, *B*, *C*, and *D* are represented by vectors 1100, 1010, 0011, and 1100, respectively. Note that *A* and *D* are represented by identical vectors because their next states are the same.

Next, the output matrix is derived from the state transition graph. The output matrix has a row for each present state and a column for each output. An entry in the matrix is a

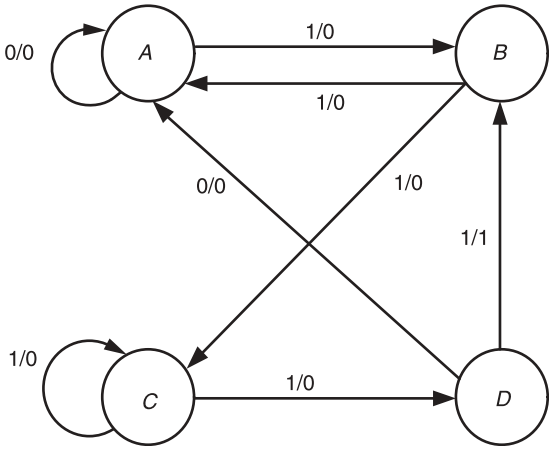


FIGURE 7.22 State transition graph.

TABLE 7.26 State Transition Matrix for Figure. 7.22

	Next States			
Present States	*A*	*B*	*C*	*D*
A	1	1	0	0
B	1	0	1	0
C	0	0	1	1
D	1	1	0	0

nonnegative integer that indicates the number of edges *going out* of a state (row) with a 1 output. The output matrix for the state transition graph of Figure 7.22 is shown in Table 7.27.

Let $S_V(X)$ be the row vector for state X in the state transition matrix, $Z_V(X)$ be the row vector for X in the output matrix, and N_b be the bits needed to encode each state. Then the attraction between states X and Y is given by

$$w(X,Y) = N_b \cdot S_V(X) \cdot S_V(Y) + Z_V(X) \cdot Z_V(Y)$$

TABLE 7.27 Output Matrix

	$Z = 1$
A	0
B	0
C	0
D	1

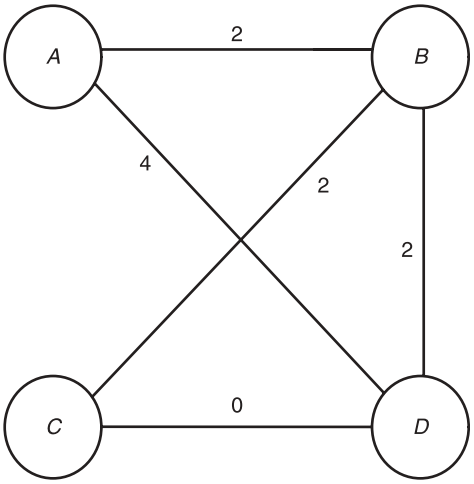


FIGURE 7.23 Attraction graph for Figure 7.22.

Thus for the state transition graph of Figure 7.22, the attraction between state pairs is given by

$$w(A,B) = 2 \cdot (1100) \cdot (1010) + (0) \cdot (0) = 2$$
$$w(A,D) = 2 \cdot (1100) \cdot (1100) + (0) \cdot (1) = 4$$
$$w(B,C) = 2 \cdot (1010) \cdot (0011) + (0) \cdot (0) = 2$$
$$w(B,D) = 2 \cdot (1010) \cdot (1100) + (0) \cdot (1) = 2$$
$$w(C,D) = 2 \cdot (0011) \cdot (1100) + (0) \cdot (1) = 0$$

Hence the attraction graph is as shown in Figure 7.23.

Fan-In Oriented Algorithm As in fan-out oriented algorithm, the state transition matrix is derived from the state transition graph. However, instead of the output matrix, an input matrix is used. In addition, the entries in the matrices are determined differently.

In the state transition matrix used in the fan-in oriented algorithm each row corresponds to a next state and each column represents a current state. An entry in the matrix is the number of edges entering a next state (row) from a current state (column). Thus the state transition matrix for the state diagram of Figure 7.22 is

	Current state			
Next state	*A*	*B*	*C*	*D*
A	1	1	0	1
B	1	0	0	1
C	0	1	1	0
D	0	0	1	0

The input matrix has as many rows as there are states in the state transition graph and a column for each output value. Each entry in the matrix is the number of edges *entering* a state (row) with the input value in a column. The input matrix for the state transition graph of Figure 7.22 is

	Input	
	0	1
A	2	1
B	0	2
C	0	2
D	0	1

The attraction between a pair of states is computed using the state transition and input matrices, and the number of bits needed to encode the states. Let N_b be the number of bits needed for state encoding, $S_V(X)$ be the row vector in the state transition matrix for state X, and $I_v(X)$ be the vector in the input matrix for state X. Then the attraction between a pair of states X and Y is

$$w(X,Y) = N_b \cdot S_V(X) \cdot S_V(Y) + I_V(X) \cdot I_V(Y)$$

Thus for the state transition graph of Figure 7.22 the attraction between state pairs is given by

$$w(A,B) = 2 \cdot (1101) \cdot (1001) + (2, 1) \cdot (0, 2) = 6$$
$$w(A,D) = 2 \cdot (1101) \cdot (0010) + (2, 1) \cdot (0, 1) = 1$$
$$w(B,C) = 2 \cdot (1001) \cdot (0110) + (0, 2) \cdot (0, 2) = 4$$
$$w(B,D) = 2 \cdot (1001) \cdot (0010) + (0, 2) \cdot (0, 1) = 2$$
$$w(C,D) = 2 \cdot (0110) \cdot (0010) + (0, 2) \cdot (0, 1) = 4$$

The resulting attraction graph is shown in Figure 7.24.

Code Assignment An attraction graph is used to assign N_b-bit codes to states. The assignment procedure is as follows:

(i) Find the N_b edges with highest weights for each node and take their sum.
(ii) Select the node with the largest total weight and assign the all 0 code to the node.
(iii) Assign adjacent codes to the N_b neighboring nodes that have the highest weights.

For the attraction graph produced by the *fan-out* algorithm, $N_b = 2$ since 2 bits are needed to encode the states. The sum of two ($N_b = 2$) edges with highest weights is assigned to each node as shown in Figure 7.25. Both nodes *A* and *D* have the highest weight (i.e., 7), thus either can be assigned 00. Assuming *A* has been assigned 00, its

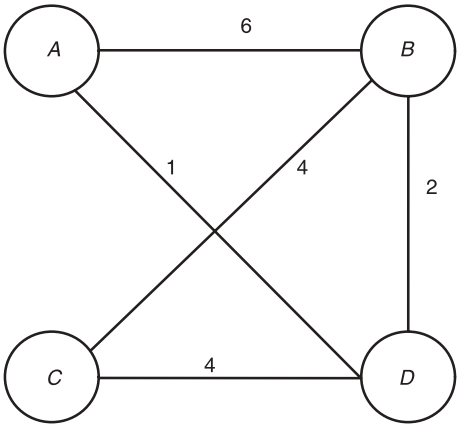


FIGURE 7.24 Attraction graph.

two neighboring states are assigned 01 and 10, respectively. The remaining 2-bit code (i.e., 11) is assigned to node *C*. Thus the state encodings are

$$\begin{array}{rcc} & y_1 & y_2 \\ A = & 0 & 0 \\ B = & 0 & 1 \\ C = & 1 & 1 \\ D = & 1 & 0 \end{array}$$

The resulting next state and output expressions as shown below require nine literals:

$$Y_1 = \bar{x}y_1y_2 + xy_2$$
$$Y_2 = x$$
$$Z = xy_1\bar{y}_2$$

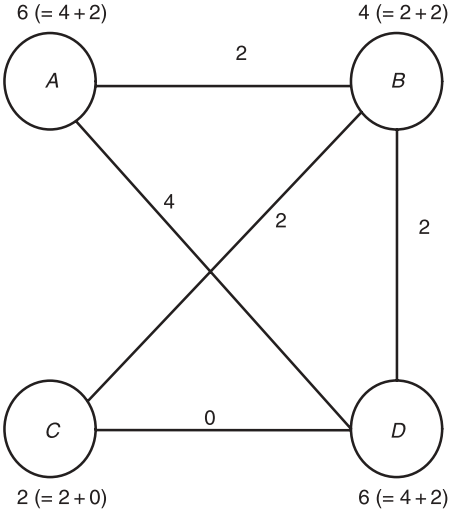


FIGURE 7.25 Nodes with computed weight.

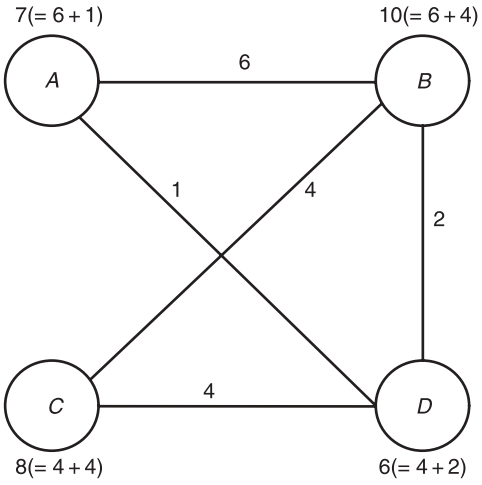


FIGURE 7.26 Fan-in attraction graph with computed node weights.

If the states are encoded in an arbitrary manner (e.g., $A = 00$, $B = 01$, $C = 10$, and $D = 11$), then the number of literals in the resulting next state and output equations will be 17.

The application of the coding procedure to the attraction graph produced by the fan-in algorithm results in the diagram of Figure 7.26.

Since B has the highest weight it is assigned 00. A and C are assigned codes adjacent to 00 (i.e., 01 and 10), and D is assigned 11. Thus the state codes for the circuit resulting from the fan-in algorithm are

$$Y_1 = \bar{x}y_1\bar{y}_2 + x\bar{y}_2$$
$$Y_2 = \bar{x}$$
$$Z = xy_1y_2$$

The total number of literals in these equations is also nine as in the case of the fan-out algorithm.

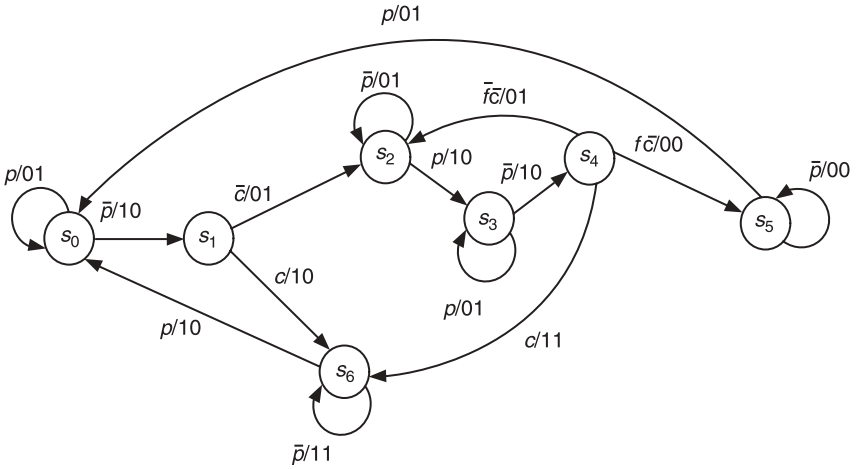


FIGURE 7.27 State transition diagram of a sequential circuit.

7.6.3 State Assignment Based on 1-Hot Code

One straightforward approach for encoding the states of a sequential circuit is to assign a 1-out-of-n code to each state, where n is the number of states in the circuit. In such an n-bit codeword only one bit is 1 (hot), and the rest of the bits are 0's. The state assignment based on 1-out-of-n code is also known as *1-hot* encoding. Let us illustrate the 1-hot encoding for the sequential circuit shown in Figure 7.27. It is a Mealy-type sequential circuit and has three inputs (p, c, f) and two outputs (z_0, z_1). The states are assigned codes as follows:

	y_0	y_1	y_2	y_3	y_4	y_5	y_6
s_0	1	0	0	0	0	0	0
s_1	0	1	0	0	0	0	0
s_2	0	0	1	0	0	0	0
s_3	0	0	0	0	1	0	0
s_4	0	0	0	0	0	1	0
s_5	0	0	0	0	0	0	1
s_6	0	0	0	1	0	0	0

The minimized next state and output expressions corresponding to this assignment are

$$
\begin{aligned}
Y_0 &= p\bar{y}_1\bar{y}_2\bar{y}_4\bar{y}_5 + py_3 \\
Y_1 &= \bar{p}y_0 \\
Y_2 &= \bar{c}\bar{f}y_5 + \bar{p}y_2 + \bar{c}y_1 \\
Y_3 &= c\bar{y}_0\bar{y}_2\bar{y}_3\bar{y}_4\bar{y}_6 + \bar{p}y_3 \\
Y_4 &= \bar{p}(y_2 + y_4) \\
Y_5 &= \bar{p}y_4 \\
Y_6 &= \bar{c}fy_5 + \bar{p}y_6 \\
z_0 &= py_3 + py_2 + Y_1 + Y_3 + Y_5 \\
z_1 &= y_4Y_4 + y_3\overline{Y}_0 + \bar{y}_3Y_0 + cy_6 + Y_2
\end{aligned}
$$

The main disadvantage of 1-hot encoding is that the resulting sequential circuit uses significantly more flip-flops than the minimum number required. On the other hand, the advantage of this approach is that a state can be identified without encoding, and the next state and output logic expressions are relatively straightforward. However, the complexity of the circuit depends on how the 1-hot codewords are assigned to the states; this is not a trivial task.

7.6.4 State Assignment Using *m*-out-of-*n* Code

An alternative for encoding the states of a sequential machine is to use m-out-of-n code. An m-out-of-n code has m 1's and $(m - n)$ 0's, with a Hamming distance of $2d$ ($d = 1, 2, \ldots, \lfloor n/2 \rfloor$) between codewords. Let us consider how to select the m and n values for representing the states of a sequential circuit. The n represents the number of flip-flops required. Note that unlike in 1-hot encoding, where the number of flip-flops is equal to the number of states in the sequential circuit, in the m-out-of-n encoding the minimum value of n is selected such that together with a properly chosen value of m, the number of codewords will be sufficient to uniquely represent each state. Table 7.28 shows the values of n and m needed for encoding different numbers of states.

TABLE 7.28 Selection of n and m Values

Number of States	n	m
4–7	4	2
7–10	5	2
11–20	6	3
21–35	7	3
37–70	8	4
71–127	9	4

TABLE 7.29 A State Table

	$x = 0$	$x = 1$
A	F,100	D,100
B	E,100	C,100
C	E,100	G,100
D	F,100	F,010
E	A,010	B,010
F	A,001	B,001
G	E,100	F,010

Let us implement the sequential circuit of Table 7.29 using m-out-of-n codes for state assignment. There are seven states in the circuit, so we can select $m = 2$ and $n = 5$ (i.e., 2-out-of-5 code). Since there are 10 possible codewords, we arbitrarily choose seven of these for state encoding:

State	y_0	y_1	y_2	y_3	y_4
A	1	1	0	0	0
B	1	0	0	1	0
C	0	1	1	0	0
D	0	1	0	1	0
E	1	0	0	0	1
F	1	0	1	0	0
G	0	1	0	0	1

The next expressions resulting from the above assignment are

$$
\begin{aligned}
Y_o &= \bar{Y}_1 Y_4 + \bar{y}_1 \bar{Y}_2 + \bar{Y}_1 Y_2 + \text{int} \\
\text{int} &= x\bar{y}_0 \bar{y}_2 \\
Y_1 &= \bar{y}_1 \bar{Y}_3 \bar{Y}_4 + \bar{y}_1 \bar{y}_3 \bar{Y}_3 + xY_4 + y_1 Y_3 \\
Y_2 &= \bar{x} y_1 \bar{y}_2 \bar{y}_4 + x\bar{y}_1 y_3 + \text{int} \\
Y_3 &= x\bar{y}_1 \bar{y}_3 + xy_0 y_1
\end{aligned}
$$

$$Y_4 = \bar{y}_0\bar{y}_3 \text{ int} + x\bar{y}_0\overline{\text{int}} + \bar{x}\bar{y}_1 y_3$$
$$z_0 = \bar{x}Y_0\bar{Y}_1 + xY_1 + Y_4$$
$$z_1 = \bar{y}_1 y_4 + \text{int}$$
$$z_2 = \bar{y}_1 y_2$$

The advantage of using m-out-of-n-code for state assignment is that if there is a fault in the next state logic, the circuit may move to an erroneous state, which will be identified by a noncodeword. If a dedicated circuit is incorporated to check whether the outputs of the memory elements are codewords, then an erroneous state can easily be detected. It is also possible to encode the outputs of a sequential circuit using m-out-of-n code such that if a noncodeword output is produced, a fault is assumed to be present in the output logic and/or in the next state logic.

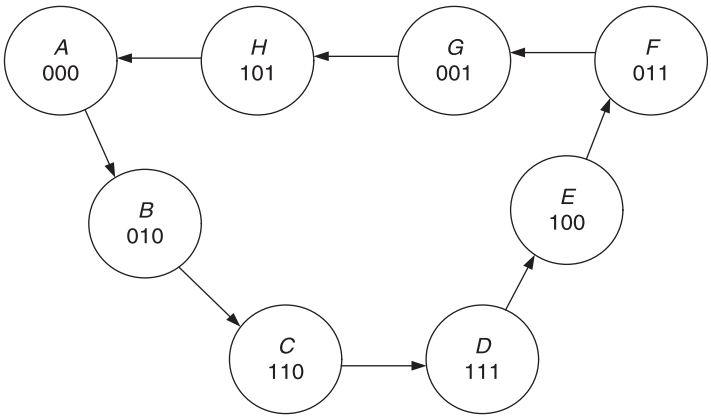
7.7 SEQUENTIAL PAL DEVICES

The PLDs considered in Chapter 3 are combinational devices: their outputs at any instant of time are functions of their inputs at that instant. Although it is possible to implement sequential logic circuits using these devices with memory elements, this results in an increase in the number of packages. Therefore the incorporation of memory elements within PLDs has a significant advantage in that a sequential circuit may be implemented using a single package. PAL types are often generic. The same basic PAL is manufactured by many different companies—Altera, Lattice (Vantis), Cypress, and others.

A number of PAL devices categorized as SPLDs (simple programmable logic devices) are currently available for implementing sequential circuits (e.g., PAL16R8, PAL22V10).

The R8 in device PAL16R8 indicates that it has eight built-in D flip-flops. Each output pin in the device has eight product terms associated with it. The seven product terms in a group drive an OR gate, while a dedicated product term controls the enable input of an inverting tristate buffer. When this product term is activated, the output is enabled and the sum of the product terms is gated to the output. On the other hand, if the product term is not activated, the tristate buffer remains in the high-impedance state and the output pin can be used as an input.

As an example, let us implement the following 3-bit random sequence generator using a PAL16R8:



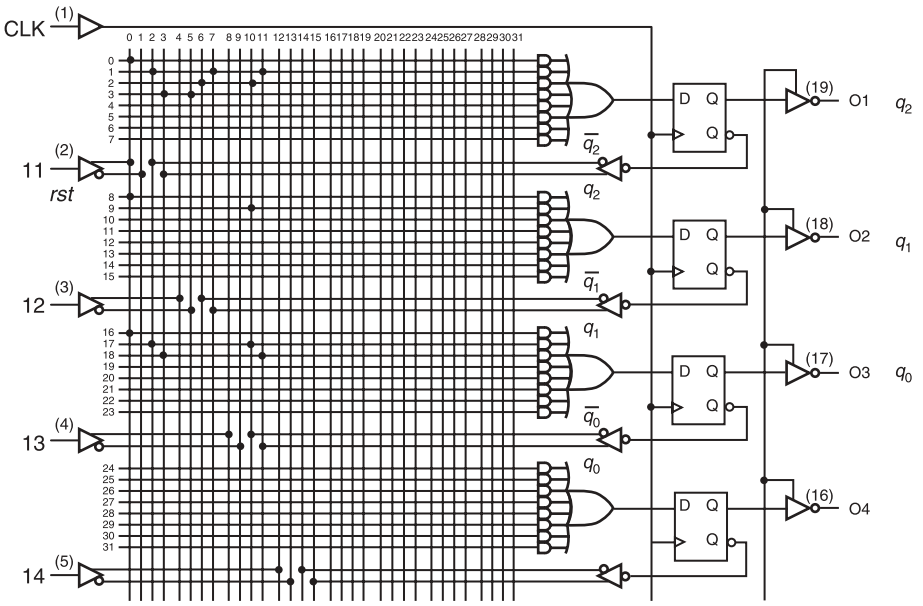


FIGURE 7.28 Programmed PAL16R8.

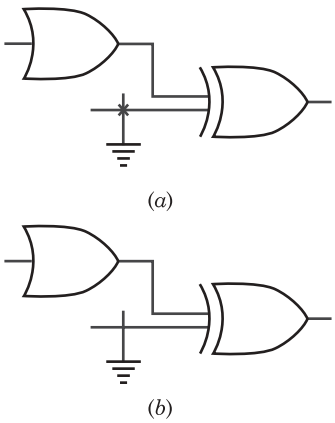
Since there are 3 bits in a state, three D flip-flops of PAL16R8 have to be used. The next state expressions for the flip-flops are

$$q_2 \cdot D = \overline{rst + \bar{q}_2 q_1 q_0 + \bar{q}_1 \bar{q}_0 + q_2 \bar{q}_1}$$

$$q_1 \cdot D = \overline{rst + \bar{q}_0}$$

$$q_0 \cdot D = \overline{rst + \bar{q}_2 \bar{q}_0 + q_2 q_0}$$

Figure 7.28 shows the partial structure of PAL16R8 with the above expressions programmed on the chip.



(*a*)

(*b*)

FIGURE 7.29 Programmable output.

The output polarity of many modern PAL devices can be programmed to be active high or active low. Figure 7.29 shows how the output polarity is programmed using an EX-OR gate. If the fuse at the input of the EX-OR gate is intact, the input remains connected to the ground as in Figure 7.29*a*, and the signal at the other input is directly transferred to the output of the EX-OR gate. On the hand, if the fuse is blown as in Figure 7.29*b* the input is considered to be high and the EX-OR gate functions like an inverter.

PAL22V10 PAL22V10 is one of the most popular PAL devices Figure 7.30 shows the logic diagram of the device. It has 12 dedicated inputs, with one input also acting as a

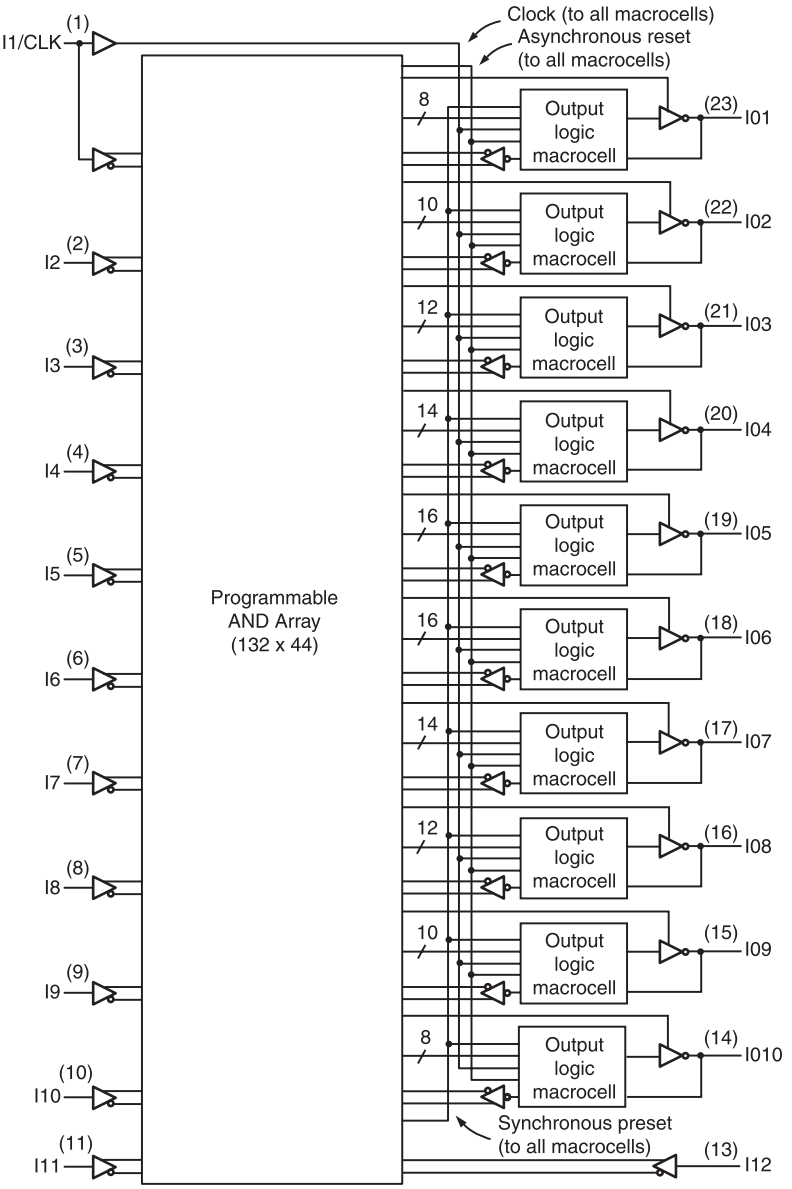


FIGURE 7.30 PAL22V10 structure.

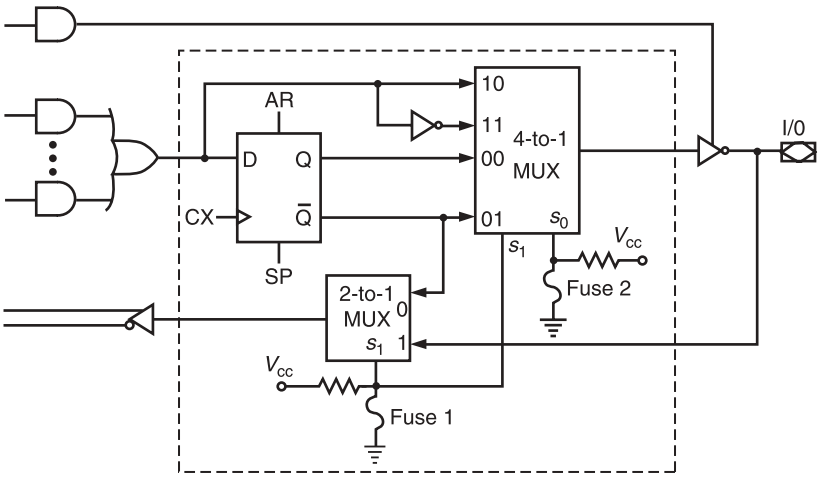


FIGURE 7.31 Logic diagram of a macrocell.

clock input. In addition, it has 10 I/O lines that can be configured either as inputs or outputs. It employs a variable product term distribution that allocates from 8 to 16 product terms to each output.

Each output pin is driven by a *macrocell*. Figure 7.31 shows the logic diagram of a macrocell. When fuse 1s blown, the selection inputs s_1 of both multiplexers are set to logic 1. Similarly, the s_0 input of the 2-to-1 multiplexer is set to logic 1 when fuse 2 is blown. If both fuses remain intact, the select inputs of both multiplexers remain at logic 0.

The macrocell operates in four different modes depending on the status of the fuses 1 and 2:

Mode 1: $s_1s_0 = 00$ (fuse 1 intact, fuse 2 intact). The output of the *D* flip-flop is transferred to the macrocell and is also fed back to the AND array. However, because of the presence of the inverter at the output of the 4-to-1 multiplexer, only the complement of the registered output is available.

Mode 2: $s_1s_0 = 01$ (fuse 1 intact, fuse 2 blown). The complement output of the *D* flip-flop is transferred to the output of the macrocell via the inverter, thus generating the true registered output. The macrocell output is also internally fed back to the AND array.

Mode 3: $s_1s_0 = 10$ (fuse 1 blown, fuse 2 intact). The inverted value of the OR gate is available at the output of the macrocell, and is also fed back to the AND array.

Mode 4: $s_1s_0 = 11$ (fuse 1 blown, fuse 2 blown). The output of the OR gate is available at the output of the macrocell, and is also fed back to the AND array.

Note that the tristate inverter at the macrocell output has to be enabled by the dedicated AND gate to make the output available; otherwise the I/O pin acts as an additional input if fuse 1 is blown.

All 10 *D* flip-flops in a 22V10 share an asynchronous reset (AR) product term and a synchronous preset (SP) product term. When the preset term is activated, all the *D* flip-flops in the device are loaded with logic 1's on the positive edge of the clock pulse. If the reset product term is enabled, the flip-flops are loaded with logic 0's independent of the clock. It is not possible, however, to individually preset and reset a flip-flop.

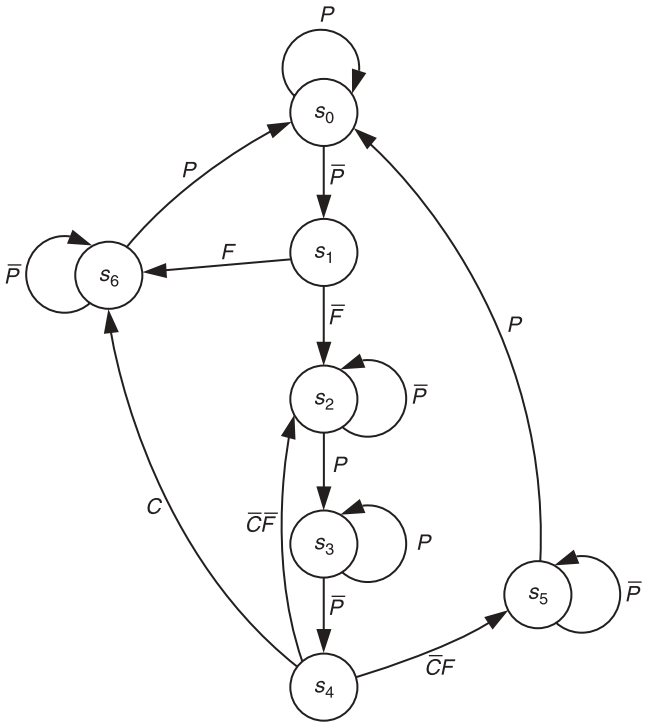


FIGURE 7.32 A state transition graph [2].

It should be clear from the operating modes of PAL22V10 that it can be programmed to implement arbitrary state machines. *A* desired state is first preloaded in the flip-flops of the device.

As an example, the sequential circuit specified by the state transition graph of Figure 7.32 is implemented using the device. The next state equations of the circuit are derived assuming the following state assignment:

	y_2	y_1	y_0
A	1	0	0
B	1	0	1
C	0	1	0
D	1	1	0
E	1	1	1
F	0	0	1
G	0	0	0

The next state expressions are

$$Y_0 = \bar{P}y_0\bar{y}_2 + \bar{P}\bar{y}_0y_2 + \bar{C}Fy_0y_1$$
$$Y_1 = \bar{y}_0y_1 + \bar{C}\bar{F}y_1 + \bar{F}y_2\bar{y}_1y_0$$
$$Y_2 = P\bar{y}_0 + P\bar{y}_3 + \bar{y}_0y_3$$

These are mapped onto a PAL22V10 as shown in Figure 7.33.

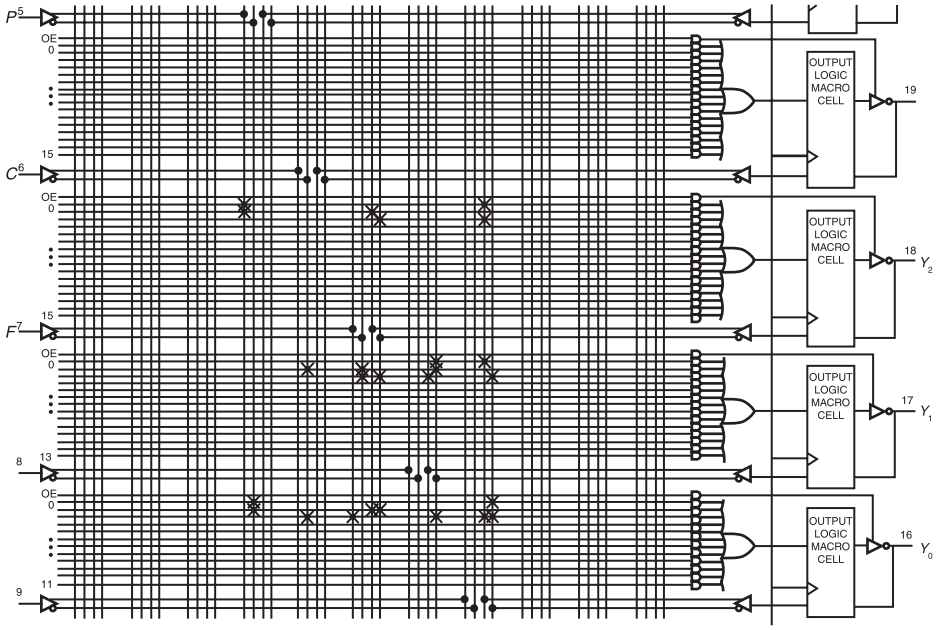
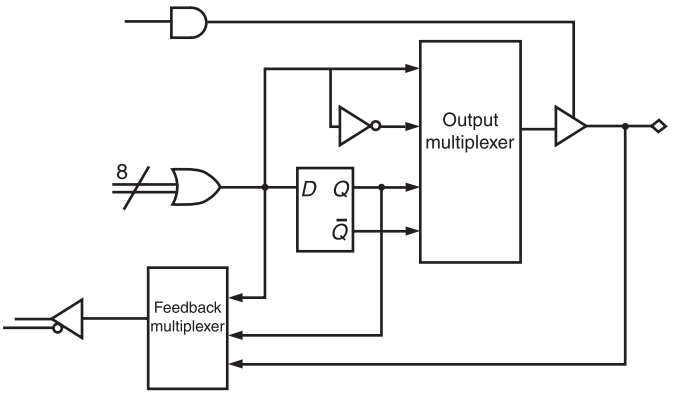


FIGURE 7.33 A section of programmed 22V10.

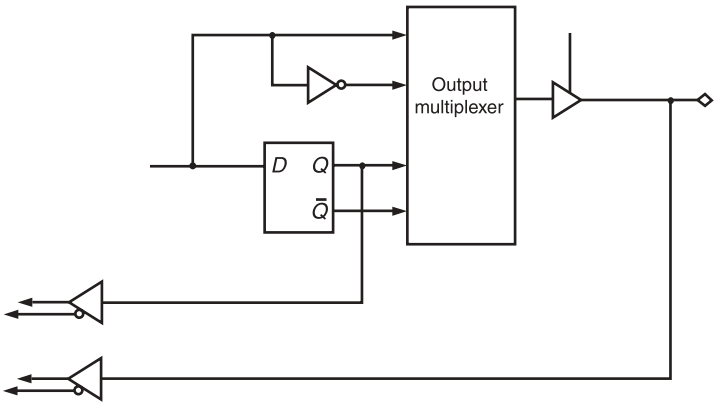
Complex PLDs In the past all PLDs were manufactured using fuse-based TTL (transistor transistor logic) technology. A drawback of fuse-based programmable logic devices is that they cannot be reprogrammed. Thus they are not suitable in applications requiring alteration of logic functions or in situations where a new logic design has to be modified several times before a satisfactory design is obtained. This, in addition to high power consumption in TTL devices, has resulted in a gradual shift to CMOS (complementary metal on silicon) technology (see Appendix). CMOS PLDs consume significantly less power than TTL devices. Modern CMOS PLDs include many PAL-like blocks that are interconnected by a programmable switch matrix; these devices are popularly known as *complex PLDs* (CPLDs).

One significant advantage of CMOS devices is that they use EPROM cells or EEPROM cells instead of fuses as programmable connections. *A* fuse takes up a large amount of silicon area whereas the EPROM and EEPROM cells are significantly smaller than fuses; thus more functions can be packed onto a smaller device. The devices based on EPROM cells are known as *EPLDs* (erasable programmable logic devices), and those based on EEPROM cells are referred to as *EEPLDs* (electrically erasable programmable logic devices).

EPLDs Altera Corp. developed the first EPLDs in the form of the MAX family of chips. Since then, many other companies have marketed EPLDs. There are three series of chips in the MAX family—MAX 5000, MAX 7000, and MAX 9000. The 7000 series is widely used; the 9000 series is similar to the 7000 series but the chips have higher capacity. All chips in the MAX family have a programmable AND/fixed OR structure, whose output feeds a *macrocell* (Fig. 7.34*a*). The output multiplexer in Figure 7.34*a* allows the selection of the output OR gate, the flip-flop, or their complements. The feedback



(*a*)



(*b*)

FIGURE 7.34 (*a*) An EPLD Macrocell and (*b*) dual feedback loop.

multiplexer allows feedback from the output of the OR gate, the output of the flip-flop, or directly from the I/O pin. If the output of the flip-flop is not made available on the I/O pin, the flip-flop is considered to be *buried* and the I/O pin cannot be used. Similarly, if the I/O pin is used as an input pin, the flip-flop cannot be used. In other words, the macrocell of Figure 7.34*a* allows only a single feedback into the AND array.

Certain EPLD allow dual feedback paths into the AND array (Fig. 7.34*b*). One feedback path comes directly from the I/O pin, and the other comes from the internal logic. Basically, dual feedback is achieved by eliminating the feedback multiplexer of Figure 7.34*a*. If the tristate buffer in Figure 7.34*b* is disabled, the internal logic becomes isolated from the I/O pin, thus allowing the internal flip-flop to be buried and the I/O pin to be used as an input line.

Figure 7.35 shows the architecture of the MAX 7000 series chips. It is composed of an array of blocks known as logic array blocks (LABs). The inputs and outputs of a LAB can be connected to any other LAB via a global bus called the *programmable interconnect array* (PIA). The PIA is fed by all dedicated inputs, I/O pins, and macrocells. All chips in the MAX family have eight dedicated input lines.

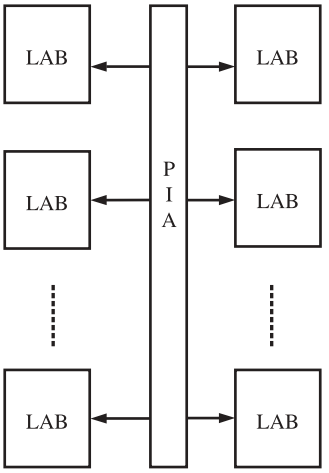


FIGURE 7.35 Block diagram of MAX 7000 series chips.

