Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Andrew Gilfillan*
Vice-President, Production: *Vince O'Brien*
Executive Marketing Manager: *Tim Galligan*
Marketing Assistant: *Jon Bryant*
Permissions Project Manager: *Karen Sanatar*
Senior Managing Editor: *Scott Disanno*
Production Project Manager/Editorial Production Manager: *Greg Dulles*
Cover Designer: *Jayne Conte*
Cover Photo: *Michael D. Ciletti*
Composition: *Jouve India Private Limited*
Full-Service Project Management: *Jouve India Private Limited*
Printer/Binder: *Edwards Brothers*
Typeface: Times Ten 10/12

About the cover: "Spider Rock in Canyon de Chelley," Chinle, Arizona, USA, January 2011. Photograph courtesy of mdc Images, LLC (www.mdcilettiphotography.com). Used by permission.

Library of Congress Cataloging-in-Publication Data

Mano, M. Morris, 1927–
Digital design : with an introduction to the verilog hdl / M. Morris Mano, Michael D. Ciletti.—5th ed.
p. cm.
Includes index.
ISBN-13: 978-0-13-277420-8
ISBN-10: 0-13-277420-8
1. Electronic digital computers—Circuits. 2. Logic circuits. 3. Logic design. 4. Digital integrated circuits. I. Ciletti, Michael D. II. Title.
TK7888.3.M343 2011
621.39'5—dc23

2011039094

PEARSON

10 9 8 7 6 5 4 3 2 1

ISBN-13: 978-0-13-277420-8
ISBN-10: 0-13-277420-8

students to work on signficant independent design projects and to succeed in a later course in computer architecture and advanced digital design.

Instructor Resources

Instructors can download the following classroom-ready resources from the publisher's website for the text (www.pearsonhighered.com/mano):

- Source code and test benches for all Verilog HDL examples in the test
- All figures and tables in the text
- Source code for all HDL models in the solutions manual
- A downloadable solutions manual with graphics suitable for classroom presentation

HDL Simulators

The Companion Website identifies web URLs for two simulators provided by SynaptiCAD. The first simulator is *VeriLogger Pro*, a traditional Verilog simulator that can be used to simulate the HDL examples in the book and to verify the solutions of HDL problems. This simulator accepts the syntax of the IEEE-1995 standard and will be useful to those who have legacy models. As an interactive simulator, *Verilogger Extreme* accepts the syntax of IEEE 2001 as well as IEEE-1995, allowing the designer to simulate and analyze design ideas before a complete simulation model or schematic is available. This technology is particularly useful for students because they can quickly enter Boolean and *D* flip-flop or latch input equations to check equivalency or to experiment with flip-flops and latch designs. Students can access the Companion Website at www.pearsonhighered.com/mano.

Chapter Summary

The following is a brief summary of the topics that are covered in each chapter.

Chapter 1 presents the various binary systems suitable for representing information in digital systems. The binary number system is explained and binary codes are illustrated. Examples are given for addition and subtraction of signed binary numbers and decimal numbers in binary-coded decimal (BCD) format.

Chapter 2 introduces the basic postulates of Boolean algebra and shows the correlation between Boolean expressions and their corresponding logic diagrams. All possible logic operations for two variables are investigated, and the most useful logic gates used in the design of digital systems are identified. This chapter also introduces basic CMOS logic gates.

Chapter 3 covers the map method for simplifying Boolean expressions. The map method is also used to simplify digital circuits constructed with AND-OR, NAND, or NOR gates. All other possible two-level gate circuits are considered, and their method of implementation is explained. Verilog HDL is introduced together with simple examples of gate-level models.

Chapter 4 outlines the formal procedures for the analysis and design of combinational circuits. Some basic components used in the design of digital systems, such as adders and code converters, are introduced as design examples. Frequently used digital logic functions such as parallel adders and subtractors, decoders, encoders, and multiplexers are explained, and their use in the design of combinational circuits is illustrated. HDL examples are given in gate-level, dataflow, and behavioral models to show the alternative ways available for describing combinational circuits in Verilog HDL. The procedure for writing a simple test bench to provide stimulus to an HDL design is presented.

Chapter 5 outlines the formal procedures for analyzing and designing clocked (synchronous) sequential circuits. The gate structure of several types of flip-flops is presented together with a discussion on the difference between level and edge triggering. Specific examples are used to show the derivation of the state table and state diagram when analyzing a sequential circuit. A number of design examples are presented with emphasis on sequential circuits that use D-type flip-flops. Behavioral modeling in Verilog HDL for sequential circuits is explained. HDL Examples are given to illustrate Mealy and Moore models of sequential circuits.

Chapter 6 deals with various sequential circuit components such as registers, shift registers, and counters. These digital components are the basic building blocks from which more complex digital systems are constructed. HDL descriptions of shift registers and counters are presented.

Chapter 7 deals with random access memory (RAM) and programmable logic devices. Memory decoding and error correction schemes are discussed. Combinational and sequential programmable devices such as ROMs, PLAs, PALs, CPLDs, and FPGAs are presented.

Chapter 8 deals with the register transfer level (RTL) representation of digital systems. The algorithmic state machine (ASM) chart is introduced. A number of examples demonstrate the use of the ASM chart, ASMD chart, RTL representation, and HDL description in the design of digital systems. The design of a finite state machine to control a datapath is presented in detail, including the realistic situation in which status signals from the datapath are used by the state machine that controls it. This chapter is the most important chapter in the book as it provides the student with a systematic approach to more advanced design projects.

Chapter 9 outlines experiments that can be performed in the laboratory with hardware that is readily available commercially. The operation of the ICs used in the experiments is explained by referring to diagrams of similar components introduced in previous chapters. Each experiment is presented informally and the student is expected to design the circuit and formulate a procedure for checking its operation in the laboratory. The lab experiments can be used in a stand-alone manner too and can be accomplished by a traditional approach, with a breadboard and TTL circuits, or with an HDL/synthesis approach using FPGAs. Today, software for synthesizing an HDL model and implementing a circuit with an FPGA is available at no cost from vendors of FPGAs, allowing students to conduct a significant amount of work in their personal environment before using prototyping boards and other resources in a lab.

Therefore, the answer is $(0.6875)_{10} = (0.\ a_{-1}\ a_{-2}\ a_{-3}\ a_{-4})_2 = (0.1011)_2$.

To convert a decimal fraction to a number expressed in base *r,* a similar procedure is used. However, multiplication is by *r* instead of 2, and the coefficients found from the integers may range in value from 0 to $r - 1$ instead of 0 and 1.

■

EXAMPLE 1.4

Convert $(0.513)_{10}$ to octal.

$$0.513 \times 8 = 4.104$$
$$0.104 \times 8 = 0.832$$
$$0.832 \times 8 = 6.656$$
$$0.656 \times 8 = 5.248$$
$$0.248 \times 8 = 1.984$$
$$0.984 \times 8 = 7.872$$

The answer, to seven significant figures, is obtained from the integer part of the products:

$$(0.513)_{10} = (0.406517\ldots)_8$$

The conversion of decimal numbers with both integer and fraction parts is done by converting the integer and the fraction separately and then combining the two answers. Using the results of Examples 1.1 and 1.3, we obtain

$$(41.6875)_{10} = (101001.1011)_2$$

From Examples 1.2 and 1.4, we have

$$(153.513)_{10} = (231.406517)_8$$

■

1.4 OCTAL AND HEXADECIMAL NUMBERS

The conversion from and to binary, octal, and hexadecimal plays an important role in digital computers, because shorter patterns of hex characters are easier to recognize than long patterns of 1's and 0's. Since $2^3 = 8$ and $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits. The first 16 numbers in the decimal, binary, octal, and hexadecimal number systems are listed in Table 1.2.

The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three digits each, starting from the binary point and proceeding to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

$$\begin{array}{cccccccccc} (10 & 110 & 001 & 101 & 011 & \cdot & 111 & 100 & 000 & 110)_2 = (26153.7406)_8 \\ 2 & 6 & 1 & 5 & 3 & & 7 & 4 & 0 & 6 \end{array}$$

Table 1.2
Numbers with Different Bases

Decimal (base 10)	Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	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

Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of *four* digits:

$$\begin{array}{cccccccc} (10 & 1100 & 0110 & 1011 & \cdot & 1111 & 0010)_2 = (2C6B.F2)_{16} \\ 2 & C & 6 & B & & F & 2 \end{array}$$

The corresponding hexadecimal (or octal) digit for each group of binary digits is easily remembered from the values listed in Table 1.2.

Conversion from octal or hexadecimal to binary is done by reversing the preceding procedure. Each octal digit is converted to its three-digit binary equivalent. Similarly, each hexadecimal digit is converted to its four-digit binary equivalent. The procedure is illustrated in the following examples:

$$\begin{array}{ccccccccc} (673.124)_8 = (110 & 111 & 011 & \cdot & 001 & 010 & 100)_2 \\ 6 & 7 & 3 & & 1 & 2 & 4 \end{array}$$

and

$$\begin{array}{cccccc} (306.D)_{16} = (0011 & 0000 & 0110 & \cdot & 1101)_2 \\ 3 & 0 & 6 & & D \end{array}$$

Binary numbers are difficult to work with because they require three or four times as many digits as their decimal equivalents. For example, the binary number 111111111111 is equivalent to decimal 4095. However, digital computers use binary numbers, and it is sometimes necessary for the human operator or user to communicate directly with the

machine by means of such numbers. One scheme that retains the binary system in the computer, but reduces the number of digits the human must consider, utilizes the relationship between the binary number system and the octal or hexadecimal system. By this method, the human thinks in terms of octal or hexadecimal numbers and performs the required conversion by inspection when direct communication with the machine is necessary. Thus, the binary number 111111111111 has 12 digits and is expressed in octal as 7777 (4 digits) or in hexadecimal as FFF (3 digits). During communication between people (about binary numbers in the computer), the octal or hexadecimal representation is more desirable because it can be expressed more compactly with a third or a quarter of the number of digits required for the equivalent binary number. Thus, **most computer manuals use either octal or hexadecimal numbers to specify binary quantities.** The choice between them is arbitrary, although hexadecimal tends to win out, since it can represent a byte with two digits.

1.5 COMPLEMENTS OF NUMBERS

Complements are used in digital computers to **simplify the subtraction operation** and for logical manipulation. Simplifying operations leads to simpler, less expensive circuits to implement the operations. There are two types of complements for each base-r system: the radix complement and the diminished radix complement. The first is referred to as the r's complement and the second as the $(r - 1)$'s complement. When the value of the base r is substituted in the name, the two types are referred to as the 2's complement and 1's complement for binary numbers and the 10's complement and 9's complement for decimal numbers.

Diminished Radix Complement

Given a number N in base r having n digits, the $(r - 1)$'s complement of N, i.e., its diminished radix complement, is defined as $(r^n - 1) - N$. For decimal numbers, $r = 10$ and $r - 1 = 9$, so the 9's complement of N is $(10^n - 1) - N$. In this case, 10^n represents a number that consists of a single 1 followed by n 0's. $10^n - 1$ is a number represented by n 9's. For example, if $n = 4$, we have $10^4 = 10{,}000$ and $10^4 - 1 = 9999$. It follows that the 9's complement of a decimal number is obtained by subtracting each digit from 9. Here are some numerical examples:

The 9's complement of 546700 is $999999 - 546700 = 453299$.

The 9's complement of 012398 is $999999 - 012398 = 987601$.

For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, if $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus, the 1's complement of a binary number is obtained by subtracting each digit from 1. However, when subtracting binary digits from 1, we can

EXAMPLE 1.8

Repeat Example 1.7, but this time using 1's complement.

(a) $X - Y = 1010100 - 1000011$

$$\begin{aligned} X &= \quad 1010100 \\ \text{1's complement of } Y &= + \quad 0111100 \\ \text{Sum} &= \quad 10010000 \\ \text{End-around carry} &= + \qquad 1 \\ \textit{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

(b) $Y - X = 1000011 - 1010100$

$$\begin{aligned} Y &= \quad 1000011 \\ \text{1's complement of } X &= + \quad 0101011 \\ \text{Sum} &= \quad 1101110 \end{aligned}$$

There is no end carry. Therefore, the answer is $Y - X = -(\text{1's complement of } 1101110) = -0010001$.

■

Note that the negative result is obtained by taking the 1's complement of the sum, since this is the type of complement used. The procedure with end-around carry is also applicable to subtracting unsigned decimal numbers with 9's complement.

1.6 SIGNED BINARY NUMBERS

Positive integers (including zero) can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with binary digits. It is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit 0 for positive and 1 for negative.

It is important to realize that both signed and unsigned binary numbers consist of a string of bits when represented in a computer. The user determines whether the number is signed or unsigned. If the binary number is signed, then the leftmost bit represents the sign and the rest of the bits represent the number. If the binary number is assumed to be unsigned, then the leftmost bit is the most significant bit of the number. For example, the string of bits 01001 can be considered as 9 (unsigned binary) or as +9 (signed binary) because the leftmost bit is 0. The string of bits 11001 represents the binary equivalent of 25 when considered as an unsigned number and the binary equivalent of −9 when considered as a signed number. This is because the 1 that is in the leftmost position designates a negative and the other four bits represent binary 9. Usually, there is no confusion in interpreting the bits if the type of representation for the number is known in advance.

Register Transfer

A digital system is characterized by its registers and the components that perform data processing. In digital systems, a *register transfer* operation is a basic operation that consists of a transfer of binary information from one set of registers into another set of registers. The transfer may be direct, from one register to another, or may pass through data-processing circuits to perform an operation. Figure 1.1 illustrates the transfer of information among registers and demonstrates pictorially the transfer of binary information from a keyboard into a register in the memory unit. The input unit is assumed to have a keyboard, a control circuit, and an input register. Each time a key is struck, the control circuit enters an equivalent eight-bit alphanumeric character code into the input register. We shall assume that the code used is the ASCII code with an odd-parity bit. The information from the input register is transferred into the eight least significant cells of a processor register. After every transfer, the input register is cleared to enable the control to insert a new eight-bit code when the keyboard is struck again. Each eight-bit character transferred to the processor register is preceded by a shift of the previous character to the next eight cells on its left. When a transfer of four characters is completed, the processor register is full, and its contents are transferred into a memory register. The content stored in the

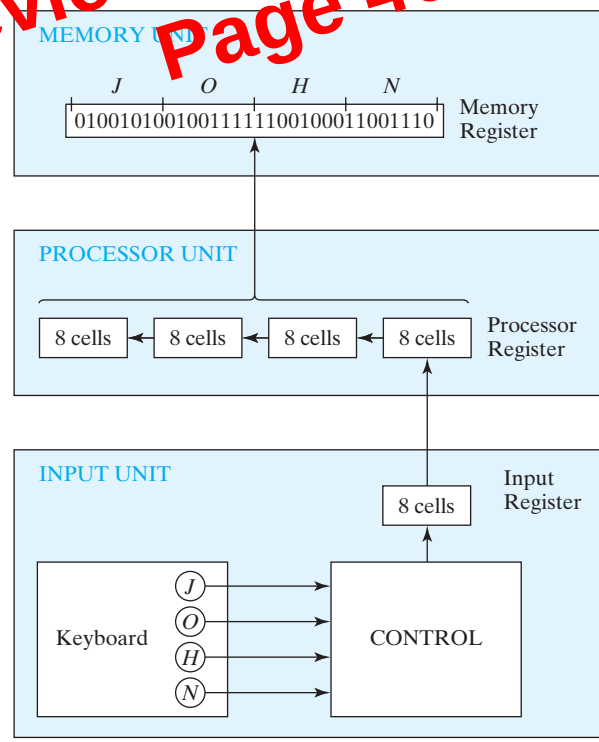


FIGURE 1.1
Transfer of information among registers

memory register shown in Fig. 1.1 came from the transfer of the characters "J," "O," "H," and "N" after the four appropriate keys were struck.

To process discrete quantities of information in binary form, a computer must be provided with devices that hold the data to be processed and with circuit elements that manipulate individual bits of information. **The device most commonly used for holding data is a register.** Binary variables are manipulated by means of digital logic circuits. Figure 1.2 illustrates the process of adding two 10-bit binary numbers. The memory unit, which normally consists of millions of registers, is shown with only three of its registers. The part of the processor unit shown consists of three registers—*R1*, *R2*, and *R3*—together with digital logic circuits that manipulate the bits of *R1* and *R2* and transfer into *R3* a binary number equal to their arithmetic sum. Memory registers store information and are incapable of processing the two operands. However, the information stored in memory can be transferred to processor registers, and the results obtained in processor registers can be transferred back into a memory register for storage until needed again. The diagram shows the contents of two operands transferred from two memory registers

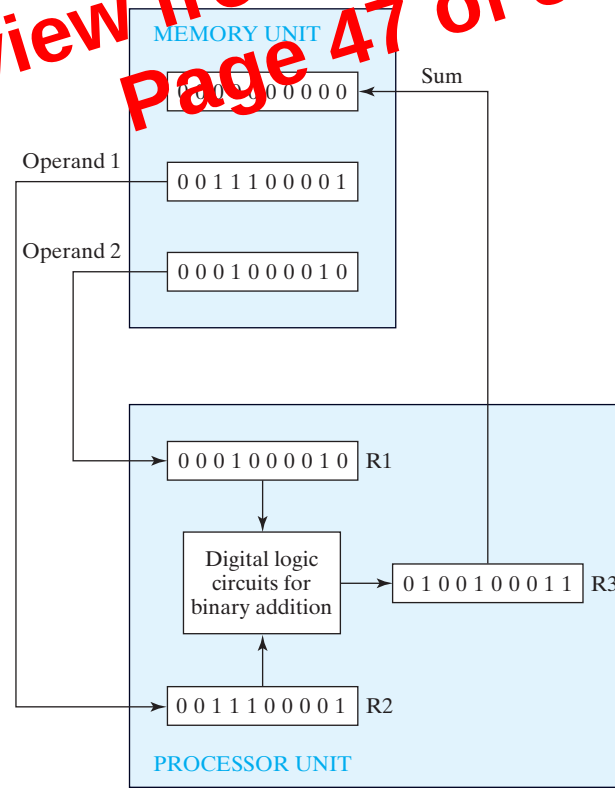


FIGURE 1.2
Example of binary information processing

Table 1.8
Truth Tables of Logical Operations

AND			OR			NOT	
x	y	$x \cdot y$	x	y	$x + y$	x	x'
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

AND and OR are the same as those used for multiplication and addition. However, **binary logic should not be confused with binary arithmetic.** One should realize that an arithmetic variable designates a number that may consist of many digits. A logic variable is always either 1 or 0. For example, in binary arithmetic we have $1 + 1 = 10$ (read "one plus one is equal to 2"), whereas in binary logic we have $1 + 1 = 1$ (read "one OR one is equal to one").

For each combination of the values of x and y, there is a value of z specified by the definition of the logical operation. Definitions of logical operations may be listed in a compact form called *truth tables*. A truth table is a table of all possible combinations of the variables, showing the relation between the values that the variables may take and the result of the operation. The truth tables for the operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs. For each combination, the result of the operation is then listed in a separate row. The truth tables for AND, OR, and NOT are given in Table 1.8. These tables clearly demonstrate the definition of the operations.

Logic Gates

Logic gates are electronic circuits that operate on one or more input signals to produce an output signal. Electrical signals such as voltages or currents exist as analog signals having values over a given continuous range, say, 0 to 3 V, but in a digital system these voltages are interpreted to be either of two recognizable values, 0 or 1. Voltage-operated logic circuits respond to two separate voltage levels that represent a binary variable equal to logic 1 or logic 0. For example, a particular digital system may define logic 0 as a signal equal to 0 V and logic 1 as a signal equal to 3 V. In practice, each voltage level has an acceptable range, as shown in Fig. 1.3. The input terminals of digital circuits accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within the specified range. The intermediate region between the allowed regions is crossed only during a state transition. Any desired information for computing or control can be operated on by passing binary signals through various combinations of logic gates, with each signal representing a particular binary variable. When the physical signal is in a particular range it is interpreted to be either a 0 or a 1.

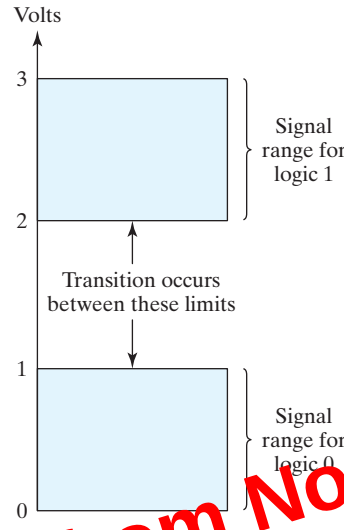
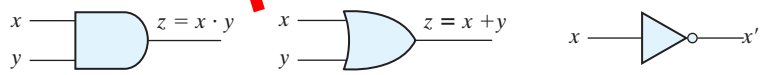


FIGURE 1.3
Signal levels for binary logic values



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

FIGURE 1.4
Symbols for digital logic circuits

The graphic symbols used to designate the three types of gates are shown in Fig. 1.4. The gates are blocks of hardware that produce the equivalent of logic-1 or logic-0 output signals if input logic requirements are satisfied. The input signals x and y in the AND and OR gates may exist in one of four possible states: 00, 10, 11, or 01. These input signals are shown in Fig. 1.5 together with the corresponding output signal for each gate. The timing diagrams illustrate the idealized response of each gate to the four input signal combinations. The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels. In reality, the transitions between logic values occur quickly, but not instantaneously. The low level represents logic 0, the high level logic 1. The AND gate responds with a logic 1 output signal when both input signals are logic 1. The OR gate responds with a logic 1 output signal if any input signal is logic 1. The NOT gate is commonly referred to as an inverter. The reason for this name is apparent from the signal response in the timing diagram, which shows that the output signal inverts the logic sense of the input signal.

WEB SEARCH TOPICS

BCD code
ASCII
Storage register
Binary logic
BCD addition
Binary codes
Binary numbers
Excess-3 code

These rules are exactly the same as the AND, OR, and NOT operations, respectively, defined in Table 1.8. We must now show that the Huntington postulates are valid for the set $B = \{0, 1\}$ and the two binary operators $+$ and $\cdot$.

1. That the structure is *closed* with respect to the two operators is obvious from the tables, since the result of each operation is either 1 or 0 and $1, 0 \in B$.
2. From the tables, we see that

   (a) $0 + 0 = 0 \qquad 0 + 1 = 1 + 0 = 1;$
   (b) $1 \cdot 1 = 1 \qquad 1 \cdot 0 = 0 \cdot 1 = 0.$

   This establishes the two *identity elements,* 0 for $+$ and 1 for $\cdot$, as defined by postulate 2.
3. The *commutative* laws are obvious from the symmetry of the binary operator tables.
4. (a) The *distributive* law $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ can be shown to hold from the operator tables by forming a truth table of all possible values of x, y, and z. For each combination, we derive $x \cdot (y + z)$ and show that the value is the same as the value of $(x \cdot y) + (x \cdot z)$:

x	y	z	$y + z$	$x \cdot (y + z)$	$x \cdot y$	$x \cdot z$	$(x \cdot y) + (x \cdot z)$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

   (b) The *distributive* law of $+$ over $\cdot$ can be shown to hold by means of a truth table similar to the one in part (a).

5. From the complement table, it is easily shown that

   (a) $x + x' = 1$, since $0 + 0' = 0 + 1 = 1$ and $1 + 1' = 1 + 0 = 1$.
   (b) $x \cdot x' = 0$, since $0 \cdot 0' = 0 \cdot 1 = 0$ and $1 \cdot 1' = 1 \cdot 0 = 0$.

   Thus, postulate 1 is verified.
6. Postulate 6 is satisfied because the two-valued Boolean algebra has two elements, 1 and 0, with $1 \neq 0$.

We have just established a two-valued Boolean algebra having a set of two elements, 1 and 0, two binary operators with rules equivalent to the AND and OR operations, and a complement operator equivalent to the NOT operator. Thus, Boolean algebra has been defined in a formal mathematical manner and has been shown to be equivalent to the binary logic presented heuristically in Section 1.9. The heuristic presentation is helpful in understanding the application of Boolean algebra to gate-type circuits. The formal

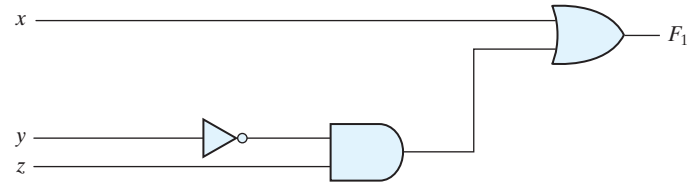


FIGURE 2.1
Gate implementation of $F_1 = x + y'z$

that combines x with $y'z$. In logic-circuit diagrams, the variables of the function are taken as the inputs of the circuit and the binary variable F_1 is taken as the output of the circuit. The schematic expresses the relationship between the output of the circuit and its inputs. Rather than listing each combination of inputs and outputs, it indicates how to compute the logic value of each output from the logic values of the inputs.

There is only one way that a Boolean function can be represented in a truth table. However, when the function is in algebraic form, it can be expressed in a variety of ways, all of which have equivalent logic. The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram. Conversely, the interconnection of gates will dictate the logic expression. Here is a key fact that motivates our use of Boolean algebra: By manipulating a Boolean expression according to the rules of Boolean algebra, it is sometimes possible to obtain a simpler expression for the same function and thus reduce the number of gates in the circuit and the number of inputs to the gate. Designers are motivated to reduce the complexity and number of gates because their effort can significantly reduce the cost of a circuit. Consider, for example, the following Boolean function:

$$F_2 = x'y'z + x'yz + xy'$$

A schematic of an implementation of this function with logic gates is shown in Fig. 2.2(a). Input variables x and y are complemented with inverters to obtain x' and y'. The three terms in the expression are implemented with three AND gates. The OR gate forms the logical OR of the three terms. The truth table for F_2 is listed in Table 2.2. The function is equal to 1 when $xyz = 001$ or 011 or when $xy = 10$ (irrespective of the value of z) and is equal to 0 otherwise. This set of conditions produces four 1's and four 0's for F_2.

Now consider the possible simplification of the function by applying some of the identities of Boolean algebra:

$$F_2 = x'y'z + x'yz + xy' = x'z(y' + y) + xy' = x'z + xy'$$

The function is reduced to only two terms and can be implemented with gates as shown in Fig. 2.2(b). It is obvious that the circuit in (b) is simpler than the one in (a), yet both implement the same function. By means of a truth table, it is possible to verify that the two expressions are equivalent. The simplified expression is equal to 1 when $xz = 01$ or when $xy = 10$. This produces the same four 1's in the truth table. Since both expressions

Table 2.6
Truth Table for $F = xy + x'z$

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Since there is a total of eight minterms or maxterms in a function of three variables, we determine the missing terms to be 0, 2, 4, and 5. The function expressed as a product of maxterms is

$$F(x, y, z) = \Pi(0, 2, 4, 5)$$

the same answer as obtained in Example 2.5.

Standard Forms

The two canonical forms of Boolean algebra are basic forms that one obtains from reading a given function from the truth table. These forms are very seldom the ones with the least number of literals, because each minterm or maxterm must contain, by definition, *all* the variables, either complemented or uncomplemented.

Another way to express Boolean functions is in *standard* form. In this configuration, the terms that form the function may contain one, two, or any number of literals. There are two types of standard forms: the sum of products and products of sums.

The *sum of products* is a Boolean expression containing AND terms, called *product terms,* with one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed as a sum of products is

$$F_1 = y' + xy + x'yz'$$

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

The logic diagram of a sum-of-products expression consists of a group of AND gates followed by a single OR gate. This configuration pattern is shown in Fig. 2.3(a). Each product term requires an AND gate, except for a term with a single literal. The logic sum is formed with an OR gate whose inputs are the outputs of the AND gates and the single literal. It is assumed that the input variables are directly available in their complements, so inverters are not included in the diagram. This circuit configuration is referred to as a *two-level implementation*.

PROBLEMS

(Answers to problems marked with * appear at the end of the text.)

2.1 Demonstrate the validity of the following identities by means of truth tables:
(a) DeMorgan's theorem for three variables: $(x + y + z)' = x'y'z'$ and $(xyz)' = x' + y' + z'$
(b) The distributive law: $x + yz = (x + y)(x + z)$
(c) The distributive law: $x(y + z) = xy + xz$
(d) The associative law: $x + (y + z) = (x + y) + z$
(e) The associative law and $x(yz) = (xy)z$

2.2 Simplify the following Boolean expressions to a minimum number of literals:
(a)* $xy + xy'$
(b)* $(x + y)(x + y')$
(c)* $xyz + x'y + xyz'$
(d)* $(A + B)'(A' + B')'$
(e) $(a + b + c')(a'b' + c)$
(f) $a'bc + abc' + ab' + a'bc'$

2.3 Simplify the following Boolean expressions to a minimum number of literals:
(a)* $ABC + A'B + ABC'$
(b)* $x'yz + xz$
(c)* $(x + y)'(x' + y')$
(d)* $xy + x(wz + wz')$
(e)* $(BC' + A'D)(AB' + CD')$
(f) $(a' + c')(a + b + c')$

2.4 Reduce the following Boolean expressions to the indicated number of literals:
(a) $A'B' + ABC + AC'$ to three literals
(b) $(x'y' + z)' + z + xy$ to three literals
(c)* $A'B(D' + C'D) + B(A + A'CD)$ to one literal
(d)* $(A' + C)(A' + C')(A + B + C'D)$ to four literals
(e) $ABC'D + A'BD + ABCD$ to two literals

2.5 Draw logic diagrams of the circuits that implement the original and simplified expressions in Problem 2.2.

2.6 Draw logic diagrams of the circuits that implement the original and simplified expressions in Problem 2.3.

2.7 Draw logic diagrams of the circuits that implement the original and simplified expressions in Problem 2.4.

2.8 Find the complement of $F = wx + yz$; then show that $FF' = 0$ and $F + F' = 1$.

2.9 Find the complement of the following expressions:
(a)* $xy' + x'y$
(b) $(a + c)(a + b')(a' + b + c')$
(c) $z + z'(v'w + xy)$

2.10 Given the Boolean functions F_1 and F_2, show that
(a) The Boolean function $E = F_1 + F_2$ contains the sum of the minterms of F_1 and F_2.
(b) The Boolean function $G = F_1F_2$ contains only the minterms that are common to F_1 and F_2.

2.11 List the truth table of the function:
(a)* $F = xy + xy' + y'z$
(b) $F = bc + a'c'$

2.12 We can perform logical operations on strings of bits by considering each pair of corresponding bits separately (called bitwise operation). Given two eight-bit strings $A = 10110001$ and $B = 10101100$, evaluate the eight-bit result after the following logical operations:
(a)* AND (b) OR (c)* XOR (d)* NOT A (e) NOT B

If a function is not expressed in sum-of-minterms form, it is possible to use the map to obtain the minterms of the function and then simplify the function to an expression with a minimum number of terms. It is necessary, however, to make sure that the algebraic expression is in sum-of-products form. Each product term can be plotted in the map in one, two, or more squares. The minterms of the function are then read directly from the map.

EXAMPLE 3.4

For the Boolean function

$$F = A'C + A'B + AB'C + BC$$

(a) Express this function as a sum of minterms.

(b) Find the minimal sum-of-products expression.

Note that F is a sum of products. Three product terms in the expression have two literals and are represented in a three-variable map by two squares each. The two squares corresponding to the first term, $A'C$, are found in Fig. 3.7 from the coincidence of A' (first row) and C (two middle columns) to give squares 001 and 011. Note that, in marking 1's in the squares, it is possible to find a 1 already placed there from a preceding term. This happens with the second term, $A'B$, which has 1's in squares 011 and 010. Square 011 is common with the first term, $A'C$, though, so only one 1 is marked in it. Continuing in this fashion, we determine that the term $AB'C$ belongs in square 101, corresponding to minterm 5, and the term BC has two 1's in squares 011 and 111. The function has a total of five minterms, as indicated by the five 1's in the map of Fig. 3.7. The minterms are read directly from the map to be 1, 2, 3, 5, and 7. The function can be expressed in sum-of-minterms form as

$$F(A, B, C) = \Sigma(1, 2, 3, 5, 7)$$

The sum-of-products expression, as originally given, has too many terms. It can be simplified, as shown in the map, to an expression with only two terms:

$$F = C + A'B$$

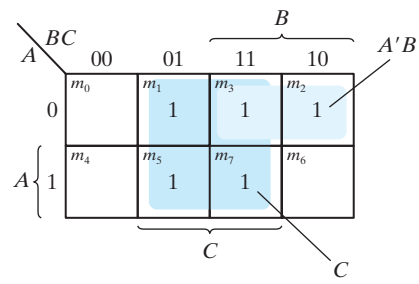


FIGURE 3.7
Map of Example 3.4, $A'C + A'B + AB'C + BC = C + A'B$

that cover minterms m_3, m_9, and m_{11}. There are four possible ways that the function can be expressed with four product terms of two literals each:

$$\begin{aligned} F &= BD + B'D' + CD + AD \\ &= BD + B'D' + CD + AB' \\ &= BD + B'D' + B'C + AD \\ &= BD + B'D' + B'C + AB' \end{aligned}$$

The previous example has demonstrated that the identification of the prime implicants in the map helps in determining the alternatives that are available for obtaining a simplified expression.

The procedure for finding the simplified expression from the map requires that we first determine all the essential prime implicants. The simplified expression is obtained from the logical sum of all the essential prime implicants plus other prime implicants that may be needed to cover any remaining minterms not covered by the essential prime implicants. Occasionally, there may be more than one way of combining squares, and each combination may produce an equally simplified expression.

Five-Variable Map

Maps for more than four variables are not as simple to use as maps for four or fewer variables. A five-variable map needs 32 squares and a six-variable map needs 64 squares. When the number of variables becomes large, the number of squares becomes excessive and the geometry for combining adjacent squares becomes more involved.

Maps for more than four variables are difficult to use and will not be considered here.

3.4 PRODUCT-OF-SUMS SIMPLIFICATION

The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the product-of-sums form can be obtained.

The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the standard sum-of-products form of a function denote the complement of the function. From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified sum-of-products expression of the complement of the function (i.e., of F'). The complement of F' gives us back the function F in product-of-sums form (a consequence of DeMorgan's theorem). Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form. The best way to show this is by example.