Each LAB has 17 macrocells, 17 expander product terms, and an I/O block as shown in Figure 7.36. The expander product terms can be shared by all other macrocells in a LAB. Figure 7.37 shows the macrocell structure. The flip-flop in a macrocell can be configured as a *D*, *JK*, *SR*, or *T* type flip-flop as well as a conventional latch. In addition, each flip-flop has individually programmable clock, clock enable, clear, and preset functions. The OR gate in each macrocell of a MAX 7000 series LAB can be fed by up to five dedicated product terms within the macrocell and can also use any or all sixteen extra product terms in the same LAB; note that most PAL devices do not allow implementation of functions exceeding eight product terms. The chip with fewest number of LABs in MAX 7000 is EPM7032. It has 2 LABs, 32 macrocells, and 37 I/O pins.

Let us illustrate the implementation of a sequential circuit to be used for controlling the operation of a washing machine; the circuit is implemented using an Altera MAX 7032 device. The block diagram of the controller is shown in Figure 7.38. The washing machine goes through the following sequence operations*:

1. The washing machine is filled with water. The temperature of the water (cold or hot) is selected by an input (hc). The machine moves to the next state when it is full.
2. A timer is set to a specified time and the machine agitates until the time is complete.
3. The water drains from the machine. When it is empty the machine moves to the next step.
4. The machine is filled with cold water. It moves to the next step when it is full.
5. The timer is set and the machine agitates until the selected time is complete.
6. The water drains from the machine. When it is empty the machine moves to the next step.
7. The timer is set and the washer spins the tub to dry the clothes until the specified time is completed.

*VHDL class project report of Jessie Weaver.

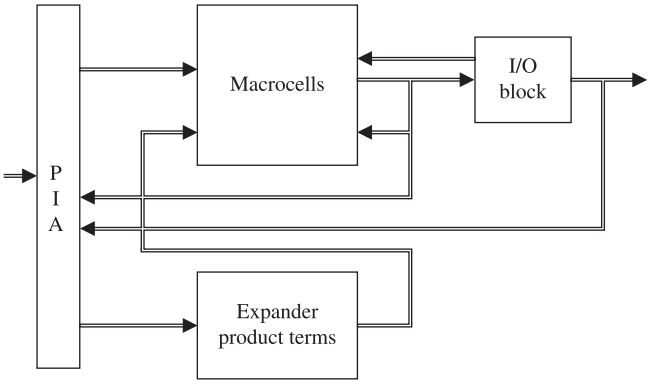


FIGURE 7.36 Logic array block (LAB).

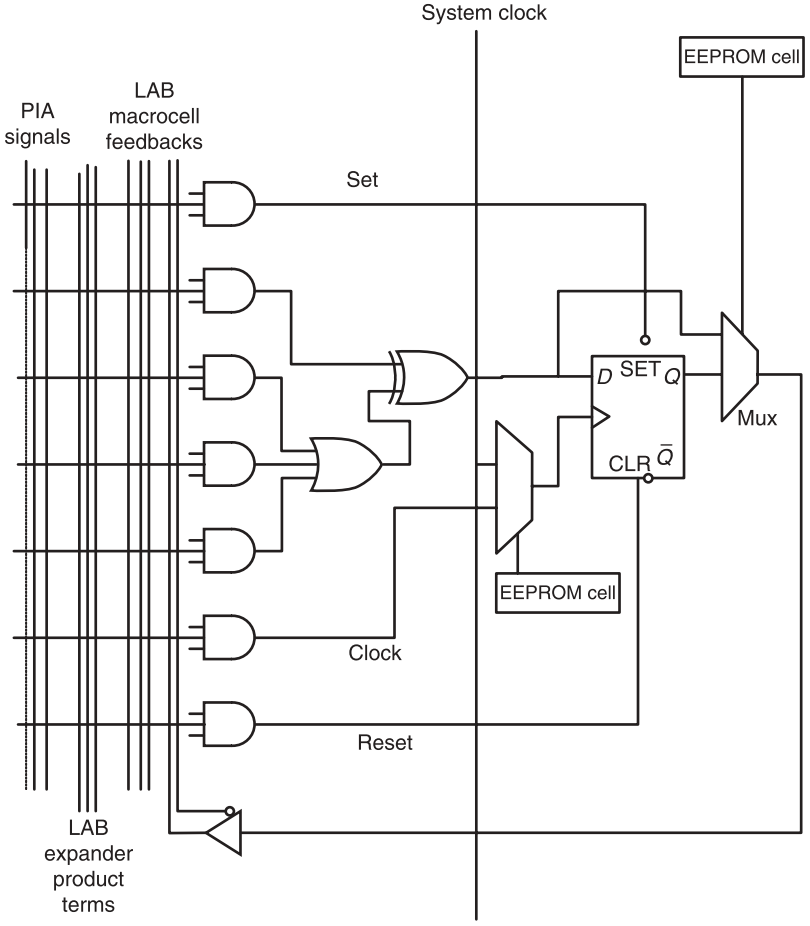


FIGURE 7.37 Macrocell.

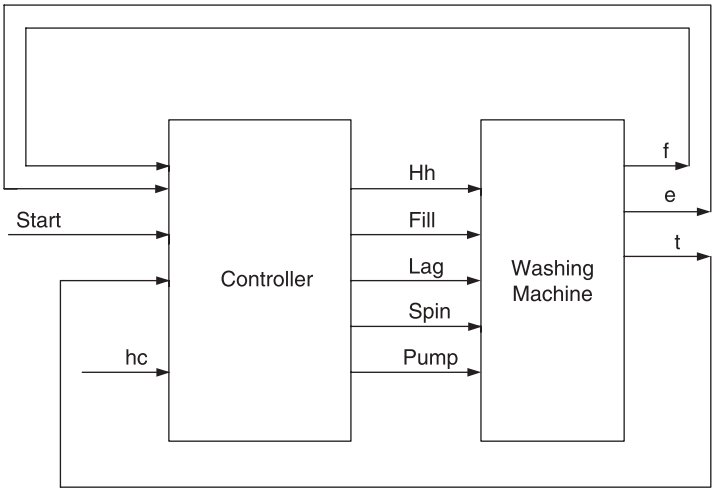


FIGURE 7.38 Block diagram of the washing machine controller circuit.

8. The machine is ready to start when it is signaled to do so by activating the S (Start/Stop) input.
9. The machine can be interrupted at any of the steps 1 to 8 by applying the reset signal. If there is water in the machine it is drained in this state.

A Moore-type sequential circuit with nine states with each state corresponding to one step of the above operations can be used for this purpose. Although states 2 and 5 and states 3 and 7 seem to perform identical operations, the correct order of operations requires states 2 and 5 (as well as states 3 and 7) to be included to determine if it is the first or the second time to agitate (drain); otherwise additional circuits will be needed to distinguish between the two. The state diagram of the Moore-type sequential circuit is shown in Figure 7.39.

The inputs and outputs of the machine are as follows:

Inputs:	hc	temperature of the water: 0 = cold, 1 = hot
	s	start/stop: 0 = start, 1 = stop
	f	0 = not full, 1 = full
	e	0 = not empty, 1 = empty
	t	0 = timer not done, 1 = timer done
Output:	Fill	0 = drain water out of washer, 1 = direct water into washer
	Lag	0 = start timer and agitate wash, 1 = don't agitate
	Spin	0 = start timer and spin wash, 1 = don't spin
	Hh	select water temperature; 0 = cold, 1 = hot
	Pump	0 = pump on, 1 = pump off

As it would be clear from the architecture of the MAX 7000 series chips, the mapping of Boolean expressions onto one of these chips can only be done by using an appropriate

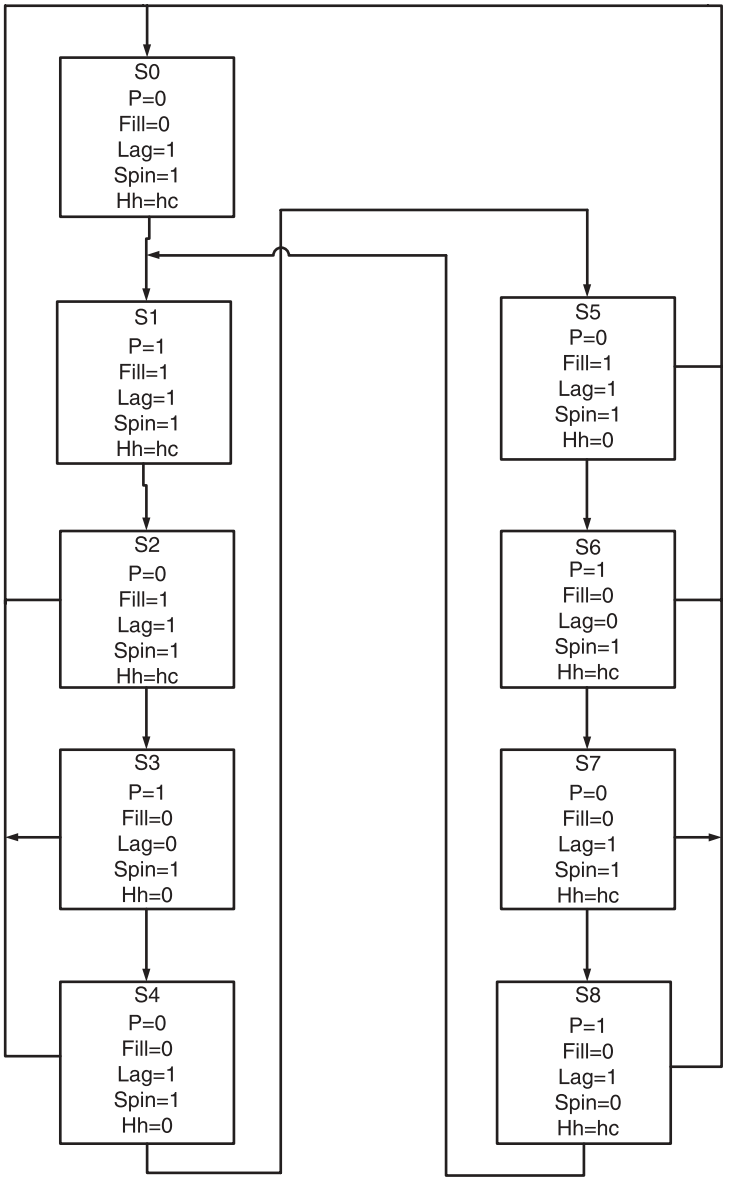


FIGURE 7.39 The state transition graph of the Moore-type sequential circuit as shown in Figure 7.38.

computer-aided design tool. The compilation of the VHDL code for the above sequential circuit by using Altera's Quartus II software provides the state assignment, the corresponding next state expressions, and the resources needed—that is, number of macrocells and expanded product terms used, identification of macrocells utilized, as well as interconnections among them—to implement the circuit on a selected MAX 7000 series chip. Assuming an EPM7032 chip is used for the implementation, the state assignment and the corresponding next state expressions generated by Quartus II are shown in Figure 7.40*a* and 7.40*b*, respectively. (Note that A1L9, A1L7, and A1L8 are the EX-OR of expanded

	Q3	Q2	Q1	Q0
S0	0	0	0	0
S1	1	1	0	1
S2	0	0	0	1
S3	0	0	1	0
S4	0	1	0	0
S5	0	0	1	1
S6	0	1	1	0
S7	0	1	0	1
S8	1	0	1	0

(*a*)

```
A1L7 = (!Q2 & Q0 & f ) # (Q2 & !Q0 & e & !Q1);

A1L8 = (s & Q0 & Q3) # (!Q0 & e & !Q1 & !Q2) # (Q0 & Q1 & f ) # (!s & Q1 & Q2);

A1L9 = (Q0 & Q3) # (Q0 & !s & !e & Q2);

Q0_p0_out = t & Q3;

Q0_p1_out = !s & !f & Q1 & Q0;

Q0_p2_out = !s & !f & Q0 & !Q2;

Q0_p3_out = !Q1 & !Q0 & e;

Q0_p4_out = Q1 & Q2 & t;

Q0_or_out = A1L9 # Q0_p0_out # Q0_p1_out # Q0_p2_out # Q0_p3_out # Q0_p4_out;

Q1_p0_out = Q1 & Q0 & f;

Q1_p1_out = !t & !s & Q1;

Q1_p2_out = !t & !s & Q3 & !Q0;

Q1_p3_out = !Q3 & Q0 & e & Q2;

Q1_p4_out = !s & Q1 & Q0;

Q1_or_out = A1L7 # Q1_p0_out # Q1_p1_out # Q1_p2_out # Q1_p3_out # Q1_p4_out;

Q2_p0_out = !Q0 & Q3 & t;

Q2_p1_out = !s & !e & Q2 & !Q0;

Q2_p2_out = !s & !e & Q2 & !Q3;

Q2_p3_out = !Q0 & Q1 & t;

Q2_p4_out = Q2 & Q1 & t;

Q2_or_out = A1L8 # Q2_p0_out # Q2_p1_out # Q2_p2_out # Q2_p3_out # Q2_p4_out;
```

FIGURE 7.40 (*a*) State assignment. (*b*) Next state and output expressions.

```
Q3_p0_out = Q0 & !Q3 & e & Q2;

Q3_p1_out = s & Q0 & Q3;

Q3_p2_out = !s & !Q0 & Q3;

Q3_p3_out = !Q0 & Q3 & t;

Q3_p4_out = !Q0 & !Q1 & e & !Q2;

Q3_or_out = Q3_p0_out # Q3_p1_out # Q3_p2_out # Q3_p3_out # Q3_p4_out;

LAG_p1_out = Q0 & !Q2;

LAG_p2_out = !Q2 & Q3;

LAG_p3_out = Q0 & Q3;

LAG_or_out = LAG_p1_out # LAG_p2_out # LAG_p3_out # !Q1;

FILL_p1_out = Q3 & Q0;

FILL_p2_out = Q0 & !Q2;

FILL_p3_out = Q0 & Q1;

FILL_or_out = FILL_p1_out # FILL_p2_out # FILL_p3_out;

SPIN = !( !Q0 & Q3);

HHL_p0_out = !Q0 & hc & Q1 & Q2;

HHL_p1_out = Q3 & Q0 & hc;

HHL_p2_out = Q0 & hc & !Q1;

HHL_p3_out = hc & !Q1 & !Q2;

HHL_p4_out = Q3 & hc & !Q2;

HHL_or_out = HHL_p0_out # HHL_p1_out # HHL_p2_out # HHL_p3_out # HHL_p4_out;

P_p1_out = Q1 & !Q0 & !Q3;

P_p2_out = Q0 & Q3;

P_p3_out = Q3 & !Q2;

P_or_out = P_p1_out # P_p2_out # P_p3_out;
```

(*b*)

FIGURE 7.40 (*Continued*).

product terms needed to implement the next state expressions for Q0, Q1, and Q2, respectively. # stands for the EX-OR operation.)

The implementation of these expressions needs 12 macrocells (4 for next state expressions, 5 for output expressions, and 3 for expanded product terms). Thus one out of the two LABs in EPM7032 is utilized in implementing the state machine.

EXERCISES

1. Find a minimal state table for each of the sequential circuits specified:

Present State	Input $x = 0$	Input $x = 1$
A	B,0	F,1
B	D,0	D,0
C	C,0	F,0
D	A,1	A,0
E	D,0	D,0
F	D,0	B,0

(a)

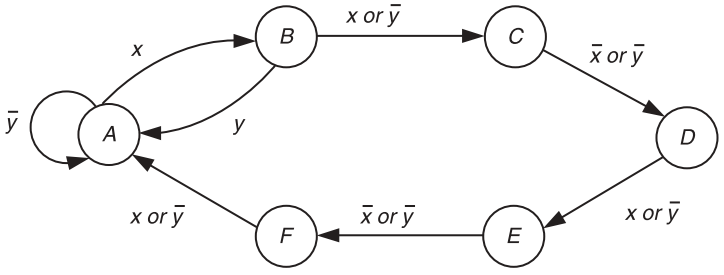
Present State	Input $x = 0$	Input $x = 1$
A	C,0	E,0
B	H,0	G,1
C	B,0	A,0
D	E,1	H,0
E	E,1	C,0
F	H,0	D,1
G	A,1	H,0
H	D,0	F,1

(b)

2. For the state tale shown below, find the output and state sequences corresponding to the input sequence: 01010110 assuming that the circuit starts in state A.

Present State	Input $x = 0$	Input $x = 1$
A	D,0	A,0
B	D,1	A,0
C	C,0	B,0
D	C,0	A,0

3. The state diagram of a sequential circuit is shown below. Implement the circuit using D flip-flops as memory elements. Assume the following state assignment: $A = 000$, $B = 001$, $C = 010$, $D = 011$, $E = 100$, $F = 101$.



4. Implement the state diagram of Exercise 3 using JK flip-flops as memory elements.

5. A synchronous sequential circuit with two inputs x_1 and x_2 and an output z is to be designed. The output z is to be 1 whenever both x_1 and x_2 receive identical groups of five input bits. Each input bit is synchronized with a clock pulse applied to the clock input. Show the minimized state table for the circuit and implement it using *D* flip-flops as memory elements.

6. A synchronous sequential circuit has one input x and one output z. the output is 1 whenever the input sequence is 0110 (e.g., if the input sequence is 000110110, the output sequence would be 000001001). Construct the state diagram for the circuit, and implement it using *JK* flip-flops as memory elements.

7. A synchronous sequential circuit is to be used for generating the parity of a continuous stream of binary digits. The output of the circuit produces a logic 1 if the number of 1's received at the input is even; otherwise the output is at logic 0. Implement the circuit using *JK* flip-flops as memory elements.

8. A synchronous sequential circuit is to be used to detect errors in a message using 2-out-of-4 code. The sequential circuit receives a coded message serially and produces an output of 1 whenever an illegal message is received. Develop a state diagram to meet this specification, and implement the circuit using *D* flip-flops.

9. Implement the sequential circuit of Exercise 3 using an *m*-out-of-*n* code.

10. A sequential circuit having a single input and a single output is to be designed according to the following specification. The output is to be at logic 0 unless an input sequence 0010 is received (e.g., if the input sequence is 0100100100, the output sequence will be 0000001001). Construct a minimum row state table for a circuit.

11. Find a minimum row state table for each of the incompletely specified sequential circuits specified by the following state tables:

Present State	Input $x = 0$	Input $x = 1$
A	B,1	H,1
B	A,0	G,0
C	–,0	F,–
D	D,0	–,1
E	C,0	D,–
F	A,–	C,0
G	–,–	B,–
H	G,0	E,–

(*a*)

Present State	Input $= x_1x_2$ 00	01	11	10
A	C,1	–,–	G,1	E,1
B	–,–	E,0	–,–	–,–
C	F,1	F,0	–,1	–,–
D	–,–	–,0	–,–	B,1
E	F,0	–,–	D,–	A,0
F	–,–	C,0	C,–	B,0
G	F,0	–,0	–,1	B,–

(*b*)

Present State	Input I_1	Input I_2	Input I_3
A	A,0	B,1	E,1
B	B,0	A,1	F,1
C	A,1	D,0	E,0
D	F,0	C,1	A,0
E	A,0	D,1	E,1
F	B,0	D,1	F,1

(*c*)

12. Find a minimum row state table for each of the sequential circuits whose states are given below by using the partitioning method.

Present State	Input $x = 0$	Input $x = 1$
A	A,0	E,1
B	E,1	C,0
C	A,1	D,1
D	F,0	G,1
E	B,1	C,0
F	F,0	E,1
	Next state, Output	

(*a*)

Present State	Input $x = 0$	Input $x = 1$
A	B,0	C,1
B	A,1	E,0
C	F,1	C,0
D	D,0	C,1
E	A,1	B,0
F	B,0	D,1
	Next state, Output	

(*b*)

13. A one-input, four-state (A, B, C, and D) sequential circuit produces the output sequence Z corresponding to an input sequence I as shown:

I	1	1	0	1	0	0	1	0	1	1	0	0	1	0
Z	1	0	0	0	1	1	1	0	1	0	0	1	0	0

If the circuit output response and the next state for the input sequence 01 are as follows

	01	
A	00	C
B	01	B
C	10	A
D	11	B

fill in the rest of the entries (state, output) in the state table circuit:

	$x = 0$	$x = 1$
A	C,0	–,1
B	A,–	–,0
C	–,–	C,–
D	A,–	–,0

14. Implement the state table of the circuit given below using 1-hot state encoding:

	Input	
Present State	$x = 0$	$x = 1$
A	A,0	E,1
B	E,1	C,0
C	A,1	D,1
D	F,0	G,1
E	B,1	C,0
F	F,0	E,1
	Next state, Output	

15. Derive the state encoding for the circuit specified by the following state table using *fan-in oriented* algorithm:

	Input	
Present State	$x = 0$	$x = 1$
A	B,0	C,1
B	A,1	E,0
C	F,1	C,1
D	D,0	C,1
E	A,1	B,0
F	B,0	D,1
	Next state, Output	

16. Derive the state encoding for the circuit specified by the following state table using *fan-out oriented* algorithm:

	Input	
Present State	$x = 0$	$x = 1$
A	B,0	A,1
B	D,1	A,1
C	B,0	D,0
D	D,0	C,1
E	E,1	E,1
F	A,0	B,0
	Next state, Output	

17. Design a sequential circuit that produces an output of 1 after it has received an input sequence 1101 or 011. Derive the design equations for the circuit assuming *D* flip-flops as memory elements. Use the principle of partitioning for generating the state assignments.

18. Minimize the following incompletely specified sequential circuits if possible; implement the circuits using *D* flip-flops as memory elements.

	Input	
Present State	$x = 0$	$x = 1$
A	*E*,0	*A*,0
B	*D*,0	*B*,0
C	*E*,1	*C*,–
D	*A*,0	*A*,1
E	*A*,–	*B*,–

(*a*)

	Input	
Present State	$x = 0$	$x = 1$
A	*C*,1	–,–
B	*B*,–	*E*,–
C	*A*,1	*D*,1
D	*F*,0	*G*,0
E	*G*,–	–,–
F	–,–	*C*,0
G	*A*,1	*B*,–

(*b*)

19. A sequential circuit is to be used to identify possible noncodewords produced by a 3-out-of-6 code generator circuit. The sequential circuit will examine the output of the code generator circuit 1 bit at a time. After all 6 bits have been examined, the circuit produces an output of 1 if it detects a noncodeword; otherwise an output of 0 is produced. Derive the excitation and the output expressions for the circuit assuming it will be implemented using a PAL22V10 device.

20. A sequential circuit is to be designed to monitor the status of a chemical experiment. Every 10 s the circuit receives an input pattern between 0001 and 1111. If the input pattern received is 0111, the experiment is assumed to be continuing perfectly; the input patterns 0110 and 1000 are also considered satisfactory. However, if the circuit receives any other binary pattern twice consecutively, the experiment must be stopped. Derive a state diagram of the sequential circuit and a complete logic diagram using *JK* flip-flops as memory elements.

REFERENCES

1. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1978.
2. P. K. Lala, *Digital System Design Using PLDs*, Prentice Hall, Englewood Cliffs, NJ, 1990.

8 Counter Design

8.1 INTRODUCTION

Counters are frequently used in computers and other digital systems. Since a counter circuit must remember its past states, it has to possess memory. Thus the sequential logic design principles discussed in Chapter 7 can be utilized in designing counter circuits. Like all other sequential logic circuits, counter circuits can be classified into two categories—*synchronous* and *asynchronous*.

In synchronous counters all memory elements are simultaneously triggered by a clock, whereas in asynchronous counters the output of each memory element activates the next memory element.

Many types of counters are used in practice. In some cases they count in pure binary; in other cases the count may differ considerably from straight binary (e.g., decade or BCD counters). This chapter examines the construction and operation of the most important types of counters.

8.2 RIPPLE (ASYNCHRONOUS) COUNTERS

In a ripple counter there is no clock or source of synchronizing pulses; however, the state changes still occur due to pulses at clock inputs of the flip-flops. Figure 8.1*a* shows a 3-bit ripple counter constructed from *JK* flip-flops. It is assumed that the flip-flops change state on the negative-going (falling) edge of the pulses appearing at their clock inputs. The counter is first cleared by applying a reset pulse; thus counting begins from 000. A timing diagram representing the sequence of logic states through which flip-flops *A*, *B*, and *C* go in counting from 0 to 7 is shown in Figure 8.1*b*. As can be seen in Figure 8.1*a*, the normal output of flip-flop *C* acts as the clock pulse of flip-flop *B*. Similarly, the output of flip-flop *B* is used as the clock pulse source for flip-flop *A*. The input (i.e., the count pulses) is applied only to the clock input of flip-flop *C*. Note that the *J* and *K* inputs of all the flip-flops are tied to logic 1.

The first negative-going pulse at the clock input of flip-flop *C* changes its output to 1. Thus the counter shows an output of 001. Since the output of flip-flop *C* is 1, the clock input of flip-flop *B* is also 1. When the second negative-going pulse occurs at the clock input of flip-flop *C*, its output makes a transition from 1 to 0. This negative-going transition *Q* at the clock input of flip-flop *B* changes its output from 0 to 1.

Principles of Modern Digital Design, by Parag K. Lala

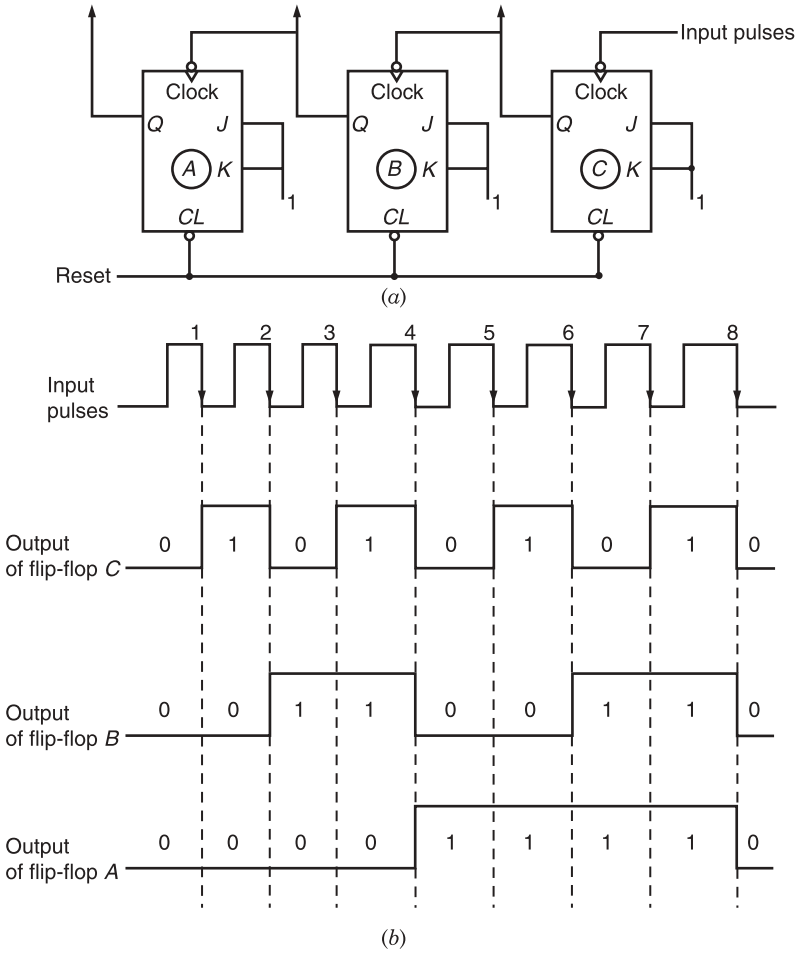


FIGURE 8.1 (*a*) A 3-bit ripple counter and (*b*) timing diagram.

Hence the output of the counter is now 010. On the third negative-going pulse at the clock input of flip-flop *C*, its output changes from 0 to 1. This positive-going transition cannot change the output of flip-flop *B*. Therefore after the third negative-going pulse, the counter output is 011. It continues to operate in this manner until the count value 111 is reached.

The next negative-going input pulse at the clock input of flip-flop *C* will change its output from 1 to 0. This transition in the output of *C* will cause the output of flip-flop *B* to change from 1 to 0, which in turn will change the output of flip-flop *A* to 0. In other words, the counter is reset to 000 after eight pulses and is ready to begin counting again as subsequent input pulses are applied at the clock input of flip-flop *C*. Thus any change in the output of flip-flop *C* moves through the counter like a ripple on water, hence the name "ripple counter."

An alternative way of implementing a 3-bit ripple counter is shown in Figure 8.2. This is implementation uses *T* flip-flops. The *T* flip-flops are triggered on the positive-going (rising) edge of the pulses appearing at their clock inputs.

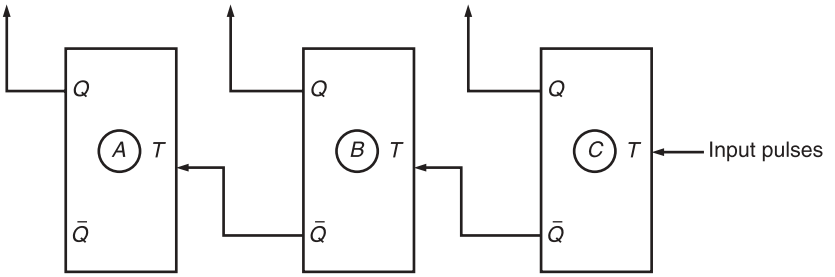


FIGURE 8.2 Three-bit ripple counter implemented with *T* flip-flops.

The 3-bit ripple counter circuit has eight ($=2^3$) different states, each one corresponding to a count value. Similarly, a counter with n flip-flops can have 2^n states. The number of states in a counter is known as its *mod* (*modulo*) *number*. Thus a 3-bit counter is a mod-8 counter. Similarly, a 6-bit counter is a mod-64 counter (i.e., it has 64 distinct states, 000000 through 111111). A mod-n counter may also be described as a *divide-by-n* counter, in the sense that the most significant flip-flop produces one pulse for every n pulses at the clock input of the least significant flip-flop. Thus the counter of Figure 8.1*a* is a divide-by-8 counter.

The ripple counter poses a problem when there is a large number of flip-flops. In such a case, the most significant flip-flop cannot change state until the propagation delay times of all other flip-flops have elapsed. For example, if each flip-flop in a 6-bit ripple counter has a propagation delay of 10 ns, it will take 60 ns when changing from a count of 31 (0111111) to a count of 32 (100000).

Since the states of the flip-flops in a ripple counter do not change simultaneously, undesirable transient states may be produced during the change from one valid state to another. For example, in going from state 011 to 100, the counter of Figure 8.1*a* generates two transient states 010 and 000:

counter state 011
↓
counter state 010 (filp-flop *C* changes)
↓
counter state 000 (filp-flop *B* changes)
↓
counter state 100 (filp-flop *A* changes)

Such transient states in a counter may have undesirable effects on other parts of a digital system, so they must be guarded against.

It is also possible to design a ripple counter such that it will only count up to a value less than the maximum possible. For example, the 3-bit ripple counter can count up to 7 (111) before resetting to 0 (000). By adding NAND gate to the counter circuit, it may be made to reset for every fifth pulse—that is, when the count is 5 (101), as shown in Figure 8.3. The inputs to the NAND gate are the outputs flip-flops *A* and *B*, so the output of the NAND gate will go to 0 whenever the outputs of flip-flops *A* and *B* are 1. This happens only when the counter makes a transition from state 101 to state 110. This makes the output of the NAND gate go to 0, which in turn clears the flip-flops within a few nanoseconds. Thus the counter

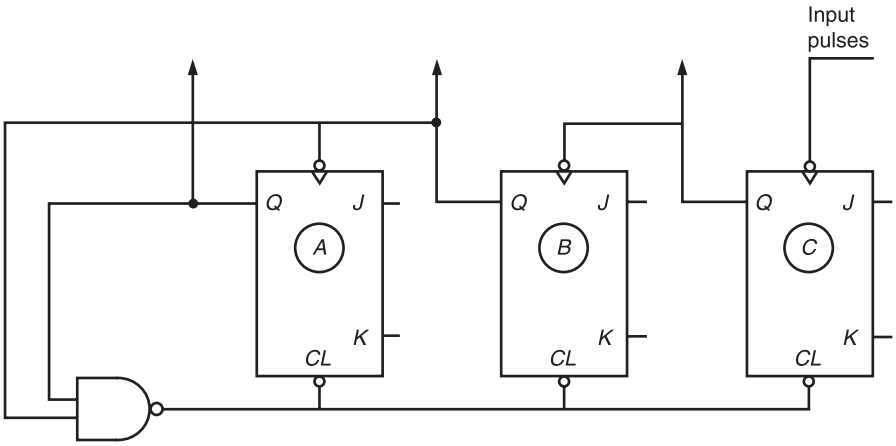


FIGURE 8.3 Mod-6 counter.

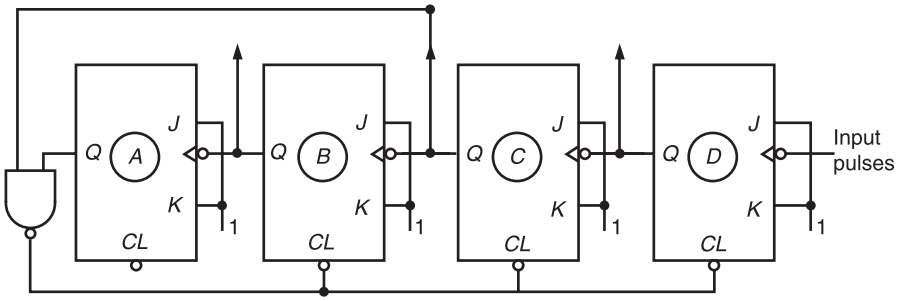
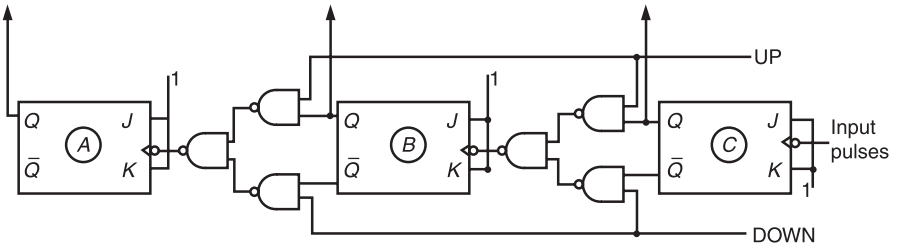


FIGURE 8.4 Decade counter.

essentially counts up to 101 and then resets to 000 (i.e., it is a mod-6 counter). In a similar manner, counters with any other modulus can be designed by using a NAND gate to detect the appropriate state for resetting the counter. For example, using a NAND gate with inputs *A* and *C*, a 4-bit counter can be reset to 0000 when the count value 1010 is reached (Fig. 8.4). In other words, the normal count will be from 0000 to 1001, resulting in a mod-10 counter, also known as a *decade counter*.

8.3 ASYNCHRONOUS UP–DOWN COUNTERS

In certain applications a counter must be able to count both up and down. Figure 8.5 shows the circuit for a 3-bit up–down counter. It counts up or down depending on the status of the control signals UP and DOWN. When the UP input is at 1 and DOWN input is at 0, the NAND network between flip-flops *B* and *C* will gate the noninverted output of flip-flop *C* into the clock input of flip-flop *B*. Similarly, the noninverted output of flip-flops *B* will be gated through the other NAND network into the clock input of flip-flop *A*. Thus the counter will count up. When the control input UP is at 0 and DOWN is at 1, the inverted outputs of flip-flop *C* and flip-flop *B* are gated into the clock inputs of flip-flops



Up	Down	Operation
0	0	No counting
1	0	Counts up
0	1	Counts down
1	1	No counting

FIGURE 8.5 A 3-bit up–down counter.

B and *A*, respectively. If the flip-flops are initially reset to 0's, then the counter will go through the following sequence as input pulses are applied:

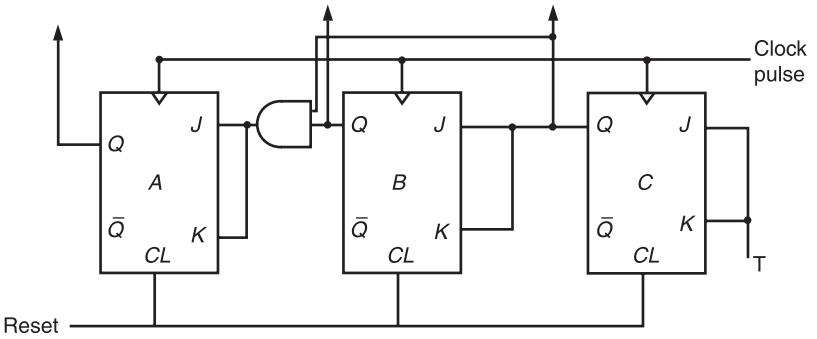
A	*B*	*C*
0	0	0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1

An asynchronous up–down counter is slower than an up counter or a down counter because of the additional propagation delay introduced by the NAND networks.

8.4 SYNCHRONOUS COUNTERS

In asynchronous counters counter the state of a flip-flop changes only when a transition occurs at the output of its preceding flip-flop. Since each flip-flop has a propagation delay, the time required by an asynchronous counter to complete its response to an input sequence is of the order of the sum of the propagation delays of the flip-flops in the counter. If the total propagation delay of the counter is greater than or equal to the period of the input pulse, then the counter does not function properly. Thus the maximum frequency at which the input pulse can be applied to a ripple counter has to be lowered if the number of flip-flops in the counter is large.

In synchronous counters, the clock inputs of all the flip-flops are connected together and are triggered by the input pulses. Thus all the flip-flops change state simultaneously (i.e., in parallel). Figure 8.6*a* shows the circuit of a 3-bit synchronous counter. The *J* and *K* inputs of flip-flop *C* are connected to logic 1. The *B* flip-flop has its *J* and *K* inputs connected to the output of flip-flop *C*, and the *J* and *K* inputs of flip-flop *A* are connected to the output of



(*a*)

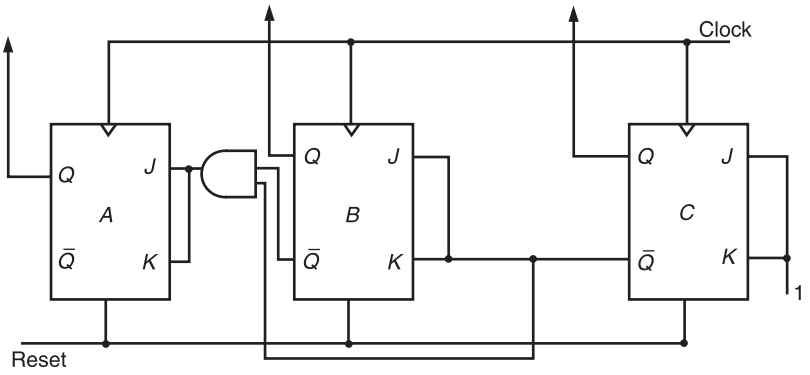
A	*B*	*C*
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

(*b*)

FIGURE 8.6 (*a*) A 3-bit synchronous counter and (*b*) the count sequence.

an AND gate that is fed by the outputs of flip-flops *B* and *C*. Assuming that the outputs of all the flip-flops are initially reset to 0, the rising edge of the first clock pulse will change the output of flip-flop *C* to 1. This will result in a 1 at the *J* and *K* inputs of flip-flop *B*. The rising edge of the second clock pulse will cause flip-flop B to change its output from 0 to 1 and flip-flop C to change its output from 1 to 0. On the positive edge of the third clock pulse, the output of flip-flop *C* will change again, from 0 to 1. Therefore both the inputs of the AND gate will be at 1 at the end of the third clock pulse. The positive edge of the fourth clock pulse will cause flip-flop *C* to change its output again; flip-flop *B* will change at the same time because its *J* and *K* inputs are at 1. Flip-flop *A* will also change state on the positive edge of the fourth clock pulse because its *J* and *K* inputs are at 1 due to the AND gate. The count sequence for the 3-bit counter is shown in Figure 8.6*b*.