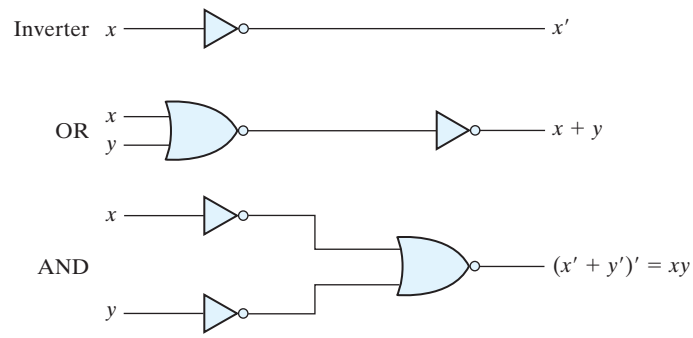


FIGURE 3.22
Logic operations with NOR gates

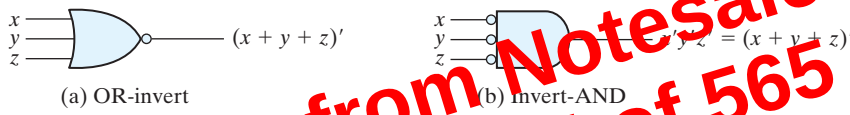


FIGURE 3.23
Two graphic symbols for the NOR gate

A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form. Remember that the simplified product-of-sums expression is obtained from the map by combining the 0's and complementing. A product-of-sums expression is implemented with a first level of OR gates that produce the sum terms followed by a second-level AND gate to produce the product. The transformation from the OR–AND diagram to a NOR diagram is achieved by changing the OR gates to NOR gates with OR-invert graphic symbols and the AND gate to a NOR gate with an invert-AND graphic symbol. A single literal term going into the second-level gate must be complemented. Figure 3.24 shows the NOR implementation of a function expressed as a product of sums:

$$F = (A + B)(C + D)E$$

The OR–AND pattern can easily be detected by the removal of the bubbles along the same line. Variable E is complemented to compensate for the third bubble at the input of the second-level gate.

The procedure for converting a multilevel AND–OR diagram to an all-NOR diagram is similar to the one presented for NAND gates. For the NOR case, we must convert each OR gate to an OR-invert symbol and each AND gate to an invert-AND symbol. Any bubble that is not compensated by another bubble along the same line needs an inverter, or the complementation of the input literal.

The transformation of the AND–OR diagram of Fig. 3.21(a) into a NOR diagram is shown in Fig. 3.25. The Boolean function for this circuit is

$$F = (AB' + A'B)(C + D')$$

3.12 Simplify the following Boolean functions:

(a)* $F(A, B, C, D) = \Pi(1, 3, 5, 7, 13, 15)$

(b) $F(A, B, C, D) = \Pi(1, 3, 6, 9, 11, 12, 14)$

3.13 Simplify the following expressions to (1) sum-of-products and (2) products-of-sums:

(a)* $x'z' + y'z' + yz' + xy$

(b) $ACD' + C'D + AB' + ABCD$

(c) $(A' + B + D')(A' + B' + C')(A' + B' + C)(B' + C + D')$

(d) $BCD' + ABC' + ACD$

3.14 Give three possible ways to express the following Boolean function with eight or fewer literals:

$$F = A'BC'D + AB'CD + A'B'C' + ACD'$$

3.15 Simplify the following Boolean function F, together with the don't-care conditions d, and then express the simplified function in sum-of-minterms form:

(a) $F(x, y, z) = \Sigma(0, 1, 4, 5, 6)$
$d(x, y, z) = \Sigma(2, 3, 7)$

(b)* $F(A, B, C, D) = \Sigma(0, 6, 8, 13, 14)$
$d(A, B, C, D) = \Sigma(2, 4, 10)$

(c) $F(A, B, C, D) = \Sigma(5, 6, 7, 12, 14, 15)$
$d(A, B, C, D) = \Sigma(3, 9, 11, 15)$

(d) $F(A, B, C, D) = \Sigma(1, 4, 12, 7, 2, 10,)$
$d(A, B, C, D) = \Sigma(0, 6, 8)$

3.16 Simplify the following functions, and implement them with two-level NAND gate circuits:

(a) $F(A, B, C, D) = AB'D + B'C + ABC + AB'C + A'C'D'$

(b) $F(A, B, C, D) = A'B'C'D + CD + AC'D$

(c) $F(A, B, C) = (A' + C' + D')(A' + C')(C' + D')$

(d) $F(A, B, C, D) = A' + B + D' + B'C$

3.17* Draw a NAND logic diagram that implements the complement of the following function:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 3, 6, 10, 11, 14)$$

3.18 Draw a logic diagram using only two-input NOR gates to implement the following function:

$$F(A, B, C, D) = (A \oplus B)'(C \oplus D)$$

3.19 Simplify the following functions, and implement them with two-level NOR gate circuits:

(a)* $F = wx' + y'z' + w'yz'$

(b) $F(w, x, y, z) = \Sigma(0, 3, 12, 15)$

(c) $F(x, y, z) = [(x + y)(x = z)]'$

3.20 Draw the multiple-level NOR circuit for the following expression:

$$CD(B + C)A + (BC' + DE')$$

3.21 Draw the multiple-level NAND circuit for the following expression:

$$w(x + y + z) + xyz$$

3.22 Convert the logic diagram of the circuit shown in Fig. 4.4 into a multiple-level NAND circuit.

3.23 Implement the following Boolean function F, together with the don't-care conditions d, using no more than two NOR gates:

$$F(A, B, C, D) = \Sigma(2, 4, 10, 12, 14,)$$
$$d(A, B, C, D) = \Sigma(0, 1, 5, 8)$$

Assume that both the normal and complement inputs are available.

7. MANO, M. M. and C. R. KIME. 2004. *Logic and Computer Design Fundamentals*, 3rd ed. Upper Saddle River, NJ: Prentice Hall.
8. MCCLUSKEY, E. J. 1986. *Logic Design Principles*. Englewood Cliffs, NJ: Prentice-Hall.
9. PALNITKAR, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall title).

WEB SEARCH TOPICS

Boolean minimization
Karnaugh map
Wired logic
Emitter-coupled logic
Open-collector logic
Quine McCluskey method
Expresso software
Consensus theorem
Don't-care conditions

Table 4.1
Truth Table for the Logic Diagram of Fig. 4.2

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

This process is illustrated with the circuit of Fig. 4.2. In Table 4.1, we form the eight possible combinations for the three input variables. The truth table for F_2 is determined directly from the values of A, B, and C, with F_2 equal to 1 for any combination that has two or three inputs equal to 1. The truth table for F_2' is the complement of F_2. The truth tables for T_1 and T_2 are the OR and AND functions of the input variables, respectively. The values for T_3 are derived from T_1 and F_2': T_3 is equal to 1 when both T_1 and F_2' are equal to 1, and T_3 is equal to 0 otherwise. Finally, F_1 is equal to 1 for those combinations in which either T_2 or T_3 or both are equal to 1. Inspection of the truth table combinations for A, B, C, F_1, and F_2 shows that it is identical to the truth table of the full adder given in Section 4.5 for x, y, z, S, and C, respectively.

Another way of analyzing a combinational circuit is by means of logic simulation. This is not practical, however, because the number of input patterns that might be needed to generate meaningful outputs could be very large. But simulation has a very practical application in verifying that the functionality of a circuit actually matches its specification. In Section 4.12, we demonstrate the logic simulation and verification of the circuit of Fig. 4.2, using Verilog HDL.

4.4 DESIGN PROCEDURE

The design of combinational circuits starts from the specification of the design objective and culminates in a logic circuit diagram or a set of Boolean functions from which the logic diagram can be obtained. The procedure involves the following steps:

1. From the specifications of the circuit, determine the required number of inputs and outputs and assign a symbol to each.
2. Derive the truth table that defines the required relationship between inputs and outputs.

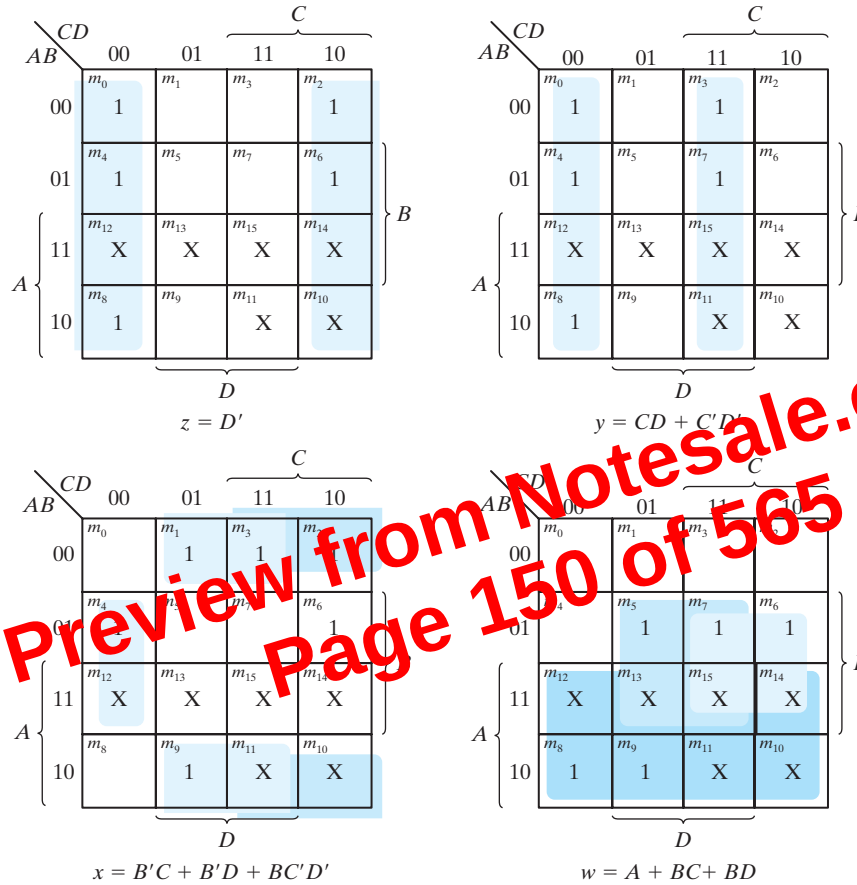


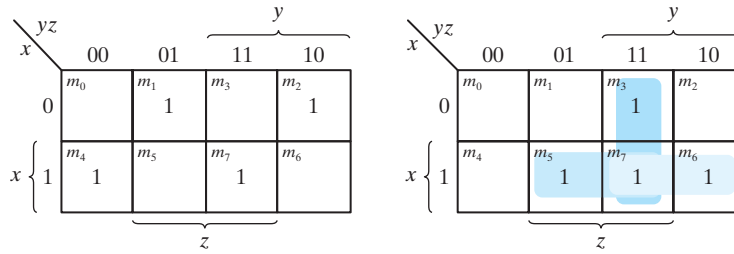
FIGURE 4.3
Maps for BCD-to-excess-3 code converter

implemented with three or more levels of gates:

$$z = D'$$
$$y = CD + C'D' = CD + (C + D)'$$
$$x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$$
$$= B'(C + D) + B(C + D)'$$
$$w = A + BC + BD = A + B(C + D)$$

The logic diagram that implements these expressions is shown in Fig. 4.4. Note that the OR gate whose output is $C + D$ has been used to implement partially each of three outputs.

Not counting input inverters, the implementation in sum-of-products form requires seven AND gates and three OR gates. The implementation of Fig. 4.4 requires four AND gates, four OR gates, and one inverter. If only the normal inputs are available, the first



(a) $S = x'y'z + x'yz' + xy'z' + xyz$ (b) $C = xy + xz + yz$

FIGURE 4.6
K-Maps for full adder

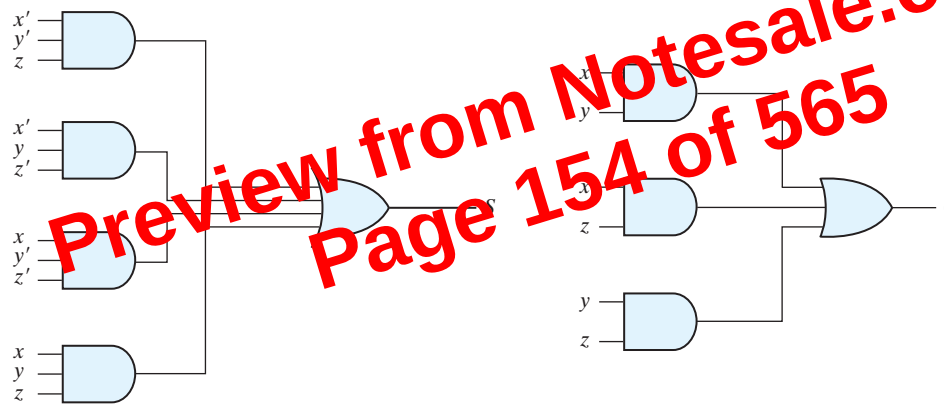


FIGURE 4.7
Implementation of full adder in sum-of-products form

in Fig. 4.8. The S output from the second half adder is the exclusive-OR of z and the output of the first half adder, giving

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

The carry output is

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

Binary Adder

A binary adder is a digital circuit that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the output carry from each full adder connected to the input carry of the next full adder in the chain.

output can be equal to 0 at any given time; all other outputs are equal to 1. The output whose value is equal to 0 represents the minterm selected by inputs A and B. The circuit is disabled when E is equal to 1, regardless of the values of the other two inputs. When the circuit is disabled, none of the outputs are equal to 0 and none of the minterms are selected. In general, a decoder may operate with complemented or uncomplemented outputs. The enable input may be activated with a 0 or with a 1 signal. Some decoders have two or more enable inputs that must satisfy a given logic condition in order to enable the circuit.

A decoder with enable input can function as a *demultiplexer*—a circuit that receives information from a single line and directs it to one of 2^n possible output lines. The selection of a specific output is controlled by the bit combination of n selection lines. The decoder of Fig. 4.19 can function as a one-to-four-line demultiplexer when E is taken as a data input line and A and B are taken as the selection inputs. The single input variable E has a path to all four outputs, but the input information is directed to only one of the output lines, as specified by the binary combination of the two selection lines A and B. This feature can be verified from the truth table of the circuit. For example, if the selection lines $AB = 10$, output D_2 will be the same as the input value E, while all other outputs are maintained at 1. Because a decoder and demultiplexer operations are obtained from the same circuit, a decoder with an enable input is referred to as a *decoder–demultiplexer*.

Decoders with enable inputs can be connected together to form a larger decoder circuit. Figure 4.20 shows two 3-to-8-line decoders with enable inputs connected to form a 4-to-16-line decoder. When $w = 0$, the top decoder is enabled and the other is disabled. The bottom decoder outputs are all 0's, and the top eight outputs generate minterms 0000 to 0111. When $w = 1$, the enable conditions are reversed: The bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 0's. This example demonstrates the usefulness of enable inputs in decoders and other

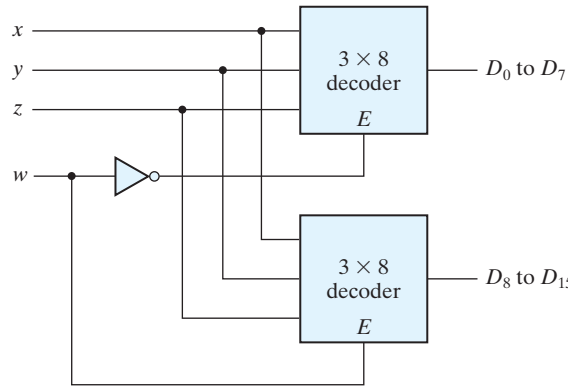


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

combinational logic components. In general, enable inputs are a convenient feature for interconnecting two or more standard components for the purpose of combining them into a similar function with more inputs and outputs.

Combinational Logic Implementation

A decoder provides the 2^n minterms of n input variables. Each asserted output of the decoder is associated with a unique pattern of input bits. Since any Boolean function can be expressed in sum-of-minterms form, a decoder that generates the minterms of the function, together with an external OR gate that forms their logical sum, provides a hardware implementation of the function. In this way, any combinational circuit with n inputs and m outputs can be implemented with an n-to-2^n-line decoder and m OR gates.

The procedure for implementing a combinational circuit by means of a decoder and OR gates requires that the Boolean function for the circuit be expressed as a sum of minterms. A decoder is then chosen that generates all the minterms of the input variables. The inputs to each OR gate are selected from the decoder outputs according to the list of minterms of each function. This procedure will be illustrated by an example that implements a full adder circuit.

From the truth table of the full adder (see Table 4.4), we obtain the functions for the combinational circuit in sum-of-minterms form:

$$S(x, y, z) = \Sigma(1, 2, 4, 7)$$

$$C(x, y, z) = \Sigma(3, 5, 6, 7)$$

Since there are three inputs and a total of eight minterms, we need a three-to-eight-line decoder. The implementation is shown in Fig. 4.21. The decoder generates the eight minterms for x, y, and z. The OR gate for output S forms the logical sum of minterms 1, 2, 4, and 7. The OR gate for output C forms the logical sum of minterms 3, 5, 6, and 7.

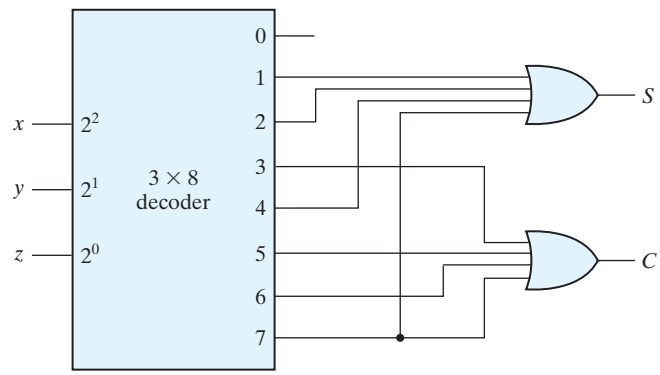


FIGURE 4.21
Implementation of a full adder with a decoder

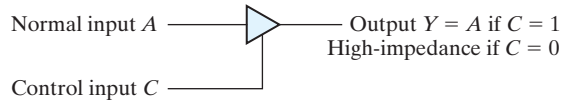
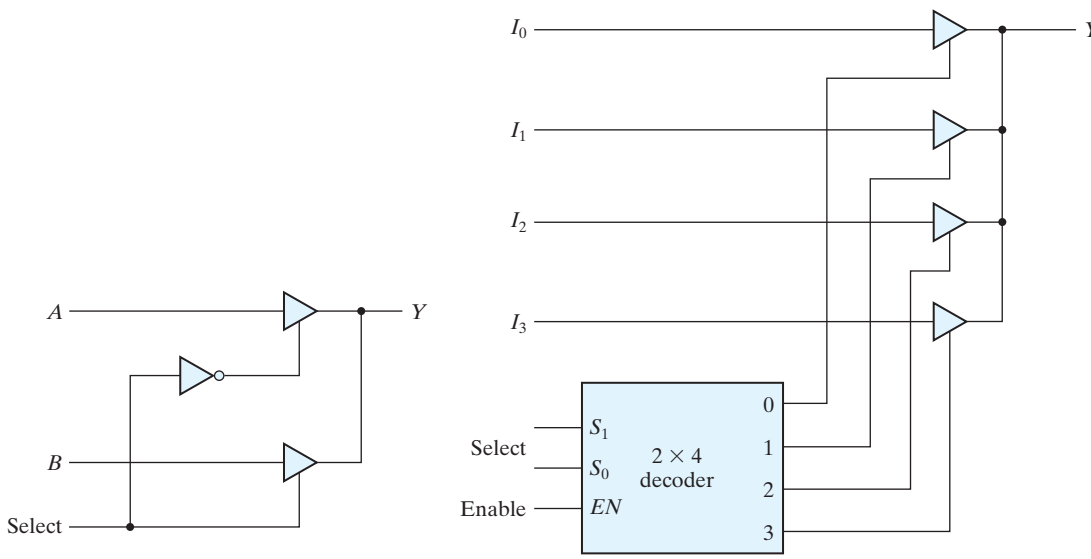


FIGURE 4.29
Graphic symbol for a three-state buffer

input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common line without endangering loading effects.

The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30. Figure 4.30(a) shows the construction of a two-to-one-line multiplexer with 2 three-state buffers and an inverter. The two outputs are connected together to form a single output line. (Note that this type of connection cannot be made with gates that do not have three-state outputs.) When the select input is 0, the upper buffer is enabled by its control input and the lower buffer is disabled. Output Y is then equal to input A. When the select input is 1, the lower buffer is enabled and Y is equal to B.

The construction of a four-to-one-line multiplexer is shown in Fig. 4.30(b). The outputs of 4 three-state buffers are connected together to form a single output line. The control inputs to the buffers determine which one of the four normal inputs I_0 through



(a) 2-to-1-line mux (b) 4-to-1-line mux

FIGURE 4.30
Multiplexers with three-state gates

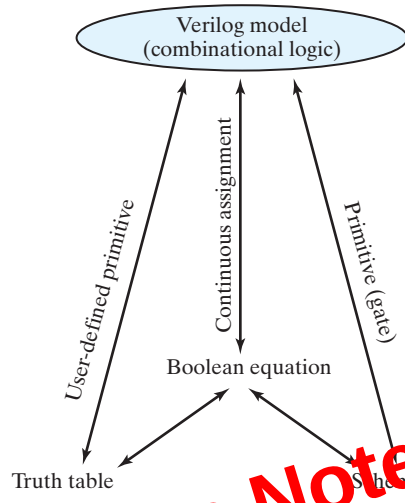


FIGURE 4.31
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

includes 12 basic gates as predefined primitives. Four of these primitive gates are of the three-state type. The other eight are the same as the ones listed in Section 2.8. They are all declared with the lowercase keywords **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, and **buf**. Primitives such as **and** are *n*-input primitives. They can have any number of scalar inputs (e.g., a three-input **and** primitive). The **buf** and **not** primitives are *n*-output primitives. A single input can drive multiple output lines distinguished by their identifiers.

The Verilog language includes a functional description of each type of gate, too. The logic of each gate is based on a four-valued system. When the gates are simulated, the simulator assigns one value to the output of each gate at any instant. In addition to the two logic values of 0 and 1, there are two other values: *unknown* and *high impedance*. An unknown value is denoted by **x** and a high impedance by **z**. An unknown value is assigned during simulation when the logic value of a signal is ambiguous—for instance, if it cannot be determined whether its value is 0 or 1 (e.g., a flip-flop without a reset condition). A high-impedance condition occurs at the output of three-state gates that are not enabled or if a wire is inadvertently left unconnected. The four-valued logic truth tables for the **and**, **or**, **xor**, and **not** primitives are shown in Table 4.9. The truth table for the other four gates is the same, except that the outputs are complemented. Note that for the **and** gate, the output is 1 only when both inputs are 1 and the output is 0 if any input is 0. Otherwise, if one input is **x** or **z**, the output is **x**. The output of the **or** gate is 0 if both inputs are 0, is 1 if any input is 1, and is **x** otherwise.

When a primitive gate is listed in a module, we say that it is *instantiated* in the module. In general, component instantiations are statements that reference lower level components in the design, essentially creating unique copies (or *instances*) of those components in the higher level module. Thus, a module that uses a gate in its description is said to

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	**x**	**z**
0	0	0	0	0
1	0	1	**x**	**x**
x	0	**x**	**x**	**x**
z	0	**x**	**x**	**x**

or	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	1	1	1
x	**x**	1	**x**	**x**
z	**x**	1	**x**	**x**

xor	0	1	**x**	**z**
0	0	1	**x**	**x**
1	1	0	**x**	**x**
x	**x**	**x**	**x**	**x**
z	**x**	**x**	**x**	**x**

not	input	output
	0	1
	1	0
	x	**x**
	z	**x**

instantiate the gate. This is instantiation as the HDL counterpart of placing and connecting parts on a circuit board.

We now present two examples of gate-level modeling. Both examples use identifiers having multiple bit widths, called *vectors*. The syntax specifying a vector includes within square brackets two numbers separated with a colon. The following Verilog statements specify two vectors:

```
output [0: 3] D;
wire   [7: 0] SUM;
```

The first statement declares an output vector *D* with four bits, 0 through 3. The second declares a wire vector *SUM* with eight bits numbered 7 through 0. (*Note*: The first (left-most) number (array index) listed is always the most significant bit of the vector.) The individual bits are specified within square brackets, so *D[2]* specifies bit 2 of *D*. It is also possible to address parts (contiguous bits) of vectors. For example, *SUM[2: 0]* specifies the three least significant bits of vector *SUM*.

HDL Example 4.1 shows the gate-level description of a two-to-four-line decoder. (See Fig. 4.19.) This decoder has two data inputs *A* and *B* and an enable input *E*. The four outputs are specified with the vector *D*. The **wire** declaration is for internal connections. Three **not** gates produce the complement of the inputs, and four **nand** gates provide the outputs for *D*. Remember that *the output is always listed first in the port list of a primitive*, followed by the inputs. This example describes the decoder of Fig. 4.19 and follows the procedures established in Section 3.10. Note that the keywords **not** and **nand** are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

HDL Example 4.1 (Two-to-Four-Line Decoder)

```
// Gate-level description of two-to-four-line decoder
// Refer to Fig. 4.19 with symbol E replaced by enable, for clarity.

module decoder_2x4_gates (D, A, B, enable);
  output      [0: 3]          D;
  input                       A, B;
  input                       enable;
  wire                        A_not,B_not, enable_not;

  not
    G1  (A_not, A),
    G2  (B_not, B),
    G3  (enable_not, enable);
  nand
    G4  (D[0], A_not, B_not, enable_not),
    G5  (D[1], A_not, B, enable_not),
    G6  (D[2], A, B_not, enable_not),
    G7  (D[3], A, B, enable_not);
endmodule
```


Two or more modules can be combined to build a hierarchical description of a design. There are two basic types of design methodologies: top down and bottom up. In a *top-down* design, the top-level block is defined and then the subblocks necessary to build the top-level block are identified. In a *bottom-up* design, the building blocks are first identified and then combined to build the top-level block. Take, for example, the binary adder of Fig. 4.9. It can be considered as a top-block component built with four full-adder blocks, while each full adder is built with two half-adder blocks. In a top-down design, the four-bit adder is defined first, and then the two adders are described. In a bottom-up design, the half adder is defined, then each full adder is constructed, and then the four-bit adder is built from the full adders.

A bottom-up hierarchical description of a four-bit adder is shown in HDL Example 4.2. The half adder is defined by instantiating primitive gates. The next module describes the full adder by instantiating and connecting two half adders. The third module describes the four-bit adder by instantiating and connecting four full adders. Note that the first character of an identifier cannot be a number, but can be an underscore, so the module name *_4bitadder* is valid. An alternative name that is meaningful, but does not require a leading underscore, is *adder_4_bit*. The instantiation is done by using the name of the module that is instantiated together with a new (or the same) set of port names. For example, the half adder *HA1* inside the full adder module is instantiated with ports *S1*, *C1*, *x,* and *y*. This produces a half adder with outputs *S1* and *C1* and inputs *x* and *y*.

Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value. Thus, $2'b01$ specifies a two-bit binary number whose value is 01. Numbers are stored as a bit pattern in memory, but they can be referenced in decimal, octal, or hexadecimal formats with the letters ***d'o'*** and ***h'*** respectively. For example, 4′HA = 4′d10 = 4′b1010 and have the same internal representation in a simulator. If the base of the number is not specified, its interpretation defaults to decimal. If the size of the number is not specified, the system assumes that the size of the number is at least 32 bits; if a host simulator has a larger word length—say, 64 bits—the language will use that value to store unsized numbers. The integer data type (keyword **integer**) is stored in a 32-bit representation. The underscore (_) may be inserted in a number to improve readability of the code (e.g., 16′b0101_1110_0101_0011). It has no other effect.

The **case** construct has two important variations: **casex** and **casez**. The first will treat as don't-cares any bits of the **case** expression or the **case** item that have logic value **x** or **z**. The **casez** construct treats as don't-cares only the logic value **z** for the purpose of detecting a match between the **case** expression and a **case** item.

The list of case items need not be complete. If the list of **case** items does not include all possible bit patterns of the **case** expression, no match can be detected. Unlisted **case** items, i.e., bit patterns that are not explicitly decoded, can be treated by using the **default** keyword as the last item in the list of case items. The associated statement will execute when no other match is found. This feature is useful, for example, when there are more possible state codes in a sequential machine than are actually used. Having a **default** case item lets the designer map all of the unused states to a desired next state without having to elaborate each individual state, rather than allowing the synthesis tool to arbitrarily assign the next state.

The examples of behavioral descriptions of combinational circuits shown here are simple ones. Behavioral modeling and procedural assignment statements require knowledge of sequential circuits and are covered in more detail in Section 5.6.

Writing a Simple Test Bench

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation. Test benches can be quite complex and lengthy and may take longer to develop than the design that is tested. The results of a test are only as good as the test bench that is used to test a circuit. Care must be taken to write stimuli that will test a circuit thoroughly, exercising all of the operating features that are specified. However, the test benches considered here are relatively simple, since the circuits we want to test implement only combinational logic. The examples are presented to demonstrate some basic features of HDL stimulus modules. Chapter 8 considers test benches in greater depth.

In addition to employing the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit being tested. We use the term "**always** statement" loosely. Actually, **always** is a Verilog language construct specifying *how* the associated statement is to execute (subject to the event control expression). The **always** statement

PROBLEMS

(Answers to problems marked with * appear at the end of the text. Where appropriate, a logic design and its related HDL modeling problem are cross-referenced.)

4.1 Consider the combinational circuit shown in Fig. P4.1. (HDL—see Problem 4.49.)

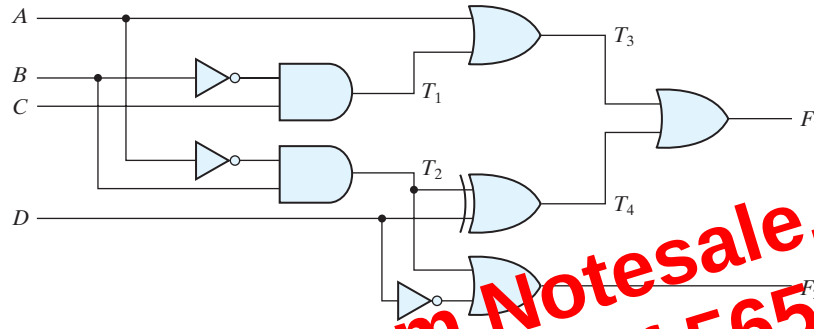


FIGURE P4.1

(a)* Derive the Boolean expressions for T_1 through T_4. Evaluate the outputs F_1 and F_2 as a function of the four inputs.

(b) List the truth table with the binary combinations of the four input variables. Then list the binary values for T_1 through T_4 and outputs F_1 and F_2 in the table.

(c) Plot the output Boolean functions obtained in part (b) on maps and show that the simplified Boolean expressions are equivalent to the ones obtained in part (a).

4.2* Obtain the simplified Boolean expressions for output F and G in terms of the input variables in the circuit of Fig. P4.2.

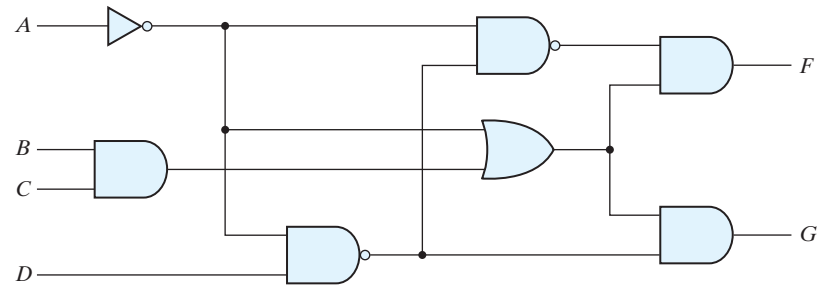


FIGURE P4.2

4.3 For the circuit shown in Fig. 4.26 (Section 4.11),

(a) Write the Boolean functions for the four outputs in terms of the input variables.

(b)* If the circuit is described in a truth table, how many rows and columns would there be in the table?

4.4 Design a combinational circuit with three inputs and one output.

(a)* The output is 1 when the binary value of the inputs is less than 3. The output is 0 otherwise.

(b) The output is 1 when the binary value of the inputs is an even number.

5. Gajski, D. D. 1997. *Principles of Digital Design*. Upper Saddle River, NJ: Prentice Hall.
6. Hayes, J. P. 1993. *Introduction to Digital Logic Design*. Reading, MA: Addison-Wesley.
7. Katz, R. H. 2005. *Contemporary Logic Design*. Upper Saddle River, NJ: Pearson Prentice Hall.
8. Mano, M. M. and C. R. Kime. 2007. *Logic and Computer Design Fundamentals*, 4th ed. Upper Saddle River, NJ: Prentice Hall.
9. Nelson, V. P., H. T. Nagle, J. D. Irwin, and B. D. Carroll. 1995. *Digital Logic Circuit Analysis and Design*. Englewood Cliffs, NJ: Prentice Hall.
10. Palnitkar, S. 1996. *Verilog HDL: A Guide to Digital Design and Synthesis*. Mountain View, CA: SunSoft Press (a Prentice Hall title).
11. Roth, C. H. 2009. *Fundamentals of Logic Design*, 6th ed. St. Paul, MN: West.
12. Thomas, D. E. and P. R. Moorby. 2002. *The Verilog Hardware Description Language*, 5th ed. Boston: Kluwer Academic Publishers.
13. Wakerly, J. F. 2005. *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice Hall.

WEB SEARCH TOPICS

Boolean equation
Combinational logic
Truth table
Exclusive–OR
Comparator
Multiplexer
Decoder
Priority encoder
Three-state inverter
Three-state buffer

0 is detected in a stream of 1s. It consists of two *D* flip-flops *A* and *B,* an input *x* and an output *y*. Since the *D* input of a flip-flop determines the value of the next state (i.e., the state reached after the clock transition), it is possible to write a set of state equations for the circuit:

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

A state equation is an algebraic expression that specifies the condition for a flip-flop state transition. The left side of the equation, with $(t + 1)$, denotes the next state of the flip-flop one clock edge later. The right side of the equation is a Boolean expression that specifies the present state and input conditions that make the next state equal to 1. Since all the variables in the Boolean expressions are a function of the present state, we can omit the designation (t) after each variable for convenience and can express the state equations in the more compact form

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

The Boolean expressions for the state equations can be derived directly from the gates that form the combinational circuit part of the sequential circuit, since the *D* values of the combinational circuit determine the next state. Similarly, the present-state value of the output can be expressed algebraically as

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

By removing the symbol *(t)* for the present state, we obtain the output Boolean equation:

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

State Table

The time sequence of inputs, outputs, and flip-flop states can be enumerated in a *state table* (sometimes called a *transition table*). The state table for the circuit of Fig. 5.15 is shown in Table 5.2. The table consists of four sections labeled *present state*, *input*, *next state*, and *output*. The present-state section shows the states of flip-flops *A* and *B* at any given time *t*. The input section gives a value of *x* for each possible present state. The next-state section shows the states of the flip-flops one clock cycle later, at time $t + 1$. The output section gives the value of *y* at time *t* for each present state and input condition.