The most important advantage of this synchronous counter is that there is no cumulative time delay because all the flip-flops are triggered in parallel. Thus the maximum operating frequency for this counter will be significantly higher than that for the corresponding ripple counter. For example, if the propagation delay of each flip-flop is 20 ns and that of the AND gate is 10 ns, the minimum clock period for the 3-bit synchronous counter is flip-flop propagation delay + AND-gate propagation delay = 20 + 10 = 30 ns, whereas the minimum clock period for the 3-bit ripple counter (Fig. 8.1) is 3 × flip-flop propagation delay = 3 × 20 = 60 ns. The 3-bit synchronous counter we considered is an up counter; its circuit can be modified slightly so that it can perform as a down counter (Fig. 8.7*a*). The corresponding count sequence is shown in Figure 8.7*b*.



(*a*)

A	*B*	*C*
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

(*b*)

FIGURE 8.7 (*a*) A 3-bit synchronous down counter and (*b*) the count sequence.

Synchronous counters can be designed using the techniques employed for designing synchronous sequential circuits (Chapter 6). The first step in the design is to list the required count sequence in a two-column transition table; the first column represents the present state of the counter, and the second column gives the next state of the counter. The counter moves from a present state to the corresponding next state after a clock pulse has been applied.

Example 8.1 Let us design a mod-5 counter using *JK* flip-flops. Figure 8.8 shows the transition table for the mod-5 counter circuit. Since there are five unique states, we need 3

Present State			Next State		
A	*B*	*C*	*A*	*B*	*C*
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	0	0	0

FIGURE 8.8 State table for mod-5 synchronous counter.

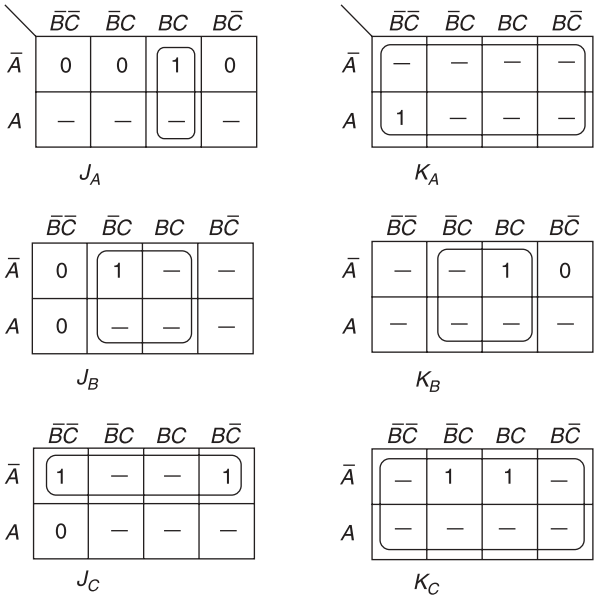


FIGURE 8.9 Excitation maps for the *JK* flip-flops in a mod-5 counter.

(=$\log_2 5$) flip-flops to implement the counter circuit. Next, we form the excitation map for the J and K inputs of each flip-flop in the circuit, as shown in Figure 8.9. The resulting input equations for the flip-flops are:

$$J_A = BC \quad K_A = 1$$
$$J_B = C \quad K_B = C$$
$$J_C = \bar{A} \quad K_C = 1$$

Implementation of the counter is shown in Figure 8.10.

In many applications it is necessary to determine whether a counter has reached a particular count value before a certain operation is started. Thus a decoder circuit has to be used to indicate the presence of the desired count value. For example, we can connect a 4-to-10 decoder at the output of the binary decade counter to convert its output to a

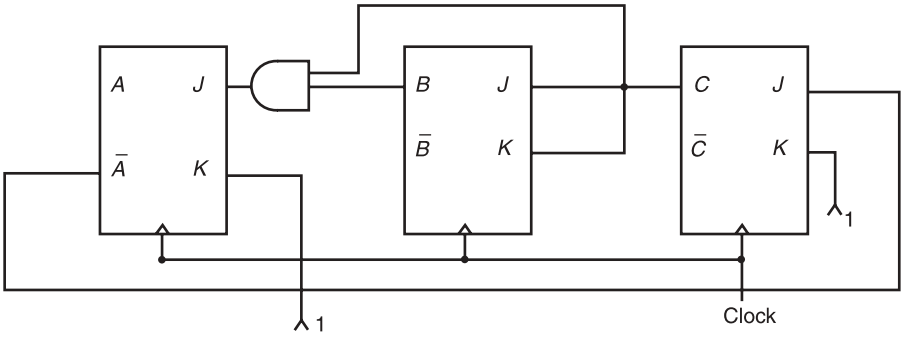


FIGURE 8.10 Mod-5 counter circuit using *JK* flip-flops.

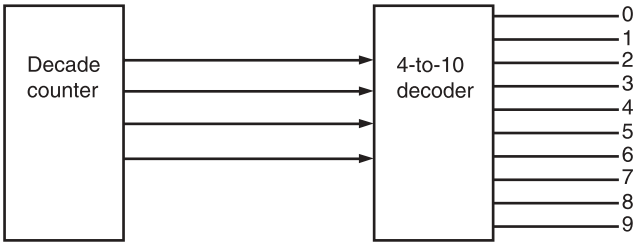


FIGURE 8.11 Conversion of the binary outputs of a counter to decimal form.

decimal representation (Fig. 8.11). The decoder output in turn can be used to drive a display device to indicate the decimal number corresponding to the binary count.

In the counter design technique we considered, all redundant states (i.e., states for which no next states are specified) are used as don't care terms. One very important point in the design of counters (or any sequential circuit) is the starting state of the counter when power is turned on. If the counter circuit starts in a redundant state, it must be ensured that the counter eventually returns to one of the states in the count sequence; a counter having this property is known as *self-starting counter*.

Unless it is important that a counter go to a particular valid state from a redundant state, it is usually possible to simplify counter logic using the redundant states and to ensure that it is self-starting. For example, in the mod-5 counter circuit of Figure 8.8 there are three redundant states:

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

By specifying a valid next state for each of the redundant states, the counter can be made self-starting. The next states for the redundant states are selected such that there is a reduction in the number of gates required for implementing the excitation equations for the flip-flops. The complete state diagram of the mod-5 counter is shown in Figure 8.12. If the counter starts in one of the redundant states, the next clock pulse will transfer it to one of the valid states and it will continue to count properly. The excitation maps for the D flip-flop implementation of the self-starting counter are shown in Figure 8.13*a*, and the corresponding circuit is shown in Figure 8.13*b*.

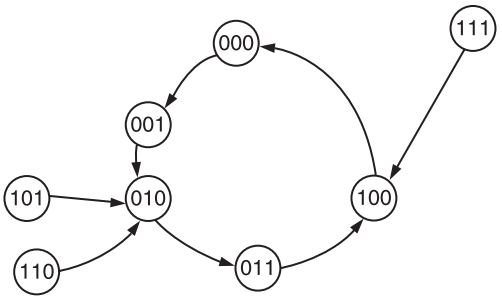


FIGURE 8.12 State diagram for the self-starting mod-5 counter.

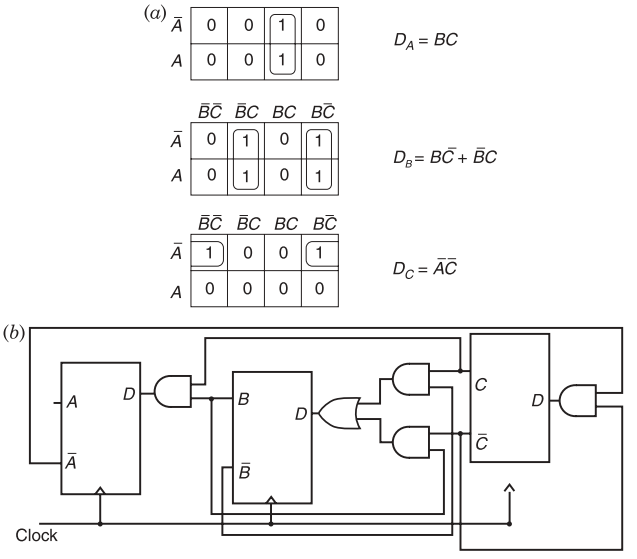


FIGURE 8.13 (*a*) Excitation maps and (*b*) self-starting mod-5 counter.

8.5 GRAY CODE COUNTERS

In a Gray code, only one bit changes in going from one code combination to another. Thus a Gray code counter can be used to eliminate the problem of momentary false count values that results from a binary counter, where more than one bit is required to change states during a transition.

Example 8.2 Let us design a 4-bit Gray code counter using *JK* flip-flops. The transition table for the counter is shown in Figure 8.14.

Present State				Next State			
A	*B*	*C*	*D*	*A*	*B*	*C*	*D*
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	1	0	0	1	0
0	0	1	0	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	0	1	0	1
0	1	0	1	0	1	0	0
0	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0
1	1	1	0	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	0	0	1
1	0	0	1	1	0	0	0
1	0	0	0	0	0	0	0

FIGURE 8.14 Transition table for 4-bit Gray code counter.

The excitation maps for the A, B, C, and D flip-flops are plotted in Figure 8.15a. The simplified functions for the J and K inputs are obtained from these maps; they are

$$JA = B\bar{C}\bar{D} \qquad KA = \bar{B}\bar{C}\bar{D}$$
$$JB = \bar{A}C\bar{D} \qquad KB = AC\bar{D}$$
$$JC = \bar{A}\bar{B}D + ABD \qquad KC = \bar{A}BD + A\bar{B}D$$
$$JD = \bar{A}B\bar{C} + AB\bar{C} + \bar{A}BC + A\bar{B}C \qquad KD = \bar{A}B\bar{C} + A\bar{B}\bar{C} + \bar{A}\bar{B}C + ABC$$

The implementation of the counter is shown in Figure 8.15b.

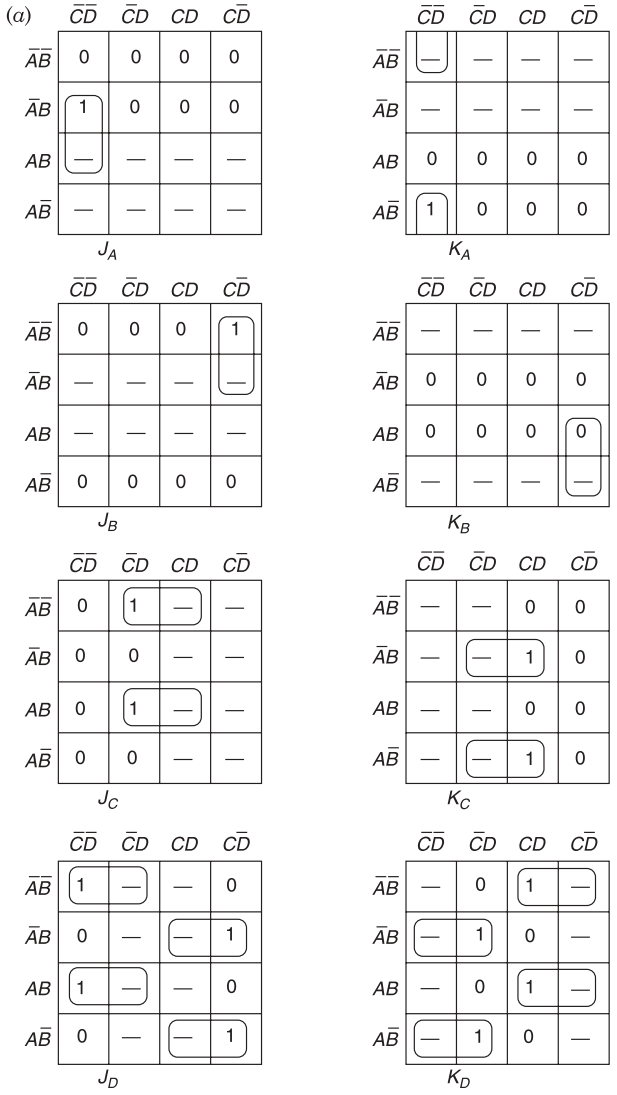
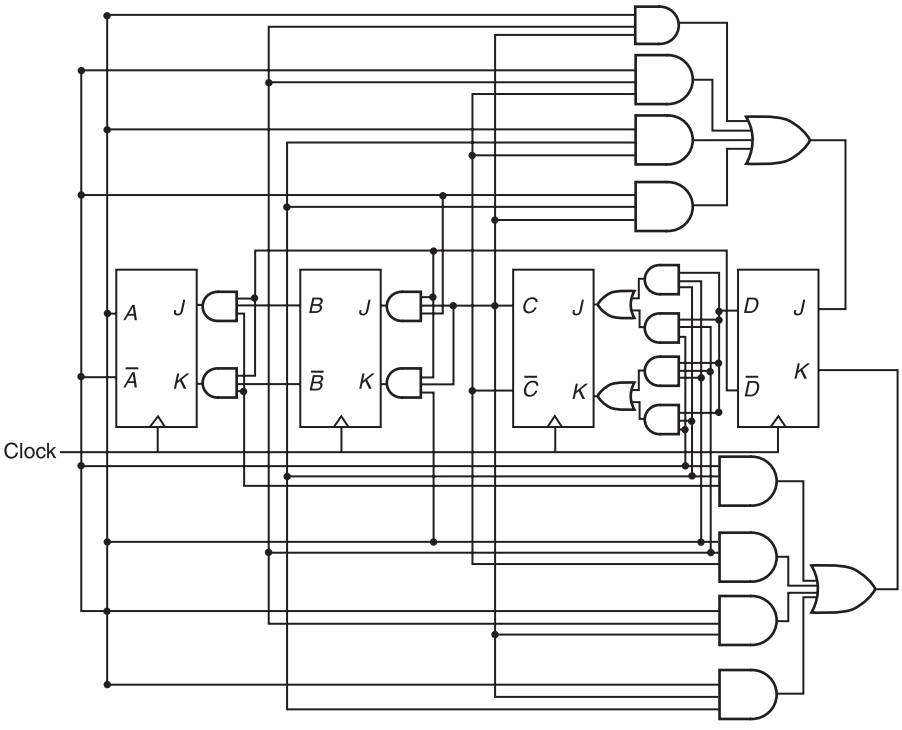


FIGURE 8.15 (a) Excitation maps and (b) 4-bit Gray code counter.

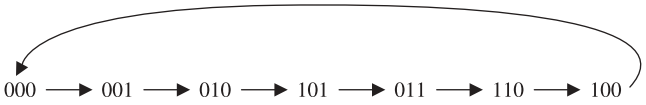


(*b*)

FIGURE 8.15 (*Continued*).

8.6 SHIFT REGISTER COUNTERS

A counter is constructed from a serial in/parallel out shift register by connecting the output of the individual flip-flops to a large network and connecting the output of that network to the input of the shift register (Fig. 8.16) so that it cycles through some predetermined number of states. For example, in the case of a 3-bit shift register the sequence is



Such a counter may be considered to be a mod-7 counter; it is also known as a *nonbinary* counter because the sequence of states does not form a consecutive sequence of binary coded numbers. The design of shift register counters can be simplified considerably by using a state *sequence tree* [1]. The state sequence tree shows how many possible sequences of different lengths can be obtained from a shift register of a given length. The derivation of the state sequence tree starts with the all 0's state of the shift register

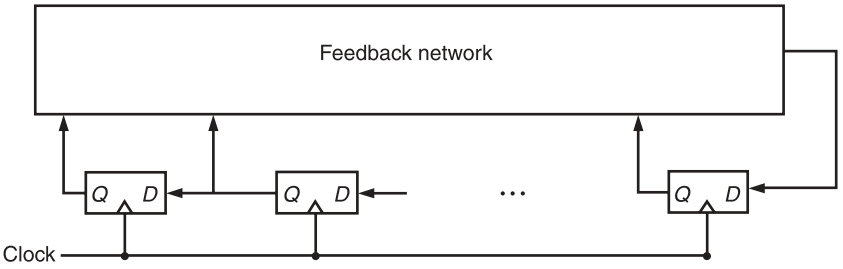
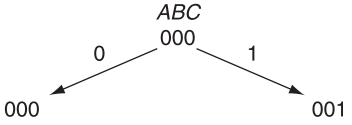
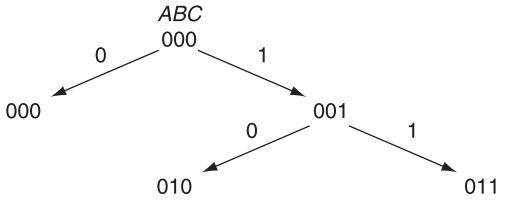


FIGURE 8.16 Shift regsister counter.

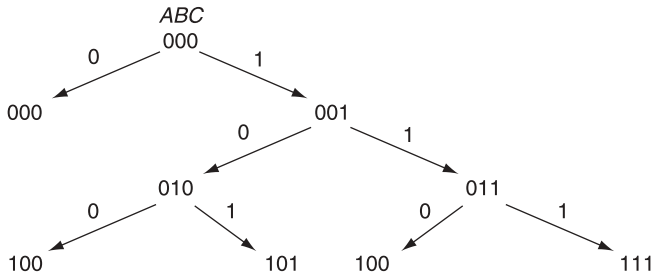
at the root of the tree. The effect of a 0 and a 1 on the counter is recorded as shown:



Since the result of shifting in a 0 does not change the state, the left side of the tree need not be considered any further. The effect of shifting in a 0 or a 1 while the shift register is in state 001 is shown next:



Since there are two new states, the next level of the tree should be



This process is continued for each branch of the tree until a newly derived state already exists in the path from the root to this node of the tree (i.e., a state repeats itself). Figure 8.17 shows the complete state sequence tree for the 3-bit shift register. A shift counter of mod *m*, where *m* is less than or equal to the total number of possible states in the shift register, can be constructed by selecting the desired sequence of states from the sequence tree.

As an example, let us design a mod-6 counter using a 3-bit shift register. It can be seen from the state sequence tree for the shift register (Fig. 8.17) that there are three different

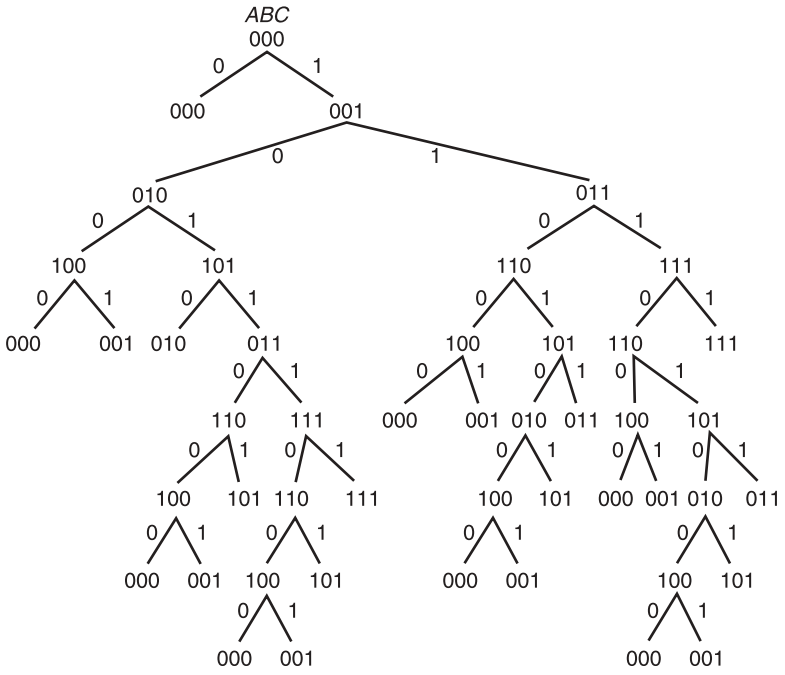
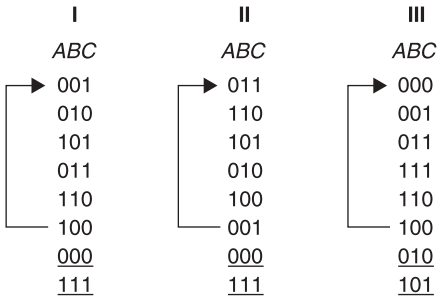


FIGURE 8.17 State sequence for a 3-bit shift register.

ways of achieving a state sequence of length 6:



In each of these three state sequences the unused states are underlined. Looking at the first state sequence, we see that if the present content of the shift register is 001 ($=\bar{A}\bar{B}C$), it must change to 010 ($\bar{A}B\bar{C}$) after a clock pulse. In other words, when the shift register contains 001, the input to it must be a 0. Similarly, when the preset content is 010, the shift input must be 1 so that after the next clock pulse its content is 101. Thus the shift input must be equal to the lowest significant bit of the next state to which the shift register should move after the clock pulse is applied. Referring to Figure 8.16, we can see that the output of the feedback network is fed to shift input. The appropriate feedback network for the three state sequences can be designed by using Karnaugh maps (Fig. 8.18). The minimal Boolean expressions for implementing the feedback networks

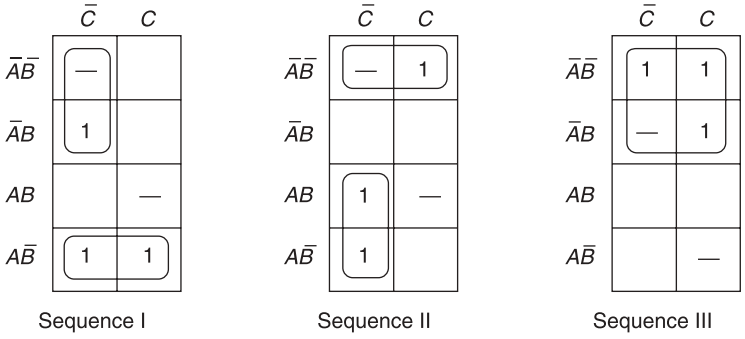


FIGURE 8.18 Karnaugh maps for feedback logic networks.

are derived as follows:

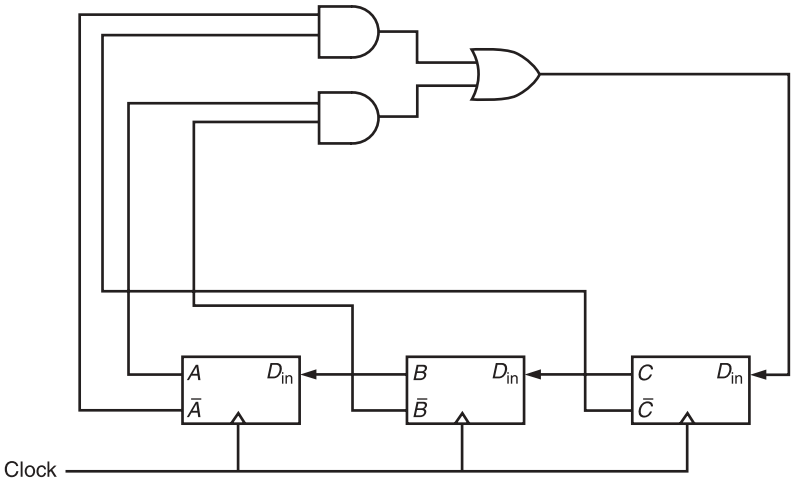
$$\text{Sequence I:} \quad D_C = \bar{A}\bar{C} + A\bar{B}$$

$$\text{Sequence II:} \quad D_C = A\bar{C} + \bar{A}\bar{B}$$

$$\text{Sequence III:} \quad D_C = \bar{A}$$

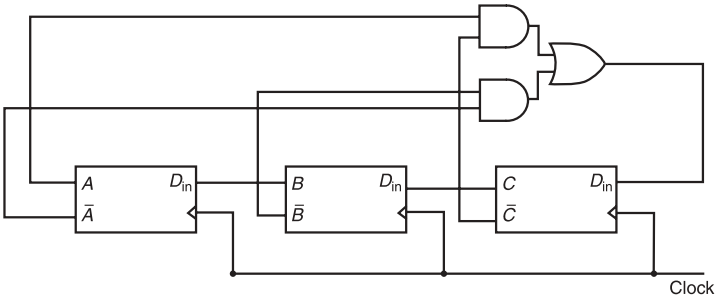
State 111 will produce a "lock up" in sequence 1(i.e., it will result in a 1 output from the feedback network, hence locking the shift register in state 111, from which it will not "return" to the main counting sequence). Hence the don't care term corresponding to state 111 in the Karnaugh map is ignored. For similar reasons, state 111 is not considered while minimizing the feedback expression for sequence II. The implementations of the counters based on state sequences I, II, and III are shown in Figure 8.19.

The design of shift counters based on a state sequence tree is not practicable using shift registers with more than four stages. However, a three-stage shift counter may be combined with a four-stage shift counter to form a composite counter having a state sequence length (i.e., mod number) higher than what is possible using either of the individual

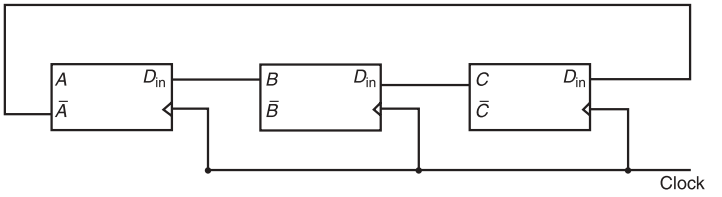


(*a*)

FIGURE 8.19 Implementations of (*a*) sequence I, (*b*) sequence II, and (*c*) sequence III.



(*b*)



(*c*)

FIGURE 8.19 (*Continued*).

counters. The sequence length of the composite counter is equal to the smallest number that is divisible by the sequence length of each of the individual counters. If the sequence

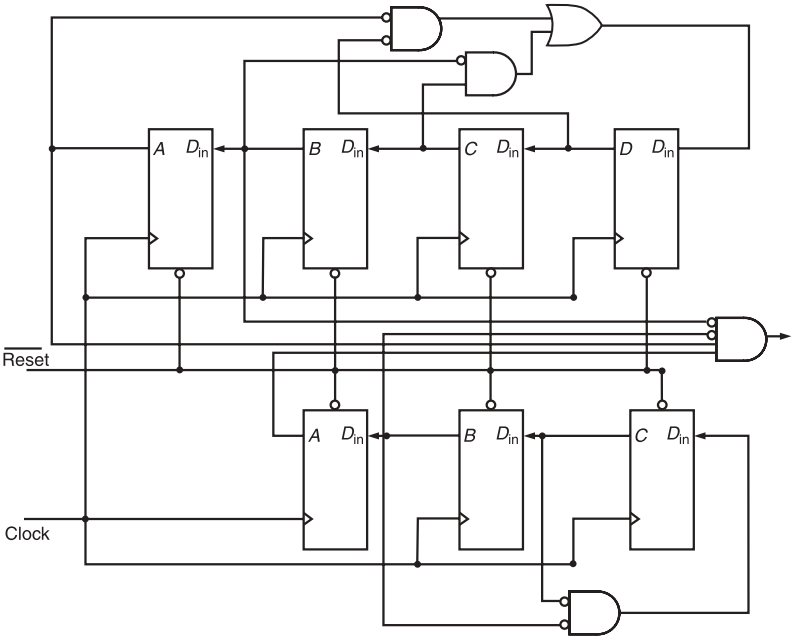


FIGURE 8.20 Mod-40 counter.

lengths have no common factors, then the sequence length of the composite counter is equal to their product.

For example, a mod-5 (three-stage) counter can be combined with a mod-8 (four-stage) counter to form a mod-40 counter, but a combination of a mod-5 and a mod-10 counter will give only a sequence length of 10, not 50. Figure 8.20 shows the implementation of a mod-40 counter based on a mod-5 and a mod-8 counter.

It is assumed that each counter produces a logic 1 output when it reaches its maximum count value. Thus the counter circuit of Figure 8.20 gives a 1 output when the mod-5 and mod-8 counters have settled in states 100 and 1000, respectively, indicating that the count sequence has been completed.

8.7 RING COUNTERS

A ring counter is basically a *circulating* shift register in which the output of the most significant stage is fed back to the input of the least significant stage. Figure 8.21 is a ring counter constructed from *D* flip-flops. The output of each stage is shifted into the next stage on the positive edge of a clock pulse. The "reset" signal clears all the flip-flops except the first one. Because flip-flop 1 is preset by a logic 1 on the reset line, the initial content of the counter is

A	*B*	*C*	*D*
0	0	0	1

The first positive clock edge shifts the output of flip-flop 4 into flip-flop 1 and the outputs of flip-flops 1, 2, and 3 into flip-flops 2, 3, and 4, respectively. Therefore the state of counter becomes

A	*B*	*C*	*D*
0	0	1	0

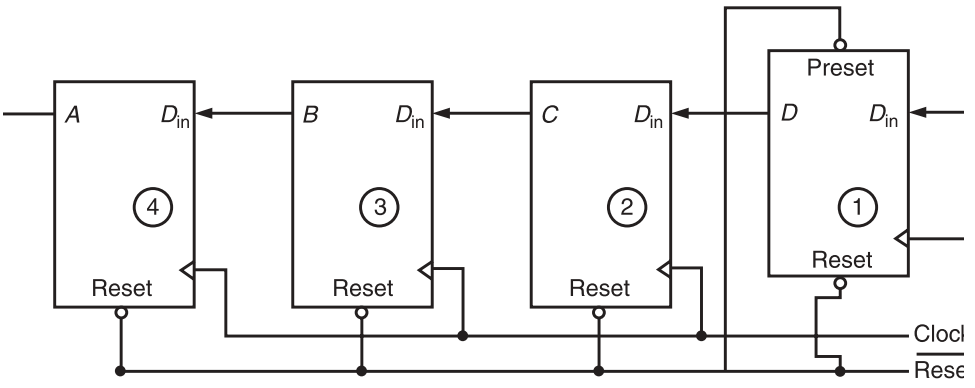


FIGURE 8.21 A 4-bit ring counter.

The second clock pulse changes the state to

A	B	C	D
0	1	0	0

At the end of the third clock pulse the counter state becomes

A	B	C	D
1	0	0	0

On the fourth clock pulse the 1 output from flip-flop 4 is transferred into flip-flop 1, thus

A	B	C	D
0	0	0	1

which is, of course, the initial state of the counter. Since the count sequence has four distinct states, the counter can be considered as a mod-4 counter. Thus the 4-bit ring counter includes only 4 of the 16 states that are posible using four flip-flops. Since an n-bit ring counter uses only n of its 2^n possible states, leaving $2^n - n$ states unused, it makes very inefficient use of flip-flops. For example, a decimal ring counter has only 10 counting states but requires 10 flip-flops, whereas a binary counter having 10 flip-flops will have 1024 ($=2^{10}$) states and can count up to 1023. The major advantage of ring counter over a binary counter is that it is *self-decoding* (i.e., no extra decoding circuit is needed to determine what state the counter is in). Each state is uniquely identified by a logic 1 at the output of the corresponding flip-flop. On the other hand, in an n-bit binary counter (except for a few count values) more than one flip-flop is on at a particular count, so additional gates are needed to generate a decoding signal for each state.

The ring counter technique can be utilized effectively to implement synchronous sequential circuits. A major problem in the realization of sequential circuits is the assignment of binary codes to the internal states of the circuit in order to reduce the complexity of circuits required (see Chapter 7). By assigning on flip-flop to one internal state, it is possible to simplify the combinational logic required to realize the complete sequential circuit. When the circuit is in a particular state, the flip-flop corresponding to that state is set to logic 1 and all other flip-flops remain reset.

Example 8.3 Let us design the sequential circuit described by the following state table:

	Input		
Present State	$x = 0$	$x = 1$	Output
	Next Stage		
A	F	C	0
B	D	C	1
C	A	D	0
D	A	E	1
E	C	F	0
F	C	G	1
G	B	G	0

Since the sequential circuit has seven states, a 7-bit ring counter is required. Let us assume that states A, B, C, D, E, F and G correspond to flip-flops 1, 2, 3, 4, 5, 6, and 7, respectively, in the ring counter. State A is the next state for both state C and state D when input $x = 0$.

Therefore whenever either flip-flop 3 or flip-flop 4 is set (i.e., the ring counter state is 0000100 or 0001000) and $x = 0$, flip-flop 1 should receive a logic 1 input signal, causing the counter to move to state 0000001. Assuming *JK* flip-flops are used to implement the ring counter, the excitation equations for flip-flop 1 are

$$J_1 = \bar{x}C + \bar{x}D$$
$$K_1 = 1$$

In other words, flip-flop 1 is set to logic 1 only if the counter is in state *C* or *D*; otherwise, it is reset. The excitation equations for the other flip-flops may be derived in a similar manner and are given by

$$\begin{aligned} J_2 &= \bar{x}G & K_2 &= 1 \\ J_3 &= \bar{x}E + \bar{x}F + xA + xB & K_3 &= 1 \\ J_4 &= \bar{x}B + xC & K_4 &= 1 \\ J_5 &= xD & K_5 &= 1 \\ J_6 &= \bar{x}A + xE & K_6 &= 1 \end{aligned}$$

The counter should move to state G whenever the present state is *F* and the input $x = 1$; hence the excitation equation for the *J* input of flip-flop 7 is

$$J_7 = xF$$

Furthermore, whenever the counter is in state *G* and $x = 1$, it should remain in stage *G*; therefore flip-flop 7 should not be reset for this input/state combination. Thus the excitation equation for the *K* input is

$$K_7 = \bar{x}$$

The output signal *Z* can be written

$$Z = B + D + F$$

The logic diagram of the complete sequential circuit is shown in Figure 8.22.

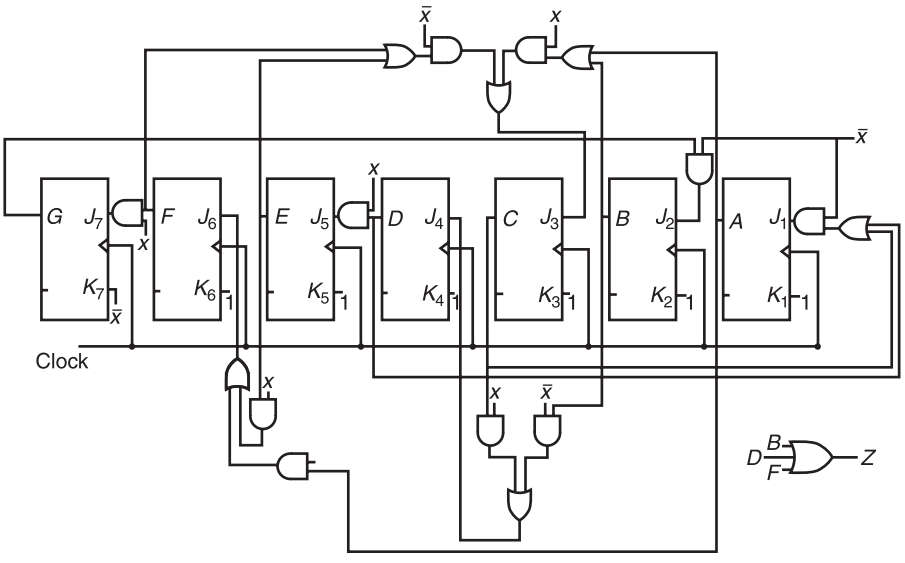


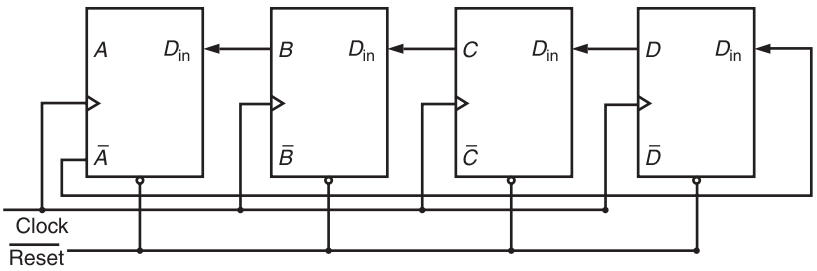
FIGURE 8.22 Implementation of seven-stage sequential circuit.

8.8 JOHNSON COUNTERS

Johnson counters are a variation of standard ring counters, with the inverted output of the last stage fed back to the input of the first stage. They are also known as *twisted ring* counters. An n-stage Johnson counter yields a count sequence of length $2n$, so it may be considered to be a mod-$2n$ counter. Figure 8.23*a* shows a 4-bit Johnson counter. The state sequence for the counter, assuming that initially all the flip-flops are reset to 0, is given in Figure 8.23*b*. At any count step only one flip-flop changes state, so in a Johnson counter a state can be decoded using a 2-input AND gate regardless of the number of flip-flops in the counter. For example, the combination $A = 0$ and $D = 0$ occurs only in one state (i.e., the first state of the count sequence shown in Figure 8.23*b*). Hence an AND gate with inputs $\bar{A}$ and $\bar{D}$ can be used to decode this state. The decoding function for each state of the 4-bit Johnson counter is shown in Figure 8.23*b*. Note that for exactly the same number of flip-flops a Johnson counter has twice the mod number of a ring counter. However, a Johnson counter requires decoding gates, whereas a ring counter does not. In general, an n-bit Johnson counter provides state sequences of even length, $2n$. However, it is possible to achieve count sequences of odd length, $2n - 1$, by using a 2-input gate for the feedback. A 4-bit *pseudo-Johnson* counter of this type and its count sequence are shown in Figure 8.24*a* and *b*, respectively.

Both the ring and Johnson counters must initially be forced into a valid state in the count sequence because they operate on a subset of the available number of states. Since a master reset signal is usually provided, this is usually not a problem; in the case of the ring counter, one of the flip-flops must be preset while the others are being reset. An alternative way to force these counters to a valid state is to use appropriate feedback logic. In the 3-bit Johnson counter shown in Figure 8.25*a*, the nonvalid states are 010 and 101. If the counter starts in one of these states, it will cycle through them indefinitely. The incorporation of feed back logic, $\bar{A}\ (\bar{B} + C)$, will make the counter self-correcting within two clock cycles (Fig. 8.25*b*).

Another apparent disadvantage of Johnson-type counters is that they are not very efficient for high count cycles. A mod-32 Johnson counter, for example, requires 16 flip-flops, whereas only 5 are required for a binary counter. However, by cascading a mod-4 and a mod-8 Johnson counter, it is possible to reduce the number of flip-flops to only 6. This arrangement is shown in Figure 8.26 The clock signal is applied to the mod-8 counter via the AND gate, which allows the clock signal to pass through only when the mod-4



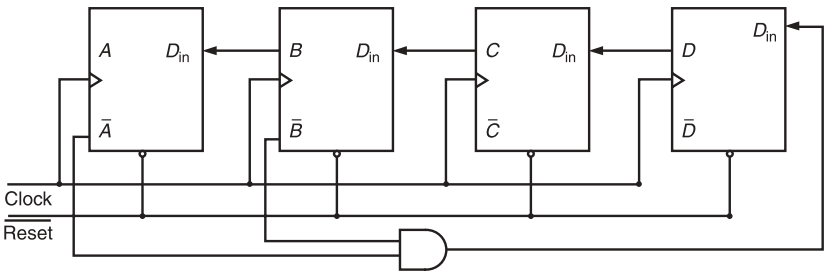
(*a*)

FIGURE 8.23 (*a*) A 4-bit Johnson counter and (*b*) the count sequence and state decoder for a 4-bit Johnson counter.

State				Decoder
A	B	C	D	
0	0	0	0	$\overline{A}\,\overline{D}$
0	0	0	1	$\overline{C}D$
0	0	1	1	$\overline{B}C$
0	1	1	1	$\overline{A}B$
1	1	1	1	AD
1	1	1	0	$C\overline{D}$
1	1	0	0	$B\overline{C}$
1	0	0	0	$A\overline{B}$

(*b*)

FIGURE 8.23 (*Continued*).



(*a*)

State			
A	B	C	D
0	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1
1	1	1	0
1	1	0	0
1	0	0	0

(*b*)

FIGURE 8.24 (*a*) A 4-bit pseudo-Johnson counter and (*b*) the count sequence.

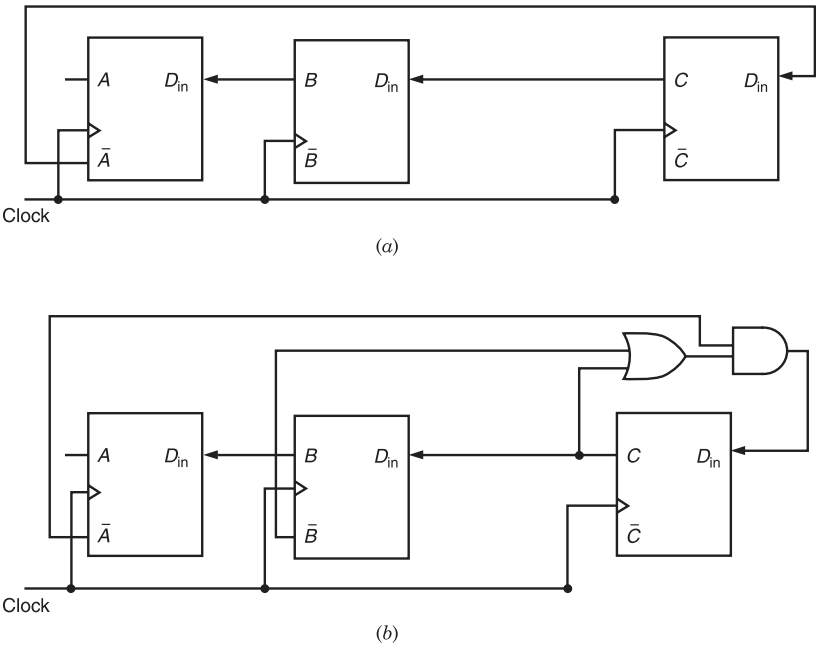


FIGURE 8.25 (*a*) A 3-bit Johnson counter and (*b*) a 3-bit self-correcting Johnson counter.

counter is in state $EF = 10$ (i.e., in the final state of its count sequence). The count sequence for the mod-32 counter is

A	B	C	D	E	F
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	0	1
		•			
		•			
		•			
1	0	0	0	1	0

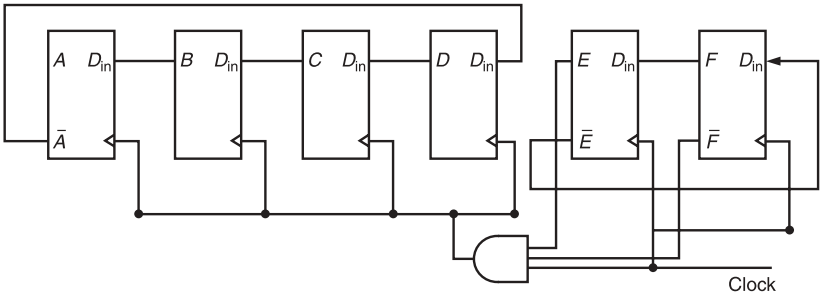


FIGURE 8.26 Mod-32 Johnson counter.

EXERCISES

1. Design a random counter to count the sequence 8, 1, 2, 0, 4, 6, 3, 5, 9, 7. Use *D* flip-flops.

2. Construct an 8-bit binary counter using *T* flip-flops as memory elements.

3. Design a ripple counter to count in excess-3 code.

4. Design a counter that can count either in mod-8 pure binary or in mod-8 Gray code, depending on a control signal being 1 or 0, respectively.

5. Implement a self-correcting counter that repeatedly generates the sequence 001, 110, 101, 010, 011. Use *D* flip-flops as memory elements.

6. Design a mod-12 counter using a shift register and appropriate feedback logic.

7. Implement a 4-bit ring counter using *JK* flip-flops, and show its count sequence.

8. Design a circuit that receives serially the output of the 4-bit ring counter in Exercise 7 and produces an output of 1 if the counter produces an illegal output combination.

9. Construct a 5-bit ring counter using *D* flip-flops. Combine the counter with a single *D* flip-flop to produce a decade counter.

10. In many applications binary ripple counters are found to be very slow. One possible approach to speed up counting is to use synchronous binary counters with carry-lookahead. Such a counter can be designed by generating a single carry-lookahead signal for each counter stage from the output of the previous stage. Derive the design equation for a 4-bit binary counter with carry-lookahead.

11. Counters are used in many designs to derive lower-frequency clock signals from the original clock signal. Show how a Johnson counter can be used to generate decoded signals of mod-2, mod-3, and mod-6 from an incoming clock signal.

12. As discussed in the text, an *n*-bit Johnson counter is a mod-2*n* counter. Show how a 4-bit Johnson counter can be converted into a mod-7 (i.e., odd modulo) counter by adding simple feed-back logic.

REFERENCE

1. J. Muth, “Designing shift counters,” *Semiconductors*, 4, 11–13 (1970).

9 Sequential Circuit Design Using VHDL

9.1 INTRODUCTION

The VHDL model of a circuit can be described at several levels (discussed in Chapter 5). Sequential circuits are typically modeled in VHDL at the behavioral level. A behavioral VHDL model of a sequential circuit focuses only on the functionality of the circuit. The actual structure of the sequential circuit is derived from the behavioral VHDL code by using a computer-aided logic design tool. The key to the behavioral level modeling is the *process* statement. As discussed in Chapter 6 a *process* can be considered as a software program that is executed in a sequential manner from top to bottom. Therefore the order of the statements in a process is important; all the statements are assumed to be executed with zero delay. All processes used in the architecture part of the VHDL model of a sequential circuit are executed concurrently, although the statements within a process are executed sequentially. In this chapter we describe memory elements as well as various sequential circuits using VHDL process statements.

9.2 *D* LATCH

A *D* latch is a *level-sensitive* device, that is, when the enable input of a latch is at logic 1, the input to the latch is copied to its output. The schematic symbol of a *D* latch is shown in Figure 9.1.

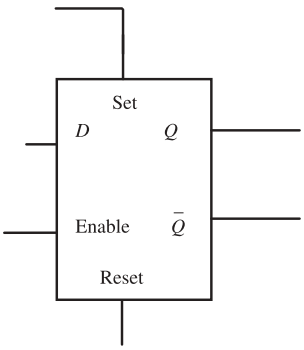


FIGURE 9.1 *D* latch.

Principles of Modern Digital Design, by Parag K. Lala

A behavioral level VHDL code of a *D* latch is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity d_latch is
 port (Enb, D: in std_logic;
       Q, Qbar: out std_logic);
end entity d_latch;

architecture behavior of d_latch is
begin
 p1: process (Enb, D)
  begin
    if Enb = '1' then
    Q <= D;
    Qbar <= not D;
  end if;
 end process;
end behavior;
```

Note that both inputs `D` and `Enb` are in the sensitivity list of the process. Thus the process is entered or activated (i.e., the statements within the process are executed) starting from the first one, each time either `Enb` or `D` changes value. If the `Enb` signal changes to logic 1 then `D` is copied to `Q`, and `Qbar` ($\overline{Q}$) takes the complement value of `D`. On the other hand, if the `Enb` signal changes to logic 0 then `D` cannot be copied to `Q`, and the latch retains its old value. Similarly, if signal `D` changes the process is also activated, and `Q` and Qbar ($\overline{Q}$) changes its value provided the `Enb` signal is at logic 1; otherwise, `Q` and $\overline{Q}$ retain their old values. The timing diagram shown in Figure 9.2 illustrates the operation of the *D* latch.

It should be mentioned that a process in the VHDL code does not require a label, but a label (e.g., `p1` for the process statement in the code for *D* latch) can increase the understandability of the VHDL code especially if more than one process is used in the architecture body of the code.

9.3 FLIP-FLOPS AND REGISTERS

9.3.1 *D* Flip-Flop

Synchronous or clocked sequential circuits use edge-triggered flip-flops as memory elements as discussed in Chapter 7. All changes in states of synchronous circuits occur on the *edge* of the clock signal used. There are several ways in which changes in a

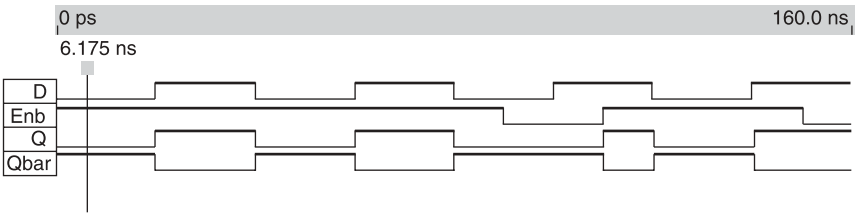


FIGURE 9.2 *D* latch timing diagram.

clock edge can be represented in VHDL code. To illustrate, the VHDL model of a positive edge-triggered *D* flip-flop is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity D_FF is
  port (D, clk: in std_logic;
        Q: out std_logic);
end entity D_FF;
architecture behavior of D_FF is
begin
  process (clk)
  begin
        if clk' event and clk = '1' then
        Q <= D;
      end if;
  end process;
end behavior;
```

The sensitivity list of the process contains only the `clk` (clock) signal since the output of a *D* flip-flop can change output only if `clk` is applied. The `clk' event and clk = '1'` condition in the if–then statement indicates that the `clk` signal has changed its value and become equal to 1. Thus the condition refers to a rising (i.e., 0-to-1) edge on the clock and is used to denote the output change in a positive-edge triggered *D* flip-flop. The `D` input is copied to the `Q` output of the flip-flop on the rising edge. Figure 9.3 shows the timing diagram of the *D* flip-flop.

A negative-edge triggered flip-flop can be specified by changing the condition in the if statement to `clk' event and clk ='0'`.

An alternative representation of the positive-edge triggered *D* flip-flop is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity D_FF is
  port (D, clk: in std_logic;
        Q: out std_logic);
end entity D_FF;
architecture behavior of D_FF is
begin
  process
   begin
```

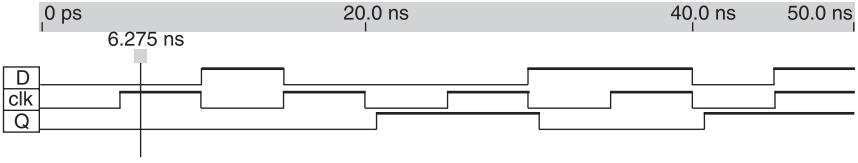


FIGURE 9.3 *D* flip-flop timing diagram.

```
        wait until clk= '1';
        Q <= D;
    end process;
end behavior;
```

In this description the `wait until clk = '1'` statement models the positive-edge trigger of the clock. The process does not require a sensitivity list in this case, the `wait until clk = '1'` statement automatically implies that the flip-flop is triggered on the rising edge of the clock. Similarly, a negative-edge triggered flip-flop can be described by changing `clk = '1'` in the `wait until` statement to `clk ='0'`.

Either a `wait until clk = '1'` or an `if clk' event and clk = '1'` statement can be used to define a clock edge; however, the latter is preferred because it is recognized by most computer-aided logic design tools.

9.3.2 *T* and *JK* Flip-Flops

A *T* flip-flop and a *JK* flip-flop change their current states on a clock edge if their inputs change. In both of these flip-flops the state can be defined as a variable or a signal. The VHDL descriptions for these flip-flops are as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity T_FF is
port (T, clk, reset: in std_logic;
Q, Qbar: out std_logic);
end T_FF;
architecture behavior of T_FF is
signal temp: std_logic;
begin
process (clk, reset) is
begin
if (reset = '1') then
temp <= '0';
elsif (clk'event and clk = '1') then
 if T = '1' then
 temp <= not temp;
 end if;
 end if;
end process;
Q <= temp;
Qbar <= not temp;
end behavior;
```

The right side of "<=" of an assignment statement in a process can be considered as an input and the left side as an output of a *D* flip-flop, and the input is transferred to the output when the clock signal is activated. Thus only the last values on the right side of

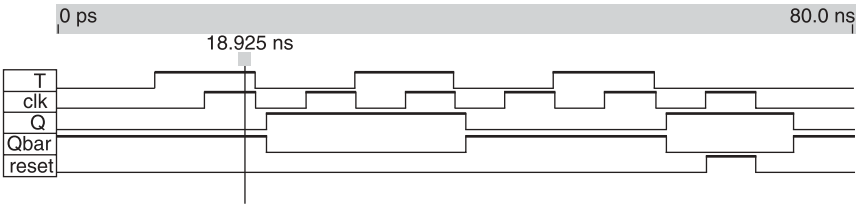


FIGURE 9.4 Timing diagram of the *T* flip-flop.

the assignment statements before the clock signal is activated are the valid outputs generated at the end of the process statement. The timing diagram of the *T* flip-flop is given in Figure 9.4.

The VHDL description for the *JK* flip is as follows:

```
library ieee;
use ieee.std_logic_1164.all;

entity JK_FF is
port (J, K, clk, Reset: in std_logic;
Q, Qbar: out std_logic);
end JK_FF;

architecture behavior of JK_FF is
signal temp: std_logic;
begin
process (clk, Reset) is
begin
if (Reset = '1') then
   temp <= '0';
  elsif (clk'event and clk = '1') then
    if (J = '1' and K = '1') then
    temp <= not temp;
    elsif (J = '1' and K = '0') then
    temp <= '1';
    elsif (J ='0' and K = '1') then
    temp <= '0';
    else
    temp <= temp;
  end if;
 end if;
end process;
Q <= temp;
Qbar <= not temp;
end behavior;
```

The timing diagram of the *JK* flip-flop is given in Figure 9.5.

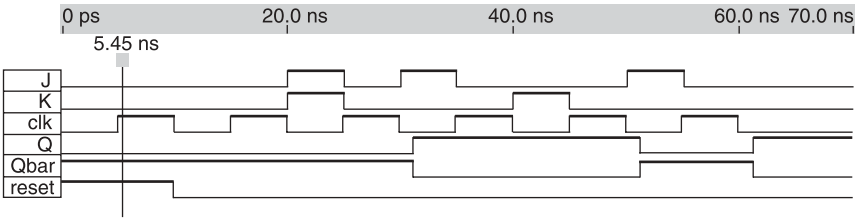


FIGURE 9.5 Timing diagram of *JK* flip-flop.

9.3.3 Synchronous and Asynchronous Reset

The outputs of flip-flops in a circuit can take either a 0 or 1 value when power is applied to the circuit. The flip-flop outputs can be forced to an all 0 state by applying a reset signal to the circuit. A reset is *synchronous* if it becomes activated only after the flip-flop is clocked. The VHDL code for a *D* flip-flop with synchronous reset is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity D_FF is
  port (D, Clk, Reset: in std_logic;
        Q: out std_logic);
end D_FF;
architecture behavior of D_FF is
begin
  process (clk)
   begin
      if clk' event and clk = '1' then
      if reset= '1' then
          Q <= '0';
      else
         Q <= D;
     end if;
   end if;
 end process;
end behavior;
```

Note that the reset input is not included in the sensitivity list because the output of the flip-flop changes value only on a clock transition. Figure 9.6 shows the timing diagram of a synchronous reset flip-flop.

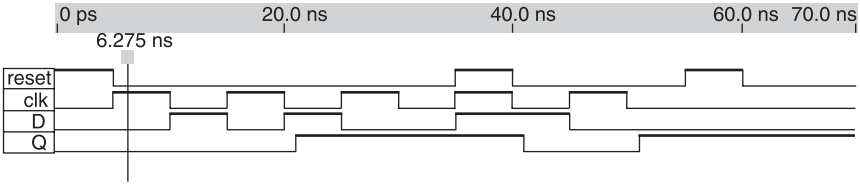


FIGURE 9.6 Timing diagram of a synchronous reset *D* flip-flop.

In an asynchronous reset the flip-flop output is affected by both the reset signal and the clock. Therefore the process statement in the VHDL description of the *D* flip-flop with asynchronous reset must include both reset and clock signals in its sensitivity list:

```
library ieee;
use ieee.std_logic_1164.all;
entity DFF is
  port (D, clk, reset: in std_logic;
        Q: out std_logic);
end DFF;
architecture behavior of DFF is
begin
   process (clk, reset)
   begin
      if reset = '1' then
         Q <= '0';
       elsif clk' event and clk = '1' then
         Q <= D;
   end if;
 end process;
end behavior;
```

In the code above it is assumed that flip-flop output `Q` becomes 0 when the reset signal is *active-high*. If the reset signal is *active-low* the architecture part of the VHDL code for the *D* flip-flop has to be modified:

```
architecture behavior of DFF is
begin
   process (clk, reset)
   begin
      if reset = '0' then
        Q <= '0';
      elsif clk' event and clk = '1' then
        Q <= D;
    end if;
  end process;
end behavior;
```

The timing diagram of the flip-flop with active-low asynchronous reset is shown in Figure 9.7.

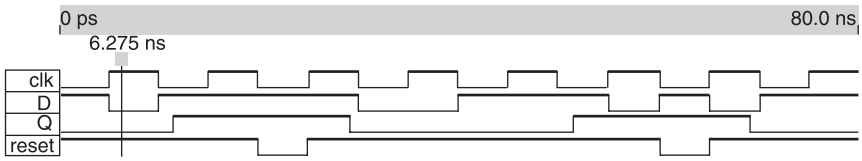


FIGURE 9.7 *D* flip-flop with active-low asynchronous reset.

9.3.4 Synchronous and Asynchronous Preset

A preset signal sets the output of a flip-flop to logic 1. If the preset is asynchronous the output changes value irrespective of the status of the clock; on the other hand, for the synchronous preset the change in the output happens in conjunction with the clock signal. The code for a *D* flip-flop with asynchronous reset and synchronous preset is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity SRDFF is
port (D, clk, preset, reset: in std_logic;
      Q: out std_logic);
end SRDFF;
architecture behavior of SRDFF is
begin
process (clk, reset)
begin
if reset = '1' then
   Q <= '0';
   elsif clk' event and clk = '1' then
   if preset= '1' then
      Q <= '1';
   else
      Q <= D;
    end if;
   end if;
  end process;
end behavior;
```

The timing diagram of the flip-flop is shown in Figure 9.8.

9.3.5 Registers

An *n*-bit register is formed by cascading *n* flip-flops. The VHDL code for a 4-bit register is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
```

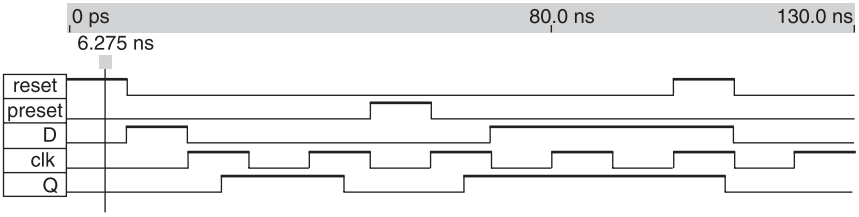


FIGURE 9.8 Timing diagram of synchronous preset and asynchronous reset *D* flip-flop.

```
entity reg_4 is
    port (reset, clk: in std_logic;
          D: in std_logic_vector (3 downto 0);
          Q: out std_logic_vector (3 downto 0));
end entity reg_4;
architecture behavior of reg_4 is
begin
   process (reset,clk)
   begin
      if reset = '1' then
         Q <= "0000";
      elsif clk' event and clk = '1' then
         Q <= D;
      end if;
    end process;
end behavior;
```

Both D and Q are defined to be 4-bit data. The sensitivity list of the process statement includes both reset and clk. As discussed earlier this indicates register Q is reset asynchronously; thus when the reset signal is active-high Q is assigned "0000" (i.e., all 4 bits of the register are assigned 0's). For a register of arbitrary length a more efficient way to define that all bits of a register are to be reset to 0's is to assign

```
(others => '0');
```

to the register. For example, if Q is a 64-bit wide register the statement to reset Q is

```
Q <= (others => '0');
```

An *n*-bit register can be formed by defining D and Q as

```
D: in std_logic_vector (n-1 downto 0);
Q: out std_logic_vector (n-1 downto 0)
```

and adding the statement

```
generic (n: integer:= value);
```

in the entity description of the register; the parameter *value* is an integer. For example, a 32-bit register can be formed by setting parameter *value* to 32 in the *generic* statement as follows:

```
entity register is
  generic (n: integer:= 32);
  port (reset, clk: in std_logic;
        D: in std_logic_vector (n-1 downto 0);
        Q: out std_logic_vector (n-1 downto 0));
end register;
```

```
architecture behavior of register is
             begin
                process (reset, clk)
                begin
                   if reset = '1' then
                      Q <= (others => '0');
                   elsif clk' event and clk = '1' then
                     Q <= D;
                   end if;
               end process;
            end behavior;
```

The above VHDL code for an n (=32)-bit register can be used to describe a register of any size by changing the value of parameter n. The statement `Q <= (others => '0')` indicates that all the bits in vector `Q` are reset to 0. The syntax `others =>` is typically used to assign `'0'`, `'1'`, `'X'`, or `'Z'` to a vector of arbitrary length. For example, this syntax is used in the code for an 8-bit tri state buffer to indicate the outputs of the buffer are in a high impedance (`Z`) state when the enable signal is not asserted:

```
library ieee;
use ieee.std_logic_1164.all;
entity trireg is
generic (n: integer:= 8);
  port (D: in std_logic_vector (0 to n-1);
       enb: in std_logic;
         Q: out std_logic_vector (0 to n-1));
         end trireg;
architecture behavior of trireg is
begin
process(enb, D)
begin
  if enb = '1' then
     Q <= D;
   else
     Q <= (others => 'Z');
   end if;
end process;
end behavior;
```

9.4 SHIFT REGISTERS

Several forms of shift registers have been considered in Chapter 7. The VHDL code for a parallel load and serial in/serial out 4-bit left-to-right shift register is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity ltorshiftreg is
  port (R: in std_logic_vector(3 downto 0);
```

```
clk, lin, load: in std_logic;
D: buffer std_logic_vector (3 downto 0));
end ltorshiftreg;
architecture behavior of ltorshiftreg is
 begin
  process (clk)
   begin
    if clk' event and clk = '1' then
       if load = '1' then
          D <= R;
       else
          D(0) <= D(1);
          D(1) <= D(2);
          D(2) <= D(3);
          D(3) <= lin;
       end if;
   end if;
  end process;
end behavior;
```

External data is loaded into the shift register when the load signal is 1; this is a synchronous load since data is stored only on a clock edge after the load signal is set at 1. The above code can be modified to form a right-to-left shift register:

```
library ieee;
use ieee.std_logic_1164.all;
entity rtolshiftreg is
  port (R: in std_logic_vector(3 downto 0);
  clk, lin, load: in std_logic;
  D: buffer std_logic_vector (3 downto 0));
end rtolshiftreg;
architecture behavior of rtolshiftreg is
 begin
 process (clk)
 begin
 if clk' event and clk = '1' then
  if load = '1' then
    D <= R;
  else
    D(3) <= rin;
    D(2) <= D(3);
    D(1) <= D(2);
    D(0) <= D(1);
   end if;
  end if;
  end process;
end behavior;
```

9.4.1 Bidirectional Shift Register

In a left-to-right shift register each left-shift operation (i.e., a shift toward the most significant bit) in effect multiplies the stored data by 2. On the other hand, each right-shift operation in a right-to-left shift register (i.e., a shift toward the least significant bit) divides the stored number by 2. It is also possible to design a serial shift register that can perform both operations. Such a shift register is known as a *bidirectional* or *reversible* shift register in which the data can be shifted either left or right. The VHDL code for an n-bit bidirectional shift register is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity bishift is
generic(n: natural := 8);
port(clk, dL, dR: in std_logic;
dir,load: in std_logic;
data: in std_logic_vector(0 to n-1);
Q: out std_logic_vector(0 to n-1));
end bishift;
architecture behavior of bishift is
signal reg: std_logic_vector (0 to n-1);
begin
process(clk)
begin
if clk' event and clk = '1' then
   If load = '1' then
   reg <= data;
   else
   if dir = '0' then reg <= (dR & reg (0 to (n-2)));
   else
   reg <= (reg (1 to (n-1))& dL);
   end if;
   end if;
end if;
end process;
Q <= reg;
end behavior;
```

In this code the shift direction is determined by control input `dir`:

$$\begin{aligned} \texttt{dir} = 0 \quad & \text{shift right} \\ = 1 \quad & \text{shift left} \end{aligned}$$

The right shift operation is done by concatenating the leftmost input `dR` and the lowest $(n-1)$ bits. Similarly, the left shift operation is done by concatenating the rightmost input `dL` and the upper $(n-1)$ bits. Notice that `reg` is declared as an 8-bit *signal*.

TABLE 9.1 Function Modes of a Universal Shift Register

s_1	s_0	Operation
0	0	Hold data
0	1	Shift right
1	0	Shift left
1	1	Load data

9.4.2 Universal Shift Register

A more general shift register, known as a *universal shift register*, can shift data from right-to-left and left-to-right, can hold the current data, and can load data in parallel. The modes of operation of universal shift register are selected as shown in Table 9.1.

The behavioral-level VHDL code for an n-bit universal shift register is shown below; it is assumed $n = 8$.

```
library ieee;
use ieee.std_logic_1164.all;
entity unishift is
generic(n: natural:= 8);
port(clk, dL, dR, rst: in std_logic;
s: in std_logic_vector (1 downto 0);
data: in std_logic_vector (n-1 downto 0);
Q: out std_logic_vector (n-1 downto 0));
end unishift;
architecture behavior of unishift is
signal reg: std_logic_vector (n-1 downto 0)
begin
process(clk, rst)
begin
if (rst = '0') then
 reg <= (others => '0');
elsif clk' event and clk = '1' then
if (s = "01") then reg <= (dL & reg ((n-1) downto 1));
  elsif (s = "10") then reg <= (reg ((n-2) downto 0)& dR);
  elsif (s = "11") then reg <= data;
  end if;
end if;
end process;
Q <= reg;
end behavior;
```

9.4.3 Barrel Shifter

A barrel shifter allows shifting or rotating of data by multiple bits in one clock cycle; the number of data bits that can be shifted is specified via the control inputs. For an n-bit data the length of control bits is $\log_2 n$. The shifted-in bits are all 0's and shifted-out bits are lost.

TABLE 9.2 Modes of Operation of 4-Bit Barrel Shifter

s_1	s_0	Number of Bits Shifted
0	0	No shift
0	1	1 bit
1	0	2 bits
1	1	3 bits

The VHDL code for a 4-bit left-to-right barrel shift register is given below. The length of the control bits is 2 and is identified by the 2-bit vector (s_1s_0) in the code. The number of bits shifted in one operation is selected by the control bits s_1 and s_0 as shown in Table 9.2.

```
library ieee;
use ieee.std_logic_1164.all;
entity barrelshift is
port(clk,load: in std_logic;
s: in std_logic_vector (1 downto 0);
data: in std_logic_vector(7 downto 0);
Q: buffer std_logic_vector(7 downto 0));
end barrelshift;
architecture behavior of barrelshift is
begin
 process (clk,load)
 begin
if load = '1' then
Q <= data;
elsif (clk' event and clk = '1') then
case s is
  when "00" => Q <= Q;
  when "01" => Q <= Q(6 downto 0)& '0';
  when "10" => Q <= Q(5 downto 0)& "00";
  when "11" => Q <= Q(4 downto 0)& "000";
 end case;
 end if;
 end process;
end behavior;
```

Note that `Q` has been declared as a `buffer std_logic vector`. Therefore a signal statement defining a new register for storing the results produced during the execution of the process statement is not necessary. In the code for the universal shift register this was necessary because `Q` was declared as an `out std_logic vector`. It should also be pointed out `when others` is not strictly necessary in this code for logic synthesis since all of the 2-bit Boolean combinations of `s` in the `case` statement are listed. However, as indicated earlier in Chapter 6 if all possible values of the expression in the `case` statement are not covered, `when others` must be used to specify output values for all left-over input combinations; otherwise either unintended latches or feedback paths in the resulting circuit will be introduced.

9.4.4 Linear Feedback Shift Registers

Linear feedback shift registers (LFSR) are used to generate pseudo-random numbers that can be high quality audio noise and also test patterns in logic circuit testing. A linear feed back shift register is formed by connecting the outputs of a selected number of stages in a shift register to its input through an EX-OR network. Since the feedback path involves only EX-OR operation (i.e., mod-2 addition of the chosen stages of the shift register), it is said to be a linear feedback shift register (LFSR). The output of any stage in an LFSR is a function of the initial state of the bits in the register and of the outputs of the states that are fed back. Thus the selection of feedback paths is crucial in constructing an LFSR that performs as required.

An *n-bit* LFSR may be represented by a polynomial that is irreducible and primitive. A polynomial that cannot be factored is called irreducible. For example, the polynomials $f(x) = x^3 + x + 1$ and $g(x) = x^3 + 1$ are irreducible polynomials of degree 3; the degree of a polynomial is the largest superscript in the polynomial. A polynomial $p(x)$ of degree n is primitive if the remainders generated from the divisions of all polynomials with degree $c(\leq 2^n - 1)$ by $p(x)$ correspond to all possible nonzero polynomials of degree less than n. For example, the division of x, x^2, x^3, . . . , x^7 by polynomial $g(x)$ results in the same three remainders 1, x, and x^2 repeatedly. On the other hand, if we divide x, x^2, x^3, x^4, x^5, x^6, x^7 by polynomial $f(x)$, the remainders obtained are x, x^2, $x + 1$, $x^2 + x$, $x^2 + x + 1$, $x^2 + 1$, and 1, respectively. In other words, the division of all polynomials of degree 1 to 7 by $f(x)$ results in all seven nonzero polynomials of degree less than 3. Thus although both polynomials $f(x)$ and $g(x)$ are irreducible, only $f(x)$ is primitive.

If an LFSR is implemented using $f(x)$ it will repeatedly generate seven binary patterns in sequence. In general, if the polynomial used to construct an LFSR is of degree n, then the LFSR will generate all possible $2^n - 1$ nonzero binary patterns in sequence; this sequence is termed the *maximal length sequence* of the LFSR.

Figure 9.9 shows the general representation of an LFSR based on the primitive polynomial

$$a(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_2x^2 + a_1x + a_0 \tag{9.1}$$

The feedback connections needed to implement an LFSR can be derived directly from the chosen primitive polynomial. To illustrate, let us consider the following polynomial of

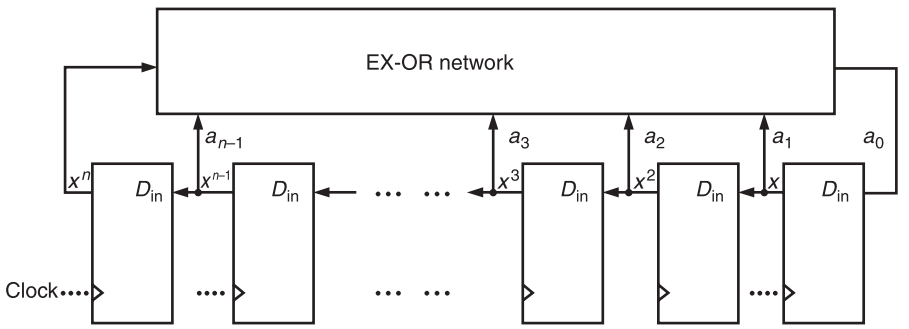


FIGURE 9.9 General representation of an LFSR.

degree 4:

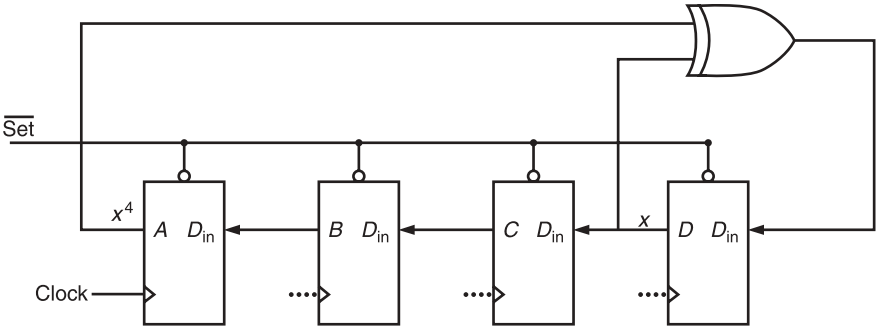
$$x^4 + x + 1$$

This can be rewritten in the form of expression (9.1):

$$a(x) = 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^1 + 1 \cdot x + 1 \cdot x^0$$

If the coefficient of x^i is 1, there is feedback from stage i, whereas if the coefficient is 0, the output of stage i is not connected to the EX-OR network. Figure 9.10*a* and 9.10*b* show the four-stage LFSR constructed by using the above polynomial and the corresponding maximal length sequence, respectively.

It is possible to construct LFSRs from certain shift registers by using feedback paths from only two stages; a partial list of such registers is given in Table 9.3 [2]. All of these LFSRs produce maximal length sequences.



(*a*)

A	*B*	*C*	*D*
1	1	1	1
1	1	1	0
1	1	0	1
1	0	1	0
0	1	0	1
1	0	1	1
0	1	1	0
1	1	0	0
1	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0
0	0	0	1
0	0	1	1
0	1	1	1

(*b*)

FIGURE 9.10 (*a*) A 4-bit LFSR and (*b*) maximal length sequence.

TABLE 9.3 Maximal Length LFSRs with Two Feedback Paths

Number of Stages	Stages from Which Feedback Paths are Derived (Corresponding Polynomial)	Length of Sequence
7	1, 7 $(x^7 + x + 1)$	127
	or 3, 7 $(x^7 + x^3 + 1)$	
9	4, 9 $(x^9 + x^4 + 1)$	511
10	3, 10 $(x^{10} + x^3 + 1)$	1,023
11	2, 11 $(x^{11} + x^2 + 1)$	2,047
15	1, 15 $(x^{15} + x^4 + 1)$	32,767
	or 4, 15 $(x^{15} + x^4 + 1)$	
	or 7, 15 $(x^{15} + x^7 + 1)$	
17	3, 17 $(x^{17} + x^3 + 1)$	131,071
	or 5, 17 $(x^{17} + x^5 + 1)$	
	or 6, 17 $(x^{17} + x^6 + 1)$	
18	7, 18 $(x^{18} + x^7 + 1)$	262,143
20	3, 20 $(x^{20} + x^3 + 1)$	1,048,575

The VHDL code of a LFSR of four stages is given below. Table 9.3 shows that by using feedback paths from only two stages a LFSR can provide maximal length sequences. The `seed` value for the LFSR is declared as a constant; any pattern except all 0's can be the seed or the starting pattern. An all 0 pattern is not used because it will lock the LFSR in that state. The LFSR goes through a sequence of 15 states and after that the sequence repeats itself. By changing the `seed` the LFSR can be made to go through a different sequence of 15 states.

```
library ieee;
use ieee.std_logic_1164.all;
entity lfsr is
port (clk, load: in std_logic;
d: buffer std_logic_vector (4 downto 1));
end lfsr;
architecture behavior of lfsr is
constant seed: std_logic_vector (4 downto 1):= "0010";
begin
  process (clk, load)
  variable feedback: std_logic;
       begin
if load = '1' then
  d <=  seed;
  elsif (clk' event and clk = '1') then
  feedback:= d(4) xor d(1);
  d <= d(3 downto 1)& feedback;
  end if;
end process;
end behavior;
```

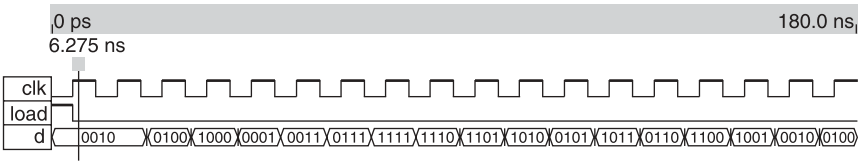


FIGURE 9.11 Simulation results of the 4-bit LFSR.

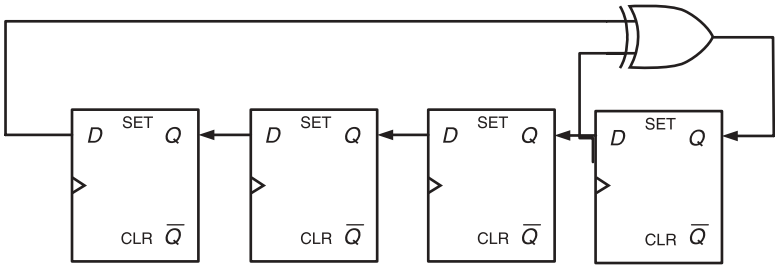


FIGURE 9.12 A 4-bit LFSR.

The simulation results of the LFSR obtained by assuming seed value 0010 are shown in Figure 9.11. The implementation of the LFSR is shown in Figure 9.12.

9.5 COUNTERS

A variety of counters are also utilized in digital systems; in Chapter 8 many types of counters were discussed. The VHDL code for a 4-bit up counter is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity upcounter is
port (clk, reset: in std_logic;
Q: out std_logic_vector (3 downto 0));
end upcounter;
architecture behavior of upcounter is
signal count: std_logic_vector (3 downto 0);
begin
   process (clk, reset)
   begin
if reset = '1' then
   count <= (others => '0');
elsif clk' event and clk = '1' then
   count <= count + 1;
end if;
end process;
Q <= count;
end behavior;
```

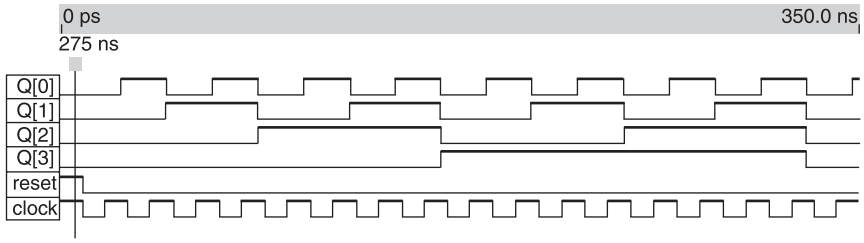


FIGURE 9.13 Simulation results of 4-bit up counter.