The derivation of a state table requires listing all possible binary combinations of present states and inputs. In this case, we have eight binary combinations from 000 to 111. The next-state values are then determined from the logic diagram or from the state equations. The next state of flip-flop *A* must satisfy the state equation

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

$K = 0$, the next state is 1. When $J = 0$ and $K = 1$, the next state is 0. When $J = K = 0$, there is no change of state and the next-state value is the same as that of the present state. When $J = K = 1$, the next-state bit is the complement of the present-state bit. Examples of the last two cases occur in the table when the present state AB is 10 and input x is 0. JA and KA are both equal to 0 and the present state of A is 1. Therefore, the next state of A remains the same and is equal to 1. In the same row of the table, JB and KB are both equal to 1. Since the present state of B is 0, the next state of B is complemented and changes to 1.

The next-state values can also be obtained by evaluating the state equations from the characteristic equation. This is done by using the following procedure:

1. Determine the flip-flop input equations in terms of the present state and input variables.
2. Substitute the input equations into the flip-flop characteristic equation to obtain the state equations.
3. Use the corresponding state equations to determine the next-state values in the state table.

The input equations for the two JK flip-flops of Fig. 5.18 were listed a couple of paragraphs ago. The characteristic equations for the flip-flops are obtained by substituting A or B for the name of the flip-flop instead of Q:

$$A(t + 1) = JA' + K'A$$
$$B(t + 1) = JB' + K'B$$

Substituting the values of J_A and K_A from the input equations, we obtain the state equation for A:

$$A(t + 1) = BA' + (Bx')'A = A'B + AB' + Ax$$

The state equation provides the bit values for the column headed "Next State" for A in the state table. Similarly, the state equation for flip-flop B can be derived from the characteristic equation by substituting the values of J_B and K_B:

$$B(t + 1) = x'B' + (A \oplus x)'B = B'x' + ABx + A'Bx'$$

The state equation provides the bit values for the column headed "Next State" for B in the state table. Note that the columns in Table 5.4 headed "Flip-Flop Inputs" are not needed when state equations are used.

The state diagram of the sequential circuit is shown in Fig. 5.19. Note that since the circuit has no outputs, the directed lines out of the circles are marked with one binary number only, to designate the value of input x.

Analysis with *T* Flip-Flops

The analysis of a sequential circuit with T flip-flops follows the same procedure outlined for JK flip-flops. The next-state values in the state table can be obtained by using either

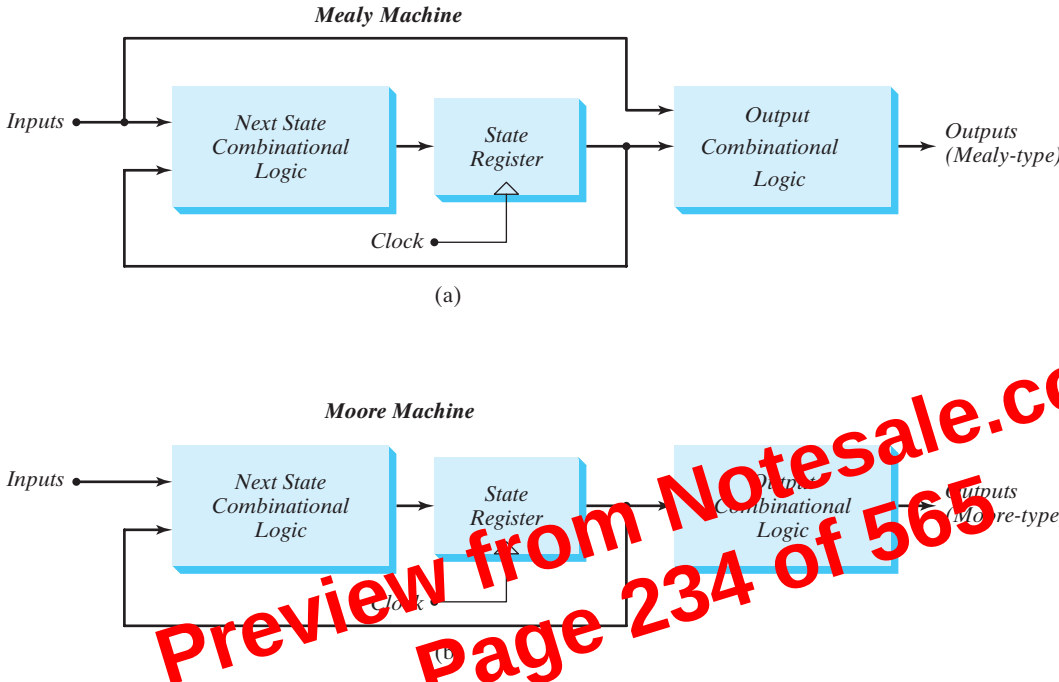


FIGURE 5.21
Block diagrams of Mealy and Moore state machines

is generated. In the Mealy model, the output is a function of both the present state and the input. In the Moore model, the output is a function of only the present state. A circuit may have both types of outputs. The two models of a sequential circuit are commonly referred to as a finite state machine, abbreviated FSM. The Mealy model of a sequential circuit is referred to as a Mealy FSM or Mealy machine. The Moore model is referred to as a Moore FSM or Moore machine.

The circuit presented previously in Fig. 5.15 is an example of a Mealy machine. Output y is a function of both input x and the present state of A and B. The corresponding state diagram in Fig. 5.16 shows both the input and output values, separated by a slash along the directed lines between the states.

An example of a Moore model is given in Fig. 5.18. Here, the output is a function of the present state only. The corresponding state diagram in Fig. 5.19 has only inputs marked along the directed lines. The outputs are the flip-flop states marked inside the circles. Another example of a Moore model is the sequential circuit of Fig. 5.20. The output depends only on flip-flop values, and that makes it a function of the present state only. The input value in the state diagram is labeled along the directed line, but the output value is indicated inside the circle together with the present state.

In a Moore model, the outputs of the sequential circuit are synchronized with the clock, because they depend only on flip-flop outputs that are synchronized with the clock. In a Mealy model, the outputs may change if the inputs change during the clock

```
 #10 t_x_in = 1;
 #30 t_x_in = 0;
 #40 t_x_in = 1;
 #50 t_x_in = 0;
 #52 t_x_in = 1;
 #54 t_x_in = 0;
 #70 t_x_in = 1;
 #80 t_x_in = 1;
 #70 t_x_in = 0;
 #90 t_x_in = 1;
 #100 t_x_in = 0;
 #120 t_x_in = 1;
 #160 t_x_in = 0;
 #170 t_x_in = 1;
 join
endmodule
```


The circuit I HDL Example 5.5 detects a 0 following a sequence of 1s in a serial bit stream. Its Verilog model uses three **always** blocks that execute concurrently and interact through common variables. The first **always** statement resets the circuit to the initial state *S0* = 00 and specifies the synchronous clocked operation. The statement *state* <= *next_state* is synchronized to a positive-edge transition of the clock. This means that any change in the value of *next_state* in the second **always** block can affect the value of *state* only as a result of a **posedge** event of *clock*. The second **always** block determines the value of the next state transition as a function of the present state and input. The value assigned to *state* by the nonblocking assignment is the value of *next_state* immediately before the rising edge of *clock*. Notice how the multiway branch condition implements the state transitions specified by the annotated edges in the state diagram of Fig. 5.16. The third **always** block specifies the output as a function of the present state and the input. Although this block is listed as a separate behavior for clarity, it could be combined with the second block. Note that the value of output *y_out* may change if the value of input *x_in* changes while the circuit is in any given state.

So let's summarize how the model describes the behavior of the machine: At every rising edge of *clock*, if *reset* is not asserted, the state of the machine is updated by the first **always** block; when *state* is updated by the first **always** block, the change in *state* is detected by the sensitivity list mechanism of the second **always** block; then the second **always** block updates the value of *next_state* (it will be used by the first **always** block at the next tick of the clock); the third **always** block also detects the change in *state* and updates the value of the output. In addition, the second and third **always** blocks detect changes in *x_in* and update *next_state* and *y_out* accordingly. The test bench provided with *Mealy_Zero_Detector* provides some waveforms to stimulate the model, producing the results shown in Fig. 5.22. Notice how *t_y_out* responds to changes in both the state and the input, and has a glitch (a transient logic value). We display both to *state*[1:0] and *next_state*[1:0] to illustrate how changes in *t_x_in* influence the value of next_state and *t_y_out*. The Mealy glitch in *t_y_out* is due to the (intentional) dynamic behavior of *t_x_in*. The input, *t_x_in,* settles

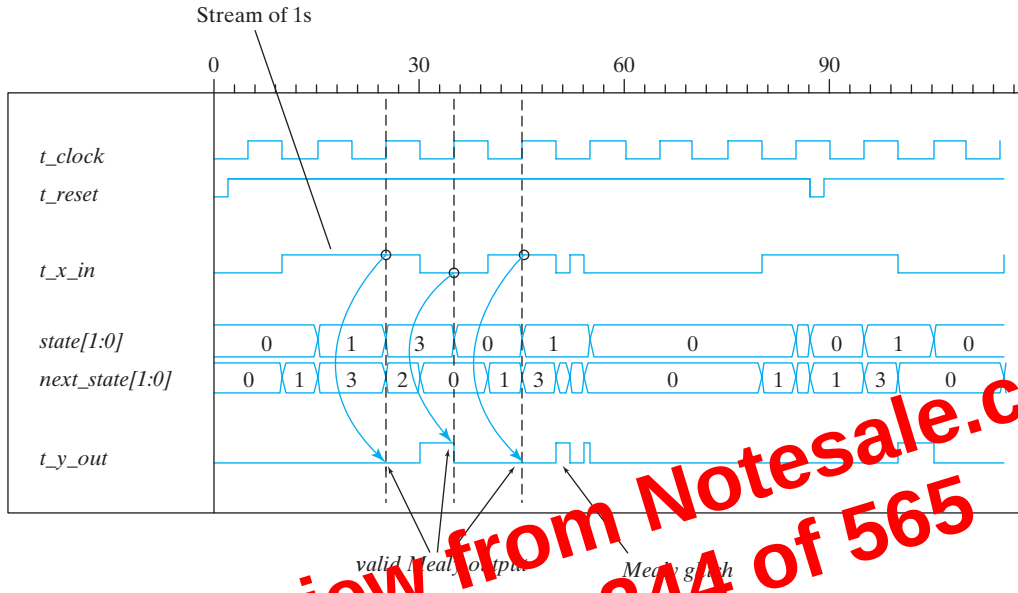


FIGURE 5.23
Simulation output of *Mealy_Zero_Detector*

to a value of 0 immediately before the clock, and at the clock, the state makes a transition from 0 to 1, which is consistent with Fig. 5.16. The output is 1 in state *S*1 immediately before the clock, and changes to 0 as the state enters *S*0.

The description of waveforms in the test bench uses the **fork** . . . **join** construct. Statements with the **fork** . . . **join** block execute in parallel, so the time delays are relative to a common reference of $t = 0$, the time at which the block begins execution.[2] It is usually more convenient to use the **fork** . . . **join** block instead of the **begin** . . . **end** block in describing waveforms. Notice that the waveform of reset is triggered "on the fly" to demonstrate that the machine recovers from an unexpected (asynchronous) reset condition during any state.

How does our Verilog model *Mealy_Zero_Detector* correspond to hardware? The first **always** block corresponds to a *D* flip-flop implementation of the state register in Fig. 5.21; the second **always** block is the combinational logic block describing the next state; the third **always** block describes the output combinational logic of the zero-detecting Mealy machine. The register operation of the state transition uses the nonblocking assignment operator (< =) because the (edge-sensitive) flip-flops of a sequential machine are updated concurrently by a common clock. The second and third **always** blocks describe combinational logic, which is level sensitive, so they use the blocking (=) assignment operator.

[2] A **fork** . . . **join** block completes execution when the last executing statement within it completes its execution.

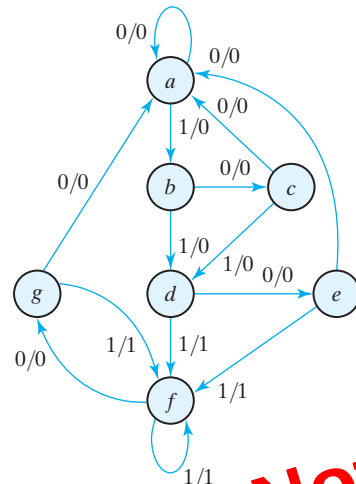


FIGURE 5.25
State diagram

by letter symbols instead of their binary values. This is in contrast to a binary counter, where the binary value sequence of the states themselves is taken as the outputs.

There are an infinite number of input sequences that may be applied to the circuit; each results in a unique output sequence. As an example, consider the input sequence 01010110100 starting from the initial state *a*. Each input of 0 or 1 produces an output of 0 or 1 and causes the circuit to go to the next state. From the state diagram, we obtain the output and state sequence for the given input sequence as follows: With the circuit in initial state *a*, an input of 0 produces an output of 0 and the circuit remains in state *a*. With present state *a* and an input of 1, the output is 0 and the next state is *b*. With present state *b* and an input of 0, the output is 0 and the next state is *c*. Continuing this process, we find the complete sequence to be as follows:

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

In each column, we have the present state, input value, and output value. The next state is written on top of the next column. It is important to realize that in this circuit the states themselves are of secondary importance, because we are interested only in output sequences caused by input sequences.

Now let us assume that we have found a sequential circuit whose state diagram has fewer than seven states, and suppose we wish to compare this circuit with the circuit whose state diagram is given by Fig. 5.25. If identical input sequences are applied to the two circuits and identical outputs occur for all input sequences, then the two circuits are said to be equivalent (as far as the input–output is concerned) and one may be replaced by the other. The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input–output relationships.

Table 5.10
Reduced State Table with Binary Assignment 1

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

saved by using simpler decoding logic. This trade-off is not guaranteed, so it must be evaluated for a given design.

Table 5.10 is the reduced state table with binary assignment 1 substituted for the letter symbols of the states. A different assignment will result in a state table with different binary values for the states. The binary form of the state table is used to derive the next-state and output-forming combinational logic part of the sequential circuit. The complexity of the combinational circuit depends on the binary state assignment chosen.

Sometimes the name *transition table* is used for a state table with a binary assignment. This convention distinguishes it from a state table with symbolic names for the states. In this book, we use the same name for both types of state tables.

5.8 DESIGN PROCEDURE

Design procedures or methodologies specify hardware that will implement a desired behavior. The design effort for small circuits may be manual, but industry relies on automated synthesis tools for designing massive integrated circuits. The sequential building block used by synthesis tools is the *D* flip-flop. Together with additional logic, it can implement the behavior of *JK* and *T* flip-flops. In fact, designers generally do not concern themselves with the type of flip-flop; rather, their focus is on correctly describing the sequential functionality that is to be implemented by the synthesis tool. Here we will illustrate manual methods using *D, JK,* and *T* flip-flops.

The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which the logic diagram can be obtained. In contrast to a combinational circuit, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalent representation, such as a state diagram.[3]

A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding a combinational

[3] We will examine later another important representation of a machine's behavior—the algorithmic state machine (ASM) chart.

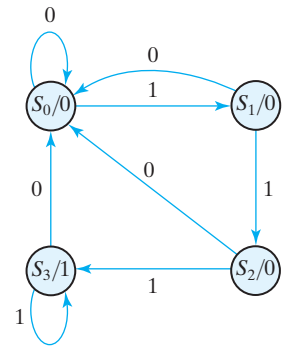


FIGURE 5.27
State diagram for sequence detector

Synthesis Using *D* Flip-Flops

Once the state diagram has been derived, the rest of the design follows a straightforward synthesis procedure. In fact, we can design the circuit by using an HDL description of the state diagram and the proper HDL synthesis tools to obtain a synthesized netlist. (The HDL description of the state diagram will be similar to HDL Example 5.6 in Section 5.6.) To design the circuit by hand, we need to assign binary codes to the states and list the state table. This is done in Table 5.11. The table is derived from the state diagram of Fig. 5.27 with a sequential binary assignment. We choose two *D* flip-flops to represent the four states, and we label their outputs *A* and *B*. There is one input *x* and one output *y*. The characteristic equation of the *D* flip-flop is $Q(t + 1) = D_Q$, which means that the next-state values in the state table specify the *D* input condition for the flip-flop. The flip-flop input equations

Table 5.11
State Table for Sequence Detector

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

and 5.26. Write a test bench to compare the state sequences and input–output behaviors of the two machines.

5.38 Write and verify an HDL behavioral description of the machine described in Problem 5.16.

5.39 Write and verify a behavioral description of the machine specified in Problem 5.17.

5.40 Write and verify a behavioral description of the machine specified in Problem 5.18.

5.41 Write and verify a behavioral description of the machine specified in Problem 5.19. (*Hint*: See the discussion of the **default** case item preceding HDL Example 4.8 in Chapter 4.)

5.42 Write and verify an HDL structural description of the circuit shown in Fig. 5.29.

5.43 Write and verify an HDL behavioral description of the three-bit binary counter in Fig. 5.34.

5.44 Write and verify a Verilog model of a *D* flip-flop having asynchronous reset.

5.45 Write and verify an HDL behavioral description of the sequence detector described in Fig. 5.28.

5.46 A synchronous finite state machine has an input *x_in* and an output *y_out*. When *x_in* changes from 0 to 1, the output *y_out* is to assert for three cycles, regardless of the value of *x_in*, and then de-assert for two cycles before the machine will respond to another assertion of *x_in*. The machine is to have active-low synchronous reset.

(a) Draw the state diagram of the machine.

(b) Write and verify a Verilog model of the machine.

5.47 Write a Verilog model of a synchronous finite state machine whose output is the sequence 0, 2, 4, 6, 8, 10, 12, 14, 0 . . . The machine is controlled by a single input, *Run*, so that counting occurs while *Run* is asserted, suspends while *Run* is de-asserted, and resumes the count when *Run* is re-asserted. Clearly state any assumptions that you make.