The library declaration `use ieee.std logic unsigned.all` in this description allows the use of `std_logic vector` signals as unsigned binary numbers that can be incremented or decremented as required. The signal statement is not necessary if `Q` is declared as `buffer std_logic vector`. Note that the `elsif` statement does not include an `else` clause because the count retains its previous value if the clock is not activated. The simulation results of the 4-bit up counter are shown in Figure 9.13.

A down counter can be described in a similar manner simply by changing the statement `count <= count + 1` to `count <= count-1`. The VHDL description of a generic up/down counter with parallel load capability is as follows:

```
library ieee;
use ieee.std logic 1164.all;
use ieee.std logic unsigned.all;
entity udcounter is
generic(n: integer: =16);
port (clk, reset, updn, load: in std_logic;
R: in std_logic_vector (n-1 downto 0));
Q: buffer std_logic_vector (n-1 downto 0));
end udcounter;
architecture behavior of udcounter is
begin
    process (clk, reset)
    begin
if reset='1' then
   Q <= (others => '0');
elsif clock' event and clk='1' then
   if load= '1' then
     Q <= R;
   elsif updn = '0' then
     Q <= Q+1;
   else
      Q  <= Q-1;
   end if;
 end if;
   end process;
end behavior;
```

Parallel data R and count value Q in the VHDL representation of a counter can be represented as integers instead of logic vectors. As an example, the code for a mod-6 up–down counter is

```
entity udcounter is
port (clock, reset, updn, load: in std_logic;
R: in integer range 0 to 5;
Q: buffer integer range 0 to 5);
end udcounter;
architecture behavior of udcounter is
begin
   process (clk, reset)
   begin
if reset = '1' then
   Q <= 0;
elsif clk' event and clk = '1' then
   if load = '1' then
        Q <= R;
      elsif updn = '0' then
         if Q=6 then
            then Q <= 0;
   else Q  <= Q+1;
        end if;
     else
     if Q = 0 then
         then Q <= 6;
         else Q  <= Q-1;
        end if;
     end if;
           end if;
          end if;
         end if;
     end process;
end behavior;
```

9.5.1 Decade Counter

The decade counter counts up to 9 starting from 0 and then automatically resets to 0; the count is incremented on each clock edge. The counter can be forced to an all 0 state by making the reset line high. The VHDL code for the decade counter is as follows:

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity decade_counter is
port(clk: in std_logic;
       reset: in std_logic;
       q: out std_logic_vector(3 downto 0));
end decade_counter;
```

```
architecture behavior of decade_counter is
signal temp: std_logic_vector(3 downto 0);
begin
process(clk, reset)
  begin
  if reset = '1' then
  temp <= (others => '0');
  elsif (clk'event and clk = '1') then
   if temp < 9 then
   temp <= temp + '1';
   else
   temp <= ''0000'';
end if;
end if;
end process;
q <= temp;
end behavior;
```

The simulation results of the decade counter are shown in Figure 9.14.

9.5.2 Gray Code Counter

A Gray code has the property that any two codewords are *adjacent* (i.e., differ by only 1 bit) (see Chapter 1). An n-bit Gray code counter has 2^n states; only one of the state bits changes during the transition from a present state to a next state. The VHDL code for a 4-bit Gray code counter is given below. The counter is basically a 4-bit up counter with its outputs converted to a Gray code. The counter can be loaded with an initial value (0 to 15) when the `load` signal is set to 0. The loaded value is converted to a Gray codeword before the counting process starts.

```
library ieee;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity graycounter is
port (reset, clk, load: in std_logic;
     D: in std_logic_vector(3 downto 0);
     Q: buffer std_logic_vector(3 downto 0);
     gray: out std_logic_vector(3 downto 0));
end graycounter;
```

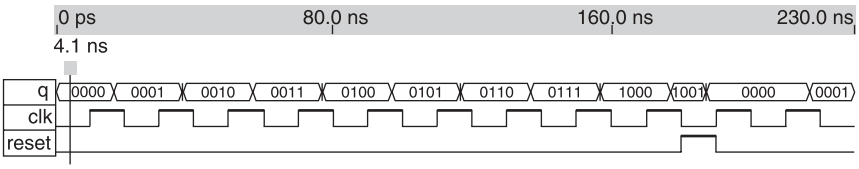


FIGURE 9.14 Simulation results of the decade counter.

```
architecture behavior of graycounter is
begin
process (reset, load, clk)
begin
  if (reset = '1') then
  Q <= (others => '0');
  elsif (load = '0') then
  Q <= D;
  elsif clk'event and clk = '1' then
  Q <= Q+1;
  end if;
end process;
gray(3) <= Q(3);
gray(2) <= Q(2) xor Q(3);
gray(1) <= Q(2) xor Q(1);
gray(0) <= Q(1) xor Q(0);
end behavior;
```

The code uses an `ieee.std_logic_unsigned.all` statement, which allows arithmetic operations on unsigned numbers in the code. The simulation results of the 4-bit Gray code counter are shown in Figure 9.15. The number 1011 is loaded into the counter first, which is converted into 1110 before the counting sequence begins. The binary equivalent of each Gray code count value is also shown in the diagram.

9.5.3 Ring Counter

A ring counter has as many states as the number of flip-flops in it. A VHDL description of an *n*-bit ring counter with *n* set to 4 is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity ringcounter is
generic (n: integer: = 4);
port (start,clk: in std_logic;
     Q: buffer std_logic_vector(0 to n-1));
end ringcounter;
```

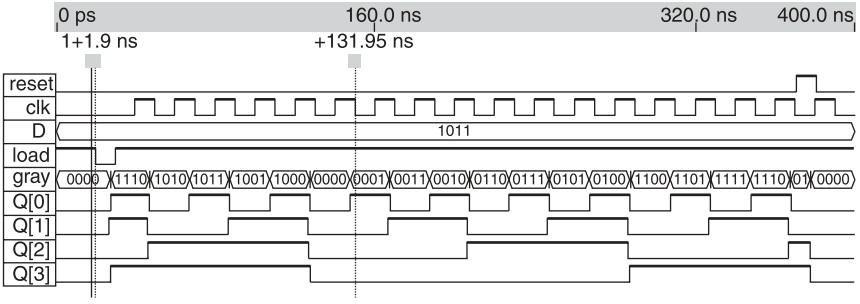


FIGURE 9.15 Simulation results of 4-bit Gray code counter.

```
architecture behavior of ringcounter is
begin
 process (clk,start)
 begin
 if start = '1' then
 Q(0) <= '1';
 Q(1 to n-1) <= (others => '0');
 elsif (clk'event and clk = '1') then
 shift: for i in 1 to n-1 loop
 Q(i) <= Q(i-1);
 end loop;
 Q(0) <= Q(n-1);
 end if;
end process;
end behavior;
```

Note that this code uses a *for loop* statement. As indicated in Chapter 6, a *for loop* statement repeats a sequence of statements for an explicitly stated number of times; a loop label is optional but is recommended to clarify the function of the loop. The statement inside the loop, `shift`, in the above code transfers the content of register `Q` left to right and is iterated $n-1(=3)$ times. The simulation results of the 4-bit ring counter are shown in Figure 9.16.

9.5.4 Johnson Counter

The VHDL specification for a Johnson counter can be obtained by setting the left most bit of the `Q` register to 1 and the remaining bits to 0, thereby making 1000 as the staring state of the counter. The VHDL description for an *n*-bit Johnson counter is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity johncount is
generic (n: integer: = 4);
port (clk, reset: in std_logic;
     Q: buffer std_logic_vector(0 to n-1));
end johncount;
architecture behavior of johncount is
begin
  process (clk, reset)
```

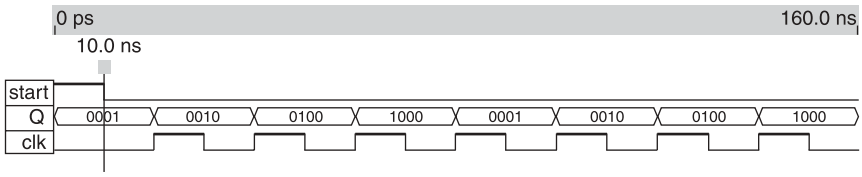


FIGURE 9.16 Functional simulation of 4-bit ring counter.

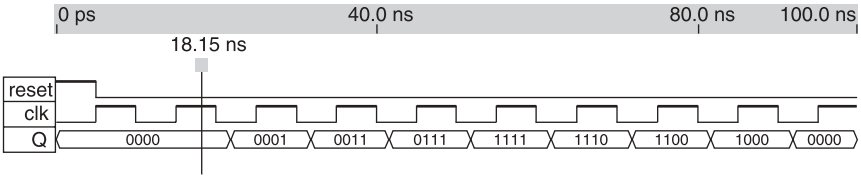


FIGURE 9.17 Function simulation of 4-bit Johnson counter.

```
  begin
       if reset = '1' then
           Q <= (others => '0');
           elsif (clk' event and clk = '1') then
shift: for i in 1 to n-1 loop
           Q(0) <= not Q(n-1);
         Q(1 to n-1) <= Q(0 to n-2);
end loop;
end if;
end process;
end behavior;
```

Note that the first statement within the *for loop* transfers the value of the most significant bit of the counter to the least significant bit. This statement can also be put just after the `end loop` clause. The simulation results of the 4-bit Johnson counter are given in Figure 9.17.

9.6 STATE MACHINES

State machines or sequential circuits can be either Moore type or Mealy type (see Chapter 7). In a Moore machine the output is a function of the present state only; therefore the output can change only on a clock transition (Fig. 9.18*a*). In a Mealy machine, on the other hand, the output is a function of the current state as well as the current input and may change if the input changes. Thus the output in a Mealy machine unlike in a Moore machine can change even if the clock signal does not change (Fig. 9.18*b*).

9.6.1 Moore-Type State Machines

We will consider VHDL code generation for Moore-type state machines first. As discussed in Chapter 7, the state register holds the present state of the state machine. The output of the state register drives the output logic block and is also fed back to the input of the next state logic block. A straightforward implementation of the Moore state machine will be to map each block in Figure 9.18*a* into a process. To illustrate, let us consider the state machine shown in Figure 9.19.

The EX-OR gate is the next state logic; it can be represented by process `p1`. The *D* flip-flop is the state register and state transitions in the flip-flop are described by process `p2`. The output logic consists of an inverter and can be represented by process `p3`. Note that *n* (the output of EX-OR) and *q* (the state bit) are internal outputs and are defined as type

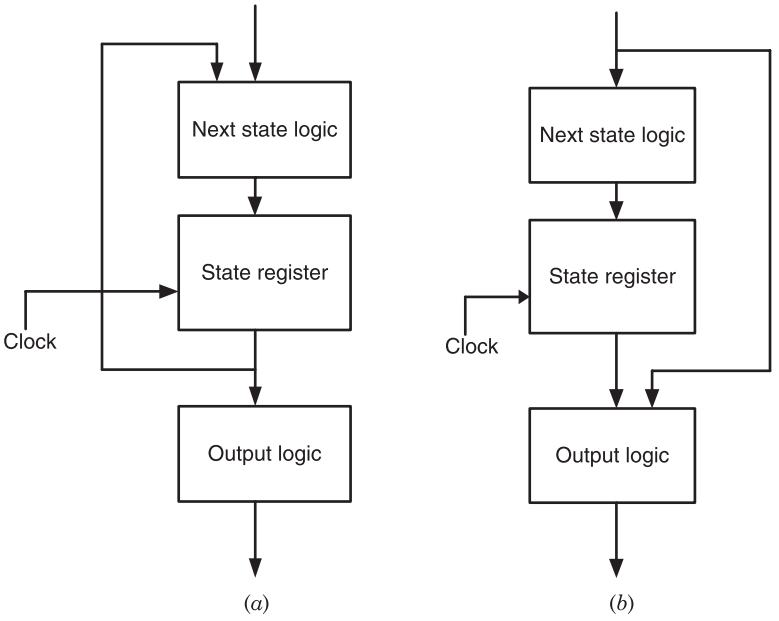


FIGURE 9.18 Models of (*a*) Moore state machine and (*b*) Mealy state machine.

signals in the VHDL code. The VHDL code of the state machine composed of three processes is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity moore3 is
port (m, clk: in std_logic;
      r: out std_logic);
end moore3;
architecture fsm1 of moore3 is
signal n, q: std_logic;
begin
 p1: process(m, q)
     begin
     n <= m xor q;
     end process;
```

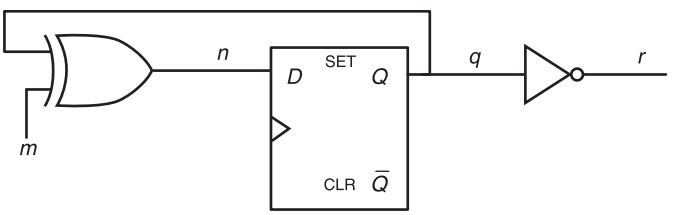


FIGURE 9.19 A Moore-type state machine.

```
 p2: process
     begin
     wait until clk = '1';
     q <= n;
     end process;
 p3: process(q)
     begin
     r <= not q;
     end process;
end fsm1;
```

Note that at the end process p2 the state bit may change its value, which in turn activates processes p1 and p3, thereby changing the value at the input of the state register as well as changing the output value.

A more efficient VHDL code could be obtained by using two processes. One of these sets the current state of the register based on the signal value computed by the next state logic; this process is activated by the clock signal. An asynchronous reset signal may also be used in this process to set the current state; however, this is an option not a requirement. The other process determines the next state value from the current state and the inputs and also computes the output; this process is sensitive to current state values and inputs. The VHDL code for the circuit of Figure 9.19 using two processes is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity moore2 is
port (m,clk: in std_logic;
r: out std_logic);
end moore2;
architecture moorearch of moore2 is
signal q:std_logic;
  begin
     p1: process(q)
         begin
         r <= not q;
         end process;
     p2: process
         begin
         wait until clk = '1';
         q <= q xor m;
         end process;
end moorearch;
```

The VHDL code for a Moore-type state machine can also be described by using a single process. For example, the VHDL code for Figure 9.19 that uses a single process is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity moorex is
 port (m,clk: in std_logic;
 r: out std_logic);
end moorex;
architecture fsm1 of moorex is
signal q:std_logic;
begin
 p1: process(clk)
 begin
 if (clk'event and clk = '1') then
 q <= q xor m;
 end if;
 end process;
 r <= q;
end fsm1;
```

Note that in addition to the process the code has a concurrent statement (included at the end of the process) to compute the output.

9.6.2 Mealy-Type State Machines

A Mealy-type state machine can always be specified by two processes. One process describes the next state logic and the state register, and the other one describes the output logic. To illustrate, let us consider the state machine shown in Figure 9.20. The VHDL code for the machine is given below; process `p1` describes the next state logic and the clocking of the state register, while process `p2` specifies the output logic. An asynchronous reset signal is used in process `p1`.

```
library ieee;
use ieee.std_logic_1164.all;
entity mealy1 is
port (m, reset, clk: in std_logic;
r: out std_logic);
end mealy1;
architecture mealyarch of mealy1 is
signal n, q: std_logic;
```

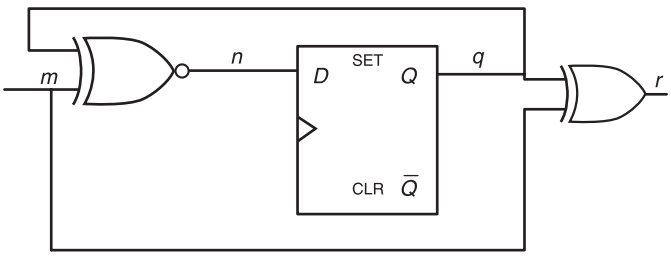


FIGURE 9.20 A Mealy-type state machine.

```
begin
p1: process(reset, clk)
begin
if reset = '1' then
q <= '0';
elsif (clk'event and clk = '1') then
q <= not (q xor m);
end if;
end process p1;
p2: process(m,q)
begin
r <= q xor m;
end process p2;
end mealyarch;
```

9.6.3 VHDL Codes for State Machines Using Enumerated Types

State machines are in general coded in VHDL directly from their state diagram or state table representations. The VHDL code is then converted into a circuit structure by a CAD system such as Quartus II. In order to specify a state diagram (or state table) in VHDL the states in the diagram are declared as *enumeration* type data; an enumeration type provides a means for defining a unique data type. For example, the states in the Moore-type state machine shown in Figure 9.21 could be declared an enumerated type (`state_type`) as follows:

```
Type state_type (A,B,C)
```

The VHDL code for the Moore-type machine is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
```

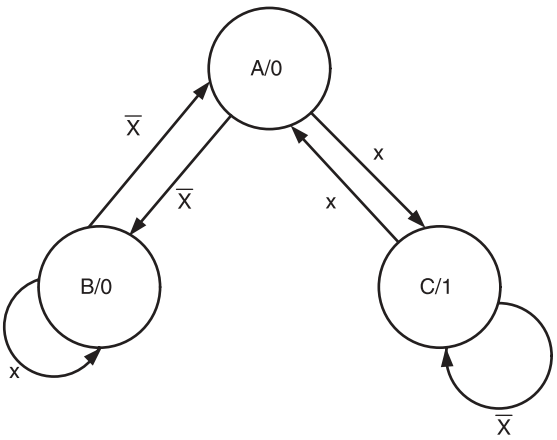


FIGURE 9.21 State diagram of a Moore-type state machine.

```
entity moorep is
port (x, reset, clk: in std_logic;
y: out std_logic);
end moorep;
architecture moorearch of moorep is
type state_type is (A,B,C);
signal PS,NS: state_type;
begin
p1: process(clk, reset)
begin
 if reset = '1' then
 PS <= A;
 elsif (clk'event and clk = '1') then
 PS <= NS;
 end if;
end process p1;
p2: process(PS,x)
begin
case PS is
    when A =>
    if (x = '0') then NS <= B;
    else NS <= C;
    end if;
    when B =>
    if (x  = '0') then NS <= A;
    else NS <= B;
    end if;
    when C =>
    if (x = '0') then NS <= C;
    else NS <= A;
    end if;
end case;
end process p2;
y <= '1' when PS = C else '0';
end moorearch;
```

The VHDL code consists of two processes. The first process as discussed previously specifies that the next state (`NS`) value is transferred to present state (`PS`) on the positive edge of the clock signal. It also indicates that if `reset` signal is set to 1, `A` becomes the present state.

The second process uses a `case` statement to determine the next state value from a present state and the value of input `x`. Initially the first element in `state_type` is considered to be the present state. The subsequent state transitions occur as specified in the `case` statement. A word of caution, if the state transitions are not correctly presented in the `case` statement, the synthesis tool will produce a circuit structure that is different from what is intended; the VHDL compiler will not be able detect such human errors.

The output of the machine is written as a separate concurrent statement outside the processes. It defines the output (`y`) to be 1 only when the circuit is in state `A`. Recall that a

process itself is a concurrent statement; thus the architecture body of the VHDL code for the Moore-type machine consists of three concurrent statements.

The Quartus II compiler allows selection of state encoding from the following: *auto*, *minimal bits*, *1-hot*, and *user-encoded*. If the *minimal bits* option is selected, the assignment will be "00" for the first state (A), "01" for the second state (B) and "10" for the third state (C). The encoding of states using the other options will be discussed later in the chapter.

As indicated earlier, a Moore-type machine can also be described using a single process instead of two. The VHDL code for the Moore-type machine of Figure 9.21 that uses a single process is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity moore5 is
port (x, reset, clk: in std_logic;
y: out std_logic);
end moore5;
architecture moorearch of moore5 is
type state_type is (A,B,C);
signal CS: state_type;
begin
p1: process(clk, reset)
begin
if reset = '1' then
CS <= A;
elsif (clk'event and clk = '1') then
case CS is
    when A =>
    if (x = '0') then CS <= B;
    else CS <= C;
    end if;
    when B =>
    if (x  = '0') then CS <= A;
    else CS <= B;
    end if;
    when C =>
    if (x  = '0') then CS <= C;
    else CS <= A;
    end if;
    end case;
    end if;
end process p1;
y <= '1' when CS = C else '0';
end moorearch;
```

In this code the signal CS represents the present state. As in the code using two processes, the state machine initially starts at state A. The case statement then determines the next state based

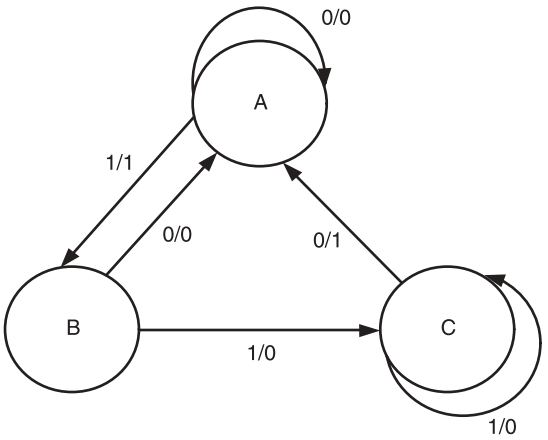


FIGURE 9.22 Sequence (110) detector circuit.

on the present state and the input value. The next state value is transferred to present state CS only at the end of the process. The output is determined based on the present state of the machine.

9.6.4 Mealy Machine in VHDL

As indicated earlier a Mealy machine can be specified in VHDL using two processes. Figure 9.22 shows the state diagram of a state machine that produces an output of 1 when it receives the sequence 110. The VHDL code for the state machine is as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity mealy3 is
port (x, reset, clk: in std_logic;
z: out std_logic);
end mealy3;
architecture mealyarch of mealy3 is
type state_type is (A,B,C);
signal PS: state_type;
begin
p1: process(clk, reset)
begin
if reset = '1' then
PS <= A;
elsif (clk'event and clk = '1')then
case PS is
  when A =>
  if (x = '0') then PS <= A;
  else PS <= B;
  end if;
  when B =>
  if (x = '0') then PS <= A;
```

```
  else PS <= C;
  end if;
  when C =>
  if (x  = '0') then PS <=  A;
  else PS <= C;
  end if;
  end case;
  end if;
end process p1;
p2: process(PS,x)
 begin
  case PS is
  when A =>
  if (x = '0') then z <= '0';
  else z <=  '1';
  end if;
  when B =>
  z <= '0';
  when C =>
  if (x = '0') then z <= '1';
  else z <= '0';
  end if;
  end case;
  end process p2;
end mealyarch;
```

Process p1 deals with the state transitions. As in the case of the Moore machine the states of the circuit are defined as state_type. Signal PS represents the state of the circuit. Process p2 is used to define the output changes in the circuit with change in input x. Note that the output changes are specified in a separate process because in a Mealy machine the output can change when the input does because the output is a function of the input and the current state. If the output is made part of process p1 then both state and output changes can happen only when the clock signal makes a transition, thus implying that the output is registered, which is not true for Mealy machines.

As an additional example, let us describe the state machine shown in Figure 9.23 in VHDL code:

```
library ieee;
use ieee.std_logic_1164.all;
entity mealy3a is
port (clk, resetn           :    in std_logic;
          a, b, c, d        :    in std_logic;
          z         :   out std_logic_vector(1 downto 0));
end mealy3a;
architecture behavior of mealy3a is
    type state_type is(K, L, M, N, P, Q, R, S, T);
    signal PS : state_type;
```

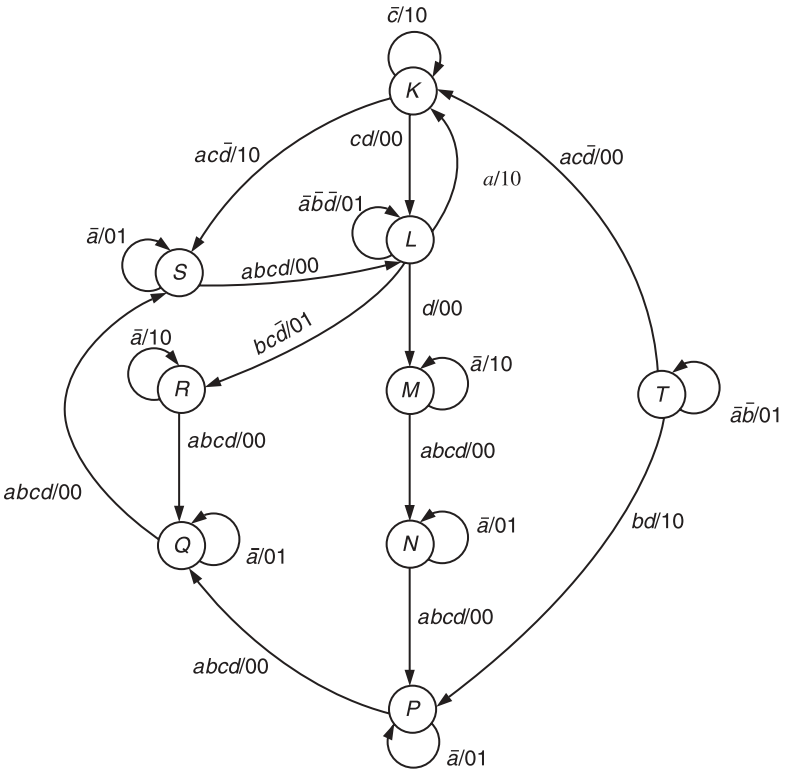


FIGURE 9.23 A state machine.

```
begin
    process (resetn, clk)
    begin
        if resetn = '0' then
            PS <= K;
        elsif (clk'event and clk = '1')then
          case PS is
            when K =>
                if c = '0' then PS <= K;
                elsif (c = '1' and d = '1') then PS <= L;
                elsif (a = '1' and c = '1' and d = '0')
                 then PS <= S;
                end if;
            when L =>
                if (a = '0' and b = '0' and d = '0')
                 then PS <= L;
                elsif d = '1' then PS <= M;
                elsif (b = '1' and c = '1' and d = '0')
                 then PS <= R;
                elsif a = '1' then PS <= K;
                end if;
```

```
            when M =>
                if a = '0' then PS <= M;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= N;
                end if;
            when N =>
                if a = '0' then PS <= N;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= P;
                end if;
            when P =>
                if a = '0' then PS <= P;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= Q;
                end if;
            when Q =>
                if a = '0' then PS <= Q;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= S;
                end if;
            when R =>
                if a = '0' then PS <= R;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= Q;
                end if;
            when S =>
                if a = '0' then PS <= S;
                elsif (a = '1' and b = '1' and c = '1'
                 and d = '1') then
                   PS <= L;
                end if;
            when T =>
                if a = '0' and b = '0'then PS <= T;
                elsif b = '1' and d = '1' then PS <= P;
                end if;
            when others =>
                PS <= K;
            end case;
          end if;
end process;
process (PS,a,b,c,d)
begin
```

```
case PS is
  when K =>
      if (a = '1' and c = '1' and d = '0') then
      Z <=  ''10'';
      elsif c = '0' then
      z  <= ''00'';
      end if;
  when M =>
      if (a = '1' and b = '1' and c = '1'
       and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''10'';
      end if;
  when N =>
      if (a = '1' and b = '1'and c = '1'
       and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''01'';
      end if;
  when P =>
      if (a = '1' and b = '1' and c = '1'
       and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''01'';
      end if;
  when Q =>
      if (a = '1' and b = '1' and c = '1'
       and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''01'';
      end if;
  when R =>
      if (a = '1' and b = '1' and c = '1'
      and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''10'';
      end if;
  when S =>
      if (a = '1' and b = '1' and c = '1'
       and d = '1') then
      z  <= ''00'';
      elsif a = '0' then
      z  <= ''01'';
```

```
                end if;
            when T =>
                if (a = '1' and c = '1' and d = '0') then
                z  <= ''00'';
                elsif (b = '1' and d = '1') then
                z  <= ''10'';
                elsif (a = '0' and b = '0') then
                z <= ''01'';
                end if;
            when L =>
                if a = '1' then
                z  <= ''10'';
                elsif (b = '1' and c = '1' and d = '0')
                 then
                z <= ''01'';
                elsif d = '1' then
                z <= ''00'';
                end if;
            when others =>
                z <= ''00'';
        end case;
    end process;
end behavior;
```

In a Mealy machine the output logic can be defined using signal assignment statements instead of using a process statement. For example, the output logic in the above code can be specified as follows:

```
Z <= ''00'' when (PS = K and c = '1' and d = '1') else
     ''10'' when (PS = K and c = '0') else
     ''01'' when (PS = K and a = '1' and c = '1' and
      d = '0') else
     ''10'' when (PS = L and a = '1') else
     ''00'' when (PS = L and d = '1') else
     ''01'' when (PS = L and b = '1' and c = '1' and
      d = '0') else
     ''00'' when (PS = M and a = '1' and b = '1' and
      c = '1' and d = '1')
        else
     ''10'' when (PS = M and a = '0') else
     ''00'' when (PS = N and a = '1' and b = '1' and c = '1'
      and d = '1')
        else
     ''01'' when (PS = N and a = '0') else
     ''00'' when (PS = P and a = '1' and b = '1' and c = '1'
      and d = '1')
```

```
   else
''01'' when (PS = P and a = '0') else
''00'' when (PS = Q and a = '1' and b = '1' and c = '1'
 and d = '1')
   else
''01'' when (PS = Q and a = '0') else
''00'' when (PS = R and a = '1' and b = '1' and c = '1'
 and d = '1')
   else
''10'' when (PS = R and a = '0') else
''00'' when (PS = S and a = '1' and b = '1' and c = '1'
 and d = '1')
   else
''01'' when (PS = S and a = '0') else
''00'' when (PS = T and a = '1' and c = '1' and
 d = '0') else
''01'' when (PS = T and a = '0' and b = '0') else
''10'' when (PS = T and b = '1' and d = '1') else
''11'';
```

9.6.5 User-Defined State Encoding

The VHDL code for the state machines discussed previously assumed state assignment using minimum bits. It is also possible to choose user-defined bits or 1-hot encoding for states. Let us illustrate first user-defined state assignment for the state machine specified by the state diagram of Figure 9.24. The state codes can be derived by using a formal state assignment technique (discussed earlier in Chapter 7).

An *enumeration* type data, `state_type`, is defined by specifying the list of all states it can take. Each enumerated element in `state_type` can be directly assigned a unique

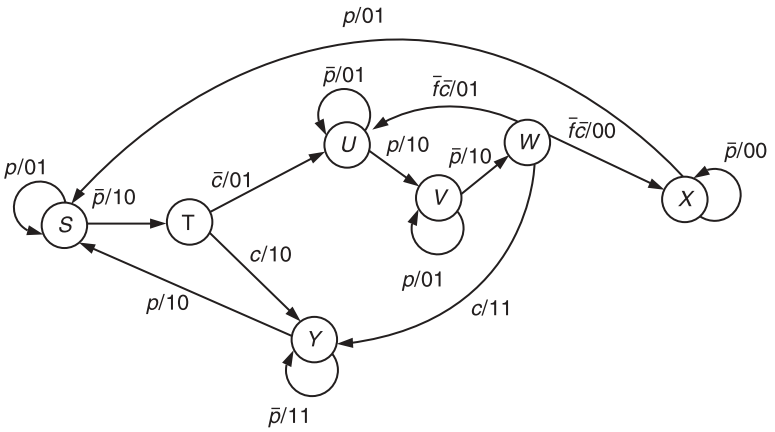


FIGURE 9.24 A state machine.

TABLE 9.4 State Codes for the Machine of Figure 9.24

	y_0	y_1	y_2
S	1	0	1
T	1	1	1
U	0	0	0
V	1	0	0
W	0	1	0
X	0	1	1
Y	0	0	1

value using the attribute `enum_encoding`. This attribute must be declared in the architecture part of the VHDL code after the `state_type` declaration statement as follows:

```
attribute enum_encoding:string;
```

This declaration indicates that the `enum_encoding` is a *string literal*, that is, a sequence of characters within double quotes. Next, the `enum_encoding` is used to assign unique code to each symbolic state declared in the `state_type` statement. To illustrate, the use of `enum_encoding` to assign codes to a state machine with three states is as follows:

```
type state_type is (A,B,C);
attribute enum_encoding: string;
attribute enum_encoding of state_type: type is ''10 11 00'';
```

This will assign patterns 10, 11, and 00 to states `A`, `B`, and `C`, respectively, that is, in the order in which the patterns are presented in the string. Each code in the string must be separated from its adjacent code(s). For a large number of states the corresponding codes can be concatenated so that the string can be represented along multiple lines.

The VHDL description of the circuit of Figure 9.24 that employs user-defined state assignment is given below; it uses the state codes (Table 9.4) derived by using the state assignment technique JEDI. The codes for the states are as follows:

```
library ieee;
use ieee.std_logic_1164.all;
entity mealy4 is
    port (clk, resetn       :    in std_logic;
          p, c, f        :  in std_logic;
          z          :    out std_logic_vector(1 downto 0));
end mealy4;
architecture behavior of mealy4 is
  type state_type is(S, T,U,V,W,X,Y);
  attribute enum_encoding: string;
  attribute enum_encoding of state_type: type is ''101 111
   000 100 010 011 001'';

     signal PS: state type;
     begin
```

```
    process (resetn, clk)
    begin
        if resetn = '0' then
            PS <= S;
        elsif (clk'event and clk = '1')then
          case PS is
            when S =>
                if p = '1' then PS <= S;
                else PS <=  T;
                end if;
            when T =>
                if c = '0' then PS <= U;
                else PS <=  Y;
                end if;
            when U =>
                if p = '0' then PS <= U;
                else PS <=  V;
                end if;
            when V =>
                if p = '0' then PS <= W;
                else PS <=  V;
                end if;
            when W =>
                if (c = '0' and f = '0')then PS <= U;
                elsif (f = '1' and c = '0')then PS <=  X;
                elsif c = '1' then PS  <= y;
                end if;
            when X =>
                if p = '0' then PS <= X;
                else PS <=  S;
                end if;
            when Y =>
                if p = '0' then PS <= Y;
                else PS <=  S;
                end if;
            end case;
          end if;
end process;
process (PS, p, c, f)
begin
z <= ''00'';
case PS is
            when S =>
                if p = '1' then
                z <=  ''01'';
                else
                z <=  ''10'';
```

```
                end if;
            when T =>
                if c = '0' then
                z <=  ''01'';
                else
                z <=  ''10'';
                end if;
            when U =>
                if p = '1' then
                z <=  ''10'';
                else
                z <=  ''01'';
                end if;
            when V =>
                if p = '1' then
                z <=  ''01'';
                else
                z <=  ''10'';
                end if;
            when W =>
                if c = '0' and f = '0' then
                z <=  ''01'';
                elsif c = '0' and f = '1' then
                z <=  ''00'';
                elsif c = '1' then
                z <=  ''11'';
                end if;
            when X =>
                if p = '1' then
                z <=  ''01'';
                else
                z <=  ''00'';
                end if;
            when Y =>
                if p = '1' then
                z <=  ''10'';
                else
                z <=  ''11'';
                end if;
        end case;
        end process;
end behavior;
```

A drawback of the `enum_encoding` attribute is that not all CAD-based state machine design tools use this for state encoding. An alternative approach for user-defined state encoding that is acceptable to any CAD tool for logic synthesis is to represent each symbolic state as a *constant*.

The VHDL code for the circuit of Figure 9.24 that uses constants for state encoding can be obtained by eliminating the following part of the code from the architecture part that uses the *enum_encoding* attribute:

```
type state_type is (S,T,U,V,W,X,Y);
attribute enum_encoding: string;
attribute enum_encoding of state_type: type is ''101 111
000 100 010 011 001'';
signal PS: state_type;
```

Instead the following is used:

```
signal PS: std_logic_vector(2 downto 0);
constant S: std_logic_vector(2 downto 0): = ''101'';
constant T: std_logic_vector(2 downto 0): = ''111'';
constant U: std_logic_vector(2 downto 0): = ''000'';
constant V: std_logic_vector(2 downto 0): = ''100'';
constant W: std_logic_vector(2 downto 0): = ''010'';
constant X: std_logic_vector(2 downto 0): = ''011'';
constant Y: std_logic_vector(2 downto 0): = ''001'';
```

Since each state is encoded using 3 bits, a 3-bit vector is needed to represent a state. A constant is used to identify a unique value of the vector (i.e., a state). The syntax for a constant is

```
constant Name: DataType:= Expression;
```

It uses the operator ``:='', and the expression is a bit string literal in this example.

9.6.6 1-Hot Encoding

In general, the use of minimum number of bits for state encoding results in more complex next state and output expressions. The use of more bits for state encoding, on the other hand, leads to more but simpler expressions. Current Filed Programmable Gate Arrays (FPGAs) have more flip-flops available for use compared to Complex Programmable Logic Devices (CPLDs). Thus for implementing large state machines on FPGAs, the use of n flip-flops for n states as required in 1-hot encoding is a feasible strategy.

The 1-hot encoding of states in VHDL can be specified by making the following simple changes to user-defined encoding using constants (shown in the previous example):

```
architecture behavior of mealy7 is
signal PS: std_logic_vector(6 downto 0);
constant S: std_logic_vector(6 downto 0): = ''1000000'';
constant T: std_logic_vector(6 downto 0): = ''0100000'';
constant U: std_logic_vector(6 downto 0): = ''0010000'';
constant V: std_logic_vector(6 downto 0): = ''0001000'';
constant W: std_logic_vector(6 downto 0): = ''0000100'';
constant X: std_logic_vector(6 downto 0): = ''0000010'';
constant Y: std_logic_vector(6 downto 0): = ''0000001'';
```

The Quartus II CAD system has four options for selecting state codes: *auto*, *minimal bits*, *1-hot*, and *user-encoded*. The *auto* option chooses codes that may not be minimal,

whereas the *minimal* assigns minimum number of bits to encode states. The *user-encoded* option allows the use of *enum_encoding* or *constants* to define states as shown in previous examples. The *1-hot* option instead of setting a single bit to 1 sets two bits to 1's and all other bits to 0's, and also the starting state is set to all 0's. However, each state can be identified by the value of a single bit as in the case of 1-hot code.

The VHDL description for the state machine of Figure 9.24 with 1-hot option can be obtained by just replacing the following line in the architecture description based on the *enum_encoding* attribute:

```
attribute enum_encoding of state_type: type is ''101 111 000
100 010 011 001'';
```

with

```
attribute enum_encoding of state_type: type is ''one-hot'';
```

The state codes used by the Quartus II CAD system for the state machine are:

S	0	0	0	0	0	0	0
T	0	0	0	0	0	1	1
U	0	0	0	0	1	0	1
V	0	0	0	1	0	0	1
W	0	0	1	0	0	0	1
X	0	1	0	0	0	0	1
Y	1	0	0	0	0	0	1

9.7 CASE STUDIES

Case Study 1: Dice Game Controller

We specify the VHLD code for the controller of the dice game "craps." The controller part of a digital system is a state machine that controls the data path. The game is played based on the following rules:

1. A player throws two dice obtaining a number between 2 and 12.
2. He/she wins if the number is 7 or 11.
3. Any number other than 7 or 11 is the player's point and is recorded.
4. The player throws the two dice again.
5. If the number is 7 or 11, he/she loses.
6. If the number is equal to the recorded value of the first two throws, he/she wins.
7. If the conditions of steps 5 and 6 are not satisfied, the game is continued to step 4.

A Moore machine having six states as shown in Figure 9.25 can be used as a controller.* The tasks of the controller at each state are as follows:

S0	Simulate the first throw of the dice
S1	Compare the number with 7 or 11; prepare to store value in register

*VHDL class project report of James Rohrbach.

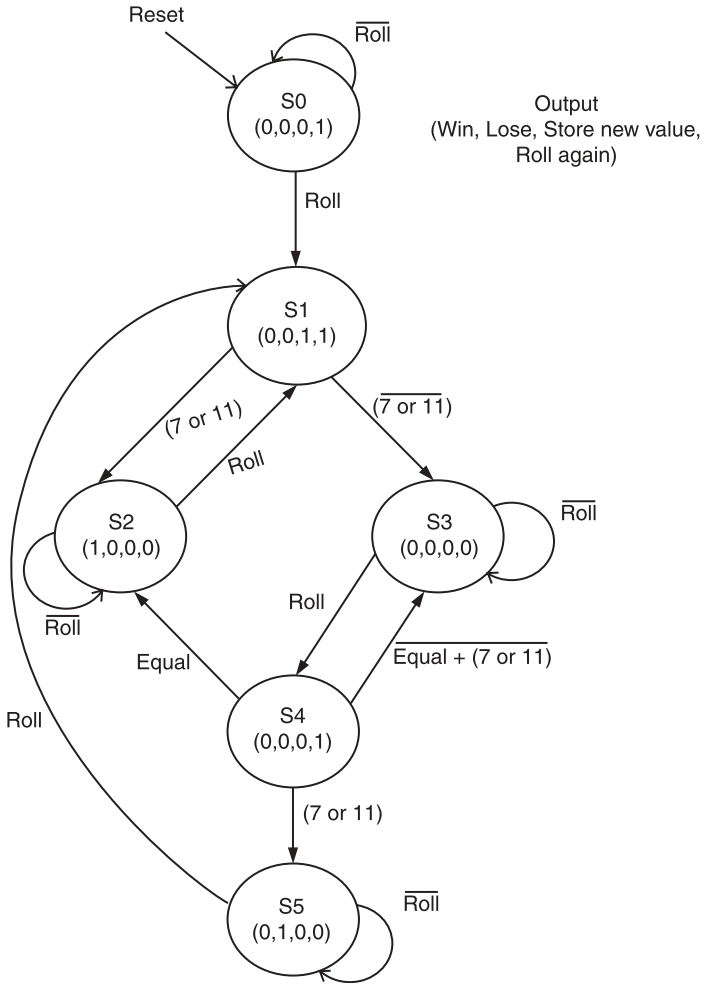


FIGURE 9.25 State diagram for the dice game controller.

S2	WIN (a halt state)
S3	Simulate further throw of dice
S4	Compare number with 7 or 11, and with the number stored in the register
S5	LOSE (a wait state)

The output at each state is represented by a 4-bit vector. The first bit of the vector is 1 for WIN, the second bit is 1 if LOSE, a 1 at the third bit stores the count value (new value) in a register, and a 1 at the fourth bit indicates the dice are to be rolled again.

We assume the following:

A single button (RP) is provided in order for the user to simulate the throw (Roll) of dice.

A counter is used to generate a "random" number from 2 to 12. The clock for the counter (FC) is separate from the clock (clk) that drives the state machine. FC is assumed to be fast enough so that the operator cannot intentionally influence the number generated.

The states in which the win/lose status are determined contain self-loops that cause the state machine to remain in that state until the user throws the dice again.

An asynchronous reset (Reset) input is provided to reset the state machine to state S0 so that the player can restart the game from any state; also a Roll again output signal is produced at states S1 and S4 to allow re-roll of the dice.

The VHDL code for the controller is given below. It uses eight processes to describe the state machine controller. Process `p1` defines the operation of the counter. Process `p2` specifies when the roll of the dice is to be started or stopped. Process `p3` checks the counter output to determine whether they are 7 or 11. Process `p4` stores the count value. Process `p5` checks if the stored count value is equal to the new count value. Process `p6` specifies the next state of the state machine based on the inputs it receives. Process `p7` defines the outputs produced at each state. Process `p8` specifies the next state when the clock or the reset signal is applied.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity Dice_Game is
     port(clk, FC, RP, Reset: in std_logic;
             Win, Lose: out std_logic;
             StoredValue, NewValue: buffer std_logic_vector
              (3 downto 0);
             C: out std_logic_vector (3 downto 0));
end Dice_Game;
architecture behavior of Dice_Game is
type State_type is (S0,S1,S2,S3,S4,S5);
signal y_present, y_next: State_type;
signal Equal, SevenOrEleven, Roll, ReRoll, StoreNew:
 std_logic;
signal NV, SV, Count: std_logic_vector(3 downto 0);
begin
p1: process(FC, Reset)
begin
       if Reset = '1' then
          Count <= "0010";
       elsif (FC' event and FC = '1') then
          if Count = "1100" then
          Count <= "0010";
          else
          Count <= Count + '1';
          end if;
       end if;
end process;
p2: process(RP, ReRoll, Reset)
begin
       if Reset = '1' or ReRoll = '1' then
          Roll <= '0';
       elsif (RP'event and RP = '1') then
```

```
            Roll <= '1';
        end if;
end process;
p3: process(RP, Reset)
begin
        if Reset = '1' then
            SevenOrEleven <= '0';
            NV <= "0000";
        elsif (RP'Event and RP = '1') then
            if (Count = "0111" or Count = "1011") then
            SevenOrEleven <= '1';
            else
            SevenOrEleven <= '0';
        end if;
        NV <= Count;
        end if;
end process;
p4: process(StoreNew, NV, Reset)
begin
        if Reset = '1' then
            SV <= "0000";
        elsif (StoreNew'event and StoreNew = '1') then
            SV <= NV;
        end if;
end process;
p5: process(SV, NV)
begin
        if SV = NV then
            Equal <= '1';
        else
            Equal <= '0';
        end if;
end process;
p6: process(Roll, Equal, SevenOrEleven, y_present)
begin
        case y_present is
            when S0 =>
                if Roll = '1' then
                y_next <= S1;
                else
                y_next <= S0;
                end if;
            when S1 =>
                if SevenOrEleven = '1' then
                y_next <= S2;
                else
                y_next <= S3;
                end if;
```

```
            when S2 =>
               if Roll = '1' then
               y_next <= S1;
               else
               y_next <= S2;
               end if;
            when S3 =>
               if Roll = '1' then
               y_next <= S4;
               else
               y_next <= S3;
               end if;
            when S4 =>
               if Equal = '1' then
               y_next <= S2;
               elsif SevenOrEleven = '1' then
               y_next <= S5;
               else
               y_next <= S3;
               end if;
            when S5 =>
               if Roll = '1' then
               y_next <= S1;
               else
               y_next <= S5;
               end if;
      end case;
end process;
p7: process(y_present)
begin
       case y_present is
            when S0 =>
               Win <= '0';
               Lose <= '0';
               StoreNew <= '0';
               ReRoll <= '0';
            when S1 =>
               Win <= '0';
               Lose <= '0';
               StoreNew <= '1';
               ReRoll <= '1';
            when S2 =>
               Win <= '1';
               Lose <= '0';
               StoreNew <= '0';
               ReRoll <= '0';
```

```
                when S3 =>
                        Win <= '0';
                        Lose <= '0';
                        StoreNew <= '0';
                        ReRoll <= '0';
                when S4 =>
                        Win <= '0';
                        Lose <= '0';
                        StoreNew <= '0';
                        ReRoll <= '1';
                when S5 =>
                        Win <= '0';
                        Lose <= '1';
                        StoreNew <= '0';
                        ReRoll <= '0';
                end case;
        end process;
        p8: process(clk, Reset)
        begin
                if Reset = '1' then
                        y_present <= S0;
                elsif(clk'Event and clk = '1') then
                        y_present <= y_next;
                end if;
        end process;
        C <= Count;
        StoredValue <= SV;
        NewValue <= NV;
end behavior;
```

The simulation results of the controller are shown in Figure 9.26.

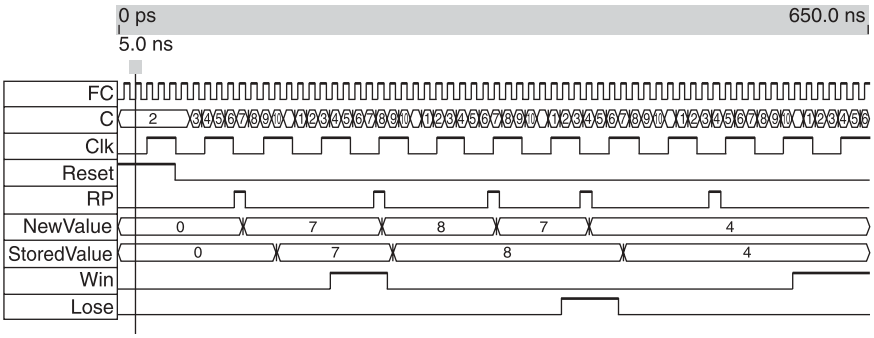


FIGURE 9.26 Simulation results of the dice game controller.

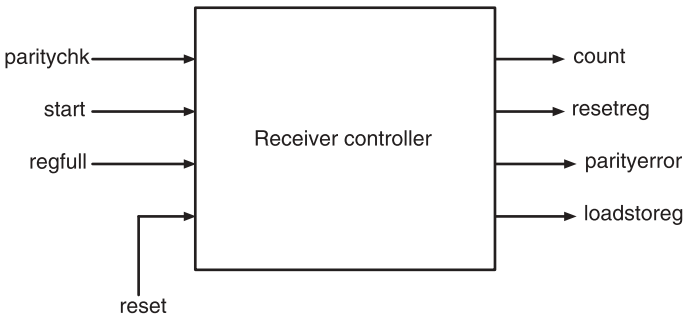


FIGURE 9.27 UART receiver controller.

Case Study 2: UART Controller

The receiver part of a UART (universal asynchronous receiver/transmitter) receives serial data from a system, converting the data to parallel. It also checks the parity of the parallel data and stores the data if it is correct; otherwise, it indicates parity error [1]:

1. Control module
2. Receiving register/shift register
3. Storage register
4. Shift pulse generator
5. Parity

The control module is the most important part of the receiver. It determines when the receiving register, which is a shift register, is to be filled with data, when it is to be reset, and when the parity check is to be done. The block diagram of the state machine is shown in Figure 9.27.

The controller has seven states as shown in Figure 9.28.*

A Reset state
B Prepare to receive data
C Start counter, reset shift register
D Receive data until the shift register is full
E Check parity
F Load data into storage register
G Indicate parity error in the shift register data

The controller has four inputs:

reset	Makes the controller stay at reset state when 1
start	Starts the counting operation
regfull	When 0 indicates the shift register is full
paritychk	When 0 indicates parity error

*VHDL class project report of Tianyu Liu.

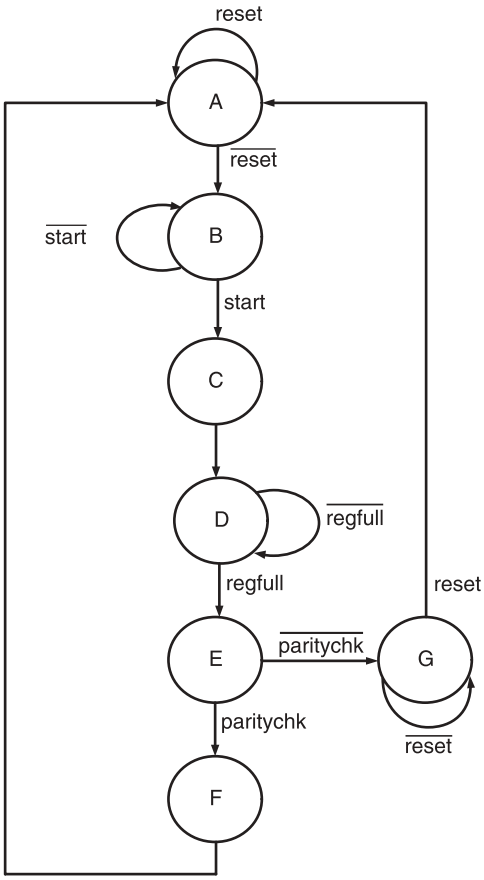


FIGURE 9.28 Controller state diagram.

The controller also has four outputs:

count	Count up
resetreg	Reset the shift register
parityerror	Indicates parity error when 0
loadstoreg	Load the storage register

The VHDL description of the controller is given below; the controller is identified as `UART` in the code:

```
library ieee;
use ieee.std_logic_1164.all;
entity UART is
port(   clk: in std_logic;
        reset, start, paritychk, regfull: in std_logic;
        count, parityerror, resetreg, loadstoreg: out
        std_logic);
end UART;
```

```
architecture behavior of UART is
signal y_present, y_next: std_logic_vector(2 downto 0);
constant A: std_logic_vector(2 downto 0):= ''000'';
constant B: std_logic_vector(2 downto 0):= ''001'';
constant C: std_logic_vector(2 downto 0):= ''011'';
constant D: std_logic_vector(2 downto 0):= ''010'';
constant E: std_logic_vector(2 downto 0):= ''110'';
constant F: std_logic_vector(2 downto 0):= ''111'';
constant G: std_logic_vector(2 downto 0):= ''100'';
begin
process( y_present, reset, start, paritychk, regfull)
begin
       case y_present is
              when A =>
                     if reset = '1' then y_next <= A;
                     else y_next <= B;
                     end if;
              when B =>
                     if start = '0' then y_next <= B;
                     else y_next <= C;
                     end if;
              when C =>
                     y_next <= D;
              when D =>
                     if regfull = '0' then y_next <= D;
                     else y_next <= E;
                     end if;
              when E =>
                     if paritychk = '0' then y_next <= G;
                     else y_next <= F;
                     end if;
              when F =>
                     y_next <= A;
              when G =>
                     if reset = '0' then y_next <= G;
                     else y_next <= A;
                     end if;
              when others =>
                     y_next <= A;
      end case;
end process;
process (clk, reset)
begin
       if reset = '1' then
                y_present <= A;
       elsif (clk'event and clk = '1') then
                y_present <= y_next;
       end if;
end process;
```

```
process (y_present)
begin
        if y_present = C then
                count <= '1';
                resetreg <= '1';
                loadstoreg <= '0';
                parityerror <= '0';
        elsif y_present = D then
                count <= '1';
                resetreg <= '0';
                loadstoreg <= '0';
                parityerror <= '0';
        elsif y_present = F then
                count <= '0';
                resetreg <= '0';
                loadstoreg <= '1';
                parityerror <= '0';
        elsif y_present = G then
                count <= '0';
                resetreg <= '0';
                loadstoreg <= '0';
                parityerror <= '1';
        else
                count <= '0';
                resetreg <= '0';
                loadstoreg <= '0';
                parityerror <= '0';
        end if;
end process;
end behavior;
```

The simulation results of the controller are shown in Figure 9.29.

Case Study 3: Controller for an Automatic Bank Teller

A controller for an automatic bank teller is to be designed.* It dispenses cash if a customer enters a correct account number and a correct amount is designated. The controller must be able to prompt the customer to enter his/her account number and the amount he/she wants to withdraw via a keyboard. These numbers must be compared first to check their validity; hence a comparator circuit must be associated with the keyboard. The comparator receives inputs from the keyboard and the controller; its outputs are fed as inputs to the controller. The controller activates a cash dispenser if the amount is correct. A clear switch must also be included so that a new customer can use the teller to start a new transaction even if the last transaction was not completed.

Specification of the Bank Teller The controller has five inputs and one output. The inputs are

key_entered	A 20-bit number
strobea	Account number

*VHDL class project report of Yucong Tao.

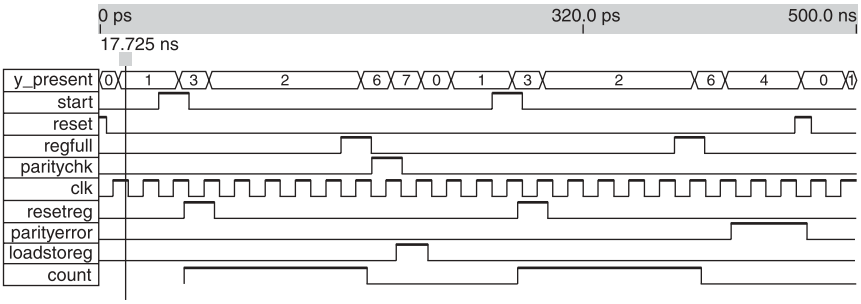


FIGURE 9.29 Simulation results of UART controller.

strobeab	Cash amount
clr	Asynchronous reset
clk	Clock signal

The output is

display_state	Controls the screen display to direct the customer

A block diagram of the bank teller is shown in Figure 9.30. The teller is further partitioned into four components:

A state machine
An account verifier
An amount checker
A cash dispenser

The *state machine* shown in Figure 9.31 has five inputs: *ac* (account checked), *am* (amount checked), *ad* (cash dispensed), *clk* (clock), and *clr* (reset). The state machine controls user display by providing a *display_state* signal, and a *dis_en* (dispenser enable) signal for the cash dispenser. The asynchronous clear signal *clr* resets the state machine in terms of both internal state and outputs.

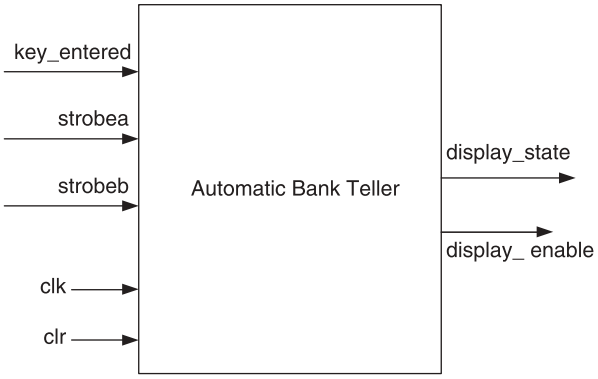


FIGURE 9.30 Bock diagram of the automatic bank teller.

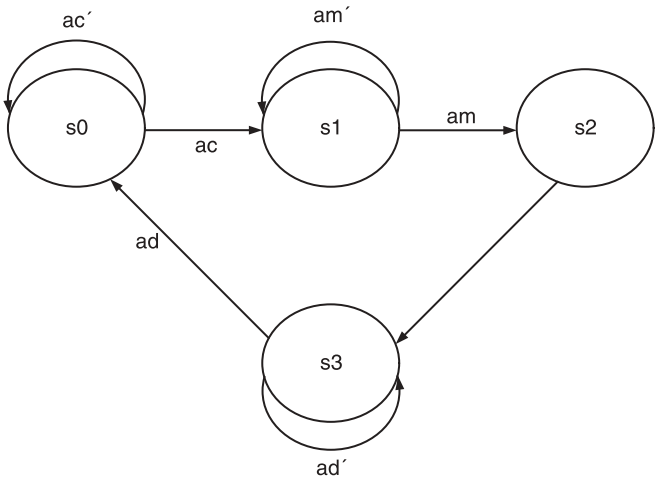


FIGURE 9.31 State diagram of automatic bank teller.

The state machine is Moore-type and has four states `s0`, `s1`, `s2`, and `s3`; s0 is the initial state with both outputs *display_state* and *dis_en* unasserted. Once input *ac* is asserted the machine moves to state `s1` at the next rising edge of the clock. In state `s1`, *display_state* is asserted and *dis_en* remains low. Upon receiving *am*, the machine moves to state `s2` where *display_state* remains low but *dis_en* is asserted. On the next clock edge the machine is forced to state `s3`. The machine remains at `s3` until *ad* is asserted, then it moves back to `s0`.

The VHDL code for the state machine is as follows:

```
library IEEE;
use ieee.std_logic_1164.all;
entity ctl is port(
ac, am, ad, clk, clr: in std_logic;
display_state, dis_en: out std_logic);
end ctl;
architecture state_machine of ctl is
type statetype is (s0, s1, s2, s3);
signal PS, NS: statetype;
begin
p1: process (PS, ac, am, ad)
begin
case PS is
when s0 => display_state <= '0'; dis_en <= '0';
if ac = '1' then
NS <= s1;
else
NS <= s0;
end if;
when s1 => display_state <= '1'; dis_en <= '0';
if am = '1' then
NS <= s2;
```

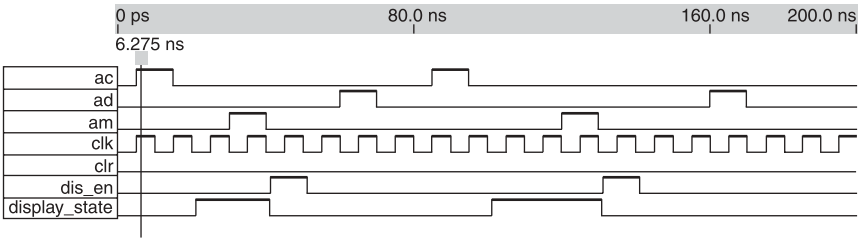


FIGURE 9.32 Simulation results of the controller for the automatic bank teller.

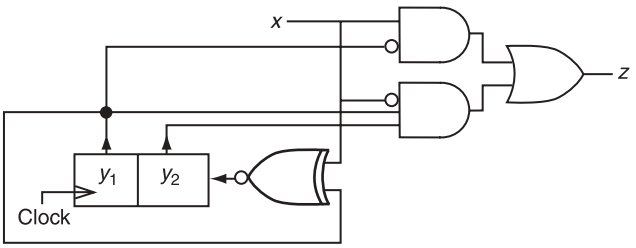
```
else
NS <= s1;
end if;
when s2 => display_state  <= '0'; dis_en <= '1';
NS <=  s3;
when s3 => display_state  <= '0'; dis_en <= '0';
if ad = '1' then
NS <= s0;
else
NS <= s3;
end if;
end case;
end process;
p2: process (clr, clk)
begin
if clr = '1' then
PS <=  s0;
dis_en <= '0';
display_state <= '0';
elsif (clk'event and clk = '1') then
PS <= NS;
end if;
end process;
end state_machine;
```

The simulation results of the automatic bank teller controller are given in Figure 9.32.

EXERCISES

1. Write the VHDL code for a *JK* flip-flop that is *set-dominant* when a control signal c is 0, and *reset-dominant* when c is 1. (In a set-dominant *JK* flip-flop, the output of the flip-flop becomes 1 when $J = K = 1$; in a reset-dominant *JK* flip-flop, the output becomes 0 when $J = K = 1$).
2. A four-stage shift register is to be used to generate two sequences of length 7 and 15. A sequence of length 7 is generated when a control signal c is set to 1; when c is set to 0, a sequence of length 15 is generated. Write the VHDL code for the circuit.

3. The shift register implementation of a sequential circuit is shown below. Write the VHDL code to specify the function of the circuit.



4. Describe a pseudorandom sequence generator based on the following polynomial using VHDL:

$$f(x) = x^5 + x^3 + 1$$

5. Write the VHDL code for a random counter that goes through the sequence

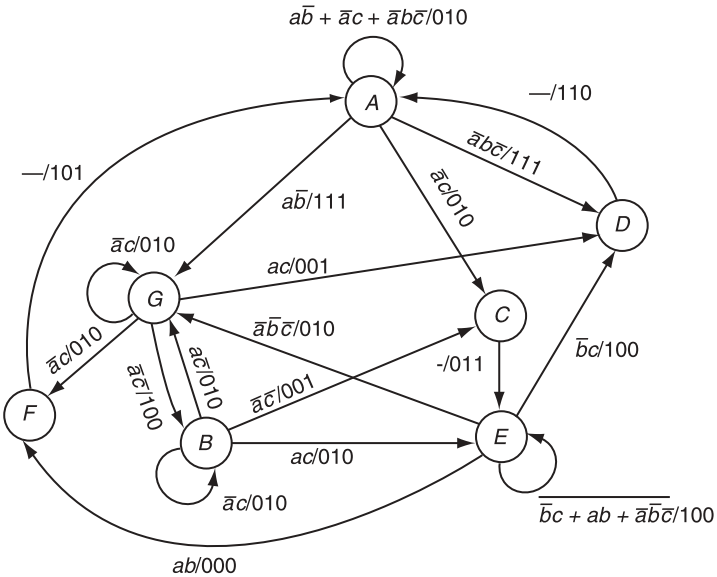
$$8 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow 4 \rightarrow 6 \rightarrow 3 \rightarrow 5 \rightarrow 9 \rightarrow 7$$

6. Write the VHDL code for a counter that can count either in mod-8 binary or Gray code depending on a control signal being 0 or 1, respectively.

7. Write the VHDL code for an 8-bit Johnson counter assuming 01110000 as the initial state.

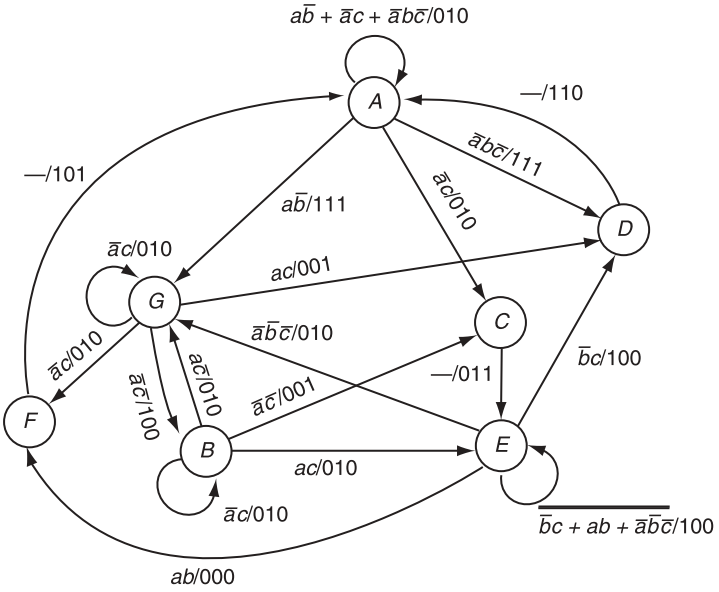
8. As discussed in the text, an n-bit Johnson counter is a mod-$2n$ counter. Write the VHDL code for a mod-7 counter constructed from a 4-bit Johnson counter.

9. Write the VHDL code for the state machine shown below:



10. Write the VHDL code for the following state machine using the state assignment A = 111, B = 001, C = 101, D = 011, E = 000, F = 110, and G = 010:

11. A sequential circuit produces an output of 1 if and only if it receives an input sequence that contains only two groups of 0's. For example, the circuit will produce an output of 1 if the input sequence is 11001111 but will generate an output of 0 if the input sequence is 11001101. Write the VHDL code for the circuit using *m-out-of-n* state encoding.



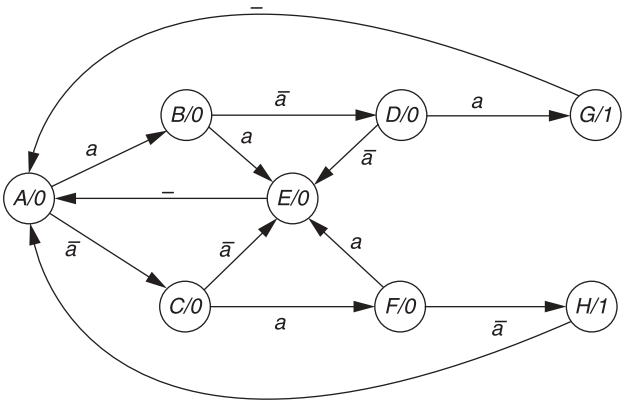
12. A sequential circuit has six states (A, B, C, D, E, F), five inputs (s, t, u, v, w), and three outputs (l, m, n). The circuit is specified below using `if-then-else` statements. Each state shows the outputs associated with the state and the state transitions. Write the VHDL specification of the circuit using a `case` statement.

```
State A: l = 0, m = 0, n = 0
   IF s = 1 THEN B ELSE A;
State B: l = 0, m = 0, n = 0;
   IF z1 = 1 THEN C            {z1 = (u + v̄)t̄; z2 = ūvt̄}
   ELSE IF z2 = 1 THEN B
   ELSE IF t = 1 THEN A
   ELSE B;
State C: l = 0, m = 0, n = 0;
   IF z1 = 1 THEN C
   ELSE IF z2 = 1 THEN D
   ELSE IF t = 1 THEN A
   ELSE C;
State D: l = 0, m = 0, n = 0;
   IF z1 = 1 THEN E
```

```
      ELSE IF z2 = 1 THEN D
      ELSE IF t = 1 THEN A
      ELSE D;
   State E: l = 1, m = 0, n = 1  {z3 = wt̄; z4 = w̄t̄}
      IF z3 = 1 THEN E
      ELSE IF z4 = 1 THEN F
      ELSE IF t = 1 THEN A
      ELSE E;
   State F: l = 0, m = 1, n = 0;
      IF t = 1 THEN A
      ELSE F;
```

13. A single-input and single-output state machine produces an output of 1 and remains at 1 when at least two 0's and two 1's have occurred at the input regardless of the order of occurrence. Write the VHDL code for the circuit.

14. Write the VHDL code for the following state machine using 1-hot code to encode the states:



15. A sequential circuit produces an output of 1 if and only if it receives an input sequence that contains only one group of 0's. For example, the circuit will produce an output of 1 if the input sequence is 11001111, but will generate an output of 0 if the input sequence is 11001101. Write the VHDL code for the circuit using k-out-of-2k state encoding.

REFERENCES

1. D. Comer, *Digital Logic and State Machine Design*, 2nd ed., John Wiley & Sons, Hoboken, NY, 1990.
2. S. Golomb, *Shift Register Sequences*, Holden Day, 1967.

10 Asynchronous Sequential Circuits

10.1 INTRODUCTION

In Chapter 7 we considered clocked (synchronous) sequential circuits; asynchronous circuits do not use clocks. If an input variable is changed in an asynchronous circuit, the circuit goes through a sequence of *unstable* states before settling down to a *stable* state. On the other hand, a synchronous circuit moves from one stable state to another without passing through any unstable states. To illustrate the concept of unstable states, let us consider the *SR* latch circuit shown in Figure 10.1. When $S = 0$ and $R = 1$, the output Q of the circuit is set to 0, and $\bar{Q} = 1$; this is a stable state because the feedback signals have no effect on the output of the latch. If R is changed to 0, the circuit still remains in the stable condition because there is no change in the output. Now if S is changed to 1, $\bar{Q}$ changes to 0; however, before Q changes to 1, there is a momentary delay during which there is an unstable state $Q = 0$, $\bar{Q} = 0$ with inputs $S = 1$ and $R = 0$. At the end of the delay, the circuit will assume the stable state $Q = 1$, $\bar{Q} = 0$. Thus an unstable state is said to exist in an asynchronous circuit if the next state at any instant of time after an input has been changed is not equal to the present state. We assume that an asynchronous circuit will eventually assume a stable state, provided the duration of the inputs is longer than the period of time the unstable state or states exist. If an input to an asynchronous circuit is changed only when it is in a stable state, never when it is unstable, then the circuit is said to be operating in *fundamental mode*. The simultaneous change of more than one input variable is avoided in asynchronous circuits because this often leads to serious timing problems.

Figure 10.2 shows the model of asynchronous circuits. Unlike synchronous circuits, these do not require separate memory elements; the propagation delays associated with the feedback paths from the outputs to the inputs provide the memory required for

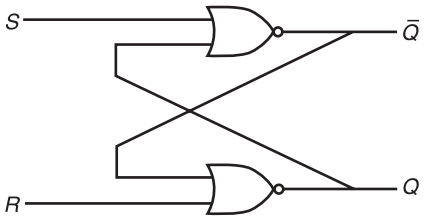


FIGURE 10.1 *SR* latch constructed from NOR gates.

Principles of Modern Digital Design, by Parag K. Lala

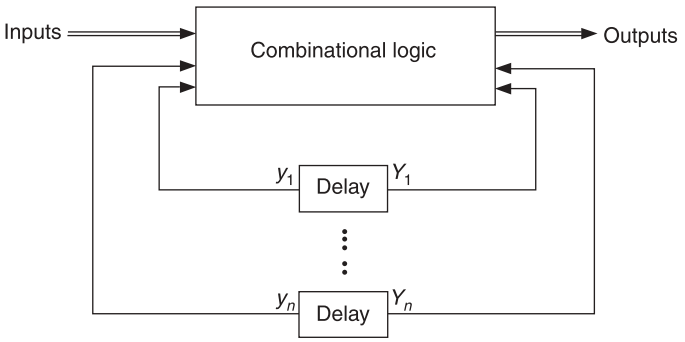


FIGURE 10.2 Model of asynchronous sequential circuits.

sequential operation. Although the delay elements shown in Figure 10.2 can be considered to have the same role as *D* flip-flops in synchronous circuits, the delays cannot be assumed to be of equal magnitude. The internal state of an asynchronous circuit is represented by the state variable y_i, and the next state variables are denoted by Y_i ($i = 1, n$).

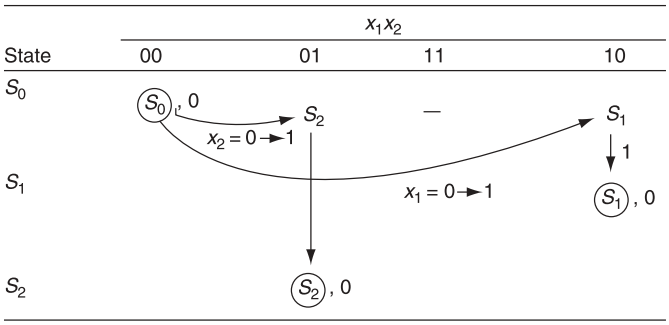
10.2 FLOW TABLE

As in the design of synchronous sequential circuits, the first step in the design of asynchronous circuits is to specify the circuit operation in a formal manner. A conventional way of describing the operation of a fundamental mode circuit is to use a *flow table*. The flow table is very similar to the state table in that it specifies all possible modes of circuit operation. As an example, we construct the flow table of a sequential circuit that has two inputs x_1x_2 and one output Z; the circuit produces an output of 1 only if the input sequence $x_1x_2 = 10, 11, 01$ is received in that order. All other sequences produce an output of 0. Initially the inputs are $x_1 = 0$, $x_2 = 0$, and the circuit is in a state designated as $\textcircled{S_0}$, where the circle around S_0 indicates that it is stable. Thus the first row of the flow table is S_0 and the entry in column $x_1x_2 = 00$ is $\textcircled{S_0}$. The output of an asynchronous circuit is associated with a stable state; hence the output entry 0 is recorded on the right of $\textcircled{S_0}$.

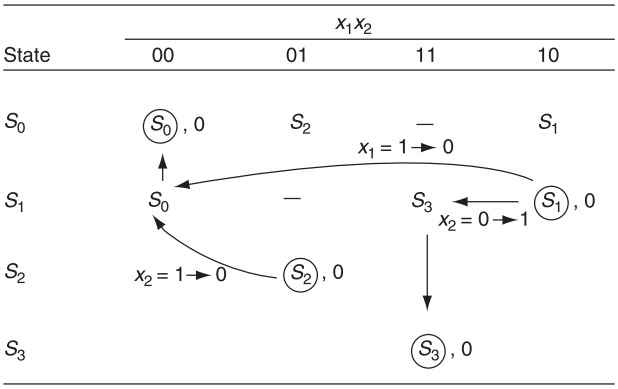
	x_1x_2			
State	00	01	11	10
S_0	$\textcircled{S_0}$, 0	—	—	—

If the input combination is now changed to $x_1x_2 = 10$, the circuit enters the stable state $\textcircled{S_1}$. To show that the change from $\textcircled{S_0}$ to $\textcircled{S_1}$ was caused by the input combination $x_1x_2 = 10$, the uncircled state S_1 is entered in column 10 of the first row. This indicated that the transition from $\textcircled{S_0}$ to $\textcircled{S_1}$ was via an unstable state. If the input x_2 is changed to 1 while the circuit is in $\textcircled{S_0}$., the circuit would enter another stable state $\textcircled{S_2}$ via the unstable

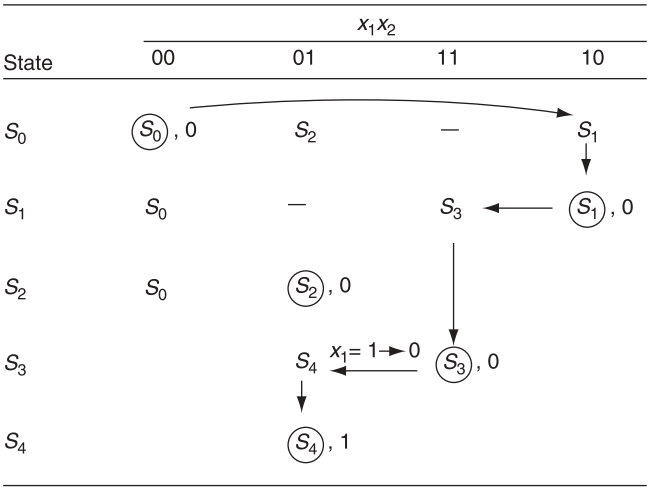
state S_2. The output of the circuit corresponding to both stable states (S_1) and (S_2) is 0. The dash in column 11 indicates that the double input change $x_1x_2 = 00$ to 11 is not allowed.



Starting in the second row, an input change from $x_1x_2 = 10$ to 00 will cause the circuit to return to the initial state (S_0). On the other hand, if the input changes from $x_1x_2 = 10$ to 11, the circuit will enter another stable state (S_3).



If input x_1 changes from 1 to 0 while the circuit is in (S_3), the desired input sequence is complete, and the circuit moves to (S_4) with the output $Z = 1$.



Starting in (S_2), if x_2 changes from 1 to 0, the circuit is reset to state S_0. If x_1 is changed from 0 to 1, the circuit cannot go to (S_3) because this will indicate that the circuit has received the input sequence (10, 11). So a new state (S_5), is included; (S_5) has a 0 output associated with it.

The change in x_2 from 1 to 0 when the circuit is in state (S_3) must take it to (S_1), since (S_1) corresponds to the first two symbols of the desired input sequence.

Starting from (S_4), if the inputs change to 00 it is not possible to get $Z = 1$ without resetting the circuit to (S_0). The change from $x_1x_2 = 01$ to 11 cannot move the circuit to S_3; however, the circuit can be taken to state (S_5).

Finally, from (S_5), if the inputs change to 01, the circuit cannot move to (S_4) because it is allowed to produce $Z = 1$ only after it resets and receives the proper input sequence. Hence it must move to (S_3). On the other hand, if x_1x_2 changes to 10, the circuit can go to (S_1) since this starts the desired sequence over again. The complete flow table is shown in Figure 10.3; dashes have been entered wherever input changes are not allowed.

It should be noted that although the primate flow table of Figure 10.3 looks like that of a Mealy model, it could have been represented in the Moore model form by adding a column of outputs such that each row has an output the same as that associated with the stable next state. The Moore model version of the primitive table is shown in Figure 10.4.

Note that the flow table constructed for the sequence detector has exactly one stable state per row. Such a table is called a *primitive flow table*. In a primitive flow table a change of an input corresponds to a change between columns without any row change. If the state in the new column is uncircled (i.e., unstable), a row change takes place, with the new row corresponding to the next state for the circuit. One important distinction between synchronous and asynchronous circuits is that in an asynchronous circuit the state changes depend on the input changes, whereas in synchronous circuits it is the clock pulse rather than the input changes that triggers the state changes.

	x_1x_2			
State	00	01	11	10
S_0	(S_0), 0	S_2	—	S_1
S_1	S_0	—	S_3	(S_1), 0
S_2	S_0	(S_2), 0	S_5	—
S_3	—	S_4	(S_3), 0	S_1
S_4	S_0	(S_4), 1	S_5	—
S_5	—	S_2	(S_5), 0	S_1

FIGURE 10.3 Flow table for the (10, 11, 01) detector.

State	x_1x_2 00	01	11	10	Output Z
S_0	(1)	2	—	5	0
S_1	1	—	3	(5)	0
S_2	1	(2)	5	—	0
S_3	—	4	(3)	5	0
S_4	1	(4)	5	—	1
S_5	—	2	(5)	5	0

FIGURE 10.4 Moore-type flow table for the (10, 11, 01) detector.

10.3 REDUCTION OF PRIMITIVE FLOW TABLES

In general, a primitive flow table constructed from a circuit specification contains redundant states that must be eliminated in order to reduce the hardware required for the circuit implementation. Since each row in a primitive flow table is identified with a unique stable state, the elimination of redundant states results in the reduction of rows in the table. The reduction process is analogous to that of the incompletely specified synchronous circuit; recall that the unspecified entries in a flow table are due to the constraint that only one input variable can change at a time. Thus two rows in a primitive flow table can be merged if the next state entries in each column corresponding to these rows are the same if both are specified. If two next state entries are the same, with one stable and the other unstable, then after merging the resultant entry is stable. There is no conflict of outputs when two rows are merged, because the outputs are only associated with stable states and two different stable states in the same column are not merged. Since the problem of eliminating redundant states in asynchronous circuits is identical to that encountered in incompletely specified synchronous circuits, the technique of Section 7.6 may be employed.

Example 10.1 Let us reduce the primitive flow table of Figure 10.5. The implication table corresponding to the flow table is shown in Figure 10.6. Note that a stable state (w) and an unstable state x are compatible if (w) is compatible with (x). An unstable state y is compatible with another unstable state z if (y) is compatible with (z). It is seen from Figure 10.6 that the compatible pairs (i.e., the rows) that can be merged are

$$(S_0, S_5)(S_1, S_6)(S_3, S_4)$$

Row S_2 cannot be merged with any other rows. Thus the reduced flow table is

State	x_1x_2 00	01	11	10
(S_0, S_5)	(S_0), 0	(S_5), 1	—	S_1
(S_1, S_6)	S_2	(S_6), 1	—	(S_1),1
(S_3, S_4)	(S_4), 1	S_6	—	(S_3), 1
(S_2)	(S_2), 1	S_5	—	S_3

	x_1x_2			
	00	01	11	10
S_0	(S_0), 0	—	—	S_1
S_1	S_2	—	—	(S_1), 1
S_2	(S_2), 1	S_5	—	S_3
S_3	S_4	—	—	(S_3), 1
S_4	(S_4), 1	S_6	—	—
S_5	S_0	(S_5), 1	—	—
S_6	S_2	(S_6), 1	—	—

FIGURE 10.5 A primitive flow table.

By replacing the rows in the reduced flow table as A, B, C, and D, the following equivalent flow table can be obtained:

	x_1x_2			
	00	01	11	10
A	(A), 0	(A), 1	—	B
B	D	(B), 1	—	(B), 1
C	(C), 1	B	—	(C), 1
D	(D), 1	A	—	C

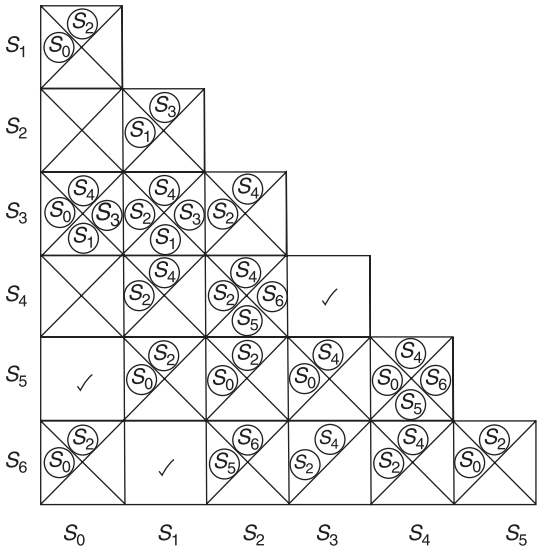


FIGURE 10.6 Implication table for Figure 10.5.

10.4 STATE ASSIGNMENT

This process is also similar to that described for synchronous sequential circuits. However, while the criterion for the selection of unique binary codes for the states in synchronous circuits was that it should result in a minimum hardware implementation, there is a more important requirement for asynchronous circuits. The assignment of secondary variables in asynchronous circuits must be such that only one variable can change during the transition of the circuit from one stable state to another.

10.4.1 Races and Cycles

A race condition results in a fundamental model circuit if more than one secondary variable is allowed to change during a state transition. The condition arises due to the unequal delays in different feedback paths in the circuit. To illustrate, let us consider the flow table shown in Figure 10.7*a*. By assigning secondary variables $A = 00$, $B = 01$, $C = 11$, and $D = 10$, the excitation table shown in Figure 10.7*b* results. Assume that the circuit is in state (00) and both inputs x_1 and x_2 are equal to 0. Now if input x_2 changes to 1, an unstable condition develops in which the present secondary variables $y_1y_2 = 00$, but the input combination

	x_1x_2			
	00	01	11	10
A	(*A*)	*C*	(*A*)	*C*
B	*A*	(*B*)	(*B*)	*D*
C	*D*	(*C*)	*B*	(*C*)
D	(*D*)	*B*	*A*	(*D*)

(*a*)

	x_1x_2			
y_1y_2	00	01	11	10
00	(00)	11	(00)	11
01	00	(01)	(01)	10
11	10	(11)	01	(11)
10	(10)	01	00	(10)

(*b*)

FIGURE 10.7 (*a*) Flow table for an asynchronous circuit and (*b*) excitation table.

$x_1x_2 = 01$ requires that the next stable state should be (11). However, due to the unknown delays in the feedback paths, y_1y_2 can change from (00) to (11) in three different ways:

1. Both y_1 and y_2 change simultaneously, giving a correct transition to (11).
2. y_1 changes before y_2; the circuit will go to state 10 first, followed by transitions to 01 and then to (01).
3. y_2 changes before y_1; the circuit goes to state (01).

Thus if either y_1 or y_2 changes before the other, an incorrect transition to state (01) will take place. Such a situation, in which the circuit may reach an incorrect stable state whenever two or more state variables change, is referred to as a *critical race*. As we shall see later, the state assignment in asynchronous circuits can be made in such a way that critical races are eliminated from the circuit.

Not all races are critical. A race is referred to as *noncritical* if the circuit reaches the correct stable state irrespective of the order in which the state variables change.

Example 10.2 Let us assume that the circuit specified by Figure 10.8 is in state $y_1y_2 = 00$ and $x_1 = 0$ and $x_2 = 0$. Now if the input x_2 is changed to 1, the circuit must move to (11). If both y_1 and y_2 change simultaneously, the desired stable state will be reached. If either y_1 or y_2 changes first, the circuit will move to 10 or 01, respectively. However, irrespective of which variable changes first, the circuit always reaches the correct stable state (11).

Let us now assume that the circuit of Figure 10.8 is in state (11) with $x_1x_2 = 01$. If the input state changes to 00, the circuit will enter unstable state 01 and then move to the stable state (00), as indicated in the flow table by an arrow leading from one unstable state to another. The absence of an arrow from an unstable state indicates that it is directed to its corresponding stable state. When a circuit goes through a sequence of unstable states before terminating in the desired stable state, it is said to have a cycle. It is important that a cycle terminate in a stable state; otherwise, the circuit will sequence through the unstable states indefinitely until the next change of input variable.

Example 10.3 Let us consider the circuit specified by Figure 10.8, and assume that it is in state (00) with input $x_1x_2 = 00$. If input x_1 is now changed to 1, the circuit makes a transition to 10. From 10 the circuit moves to 11, from 11 to 01, and then back to 00,

	x_1x_2			
y_1y_2	00	01	11	10
00	(00)	11	10	10
01	00	11	(01)	00
11	01	(11)	01	01
10	11	11	(10)	11

FIGURE 10.8 Excitation table of an asynchronous sequential circuit.

without any further change of input. The circuit will go through the unstable states indefinitely until the next change of an input.

10.4.2 Critical Race-Free State Assignment

Critical races can be avoided by assigning binary codes to the states in such a way that when a transition occurs from one state to another, only one secondary variable should change its value. In other words the principle of fundamental mode operation is extended to the state variables as well.

Example 10.4 Let us consider the reduced flow table of Figure 10.9. This is a three-row flow table; hence it requires at least two secondary variables for unique state assignment. An examination of the flow table reveals that there is a direct transition between rows:

A and C (column 10)
B and *A* (column 01)
B and *C* (column 10)
C and *A* (column 00)
C and *B* (column 11)

This information is summarized in the *transition diagram* shown in Figure 10.10. Each node in the diagram represents a state of the flow table. A directed line connects two nodes if transitions may occur between the corresponding states.

As mentioned previously, critical races can be avoided if only one state variable is allowed to change when a transition is made from one state to another. An arbitrary state assignment is shown in the transition diagram. However, inspection of it reveals that both the state variables y_1 and y_2 must change during the transition from *B* to *A*, resulting in a critical race. It is in fact impossible to assign state variables so that only one variable during a transition.

	x_1x_2			
State	00	01	11	10
A	(*A*), 0	(*A*), 0	(*A*), 0	*C*
B	(*B*), 1	*A*	(*B*), 1	*C*
C	*A*	(*C*), 1	*B*	(*C*), 1

FIGURE 10.9 A three-state flow table.

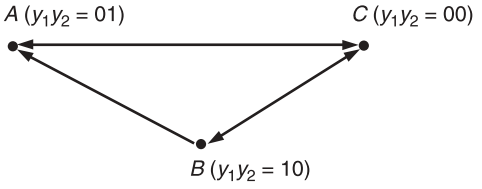


FIGURE 10.10 Transition diagram of Figure 10.9.

	x_1x_2 = 00	01	11	10
A	(A),0	(A),0	(A),0	C
B	(B),1	D	(B),1	C
C	A	(C),1	B	(C),1
D	—	A	—	—

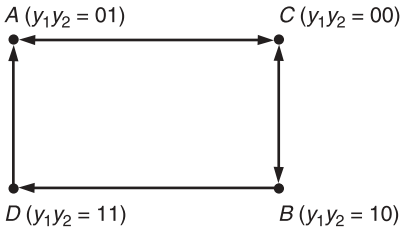


FIGURE 10.11 Critical race-free assignment with an extra state for Figure 10.9.

However, two state variables can define four states, which implies that there is a fourth or "spare" combination of secondary variables that has not been used. Figure 10.11 shows how this spare combination is utilized to overcome the critical race. If the circuit is in B under the input condition $x_1x_2 = 00$, and a change to $x_1x_2 = 01$ occurs, the circuit must go to the desired state A via the "spare" state D. This is achieved by directing the state variable changes to $y_1y_2 = 10 \rightarrow 11 \rightarrow 01$.

Thus a critical race-free assignment for a three-row flow table can be achieved by incorporating an additional row in the flow table. This row is employed to generate a path between two stable states, transitions between which would otherwise result in a critical race. The unspecified next state entries in the modified flow table corresponding to the spare state must not be assigned the same state variable combination as the spare state, otherwise an undesired stable state will be created in the fourth row.

A critical race-free state assignment is not always possible for a four-row table using two state variables.

Example 10.5 Let us consider the flow table shown in Figure 10.12. The corresponding transition diagram is shown in Figure 10.13. The transitions in the diagonal directions (e.g., D to B and C to A) indicate that no matter how the two state variables are assigned, it is not possible to satisfy the requirement that only a single variable should change during a transition. Therefore a critical race-free assignment can only be achieved by using three state variables. The race-free assignment can be obtained from the Karnaugh map of three variables, as shown in Figure 10.14*a*. Since in a Karnaugh map adjacent cells differ by a single bit, two states allocated to adjacent cells will have adjacent state assignments. There are $\binom{8}{4} = 70$ different ways of allocating the states to the cells in

	x_1x_2 = 00	01	11	10
A	(A),0	(A),0	—	B
B	C	(B),1	—	(B),1
C	(C),1	A	—	D
D	(D),1	B	—	(D),1

FIGURE 10.12 A four-row flow table.

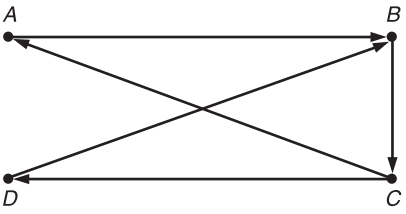
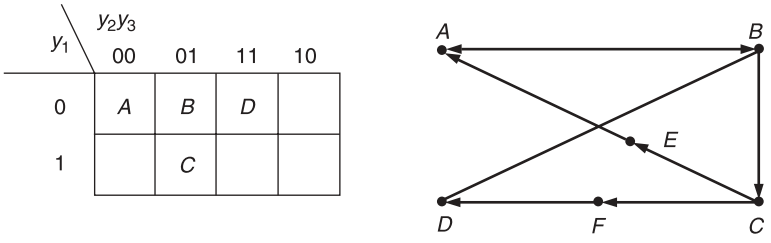


FIGURE 10.13 Transition diagram for Figure 10.12.



(*a*) (*b*)

FIGURE 10.14 (*a*) Karnaugh map for selecting race-free assignment and (*b*) race-free transition diagram.

Figure 10.14*a*. Analysis shows that these combinations fall into six different patterns [1]. One such pattern is shown in Figure 10.14*a*. In this case the state assignment is such that *A* is adjacent to *B* and *B* is adjacent to *C* and *D*. However, for a critical race-free assignment, *A* should be adjacent to *C* as should *C* to *D*. Thus the transition from *C* to *A* must be made through the spare state *E*. Similarly, the transition from *C* to *D* must be made via the spare state *F*. Hence two additional states must be included in the flow table; the transition diagram of the modified flow table is shown in Figure 10.14*b*. The modified flow table and the critical race-free assignment are shown in Figure 10.15*a* and 10.15*b*, respectively.

	x_1x_2			
	00	01	11	10
A	(*A*), 0	(*A*), 0	—	*B*
B	*C*	(*B*), 1	—	(*B*), 1
C	(*C*), 1	*E*	—	*F*
D	(*D*), 1	*B*	—	(*D*), 1
E	—	*A*	—	—
F	—	—	—	*D*

(*a*)

	y_1	y_2	y_3
A	0	0	0
B	0	0	1
C	1	0	1
D	0	1	1
E	1	0	0
F	1	1	1

(*b*)

FIGURE 10.15 (*a*) Modified flow diagram for Figure 10.12 and (*b*) critical race-free state assignment.

The critical race-free state assignment method considered here requires adding additional rows to the flow table. An alternative method for eliminating critical races assigns two combinations of state variables to each row of the flow table.

Example 10.6 Let us consider the flow table shown in Figure 10.12. Each row of the flow table is replaced with two rows as shown in Figure 10.16. For example, row A in Figure 10.12 is replaced by rows A_1 and A_2, and the stable state in each column of row A is entered in both rows A_1 and A_2. The unstable next states in each row are selected such that they are adjacent to the present state. The assignments can be obtained from the Karnaugh map as shown in Figure 10.17. The two rows corresponding to each row of the original flow table are assigned binary combinations that are complements of each other. Thus A_1 and A_2 corresponding to row A are assigned $y_1y_2y_3 = 000$ and 111, respectively. In Figure 10.12, a transition from (A) under the input change $x_1x_2 = 00 \rightarrow 10$ is directed to (B) Since the transition should change one variable only, B_1 is entered in the first row, and B_2 in the second row of column 10 in Figure 10.17; as can be seen in Figure 10.17, A_1 is adjacent to B_1 and A_2 is adjacent to B_2.

				x_1x_2			
y_1	y_2	y_3		00	01	11	10
0	0	0	A_1	(A_1), 0	(A_1), 0	—	B_1
1	1	1	A_2	(A_2), 0	(A_2), 0	—	B_2
0	0	1	B_1	C_1	(B_1), 1	—	(B_1), 1
1	1	0	B_2	C_2	(B_2), 1	—	(B_2), 1
0	1	1	C_1	(C_1), 1	A_2	—	D_1
1	0	0	C_2	(C_2), 1	A_1	—	D_2
0	1	0	D_1	(D_1), 1	B_2	—	(D_1), 1
1	0	1	D_2	(D_2), 1	B_1	—	(D_2), 1

FIGURE 10.16 Modified flow table diagram of Figure 10.12.

y_1 \ y_2y_3	00	01	11	10
0	A_1	B_1	C_1	D_1
1	C_2	D_2	A_2	B_2

FIGURE 10.17 Multiple state assignment.

Note that this method also utilizes the spare states, each state being replaced by a pair, one of which is not used in the original flow table. However, the behavior of the circuit remains unchanged irrespective of which state in a pair the circuit is in.

The major drawback of some state assignments is that the transition from one state to another in general requires a sequence of state variable changes. This has a detrimental effect on the circuit as far as speed is concerned. Significant improvement in speed can be achieved if the state assignments are made such that all the required state variable changes during a transition are completed simultaneously. Such assignments are known as *single transition time assignments* or *one-shot assignments*.

We consider one such method for critical race-free state assignment [2]. Let us assume a flow table with C columns and let column C_i $(1 \leq i \leq C)$ have x_i stable states. For example, an asynchronous circuit with 2 inputs will have 4 $(=2^2)$ columns. The procedure for the single transition time assignment consists of the following steps:

Step 1. Assign a unique combination to each stable state in a column C_i using $\lceil x_i \rceil$ number of variables.

Step 2. If the next state in column C_i corresponding to a present state is not the same as the present state itself (i.e., the present state is unstable), it is assigned the same combination of variables as the next state.

Step 3. If a column covers another column, delete the covered column. Column C_i is said to *cover* another column C_j if C_i has a 1 in every row in which C_j has a 1 whenever C_j is specified. Note that C_i also covers C_j even if the complement of C_i covers C_j.

These steps are repeated for each column of the flow table.

Example 10.7 Let us illustrate the above procedure by applying it to the flow table shown in Figure 10.18*a*. The column $x_1x_2 = 00$ has three stable states—S_0, S_2, and S_4—so 2 $(=\log_2 \lceil 3 \rceil)$ state variables y_0 and y_1 are needed to assign a unique combination to each of these states. The next state corresponding S_1 to S_2, so S_1 is assigned $y_0y_1 = 01$, the combination corresponding to S_2. Similarly, S_3 is assigned $y_0y_1 = 10$.

Column 01 has two stable states and hence requires only a single state variable y_2 to distinguish between them. Unstable states S_0, S_2, and S_4 are assigned $y_2 = 0$, 1, 0, respectively. Columns 11 and 10 have three stable states each and hence require two variables, y_3/y_4 and y_5/y_6, respectively. The complete state assignment table is shown in Figure 10.18*b*. The variable y_5 is deleted because it is covered by both y_0 and y_3. Since y_0 is identical to y_3 (i.e., they cover each other), one of them could be deleted; we delete y_3. Similarly, y_4 (identical to y_1) and y_6 (identical to y_2) are also deleted. This results in the three-variable assignment show in Figure 10.18*c*.

To prove that the state assignment of Figure 10.18*c* is indeed critical race-free, let us consider the state transition in each input column of the flow table. In column $x_1x_2 = 00$ there are two transitions: $S_1 \rightarrow S_2$ and $S_3 \rightarrow S_4$.

For $S_1 \rightarrow S_2$, $y_0y_1 = 01$; only y_2 changes value during transition. Similarly, for $S_3 \rightarrow S_4$, $y_0y_1 = 10$ and y_2 changes value. Thus there will be no critical races during these transitions.

In column $x_1x_2 = 01$ there are three transitions, $S_0 \rightarrow S_1$, $S_2 \rightarrow S_3$, and $S_4 \rightarrow S_1$. Since only y_1 changes during the $S_0 \rightarrow S_1$ transition, no critical races will occur.

(*a*)

State	x_1x_2 = 00	01	11	10
S_0	(*S*0)	S_1	(*S*0)	(*S*0)
S_1	S_2	(*S*1)	(*S*1)	S_0
S_2	(*S*2)	S_3	S_1	(*S*2)
S_3	S_4	(*S*3)	(*S*3)	S_2
S_4	(*S*4)	S_1	S_3	(*S*4)

(*b*)

	y_0	y_1	y_2	y_3	y_4	y_5	y_6
S_0	0	0	0	0	0	0	0
S_1	0	1	0	0	1	0	0
S_2	0	1	1	0	1	0	1
S_3	1	0	1	1	0	0	1
S_4	1	0	0	1	0	1	0

(*c*)

	y_0	y_1	y_2
S_0	0	0	0
S_1	0	1	0
S_2	0	1	1
S_3	1	0	1
S_4	1	0	0

FIGURE 10.18 (*a*) Flow table, (*b*) state assignment, and (*c*) reduction of state variables.

For the $S_2 \rightarrow S_3$ and $S_4 \rightarrow S_1$ transitions, two state variables (y_0 and y_1) need to change. Since none of the intermediate unstable states (000 or 110 in the case of $S_4 \rightarrow S_1$ and 001 or 111 in the case of $S_2 \rightarrow S_3$) will lead to an erroneous stable state, there is no critical race in the transition.

Similarly, it can be shown that the transitions in columns $x_1x_2 = 11$ and 10 are also critical race-free.

10.5 EXCITATION AND OUTPUT FUNCTIONS

Once the state assignment to a flow table has been made, the next step is to derive the excitation and output equations to implement the circuit. As mentioned earlier, an asynchronous circuit can be realized as a combinational circuit with feedback.

Example 10.8 Let us implement the circuit specified by the flow table of Figure 10.11. By replacing the states in Figure 10.11 with the binary assignment, the excitation table of Figure 10.19*a* results. The output table, Figure 10.19*b*, is derived by using the following rules:

(i) Assign 0 (1) to the output associated with an unstable state if it is a transient state between two stable states, both of which have outputs 0 (1) associated with them.

(ii) Assign a don't care value to the output associated with an unstable state if it is a transient state between two stable states that have different output values.

In order to implement the asynchronous circuit as a combinational circuit with feedback, the logic equations for the state variables and the output variable are derived from the Karnaugh map of functions Y_1, Y_2, and Z. Figure 10.20 shows the Karnaugh maps for Y_1, Y_2, and Z. The logic equations derived in Figure 10.20 are implemented as shown in Figure 10.21.

Asynchronous circuits can be implemented by using *SR* latches instead of feedback. The operation of the *SR* NOR latch was discussed in Chapter 4 (Section 4.3); the truth table of the latch is as follows:

S	R	Q	$\bar{Q}$
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1

	x_1x_2			
y_1y_2	00	01	11	10
01	01	01	01	00
10	10	11	10	00
00	01	00	10	00
11	—	01	—	—

y_1y_2

(*a*)

	x_1x_2			
y_1y_2	00	01	11	10
01	0	0	0	—
10	1	—	1	—
00	—	1	1	1
11	—	—	—	—

Z

(*b*)

FIGURE 10.19 (*a*) Excitation table and (*b*) output table.

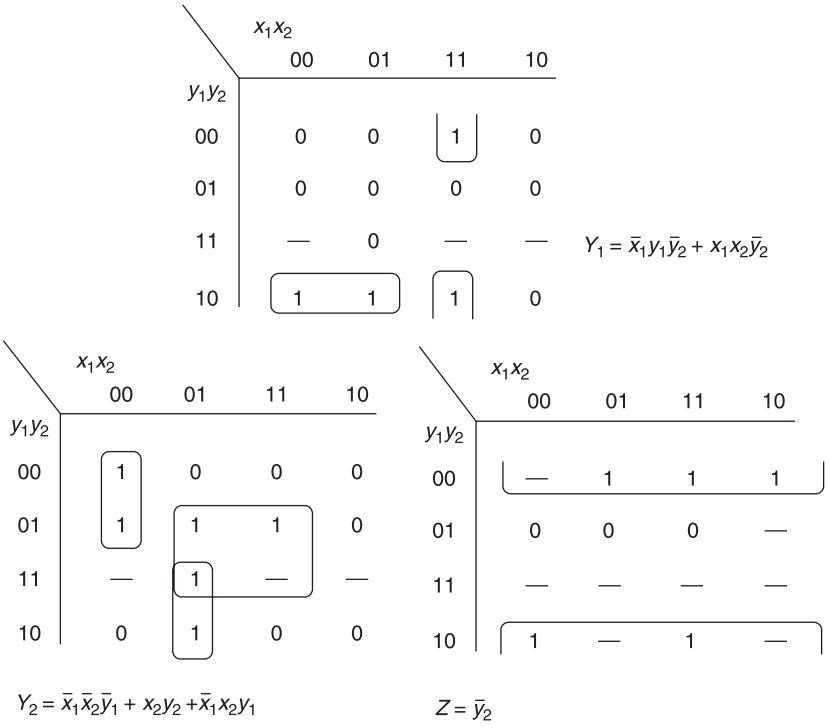


FIGURE 10.20 Karnaugh maps for the excitation and output functions.

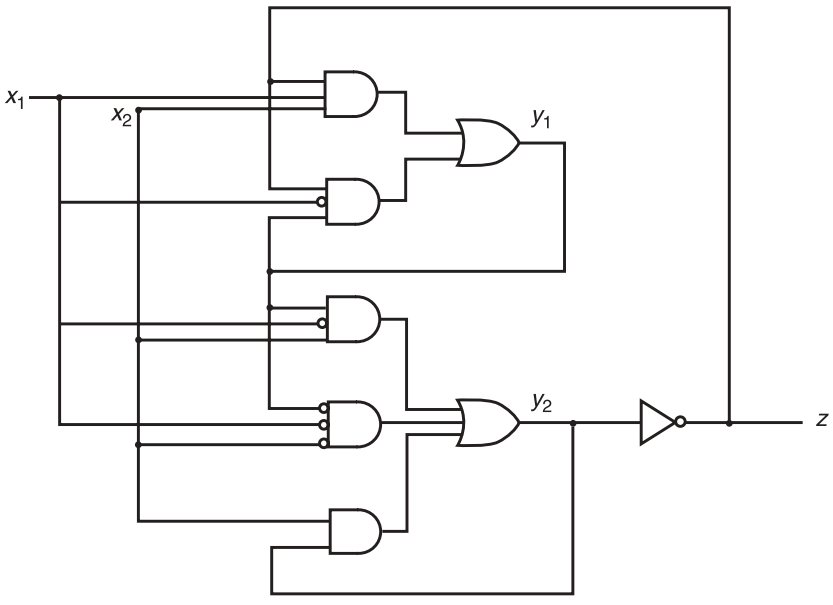


FIGURE 10.21 Realization of the asynchronous circuit.

The Karnaugh maps for an *SR* latch realization of the flow table of Figure 10.11 are derived from Figure 10.19; these are shown in Figure 10.22. The implementation of the excitation and output equations is shown in Figure 10.23.

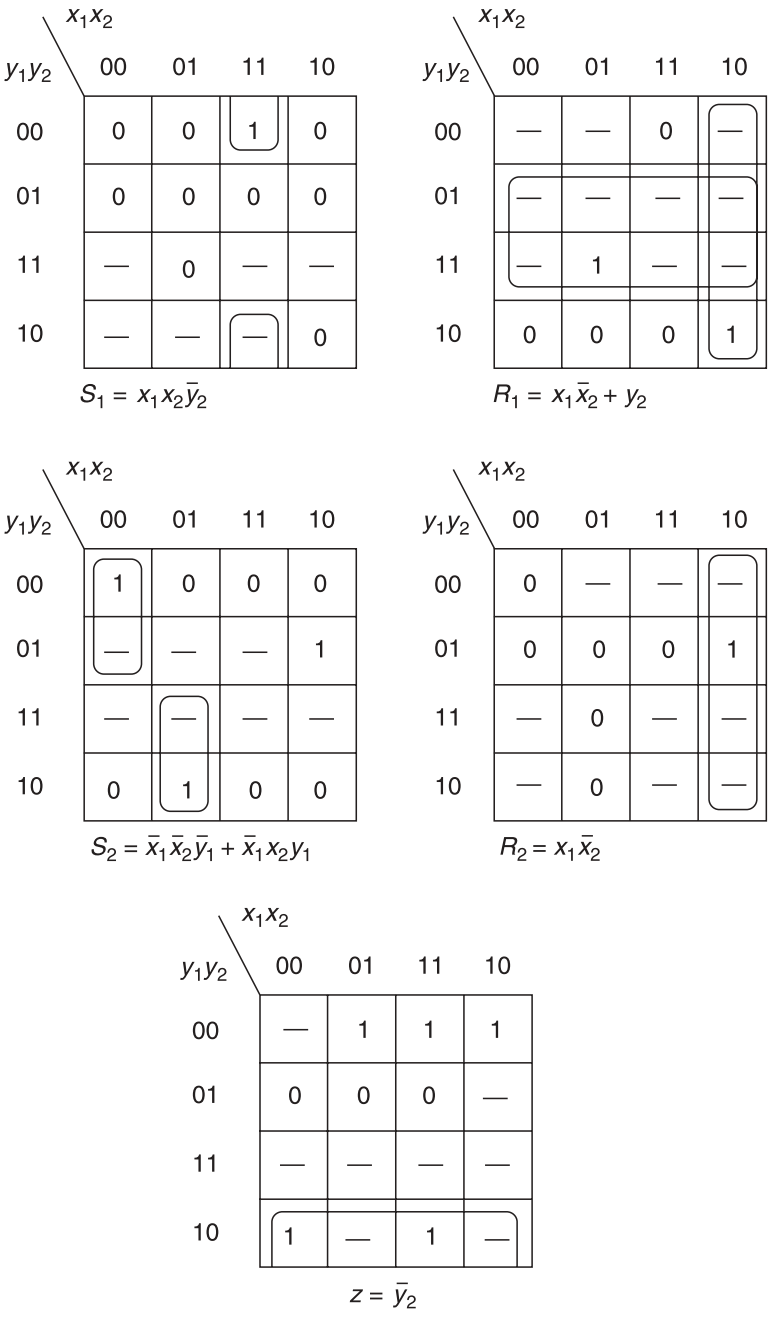


FIGURE 10.22 Karnaugh maps for *SR* latch realization.

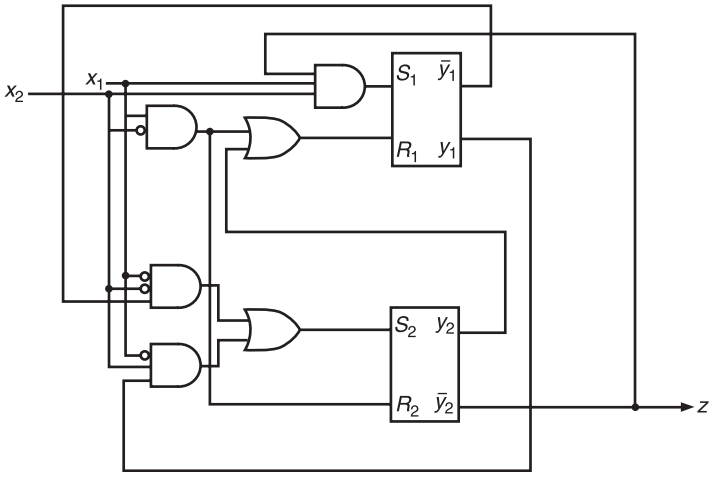


FIGURE 10.23 Implementation of the flow diagram of Figure 10.11.

10.6 HAZARDS

As discussed earlier, in asynchronous circuits all input variable and state variable changes are restricted so that only one variable can change at a time. Even then it is still possible to have erroneous outputs at times, because a variable and its complement may not change at exactly the same instant. For example, let us consider the circuit shown in Figure 10.24. Assume the inputs x_1 and x_2 are both initially at 1, which will cause the output Z to go to 1. If x_1 changes from 1 to 0, $\bar{x}_1$ also changes value, from 0 to 1, but this change does not occur at exactly the same time because of the propagation delay of the inverter. As a result, the output of the inverter will be 0 for a finite period of time, keeping the output of the AND gate at 0. Moreover, since $x_1 = 0$, the output of the other AND gate will also be 0, thus locking the output of the circuit at 0. Note that if the propagation delay of the inverter is ignored, the output of the circuit should remain at 1 when the inputs are changed from $x_1x_2 = 11$ to 01. This phenomenon in which the relative differences in delays associated with circuit elements (and interconnections) cause incorrect output is known as a *hazard*.

Hazards may also occur in combinational circuits, although not with such serious consequences as in asynchronous circuits. Figure 10.25 illustrates the occurrence of a hazard. If the circuit receives inputs $x_1x_2x_3 = 100$, the output of the circuit is at 0. If the inputs are

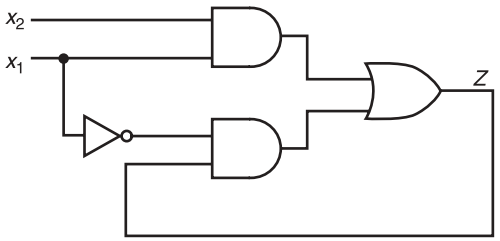


FIGURE 10.24 An asynchronous sequential circuit with a hazard.

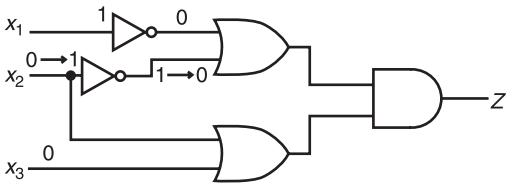


FIGURE 10.25 Combinational circuit with a hazard.

changed to $x_1x_2x_3 = 110$, the output should remain at 0; however, since the inverter fed by x_2 cannot change its output instantaneously from 1 to 0, the output of the top OR gate will be at 1. The output of the bottom OR gate will also be at 1 due to input x_2, so the circuit output momentarily goes to 1 while the inverter output is changing from 1 to 0.

There are three types of hazards: *static*, *dynamic*, and *essential*. Static and dynamic hazards may be present in combinational as well as in asynchronous circuits. Essential hazards, however, originate only in fundamental mode asynchronous circuits (see Section 10.5).

Static hazards occur when a change in a single input variable may cause a momentary change in the output that is supposed to remain constant during the change. This is often referred to as *glitch*. If the outputs before and after the change of an input variable are both 1, with a transient 0 in between, then the hazard is qualified as a *static-1 hazard*. Similarly, if the outputs before and after the change of an input variable are both 0, with a transient 1 in between, then the hazard is known as a *static-0 hazard*. The type of hazard we came across in Figure 10.25 was a static-0 hazard.

Dynamic hazards occur when the output is supposed to change due to a change in an input variable and it changes three or more times instead of only once before settling down to the proper value. Thus an output required to change from $0 \rightarrow 1$ will go through the sequence $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ due to a dynamic hazard.

Hazards can be classified into two categories: *logic hazards* and *function hazards*. Function hazards occur when more than one input variable in a circuit changes, whereas logic hazards arise because of how a circuit is realized.

10.6.1 Function Hazards

Function hazards may happen if multiple input variable changes are allowed. To illustrate function hazards, let us consider the Karnaugh map for a four-variable function, shown in Figure 10.26. Assume that initially the input combination is $x_1x_2x_3x_4 = 1001$,

x_3x_4 \ x_1x_2	00	01	11	10
00	0	0	1	0
01	0	0	0	1
11	1	0	1	0
10	0	0	0	1

FIGURE 10.26 Karnaugh map with function hazards.

and hence the corresponding output is 1. If inputs x_3 and x_4 are changed to 1 and 0, respectively the output will remain at 1, provided both x_3 and x_4 changed simultaneously; note that the output is also 1 for $x_1x_2x_3x_4 = 1010$. However, if x_3 and x_4 do not change at the same time, the input combination could be either 1011 or 1000, depending on whether x_3 or x_4 changed first. In any case this will result in a transient 0 output. The change of input combination from 1111 to 0011 will also result in a transient 0 output.

Function hazards can also be of either static or dynamic type. The change of two input variables as discussed earlier resulted in function static-0 hazard. A function dynamic hazard happens when three input variables change (e.g., from $x_1x_2x_3x_4 = 0100$ to 1001). In this case x_1, x_2, and x_4 are changing. Figure 10.27 shows the diagram formed by assuming a single input variable change at a time. For example, x_1 may change before x_2 and x_4; in that case the input combination will correspond to 1100, the leftmost entry in level 1. It is also possible that x_2 or x_4 may change first, as indicated by the middle and right entries respectively, in level 1. The entries in level 2 correspond to the second input variable change and are derived from level 1. Finally, the input variable change in level 2 completes the transition from the input combination 0100 to 1001.