5.48 Write a Verilog model of the Mealy FSM described by the state diagram in Fig. P5.48. Develop a test bench and demonstrate that the machine state transitions and output correspond to its state diagram.

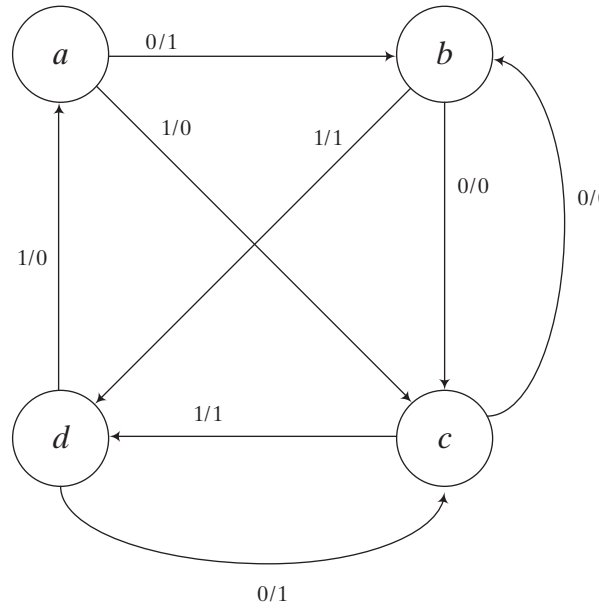


FIGURE P5.48

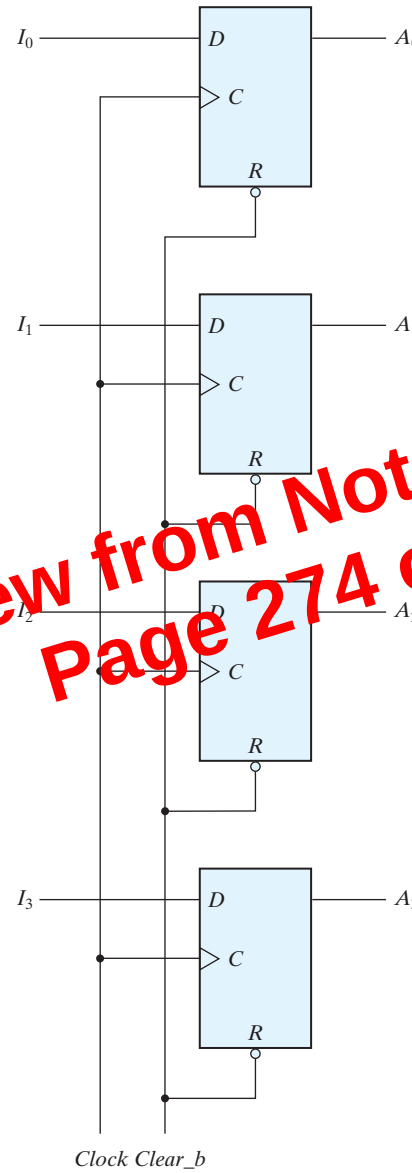


FIGURE 6.1
Four-bit register

outputs can be sampled at any time to obtain the binary information stored in the register. The input *Clear_b* goes to the active-low *R* (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously. The *Clear_b* input is useful for clearing the register to all 0's prior to its clocked operation. The *R* inputs must be maintained

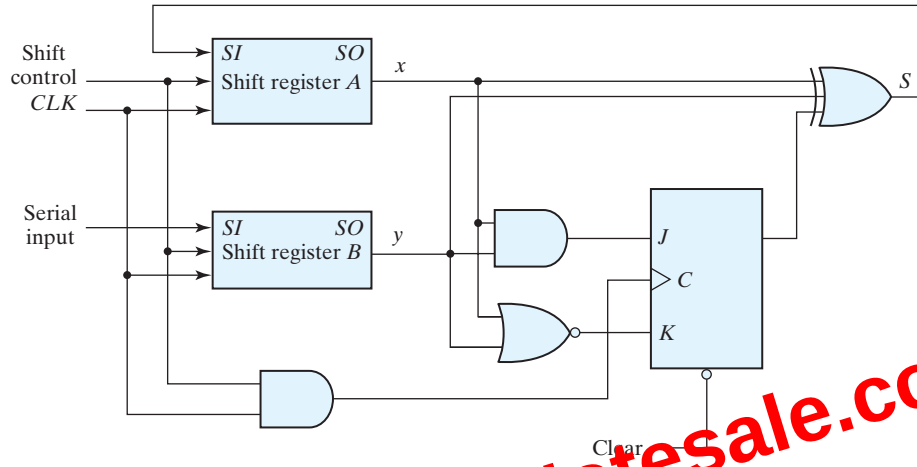
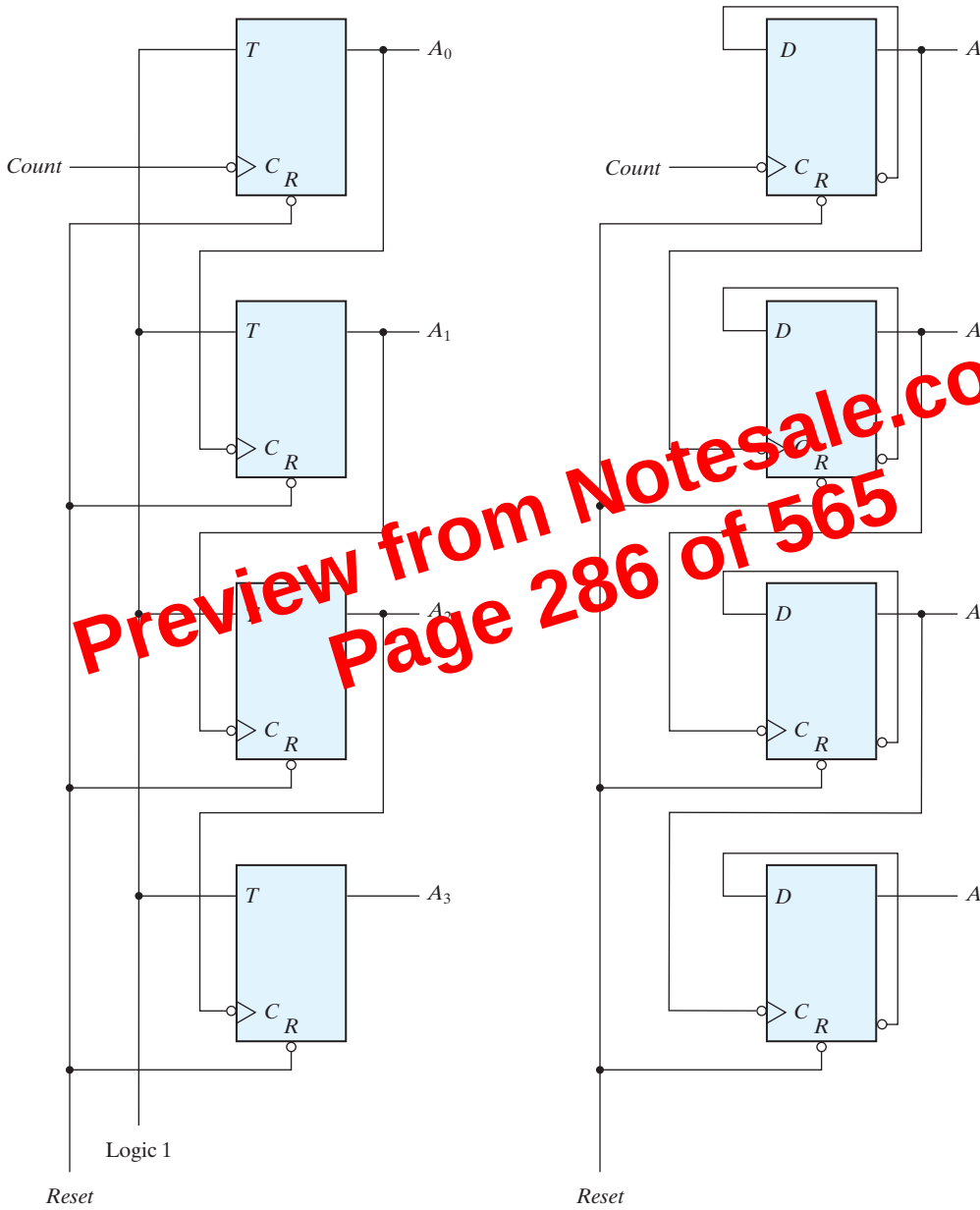


FIGURE 6.6
Second form of serial adder

3. A *shift-right* control to enable the shift-right operation and the *serial input* and *output* lines associated with the shift right.
4. A *shift-left* control to enable the shift-left operation and the *serial input* and *output* lines associated with the shift left.
5. A *parallel-load* control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. n parallel output lines.
7. A control state that leaves the information in the register unchanged in response to the clock. Other shift registers may have only some of the preceding functions, with at least one shift operation.

A register capable of shifting in one direction only is a *unidirectional* shift register. One that can shift in both directions is a *bidirectional* shift register. If the register has both shifts and parallel-load capabilities, it is referred to as a *universal shift register.*

The block diagram symbol and the circuit diagram of a four-bit universal shift register that has all the capabilities just listed are shown in Fig. 6.7. The circuit consists of four D flip-flops and four multiplexers. The four multiplexers have two common selection inputs s_1 and s_0. Input 0 in each multiplexer is selected when $s_1s_0 = 00$, input 1 is selected when $s_1s_0 = 01$, and similarly for the other two inputs. The selection inputs control the mode of operation of the register according to the function entries in Table 6.3. When $s_1s_0 = 00$, the present value of the register is applied to the D inputs of the flip-flops. This condition forms a path from the output of each flip-flop into the input of the same flip-flop, so that the output recirculates to the input in this mode of operation. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs.



(a) With *T* flip-flops

(b) With *D* flip-flops

FIGURE 6.8
Four-bit binary ripple counter

count goes from 0011 to 0010, then to 0000, and finally to 0100. The flip-flops change one at a time in succession, and the signal propagates through the counter in a ripple fashion from one stage to the next.

A binary counter with a reverse count is called a *binary countdown counter*. In a countdown counter, the binary count is decremented by 1 with every input count pulse. The count of a four-bit countdown counter starts from binary 15 and continues to binary counts 14, 13, 12, . . . , 0 and then back to 15. A list of the count sequence of a binary countdown counter shows that the least significant bit is complemented with every count pulse. Any other bit in the sequence is complemented if its previous least significant bit goes from 0 to 1. Therefore, the diagram of a binary countdown counter looks the same as the binary ripple counter in Fig. 6.8, provided that all flip-flops trigger on the positive edge of the clock. (The bubble in the *C* inputs must be absent.) If negative-edge-triggered flip-flops are used, then the *C* input of each flip-flop must be connected to the complemented output of the previous flip-flop. Then, when the true output goes from 0 to 1, the complement will go from 1 to 0 and complement the next flip-flop as required.

BCD Ripple Counter

A decimal counter follows a sequence of 10 states and returns to 0 after the count of 9. Such a counter must have at least four flip-flops to represent each decimal digit, since a decimal digit is represented by a binary code with at least four bits. The sequence of states in a decimal counter is dictated by the binary code used to represent a decimal digit. If BCD is used, the sequence of states is as shown in the state diagram of Fig. 6.9. A decimal counter is similar to a binary counter, except that the state after 1001 (the code for decimal digit 9) is 0000 (the code for decimal digit 0).

The logic diagram of a BCD ripple counter using *JK* flip-flops is shown in Fig. 6.10. The four outputs are designated by the letter symbol *Q*, with a numeric subscript equal to the binary weight of the corresponding bit in the BCD code. Note that the output of Q_1 is applied to the *C* inputs of both Q_2 and Q_8 and the output of Q_2 is applied to the *C* input of Q_4. The *J* and *K* inputs are connected either to a permanent 1 signal or to outputs of other flip-flops.

A ripple counter is an asynchronous sequential circuit. Signals that affect the flip-flop transition depend on the way they change from 1 to 0. The operation of the counter can

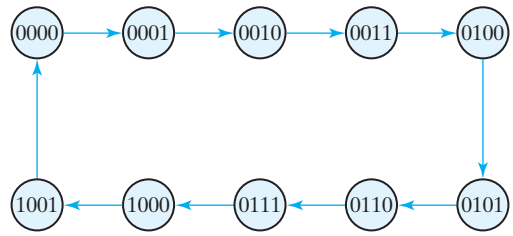


FIGURE 6.9
State diagram of a decimal BCD counter

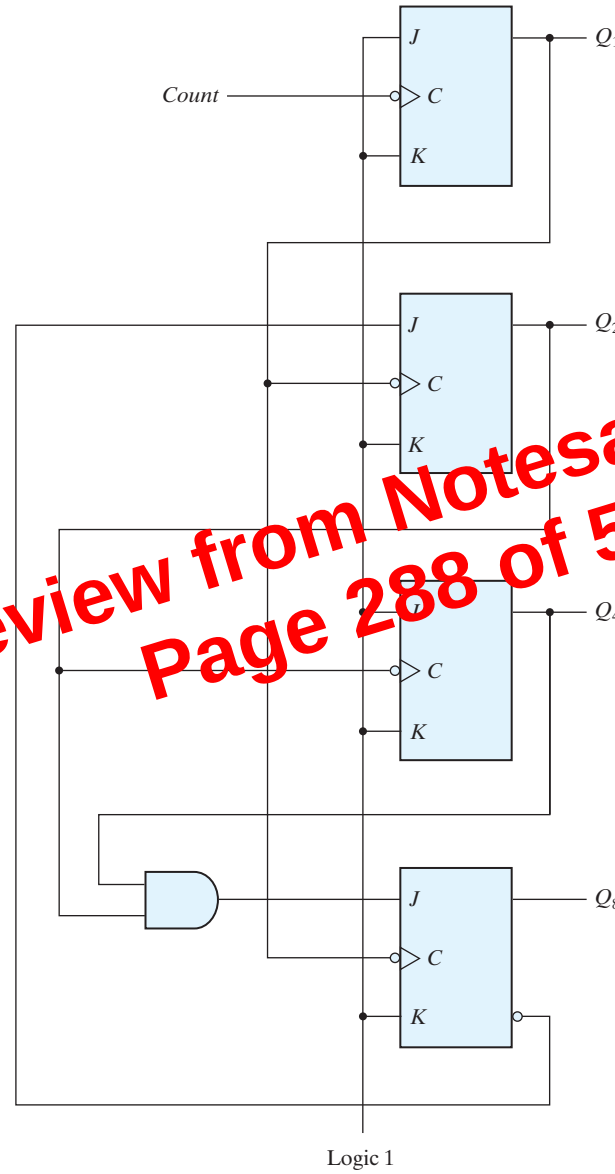


FIGURE 6.10
BCD ripple counter

be explained by a list of conditions for flip-flop transitions. These conditions are derived from the logic diagram and from knowledge of how a *JK* flip-flop operates. Remember that when the *C* input goes from 1 to 0, the flip-flop is set if $J = 1$, is cleared if $K = 1$, is complemented if $J = K = 1$, and is left unchanged if $J = K = 0$.

in the same count. When the up and down inputs are both 1, the circuit counts up. This set of conditions ensures that only one operation is performed at any given time. Note that the up input has priority over the down input.

BCD Counter

A BCD counter counts in binary-coded decimal from 0000 to 1001 and back to 0000. Because of the return to 0 after a count of 9, a BCD counter does not have a regular pattern, unlike a straight binary count. To derive the circuit of a BCD synchronous counter, it is necessary to go through a sequential circuit design procedure.

The state table of a BCD counter is listed in Table 6.5. The input conditions for the T flip-flops are obtained from the present- and next-state conditions. Also shown in the table is an output y, which is equal to 1 when the present state is 1001. In this way, y can enable the count of the next-higher significant decade while the same pulse switches the present decade from 1001 to 0000.

The flip-flop input equations can be simplified by means of maps. The unused states for minterms 10 to 15 are taken as don't-care terms. The simplified functions are

$$T_{Q1} = 1$$
$$T_{Q2} = Q'_8 Q_1$$
$$T_{Q4} = Q_2 Q_1$$
$$T_{Q8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1$$

The circuit can easily be drawn with four T flip-flops, five AND gates, and one OR gate. Synchronous BCD counters can be cascaded to form a counter for decimal numbers of any length. The cascading is done as in Fig. 6.11, except that output y must be connected to the count input of the next-higher significant decade.

Table 6.5
State Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

```
always @ (posedge CLK, negedge Clear_b)    // V2001, 2005
  if (Clear_b == 0) A_par <= 4'b0000;
  else
   case ({s1, s0})
     2'b00: A_par <= A_par;                // No change
     2'b01: A_par <= {MSB_in, A_par[3: 1]}; // Shift right
     2'b10: A_par <= {A_par[2: 0], LSB_in}; // Shift left
     2'b11: A_par <= I_par;                // Parallel load of input
   endcase
endmodule
```

Variables of type **reg** retain their value until they are assigned a new value by an assignment statement. Consider the following alternative **case** statement for the shift register model:

```
   case ({s1, s0})
     // 2'b00: A_par <= A_par;             // No change
     2'b01: A_par <= {MSB_in, A_par [3: 1]}; // Shift right
     2'b10: A_par <= {A_par [2: 0], LSB_in}; // Shift left
     2'b11: A_par <= I_par;                // Parallel load of input
   endcase
```


Without the case item 2′b00, the **case** statement would not find a match between $\{s1, s0\}$ and the case items, so register *A_par* would be left unchanged.

A structural model of the universal shift register can be described by referring to the logic diagram of Fig. 6.7(b). The diagram shows that the register has four multiplexers and four *D* flip-flops. A mux and flip-flop together are modeled as a stage of the shift register. The stage is a structural model, too, with an instantiation and interconnection of a module for a mux and another for a *D* flip-flop. For simplicity, the lowest-level modules of the structure are behavioral models of the multiplexer and flip-flop. Attention must be paid to the details of connecting the stages correctly. The structural description of the register is shown in HDL Example 6.2. The top-level module declares the inputs and outputs and then instantiates four copies of a stage of the register. The four instantiations specify the interconnections between the four stages and provide the detailed construction of the register as specified in the logic diagram. The behavioral description of the flip-flop uses a single edge-sensitive cyclic behavior (an **always** block). The assignment statements use the nonblocking assignment operator (<=) the model of the mux employs a single level-sensitive behavior, and the assignments use the blocking assignment operator (=).

HDL Example 6.2 (Universal Shift Register-Structural Model)

```
// Structural description of a 4-bit universal shift register (see Fig. 6.7)
module Shift_Register_4_str (                  // V2001, 2005
  output [3: 0] A_par,                         // Parallel output
  input [3: 0]   I_par,                        // Parallel input
```

PROBLEMS

(Answers to problems marked with * appear at the end of the book. Where appropriate, a logic design and its related HDL modeling problem are cross-referenced.)

Note: For each problem that requires writing and verifying a Verilog description, a test plan is to be written to identify which functional features are to be tested during the simulation and how they will be tested. For example, a reset on the fly could be tested by asserting the reset signal while the simulated machine is in a state other than the reset state. The test plan is to guide the development of a test bench that will implement the plan. Simulate the model using the test bench and verify that the behavior is correct. If synthesis tools and an ASIC cell library or a field programmable gate array (FPGA) tool suite are available, the Verilog descriptions developed for Problems 6.34–6.51 can be assigned as synthesis exercises. The gate-level circuit produced by the synthesis tools should be simulated and compared to the simulation results for the pre-synthesis model.

In some of the HDL problems, there may be a need to deal with the issue of unused states (see the discussion of the **default case** item preceding HDL Example 4.8 in Chapter 4).

6.1 Include a 2-input NAND gate in the register of Fig. 6.1 and connect the gate output to the *C* inputs of all the flip-flops. One input of the NAND gate receives the clock pulses from the clock generator and the other input of the NAND gate provides a parallel load control. Explain the operation of the modified register. Explain why this circuit might have operational problems.

6.2 Include a synchronous clear input to the register of Fig. 6.2. The modified register will have a parallel load capability and a synchronous clear capability. The register is cleared synchronously when the clock goes through a positive transition and the clear input is equal to 1. (HDL—see Problem 6.35(a), (b).)

6.3 What is the difference between serial and parallel transfer? Explain how to convert serial data to parallel and parallel data to serial. What type of register is needed?

6.4* The contents of a four-bit register is initially 0110. The register is shifted six times to the right with the serial input being 1011100. What is the content of the register after each shift?

6.5 The four-bit universal shift register shown in Fig. 6.7 is enclosed within one IC component package. (HDL—see Problem 6.52.)

(a) Draw a block diagram of the IC showing all inputs and outputs. Include two pins for the power supply.

(b) Draw a block diagram using two of these ICs to produce an eight-bit universal shift register.

6.6 Design a four-bit shift register with parallel load using *D* flip-flops. There are two control inputs: *shift* and *load*. When *shift* = 1, the content of the register is shifted by one position. New data are transferred into the register when *load* = 1 and *shift* = 0. If both control inputs are equal to 0, the content of the register does not change. (HDL—see Problem 6.35(c), (d).)

6.7 Draw the logic diagram of a four-bit register with four *D* flip-flops and four 4 × 1 multiplexers with mode selection inputs s_1 and s_0. The register operates according to the following function table. (HDL—see Problem 6.35(e), (f).)

6.17* Design a four-bit binary synchronous counter with *D* flip-flops.

6.18 What operation is performed in the up–down counter of Fig. 6.13 when both the up and down inputs are enabled? Modify the circuit so that when both inputs are equal to 1, the counter does not change state. (HDL—see Problem 6.35(l).)

6.19 The flip-flop input equations for a BCD counter using *T* flip-flops are given in Section 6.4. Obtain the input equations for a BCD counter that uses (a) *JK* flip-flops and (b)* *D* flip-flops. Compare the three designs to determine which one is the most efficient.

6.20 Enclose the binary counter with parallel load of Fig. 6.14 in a block diagram showing, all inputs and outputs.
(a) Show the connections of four such blocks to produce a 16-bit counter with parallel load.
(b) Construct a binary counter that counts from 0 through binary 127.

6.21* The counter of Fig. 6.14 has two control inputs—*Load* (*L*) and *Count* (*C*)—and a data input, (I_i).
(a) Derive the flip-flop input equations for *J* and *K* of the first stage in terms of *L*, *C*, and *I*.
(b) The logic diagram of the first stage of an equivalent circuit is shown in Fig. P6.21. Verify that this circuit is equivalent to the one in (a).

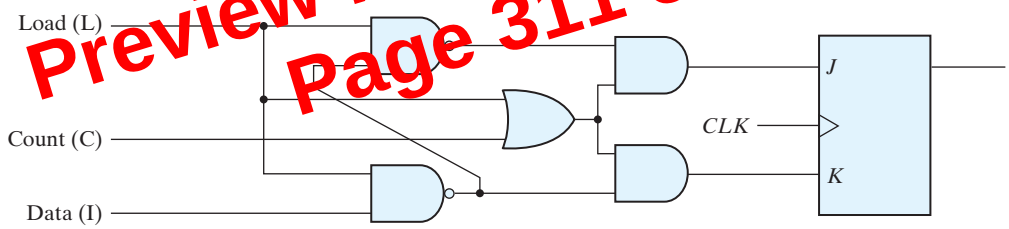


FIGURE P6.21

6.22 For the circuit of Fig. 6.14, give three alternatives for a mod-10 counter (i.e., the count evolves through a sequence of 12 distinct states).
(a) Using an AND gate and the load input.
(b) Using the output carry.
(c) Using a NAND gate and the asynchronous clear input.

6.23 Design a timing circuit that provides an output signal that stays on for exactly twelve clock cycles. A start signal sends the output to the 1 state, and after twelve clock cycles the signal returns to the 0 state. (HDL—see Problem 6.45.)

6.24* Design a counter with *T* flip-flops that goes through the following binary repeated sequence: 0, 1, 3, 7, 6, 4. Show that when binary states 010 and 101 are considered as don't care conditions, the counter may not operate properly. Find a way to correct the design. (HDL—see Problem 6.55.)

6.25 It is necessary to generate six repeated timing signals T_0 through T_5 similar to the ones shown in Fig. 6.17(c). Design the circuit using (HDL—see Problem 6.46.):
(a) flip-flops only.
(b) a counter and a decoder.

Memory address		Memory content
Binary	Decimal	
0000000000	0	1011010101011101
0000000001	1	1010101110001001
0000000010	2	0000110101000110
⋮	⋮	⋮
1111111101	1021	1001110100010100
1111111110	1022	0000110100011110
1111111111	1023	1101111000100101

FIGURE 7.3
Contents of a 1024 × 16 memory

a memory is dependent on the total number of words that can be stored in the memory and is independent of the number of bits in each word. The number of bits in the address is determined from the relationship $2^k \geq m$, where m is the total number of words and k is the number of address bits needed to satisfy the relationship.

Write and Read Operations

The two operations that RAM can perform are the write and read operations. As alluded to earlier, the write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired operation.

The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Apply the data bits that must be stored in memory to the data input lines.
3. Activate the *write* input.

The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Apply the binary address of the desired word to the address lines.
2. Activate the *read* input.

The 4 parity bits, P_1, P_2, P_4, and P_8, are in positions 1, 2, 4, and 8, respectively. The 8 bits of the data word are in the remaining positions. Each parity bit is calculated as follows:

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 5, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

Remember that the exclusive-OR operation performs the odd function: It is equal to 1 for an odd number of 1's in the variables and to 0 for an even number of 1's. Thus, each parity bit is set so that the total number of 1's in the checked positions, including the parity bit, is always even.

The 8-bit data word is stored in memory together with the 4 parity bits as a 12-bit composite word. Substituting the 4 P bits in their proper positions, we obtain the 12-bit composite word stored in memory:

	0	0	1	1	0	0	1	0	1	0	0
Bit position:	1	2	3	4	5	6	7	8	9	10	11	12

When the 12 bits are read from memory, they are checked again for errors. The parity is checked over the same combination of bits, including the parity bit. The 4 check bits are evaluated as follows:

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

A 0 check bit designates even parity over the checked bits and a 1 designates odd parity. Since the bits were stored with even parity, the result, $C = C_8C_4C_2C_1 = 0000$, indicates that no error has occurred. However, if $C \neq 0$, then the 4-bit binary number formed by the check bits gives the position of the erroneous bit. For example, consider the following three cases:

Bit position:	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

In the first case, there is no error in the 12-bit word. In the second case, there is an error in bit position number 1 because it changed from 0 to 1. The third case shows

2, 3, 6, 7, and so on. Comparing these numbers with the bit positions used in generating and checking parity bits in the Hamming code, we note the relationship between the bit groupings in the code and the position of the 1-bits in the binary count sequence. Note that each group of bits starts with a number that is a power of 2: 1, 2, 4, 8, 16, etc. These numbers are also the position numbers for the parity bits.

Single-Error Correction, Double-Error Detection

The Hamming code can detect and correct only a single error. By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors. If we include this additional parity bit, then the previous 12-bit coded word becomes 001110010100P_{13}, where P_{13} is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word 0011100101001 (even parity). When the 13-bit word is read from memory, the check bits are evaluated, as is the parity P over the entire 13 bits. If $P = 0$, the parity is correct (even parity), but if $P = 1$, then the parity over the 13 bits is incorrect (odd parity). The following four cases can arise:

If $C = 0$ and $P = 0$, no error occurred.
If $C \neq 0$ and $P = 1$, a single error occurred that can be corrected.
If $C \neq 0$ and $P = 0$, a double error occurred that is detected, but that cannot be corrected.
If $C = 0$ and $P = 1$, an error occurred in the P_{13} bit.

This scheme may detect more than two errors, but is not guaranteed to detect all such errors.

Integrated circuits use a modified Hamming code to generate and check parity bits for single-error correction and double-error detection. The modified Hamming code uses a more efficient parity configuration that balances the number of bits used to calculate the XOR operation. A typical integrated circuit that uses an 8-bit data word and a 5-bit check word is IC type 74637. Other integrated circuits are available for data words of 16 and 32 bits. These circuits can be used in conjunction with a memory unit to correct a single error or detect double errors during write and read operations.

7.5 READ-ONLY MEMORY

A read-only memory (ROM) is essentially a memory device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. Once the pattern is established, it stays within the unit even when power is turned off and on again.

A block diagram of a ROM consisting of k inputs and n outputs is shown in Fig. 7.9. The inputs provide the address for memory, and the outputs give the data bits of the stored word that is selected by the address. The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words. Note that ROM does not have data inputs, because it does not have a write operation. Integrated

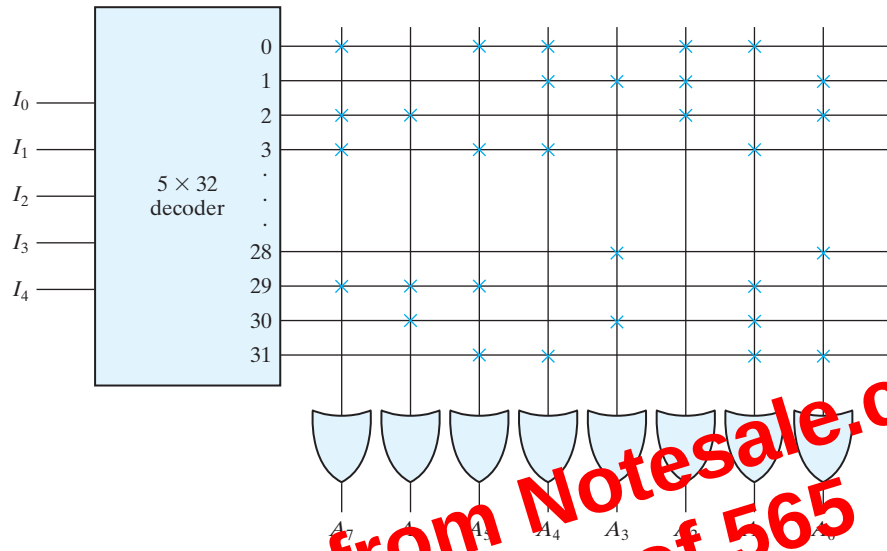


FIGURE 7.11
Programming the ROM according to Table 7.2

Combinational Circuit Implementation

In Section 4.9, it was shown that a decoder generates the 2^k minterms of the k input variables. By inserting OR gates to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. The ROM is essentially a device that includes both the decoder and the OR gates within a single device to form a minterm generator. By choosing connections for those minterms which are included in the function, the ROM outputs can be programmed to represent the Boolean functions of the output variables in a combinational circuit.

The internal operation of a ROM can be interpreted in two ways. The first interpretation is that of a memory unit that contains a fixed pattern of stored words. The second interpretation is that of a unit which implements a combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed as a sum of minterms. For example, the ROM of Fig. 7.11 may be considered to be a combinational circuit with eight outputs, each a function of the five input variables. Output A_7 can be expressed in sum of minterms as

$$A_7(I_4, I_3, I_2, I_1, I_0) = \Sigma(0, 2, 3, \ldots, 29)$$

(The three dots represent minterms 4 through 27, which are not specified in the figure.) A connection marked with $\times$ in the figure produces a minterm for the sum. All other crosspoints are not connected and are not included in the sum.

In practice, when a combinational circuit is designed by means of a ROM, it is not necessary to design the logic or to show the internal gate connections inside the unit. All that the designer has to do is specify the particular ROM by its IC number and provide the applicable truth table. The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the truth table.

The size of a PLA is specified by the number of inputs, the number of product terms, and the number of outputs. A typical integrated circuit PLA may have 16 inputs, 48 product terms, and eight outputs. For n inputs, k product terms, and m outputs, the internal logic of the PLA consists of n buffer–inverter gates, k AND gates, m OR gates, and m XOR gates. There are $2n \times k$ connections between the inputs and the AND array, $k \times m$ connections between the AND and OR arrays, and m connections associated with the XOR gates.

In designing a digital system with a PLA, there is no need to show the internal connections of the unit as was done in Fig. 7.14. All that is needed is a PLA programming table from which the PLA can be programmed to supply the required logic. As with a ROM, the PLA may be mask programmable or field programmable. With mask programming, the customer submits a PLA program table to the manufacturer. This table is used by the vendor to produce a custom-made PLA that has the required internal logic specified by the customer. A second type of PLA that is available is the field-programmable logic array, or FPLA, which can be programmed by the user by means of a commercial hardware programmer unit.

In implementing a combinational circuit with a PLA, careful investigation must be undertaken in order to reduce the number of distinct product terms, since a PLA has a finite number of AND gates. This can be done by simplifying each Boolean function to a minimum number of terms. The number of literals in a term is not important, since all the input variables are available anyway. Both the true value and the complement of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

EXAMPLE 7.2

Implement the following two Boolean functions with a PLA:

$$F_1(A, B, C) = \sum(0, 1, 2, 4)$$

$$F_2(A, B, C) = \sum(0, 5, 6, 7)$$

The two functions are simplified in the maps of Fig. 7.15. Both the true value and the complement of the functions are simplified into sum-of-products form. The combination that gives the minimum number of product terms is

$$F_1 = (AB + AC + BC)'$$

and

$$F_2 = AB + AC + A'B'C'$$

This combination gives four distinct product terms: AB, AC, BC, and $A'B'C'$. The PLA programming table for the combination is shown in the figure. Note that output F_1 is the true output, even though a C is marked over it in the table. This is because F_1 is generated with an AND–OR circuit and is available at the output of the OR gate. The XOR gate complements the function to produce the true F_1 output.

PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$A'B'C'$	4	0	0	0	–	1

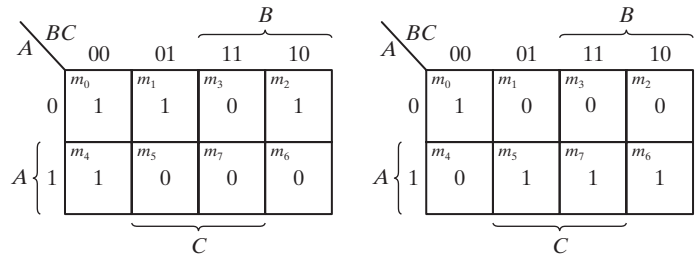


FIGURE 7.15
Solution to Example 7.2

The combinational circuit used in Example 7.2 is too simple for implementing with a PLA. It was presented merely for purposes of illustration. A typical PLA has a large number of inputs and product terms. The simplification of Boolean functions with so many variables should be carried out by means of computer-assisted simplification procedures. The computer-aided design (CAD) program simplifies each function and its complement to a minimum number of terms. The program then selects a minimum number of product terms that cover all functions in the form in which they are true or in their complemented form. The PLA programming table is then generated and the required fuse map obtained. The fuse map is applied to an FPLA programmer that goes through the hardware procedure of blowing the internal fuses in the integrated circuit.