As can be verified from Figure 10.26, only for the combination 1100 in level 1 is the output 1; for all other combinations the output is 0. The output is 0 for all input combinations in level 2. After the third variable change, the desired input combination $x_1x_2x_3x_4 = 1001$ results, which produces an output of 1. Thus the transition from 0100 to 1001 via 1100 and 1000 (or 1101) results in the output of $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$; in other words, the output shows dynamic hazard if the input variable x_1 changes before x_2 or x_4. In general, function hazards cannot be completely eliminated by modifying the circuit, hence the constraint of one input variable change at a time. However, because of the unpredictable nature of inputs, this restriction is not always applicable.

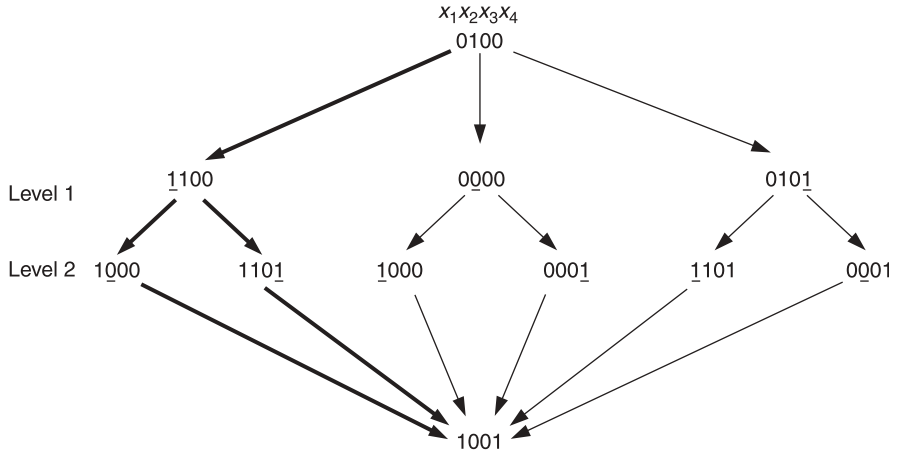


FIGURE 10.27 Transition from $x_1x_2x_3x_4 = 0100$ to 1001 (the underscore indicates that the corresponding input variable has changed).

10.6.2 Logic Hazards

Logic hazards depend on the realization of a function and can happen even if a single input variable is allowed to change at a time. Such hazards can be either static or dynamic.

Static Logic Hazards The characteristic of static logic hazards is that during an input variable change, there is a momentary change in the output, which is required to stay unchanged. Such hazards can be located from the Karnaugh map of a function. To illustrate, let us consider the function

$$f(x_1, x_2, x_3, x_4) = (x_1 + x_4)(x_2 + \bar{x}_4) + \bar{x}_1\bar{x}_2x_3$$

The Karnaugh map for the function is shown in Figure 10.28. Let the input combination at a particular time be $x_1x_2x_3x_4 = 0111$. The change in $\bar{x}_2$ from 1 to 0 makes the input combination 0011. Note that the transition from 0111 to 0011 involves changing the prime implicant from PI_2 to PI_3. Although the output for both input combinations 0111 and 0011 is 1, during the transition period $\bar{x}_2$ (the only variable that is changing) may not be equal to 1, thus resulting in a static-1 hazard. Similarly, a change in the input combination from 1101 to 1100 will also give rise to a static-1 hazard. Thus a static hazard can exist when an input variable change causes a movement between two minterms not covered by the same prime implicant, as indicated by the arrows in the Karnaugh map.

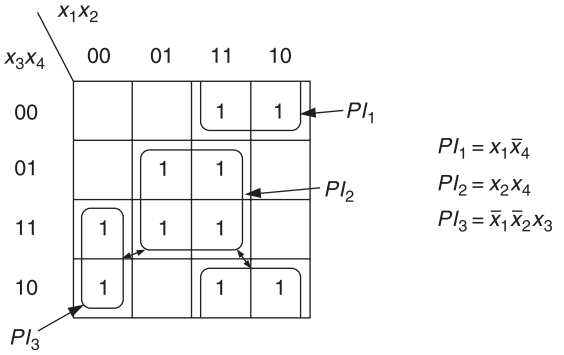


FIGURE 10.28 Karnaugh map for $f(x_1, x_2, x_3, x_4) = (x_1 + \bar{x}_4)(x_2 + \bar{x}_4) + \bar{x}_1\bar{x}_2\bar{x}_3$.

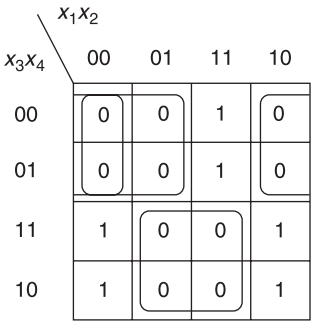


FIGURE 10.29 Karnaugh map for $f(x_1, x_2, x_3, x_4) = (x_1 + x_3)(x_2 + x_3) + (\bar{x}_2 + \bar{x}_3)$.

Let us now consider how static-0 hazards can be located from the Karnaugh map of a function. Figure 10.29 shows the Karnaugh map for the function

$$f(x_1, x_2, x_3, x_4) = (x_1 + x_3)(x_2 + x_3)(\bar{x}_2 + \bar{x}_3)$$

A static-0 hazard exists if $x_1x_2x_3x_4$ changes from 0101 to 0111; this is because the corresponding cells in that Karnaugh map are not covered by the same prime implicant.

Application of Three-Valued Logic in Hazard Detection Three-valued (ternary) logic is based on the assumption that when a logic variable changes from 0 to 1 or vice versa, it goes through a transition period where its value may be interpreted as either a 0 or 1. This indeterminate value may be represented by the symbol x, indicating the fact that the value is unknown. The truth tables for the AND, OR, and NOT functions, based on three-valued logic, can be written as follows, where a and b are inputs and Z is the output:

AND			OR			NOT	
a	b	Z	a	b	Z	a	Z
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
0	x	0	0	x	x	x	x
1	0	0	1	0	1		
1	1	1	1	1	1		
1	x	x	1	x	1		
x	0	0	x	0	x		
x	1	x	x	1	1		
x	x	x	x	x	x		

It has been shown that a static logic hazard exists in a combinational circuit for the input change

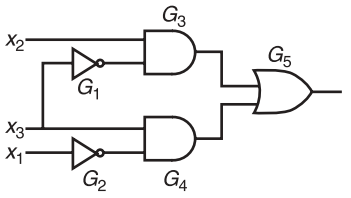
$$I_r = (I_I, \ldots, I_P, I_{P+1}, \ldots, I_n)$$

to

$$I_{t+1} = (I_1 \ldots, I_P, I_{P+1}, \ldots, I_n)$$

if the output value corresponding to the input combination I_t or I_{t+1} is the same and during the transition from I_t to I_{t+1} the output is indeterminate (i.e., x) [3]. Note that both single and multiple input changes are allowed.

Example 10.9 Let us analyze the circuit of Figure 10.30*a* to determine whether or not it contains a static hazard for the input change $x_1x_2x_3 = 010$ to 011. This is accomplished by evaluating the entire circuit using the ternary function of each gate in the circuit. Figure 10.30*b* shows the output values of each gate in response to the input change. The x at the output of G_5 indicates the existence of a hazard for the given input transition.

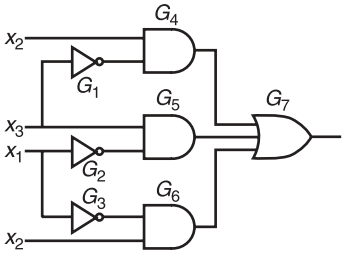


(*a*)

x_1	x_2	x_3	G_1	G_2	G_3	G_4	G_5
0	1	0	1	1	1	0	1
0	1	*x*	*x*	1	*x*	*x*	*x*
0	1	1	0	1	0	1	1

(*b*)

FIGURE 10.30 (*a*) Circuit with a static-1 hazard and (*b*) individual gate outputs.

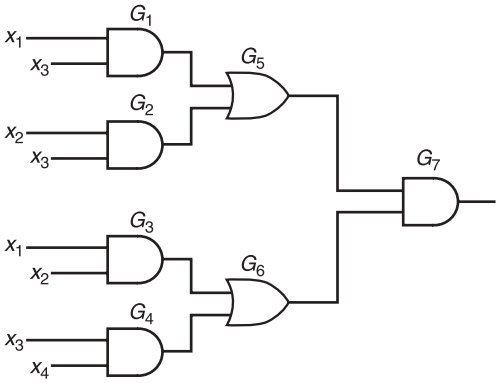


x_1	x_2	x_3	G_1	G_2	G_3	G_4	G_5	G_6	G_7
0	1	0	1	1	1	1	0	1	1
0	1	*x*	*x*	1	1	*x*	*x*	1	1
0	1	1	0	1	1	0	1	1	1

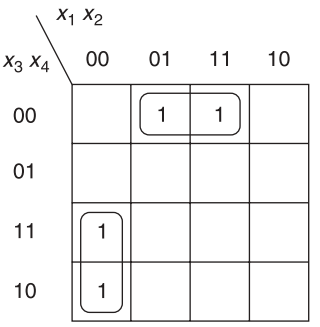
FIGURE 10.31 Hazard-free realization of Figure 10.30*a*.

Static hazards can be eliminated by grouping the midterms that cause the hazard, into one prime implicant. For example, in Figure 10.30 the two minterms in question can be combined to form the prime implicant x_1x_2. An AND gate corresponding to the prime implicant is incorporated into the circuit to make it free of static hazards, as shown in Figure 10.31.

Dynamic Logic Hazards These hazards give rise to multiple transitions at an output. They result because of the existence of at least three different paths, each with different delay time, along which the change in a single input variable can propagate through the circuit. To illustrate, let us consider the circuit shown in Figure 10.32*a*; the corresponding Karnaugh map is shown in Figure 10.32*b*.



(*a*)



(*b*)

FIGURE 10.32 (*a*) Circuit with a dynamic hazard and (*b*) Karnaugh map for the circuit function.

It can be observed from the Karnaugh map that there are no static hazards in the circuit. However, a change in x_3 can cause a dynamic hazard, because there are three different paths through the circuit for the variable x_3. Assume that the initial input combination for the circuit is $x_1x_2x_3x_4 = 0110$, and let x_3 change to 0. The three paths from x_3 to the output are

(i) Via gates G_1, G_5, and G_7
(ii) Via gates G_2, G_5, and G_7
(iii) Via gates G_4, G_6, and G_7

If G_4 changes first from 0 to 1, then the output changes from 0 to 1. Next, if G_1 changes from 1 to 0, the output goes from 1 to 0. Finally, G_2 changes from 0 to 1, which makes the output settle at 1. Thus as a result of the change in the input from 0110 to 0100, the output has changed $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$, which indicates the existence of a dynamic hazard.

Dynamic hazards cannot be eliminated by adding redundant gates as in the case of static hazards. They can be overcome only by implementing the function in a different way.

10.6.3 Essential Hazards

Essential hazards are peculiar to asynchronous sequential circuits. This type of hazard happens if the changes in an input variable propagates through the circuit via two or more paths that have unequal delays. Such a hazard can cause the circuit to terminate in an incorrect stable state.

To illustrate, let us consider the circuit shown in Figure 10.33. The excitation table for the circuit is as follows:

	x_1, x_2			
y_1y_2	00	01	11	10
00	(00)	(00)	01	(00)
01	—	00	(01)	11
11	(11)	(11)	10	(11)
10	—	11	(10)	00

Let us assume that initially $x_1 = 1$, $x_2 = 0$, $y_1 = 0$, and $y_2 = 0$. Thus $Y_1 = 0$ and $Y_2 = 0$, and the circuit is in stable state 00. Now let x_2 change from 0 to 1, and assume that the inverter that produces $\bar{x}_2$ has a propagation delay that is larger than the delays of other gates in the circuit including the feedback delay. The sequences of changes resulting from the change of x_2 from 0 to 1 are as follows:

(i) x_2 is changed from 0 to 1.
(ii) Y_2 changes from 0 to 1, which in turn changes y_2 to 1. Thus the circuit moves to state 01.

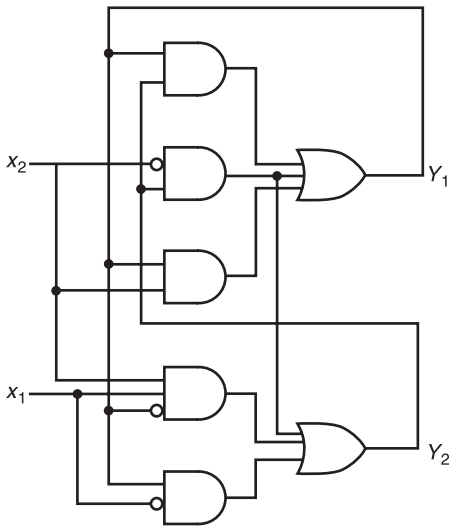


FIGURE 10.33 Circuit with essential hazards.

(iii) Since the output of the inverter generating $\bar{x}_2$ has not changed its value (i.e., its output is still at 1), Y_1 changes from 0 to 1 and Y_2 remains 1. Thus the circuit comes to state 11 (i.e., $y_1y_2 = 11$).

(iv) Assuming that the output of the inverter generating $\bar{x}_2$ has now changed from 1 to 0, Y_2 changes from 1 to 0 whereas Y_1 remains 1. The circuit now moves to the stable state 10. Thus the circuit terminates in the stable state 10 after passing through two unstable states as shown:

However, if the changes of x_1 from 0 to 1 and the change of x_2 from 0 to 1 occurred at the same time, then the circuit would have moved from 00 to 01, as can be seen in the excitation table of the circuit.

The circuit has some additional essential hazards. These happen if x_2 is changed while the circuit is in 01, 11, and 10. The essential hazards cannot be eliminated by circuit modifications, as in the case of static hazards. The only way they can be avoided is to insert delay elements in the feedback paths from the next state variables. This ensures that the change in the state variables is not fed back until the change of the input variable has been completed.

EXERCISES

1. An asynchronous sequential circuit has two inputs, x_1 and x_2, and an output Z. Input x_2 is driven by a noise-free switch. The circuit is to be designed such that a pulse on input x_1 occurring after x_2 has been pressed is transmitted to Z. Derive the excitation and output equations for the circuit.

2. The flow table of an asynchronous sequential circuit is shown below. Identify all the races that will occur if the circuit is implemented from the state table. Determine whether the races will be critical or noncritical. Derive a race-free state assignment for the circuit.

Present State	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
A	(A)	(A)	D	B
B	D	(B)	C	(B)
C	A	A	(C)	(C)
D	(D)	C	(D)	C

3. Derive a race-free state assignment for the flow table shown below. Implement a circuit corresponding to the flow table using this state assignment. Use NAND gates only.

Present State	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
A	(A), 0	(A), 0	B, 0	(A), 0
B	–,–	C	(B), 0	A
C	(C), 1	(C), 1	(C), 1	D
D	A	–,–	C	(D), 1

4. Find all the races in the following table and indicate if they are critical or not.

y_1y_2	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
00	(00)	11	(00)	11
01	11	(01)	11	11
10	00	(10)	11	11
11	(11)	(11)	00	(11)

Find a state assignment that is critical race-free.

5. A circuit with three inputs (x_1, x_2, and x_3) and four outputs (Z_1, Z_2, Z_3, and Z_4) is to be designed. Z_1 takes on the value 1 if input x_1 is changed first, followed by x_2 and then x_3. Z_2 takes on the value 1 if x_2 is changed first, followed by x_1. If the order of change is x_3 first followed by x_1 and then x_2, output Z_3 assumes the value 1. For any other order of input change, output Z_4 takes on the value 1, otherwise it remains at 0.

a. Find a minimum row flow table and a state assignment.

b. Implement the circuit using NAND gates only.

6. A circuit for implementing a combinational lock is to be designed. The circuit has two inputs, x_1 and x_2, and an output Z. The lock opens ($Z = 1$) if there is a sequence of four consecutive changes with x_2 set at 1. Find a minimum row flow table for the circuit.

7. A sequential circuit has two inputs (x_1 and x_2) and one output (z). The output becomes 1 whenever x_1 changes from 0 to 1, and becomes 0 wherever x_2 changes from 1 to 0. Find a minimum row flow table for the circuit.

8. The excitation and the output equations for a two-input (x_1, x_2) and one output (Z) asynchronous sequential circuit are as follows:

$$Y_1 = (x_1 + x_2)y_1 + \bar{x}_1\bar{x}_2y_2$$
$$Y_2 = (x_1 + x_2)\bar{y}_1 + \bar{x}_1\bar{x}_2y_2$$
$$Z = y_1 + y_2$$

Derive the flow table for the circuit.

9. Reduce the primitive flow table shown below using the approach discussed in Section 10.2.

Present State	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
A	(A,0)	–,–	–,–	B
B	C	–,–	–,–	(B,1)
C	(C,1)	F	–,–	D
D	E	–,–	–,–	D,1
E	(E,1)	G	–,–	–,–
F	A	(F,0)	–,–	–,–
G	C	(G,1)	–,–	–,–

10. The primitive flow table of an asynchronous sequential circuit is shown below.

Present State	$\bar{x}_1\bar{x}_2$	$\bar{x}_1x_2$	x_1x_2	$x_1\bar{x}_2$
A	(A,1)	B	–,–	C
B	E	(B,0)	D	–,–
C	A	–,–	F	(C,1)
D	–,–	G	(D,1)	H
E	(E,0)	B	–,–	C
F	–,–	G	(F,0)	H
G	E	(G,1)	D	–,–
H	A	–,–	F	(H,0)

Reduce the flow table using the approach discussed in Section 10.2. Also, device the logic diagram of circuit.

11. Determine a one-shot state assignment for each of the following flow tables assuming fundamental mode operation:

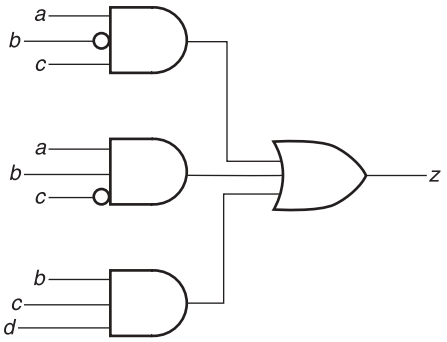
Present State	x_1x_2			
	00	01	11	10
A	B	(A)	(A)	–,–
B	(B)	F	D	(B)
C	B	(C)	(C)	–,–
D	E	(D)	(D)	–,–
E	(E)	C	A	(E)
F	B	(F)	C	–,–

(*a*)

Present State	x_1x_2			
	00	01	11	10
A	(A)	B	(A)	–,–
B	A	(B)	C	(B)
C	A	D	(C)	(C)
D	(D)	(D)	A	B

(*b*)

12. Determine whether the following circuit has a static hazard or not. If there is, specify the input condition that creates the hazard.



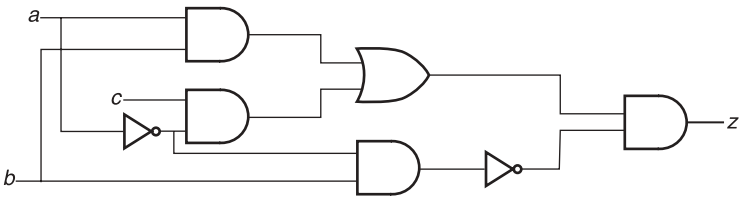
13. Find all the static hazards in the Karnaugh map of a four-variable function. Implement the circuit such that these hazards will be eliminated.

	$\bar{a}\bar{b}$	$\bar{a}b$	ab	$a\bar{b}$
$\bar{c}\bar{d}$			1	
$\bar{c}d$	1		1	
cd	1	1	1	
$c\bar{d}$				

14. Find a static hazard-free realization of the following function:

$$f(a, b, c, d) = \Sigma m(2, 3, 6, 7, 10, 11, 13, 15)$$

15. A dynamic hazard occurs in the following circuit when the output changes from 1 to 0. Specify the values of the input variables before and after the occurrence of the hazard. Assume that the inverters in the circuit are much slower than the other gates in the circuit.



REFERENCES

1. M. P. Marcus, *Switching Circuits for Engineers*, Prentice-Hall, Englewood Cliffs, NJ, 1975.
2. C. L. Liu, "A state variable assignment procedure for asynchronous sequential circuits," *Jour. ACM,* April 1963, pp. 209–215.
3. E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM Jour. Res. and Dev*., March 1965, pp. 90–99.

Appendix: CMOS Logic

CMOS technology is increasingly being used to design high-density chips. The notable advantages of CMOS are low power consumption and high noise immunity (noise immunity is the maximum noise voltage that can be tolerated). There are several ways to process CMOS circuits, each offering its own special advantage. The basic problem is to create both *n*-channel and *p*-channel transistors on a single silicon substrate. Figure A.1 shows the structure of a CMOS inverter. A *p*-channel transistor is created in an *n*-type substrate in the normal way; however, an *n*-channel transistor requires an island of *p*-type material. The source and the substrate of the *p*-channel transistor are connected to V_{DD}, whereas the source and the substrate of the *n*-channel transistor are connected to ground. V_{in} is applied to the gates of both transistors simultaneously. Note that both the transistors are enhancement-mode transistors and the *p*-channel transistor is employed as the load element. The schematic circuit representation of the inverter is shown in Figure A.2.

When V_{in} is high, the *n*-channel transistor conducts and the *p*-channel transistor becomes nonconductive. When V_{in} is low, the *p*-channel transistor becomes conductive and the *n*-channel transistor does not conduct. Because only one transistor conducts at any given time, there is little current flow from V_{DD} to ground through the transistors and power consumption is low. It should be noted that both the transistors are partially ON during the switching operation itself; however, this happens for only a fraction of the

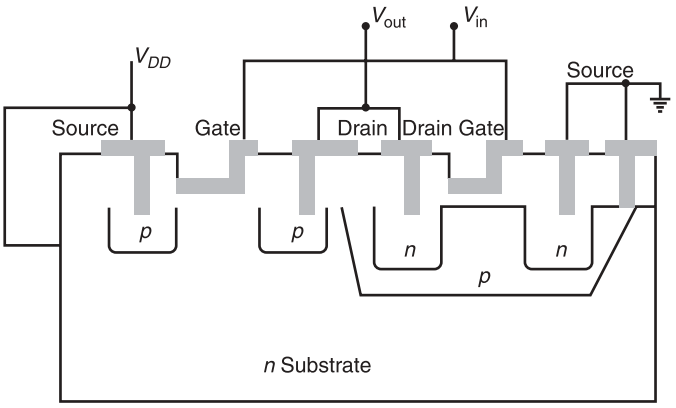


FIGURE A.1 Physical structure of a CMOS inverter.

Principles of Modern Digital Design, by Parag K. Lala

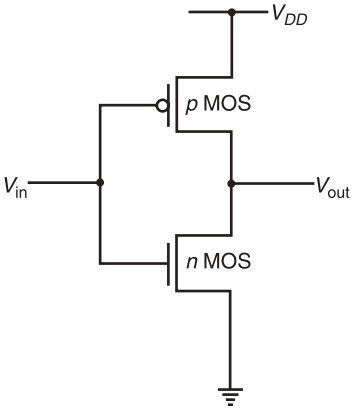


FIGURE A.2 Circuit symbol for CMOS inverter.

operating interval, so the current flow is in the microampere range. The transfer curve (i.e., the plot of V_{out} against V_{in}) of the inverter is illustrated in Figure A.3.

The basic CMOS NAND and NOR gates are shown in Figure A.4*a* and A.4*b*. A low input voltage on input *A* in the NAND circuit turns transistor T_1 ON and turns transistor T_3 OFF. Because no current can flow through T_3, the output voltage approaches V_{DD}. Similarly, a low input voltage on input *B* results in a high output voltage. Only when there are high input voltages on both inputs *A* and *B* does the current flow from V_{DD} to ground and the output goes low.

The CMOS NOR circuit (Fig. A.4*b*) is implemented by configuring the *p*-channel transistors in series and the *n*-channel transistors in parallel. A high input voltage on either input causes one of the *p*-channel transistors to be OFF and one of the *n*-channel transistors to be ON, resulting in a low output voltage. Only if both the input voltages are low do both the *p*-channel transistors turn ON, with both *n*-channel transistors OFF, to provide a high output.

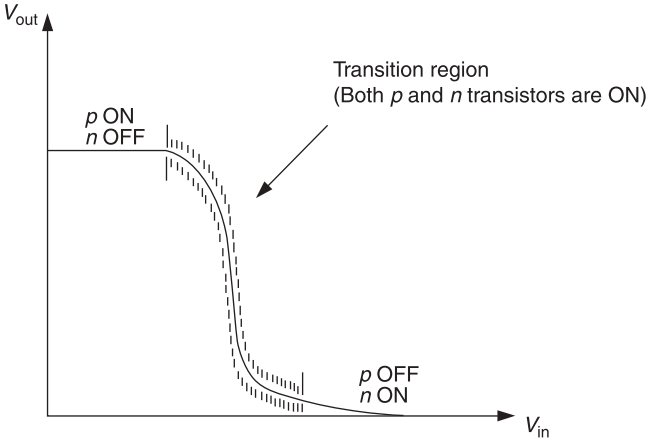


FIGURE A.3 CMOS transfer curve.

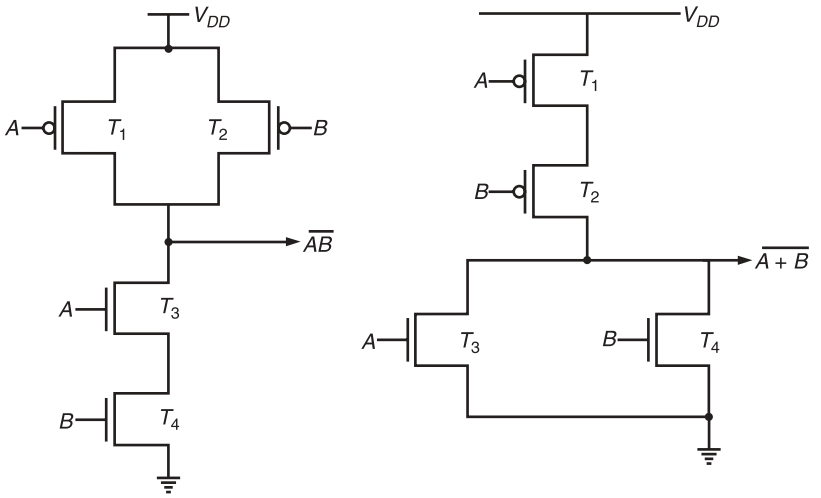


FIGURE A.4 (*a*) NAND circuit and (*b*) NOR circuit.

A.1 TRANSMISSION GATES

One of the important advantages of CMOS circuits is that they enable the construction of a nearly perfect switch—the *transmission gate*. It consists of a *p*-channel transistor connected in parallel with an *n*-channel transistor as shown in Figure A.5*a*. The transistor sources are connected to the input and their drains are connected to the output. A control voltage C is applied to the gate of the *n*-channel transistor, and its inverted value $\bar{C}$ is applied to the gate of the *p*-channel transistor.

When C is at logic 0, both transistors are nonconducting; thus the output is disconnected from the input. On the other hand, if C is at logic 1; the *p*-channel transistor transfers a high input voltage to the output, whereas the *n*-channel transistor transfers a low input voltage to the output. Thus as long as the control voltage C is high, the input is transmitted to the output. A symbol for the transmission gate is shown in Figure A.5*b*. It should be understood

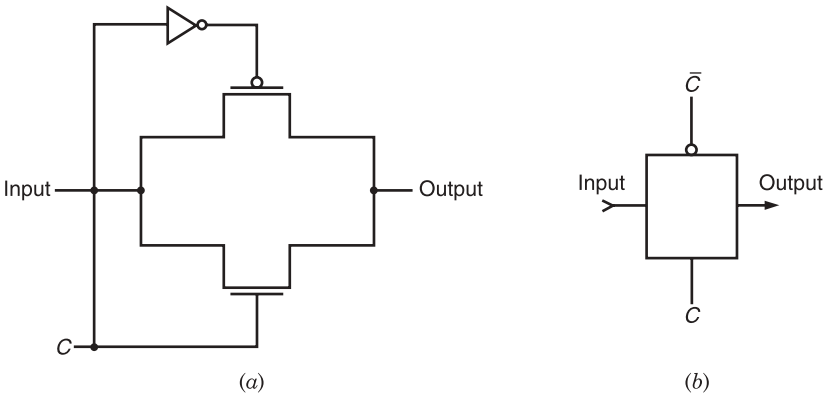


FIGURE A.5 (*a*) Transmission gate and (*b*) symbol.

Control Input *C*	Input	Output
LOW	LOW	Open circuit
LOW	HIGH	Open circuit
HIGH	LOW	LOW
HIGH	HIGH	HIGH

FIGURE A.6 Input–output relationship of the transmission gate.

that the transmission gate can transfer signals in both directions, although one end is arbitrarily labeled *input* and the other *output*. The behavior of the transmission gate is summarized in Figure A.6. It can be seen from Figure A.6 that the transmission gate is a *tristate device*. In other words, it has three possible outputs—open circuit, low, and high. However, it only has two logic levels, since open circuit really means high impedance and is not a logic level.

Figure A.7 illustrates a typical application of transmission gates. The outputs of four transmission gates are tied together to a common line, *Z*. It is desired to transmit the signals *A*, *B*, *C*, and *D* one at a time to line *Z*. This can be accomplished by making the control input of only one transmission gate at a time high while keeping the other three low. The particular transmission gate to be used can be selected by applying the appropriate input to the 2-to-4 decoder.

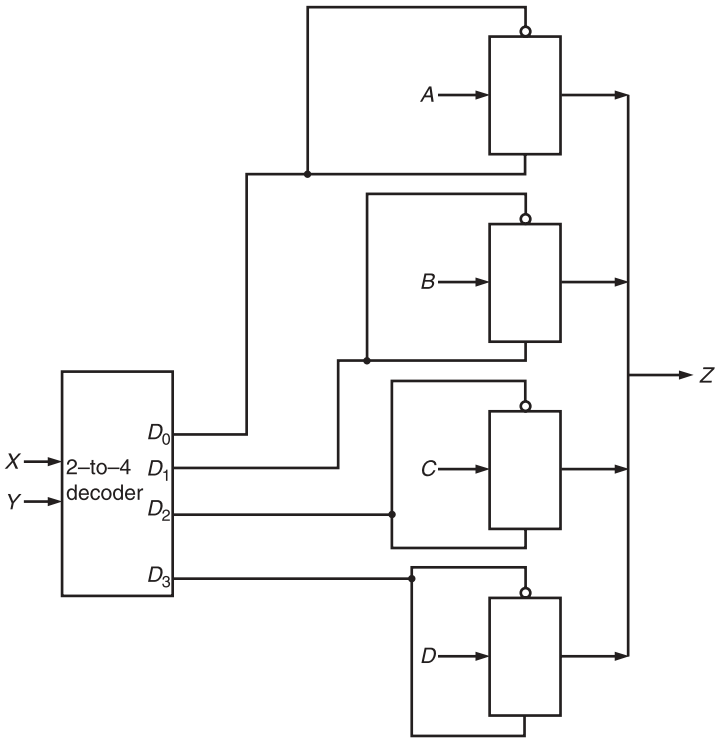


FIGURE A.7 Four signals connected to a common line using transmission gates.

A.2 CLOCKED CMOS CIRCUITS

The discussion of CMOS technology has so far concentrated on fully complementary circuits in which each gate consists of a pair of nMOS and pMOS transistors. The problem with the fully complementary approach is that for complex circuits a significant amount of chip area is wasted. This area penalty can be avoided by using clocked CMOS logic. Figure A.8 illustrates such a circuit. The basic feature of all clocked CMOS circuits is that the output node is *precharged* to V_{DD} when the clock is 0. The inputs of the circuit can be changed only during the precharge phase. When the clock goes to 1, the path to V_{DD} is opened and the path to ground is closed. Therefore, depending on the input conditions, the output will either remain high or will be pulled down during this phase, which is known as the *evaluate* phase. For example, in Figure A.9, the output Z is precharged to 1 during the time when the clock = 0. During the evaluate phase, the output Z will be pulled to ground if the function $[(A + C)B + D(E + F)] = 1$; otherwise, it will remain at 1.

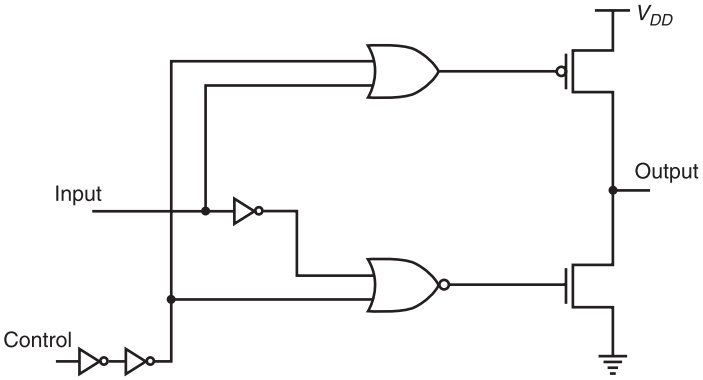


FIGURE A.8 Tristate output of an inverting buffer.

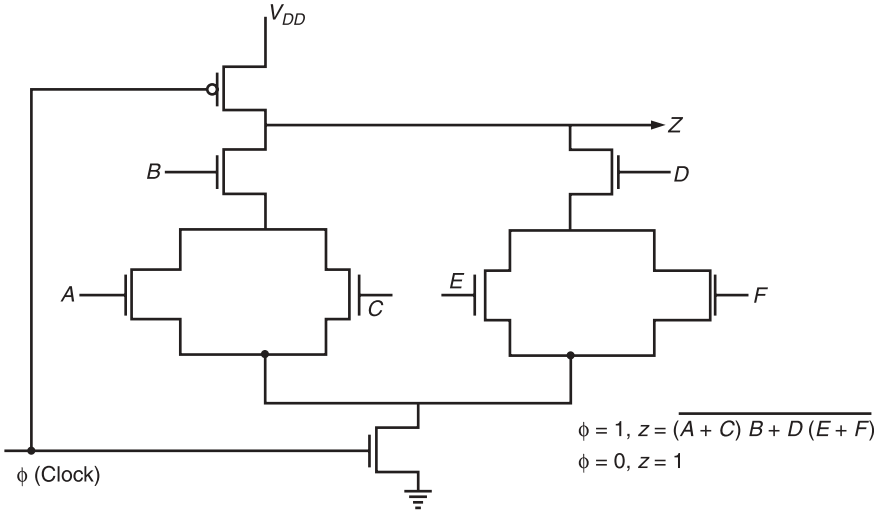


FIGURE A.9 Clocked CMOS logic.

The advantage of a clocked CMOS circuit is that it uses only an *n*-network, together with a *p*-transistor and an *n*-transistor. This results in the reduction of the load capacitance, with a consequent increase in speed. However, there are several disadvantages associated with dynamic CMOS circuits (e.g., the inputs must be changed during the precharge phase, and multiple stages cannot be cascaded to realize a function).

A.3 CMOS DOMINO LOGIC

The CMOS domino circuits and the clocked CMOS circuits have some common characteristics. Figure A.10 illustrates a domino circuit. When the clock signal is 0, transistor T_1

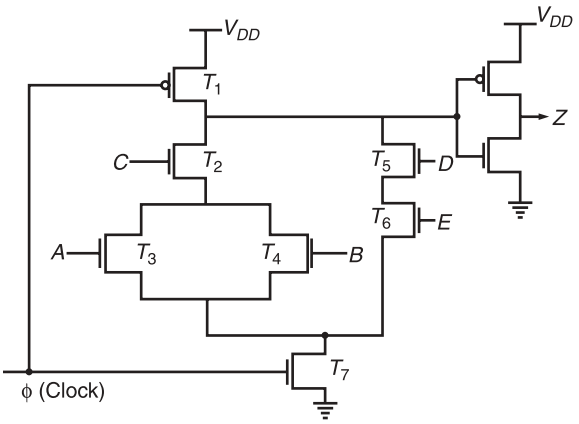


FIGURE A.10 A domino logic network.

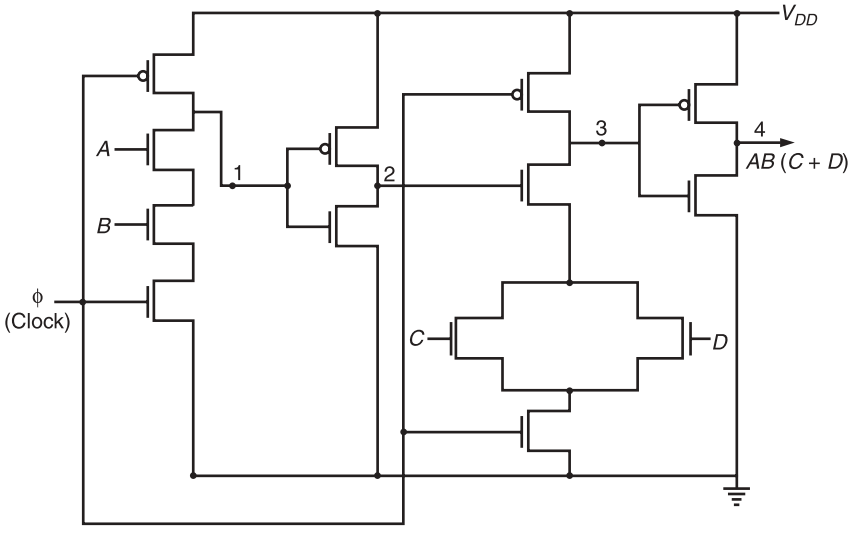


FIGURE A.11 Two-stage CMOS domino circuit.

is switched ON and transistor T_7 is switched OFF. Alternatively, T_1 is OFF and T_7 is ON when the clock signal is 1. Thus, as in clocked CMOS logic, the output is precharged high if the path to ground is open and the precharge is stopped if the path to ground is closed. The output of the clocked CMOS stage is connected to a static CMOS buffer, which feeds all subsequent logic stages. During the precharge phase the dynamic stage has a high output, so the output of the buffer will be 0. This means that all transistors in the subsequent logic stages will be turned OFF during the precharge phase. In addition, the clocked part of the circuit can only make a high-to-low transition during the evaluate phase; therefore the buffer output can only change from low to high. As a result, the output of the circuit will be hazard-free and cannot change again until the next precharge phase. Many circuits, such as the one in Figure A.10 may be cascaded to realize a function in which data is transferred from one stage to another like a series of falling dominos; hence the name *domino* logic. Figure A.11 represents a two-stage domino CMOS circuit. During the precharge phase nodes 1 and 3 are high, and nodes 2 and 4 (output node) are low. Let us assume that the inputs are $A = 1$, $B = 1$, $C = 0$, and $D = 1$. During the evaluate phase, node 1 goes low, which makes node 2 go high. Since one of the inputs to the second stage of the circuit (input D) is high, node 3 is pulled low, causing node 4 to go high.

Domino CMOS circuits provide significant speed enhancement and savings in chip area. One limitation of this structure is that each stage of the circuit must be buffered.

INDEX

Principles of Modern Digital Design, by Parag K. Lala