7.7 PROGRAMMABLE ARRAY LOGIC

The PAL is a programmable logic device with a fixed OR array and a programmable AND array. Because only the AND gates are programmable, the PAL is easier to program than, but is not as flexible as, the PLA. Figure 7.16 shows the logic configuration of a typical PAL with four inputs and four outputs. Each input has a buffer–inverter gate, and each output is generated by a fixed OR gate. There are four sections in the unit, each composed of an AND–OR array that is *three wide*, the term used to indicate that there are three programmable AND gates in each section and one fixed OR gate. Each AND gate has 10 programmable input connections, shown in the diagram by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple-input configuration of the AND gate. One of the outputs is connected to a buffer–inverter gate and then fed back into two inputs of the AND gates.

Commercial PAL devices contain more gates than the one shown in Fig. 7.16. A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections, each consisting of an eight-wide AND–OR array. The output terminals are sometimes driven by three-state buffers or inverters.

In designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the situation with a PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself, without regard

programming table for the four Boolean functions. The table is divided into four sections with three product terms in each, to conform with the PAL of Fig. 7.16. The first two sections need only two product terms to implement the Boolean function. The last section, for output z, needs four product terms. Using the output from w, we can reduce the function to three terms.

The fuse map for the PAL as specified in the programming table is shown in Fig. 7.17. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash, we mark the diagram with blown fuses in both the true and complement inputs. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the true value and the complement of each input variable, we have $AA' = 0$ and the output of the AND gate is always 0.

As with all PLDs, the design with PALs is facilitated by using CAD techniques. The blowing of internal fuses is a hardware procedure done with the help of special electronic instruments.

7.8 SEQUENTIAL PROGRAMMABLE DEVICES

Digital systems are designed with flip-flops and gates. Since the combinational PLD consists only of gates, it is necessary to include external flip-flops when they are used in the design. Sequential programmable devices include both gates and flip-flops. In this way, the device can be programmed to perform a variety of sequential-circuit functions. There are several types of sequential programmable devices available commercially, and each device has vendor-specific variants within each type. The internal logic of these devices is too complex to be shown here. Therefore, we will describe three major types without going into their detailed construction:

1. Sequential (or simple) programmable logic device (SPLD)
2. Complex programmable logic device (CPLD)
3. Field-programmable gate array (FPGA)

The sequential PLD is sometimes referred to as a simple PLD to differentiate it from the complex PLD. The SPLD includes flip-flops, in addition to the AND–OR array, within the integrated circuit chip. The result is a sequential circuit as shown in Fig. 7.18. A PAL or PLA is modified by including a number of flip-flops connected to form a register. The circuit outputs can be taken from the OR gates or from the outputs of the

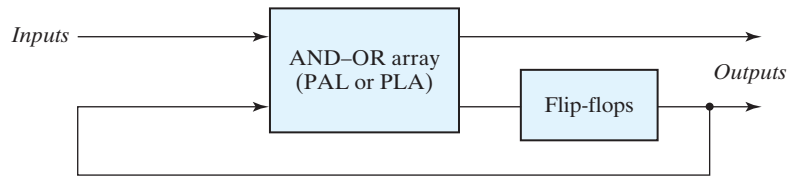


FIGURE 7.18
Sequential programmable logic device

operations. These operations are implemented with digital hardware components such as adders, decoders, multiplexers, counters, and shift registers. Control information provides command signals that coordinate and execute the various operations in the data section of the machine in order to accomplish the desired data-processing tasks.

The design of the logic of a digital system can be divided into two distinct efforts. One part is concerned with designing the digital circuits that perform the data-processing operations. The other part is concerned with designing the control circuits that determine the sequence in which the various manipulations of data are performed.

The relationship between the control logic and the data-processing operations in a digital system is shown in Fig. 8.2. The data-processing path, commonly referred to as the *datapath unit*, manipulates data in registers according to the system's requirements. The *control unit* issues a sequence of commands to the datapath unit. Note that an internal feedback path from the datapath unit to the control unit provides status conditions that the control unit uses together with the external (primary) inputs to determine the sequence of control signals (outputs of the control unit) that direct the operation of the datapath unit. We'll see later that understanding how to model this feedback relationship with an HDL is very important.

The control logic that generates the signals for sequencing the operations in the datapath unit is a finite state machine (FSM), i.e., a synchronous sequential circuit. The control commands for the system are produced by the FSM as functions of the primary inputs, the status signals, and the state of the machine. In a given state, the outputs of the controller are the inputs to the datapath unit and determine the operations that it will execute. Depending on status conditions and other external inputs, the FSM goes to its next state to initiate other operations. The digital circuits that act as the control logic provide a time sequence of signals for initiating the operations in the datapath and also determine the next state of the control subsystem itself.

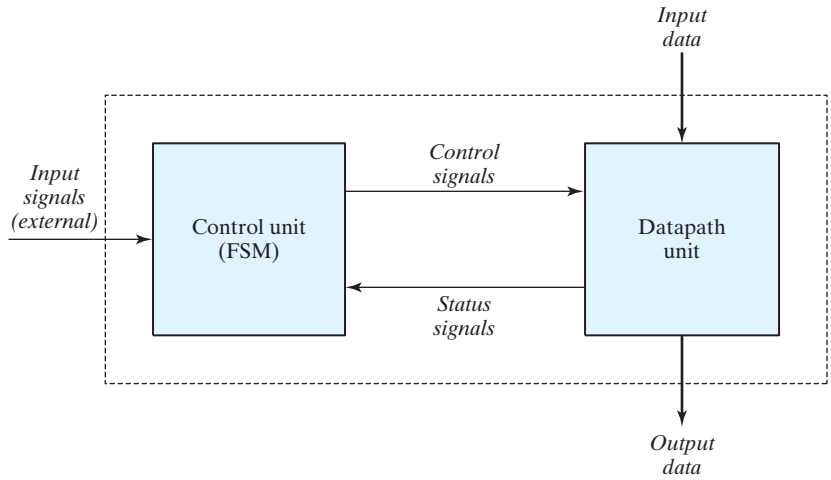


FIGURE 8.2
Control and datapath interaction

The steps to form an ASMD chart are:

1. Form an ASM chart showing only the states of the controller and the input signals[2] that cause state transitions,
2. Convert the ASM chart into an ASMD chart by annotating the edges of the ASM chart to indicate the concurrent register operations of the datapath unit (i.e., register operations that are concurrent with a state transition), and
3. Modify the ASMD chart to identify the control signals that are generated by the controller and that cause the indicated operations in the datapath unit.

The ASMD chart produced by this process clearly and completely specifies the finite state machine of the controller, identifies the registers operations of the datapath unit, identifies signals reporting the status of the datapath to the controller, and links register operations to the signals that control them.

One important use of a state machine is to control register operations on a datapath in a sequential machine that has been partitioned into a controller and a datapath. An ASMD chart links the ASM chart of the controller to the datapath it controls in a manner that serves as a universal model representing all synchronous digital hardware design. ASMD charts help clarify the design of a sequential machine by separating the design of its datapath from the design of the controller, while maintaining a clear relationship between the two units. Register operations that occur concurrently with state transitions are annotated on a path of the chart, rather than in state boxes or in conditional boxes on the path because these registers are not part of the controller. The outputs generated by the controller are the signals that control the registers of the datapath and cause the register operations annotated on the ASMD chart.

8.5 DESIGN EXAMPLE (ASMD CHART)

We will now present a simple example demonstrating the use of the ASMD chart and the register transfer representation. We start from the initial specifications of a system and proceed with the development of an appropriate ASMD chart from which the digital hardware is then designed.

The datapath unit is to consist of two *JK* flip-flops *E* and *F*, and one four-bit binary counter *A[3:0]*. The individual flip-flops in *A* are denoted by A_3, A_2, A_1, and A_0, with A_3 holding the most significant bit of the count. A signal, *Start*, initiates the system's operation by clearing the counter *A* and flip-flop *F*. At each subsequent clock pulse, the counter is incremented by 1 until the operations stop. Counter bits A_2 and A_3 determine the sequence of operations:

If $A_2 = 0$, *E* is cleared to 0 and the count continues.

If $A_2 = 1$, *E* is set to 1; then, if $A_3 = 0$, the count continues, but if $A_3 = 1$, *F* is set to 1 on the next clock pulse and the system stops counting.

[2] In general, the inputs to the control unit are external (primary) inputs and status signals that originate in the datapath unit.

the ASMD chart (Fig. 8.9(d)). Note that nonblocking assignments are used (with symbol <=) for the register transfer operations. This ensures that the register operations and state transitions are concurrent, a feature that is especially crucial during control state *S_1*. In this state, *A* is incremented by 1 and the value of *A2* (*A[2]*) is checked to determine the operation to execute at register *E* at the next clock. To accomplish a valid synchronous design, it is necessary to ensure that *A[2]* is checked before *A* is incremented. If blocking assignments were used, one would have to place the two statements that check *E* first and the *A* statement that increments last. However, by using nonblocking assignments, we accomplish the required synchronization without being concerned about the order in which the statements are listed. The counter *A* in *Datapath_RTL* is cleared synchronously because *clr_A_F* is synchronized to the clock.

The cyclic behaviors of the controller and the datapath interact in a chain reaction: At the active edge of the clock, the state and datapath registers are updated. A change in the state, a primary input, or a status input causes the level-sensitive behaviors of the controller to update the value of the next state and the outputs. The updated values are used at the next active edge of the clock to determine the state transition and the updates of the datapath.

Note that the manual method of design developed (1) a block diagram (Fig. 8.9(a)) showing the interface between the datapath and the controller, (2) an ASMD chart for the system (Fig. 8.9(d)), (3) the logic equations for the inputs to the flip-flops of the controller, and (4) a circuit that implements the controller (Fig. 8.12). In contrast, an RTL model describes the state transitions of the controller and the operations of the datapath as a step toward automatically synthesizing the circuit that implements them. The descriptions of the datapath and controller are derived directly from the ASMD chart in both cases.

HDL Example 8.2

```
// RTL description of design example (see Fig. 8.11)
module Design_Example_RTL (A, E, F, Start, clock, reset_b);
  // Specify ports of the top-level module of the design
  // See block diagram, Fig. 8.10
  output [3: 0] A;
  output        E, F;
  input         Start, clock, reset_b;
  // Instantiate controller and datapath units
  Controller_RTL M0 (set_E, clr_E, set_F, clr_A_F, incr_A, A[2], A[3], Start, clock, reset_b);
  Datapath_RTL M1 (A, E, F, set_E, clr_E, set_F, clr_A_F, incr_A, clock);
endmodule
module Controller_RTL (set_E, clr_E, set_F, clr_A_F, incr_A, A2, A3, Start, clock, reset_b);
    output reg      set_E, clr_E, set_F, clr_A_F, incr_A;
    input           Start, A2, A3, clock, reset_b;
```

```
module JK_flip_flop_2 (Q, Q_not, J, K, CLK);
 output      Q, Q_not;
 input       J, K, CLK;
 reg         Q;
 assign      Q_not = ~Q;
 always @ (posedge CLK)
  case ({J, K})
   2'b00:    Q <= Q;
   2'b01:    Q <= 1'b0;
   2'b10:    Q <= 1'b1;
   2'b11:    Q <= ~Q;
  endcase
endmodule

module t_Design_Example_STR;
 reg Start, clock, reset_b;
 wire [3: 0] A;
 wire        E, F;

 // Instantiate design example
 Design_Example_STR M0 (A, E, F, Start, clock, reset_b);
// Describe stimulus waveforms

initial #500 $finish;              // Stopwatch
initial
 begin
  reset_b = 0;
  Start = 0;
  clock = 0;
  #5 reset_b = 1; Start = 1;
  repeat (32)
   begin
    #5 clock = ~ clock;            // Clock generator
   end
 end
initial
 $monitor ("A = %b E = %b F = %b time = %0d", A, E, F, $time);
endmodule
```


The structural description was tested with the test bench that verified the RTL description to produce the results shown in Fig. 8.13. The only change necessary is the replacement of the instantiation of the example from *Design_Example_RTL* by *Design_Example_STR*. The simulation results for *Design_Example_STR* matched those for *Design_Example_RTL*. However, a comparison of the two descriptions indicates that the RTL style is easier

product in the time span of a single clock cycle will synthesize the circuit of a parallel multiplier like the one discussed in Section 4.7. On the other hand, an RTL model of the algorithm adds shifted copies of the multiplicand to an accumulated partial product. The values of the multiplier, multiplicand, and partial product are stored in registers, and the operations of shifting and adding their contents are executed under the control of a state machine. Among the many possibilities for distributing the effort of multiplication over multiple clock cycles, we will consider that in which only one partial product is formed and accumulated in a single cycle of the clock. (One alternative would be to use additional hardware to form and accumulate two partial products in a clock cycle, but this would require more logic gates and either faster circuits or a slower clock.) Instead of providing digital circuits to store and add simultaneously as many binary numbers as there are 1's in the multiplier, it is less expensive to provide only the hardware needed to sum two binary numbers and accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product being formed is shifted to the right. This leaves the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit of the multiplier is 0, there is no need to add all 0's to the partial product since doing so will not alter its resulting value.

Register Configuration

A block diagram for the sequential binary multiplier is shown in Fig. 8.14(a), and the register configuration of the datapath is shown in Fig. 8.14(b). The multiplicand is stored in register B, the multiplier is stored in register Q, and the partial product is formed in register A and stored in A and Q. A parallel adder adds the contents of register B to register A. The C flip-flop stores the carry after the addition. The counter P is initially set to hold a binary number equal to the number of bits in the multiplier. This counter is decremented after the formation of each partial product. When the content of the counter reaches zero, the product is formed in the double register A and Q, and the process stops. The control logic stays in an initial state until *Start* becomes 1. The system then performs the multiplication. The sum of A and B forms the n most significant bits of the partial product, which is transferred to A. The output carry from the addition, whether 0 or 1, is transferred to C. Both the partial product in A and the multiplier in Q are shifted to the right. The least significant bit of A is shifted into the most significant position of Q, the carry from C is shifted into the most significant position of A, and 0 is shifted into C. After the shift-right operation, one bit of the partial product is transferred into Q while the multiplier bits in Q are shifted one position to the right. In this manner, the least significant bit of register Q, designated by *Q[0]*, holds the bit of the multiplier that must be inspected next. The control logic determines whether to add or not on the basis of this input bit. The control logic also receives a signal, *Zero*, from a circuit that checks counter P for zero. *Q[0]* and *Zero* are status inputs for the control unit. The input signal *Start* is an external control input. The outputs of the control logic launch the required operations in the registers of the datapath unit.

in the block diagram of Fig. 8.14(a). The machine will be parameterized for a five-bit datapath to enable a comparison between its simulation data and the result of the multiplication with the numerical example listed in Table 8.5. The same model can be used for a datapath having a different size merely by changing the value of the parameters. The second part of the description declares all registers in the controller and the datapath, as well as the one-hot encoding of the states. The third part specifies implicit combinational logic (continuous assignment statements) for the concatenated register *CAQ*, the *Zero* status signal, and the *Ready* output signal. The continuous assignments for *Zero* and *Ready* are accomplished by assigning a Boolean expression to their **wire** declarations. The next section describes the control unit, using a single edge-sensitive cyclic behavior to describe the state transitions, and a level-sensitive cyclic behavior to describe the combinational logic for the next state and the outputs. Again, note that default assignments are made to *next_state, Load_regs, Decr_P, Add_regs*, and *Shift_regs*. The subsequent logic of the case statement assigns their value by exception. The state transitions and the output logic are written directly from the ASMD chart of Fig. 8.15(b).

The datapath unit describes the register operations within a separate edge-sensitive cyclic behavior.[3] (For clarity, separate cyclic behaviors are used; we do not mix the description of the datapath with the description of the controller.) Each control input is decoded and is used to specify the associated operations. The addition and subtraction operations will be implemented in hardware by combinational logic. Signal *Load_regs* causes the counter and the other registers to be loaded with their initial values, etc. Because the controller and datapath have been partitioned into separate units, the control signals completely specify the behavior of the datapath; explicit information about the state of the controller is not needed and is not made available to the datapath unit.

The next-state logic of the controller includes a default case item to direct a synthesis tool to map any of the unused codes to *S_idle*. The default case item and the default assignments preceding the **case** statement ensure that the machine will recover if it somehow enters an unused state. They also prevent unintentional synthesis of latches. (Remember, a synthesis tool will synthesize latches when what was intended to be combinational logic in fact fails to completely specify the input–output function of the logic.)

HDL Example 8.5 (Sequential Multiplier)

```
module Sequential_Binary_Multiplier (Product, Ready, Multiplicand, Multiplier, Start,
clock, reset_b);
// Default configuration: five-bit datapath
 parameter                          dp_width = 5;      // Set to width of datapath
 output      [2*dp_width -1: 0]     Product;
 output                             Ready;
 input       [dp_width -1: 0]       Multiplicand, Multiplier;
 input                              Start, clock, reset_b;
```

[3] The width of the datapath here is *dp-width*.

```
module Datapath_STR (count, E, Zero, data, Load_regs, Shift_left, Incr_R2, clock);
 parameter                              R1_size = 8, R2_size = 4;
 output [R2_size -1: 0]  count;
 output                                 E, Zero;
 input   [R1_size -1: 0]                data;
 input                                  Load_regs, Shift_left, Incr_R2, clock;
 wire    [R1_size -1: 0]                R1;
 wire                                   Zero;
 supply0                                Gnd;
 supply1                                Pwr;
 assign Zero = (R1 == 0);               // implicit combinational logic
 Shift_Reg              M1              (R1, data, Gnd, Shift_left, Load_regs, clock, Pwr);
 Counter                M2              (count, Load_regs, Incr_R2, clock, Pwr);
 D_flip_flop_AR         M3              (E, w1, clock, Pwr);
 and                                    (w1, R1[R1_size -1], Shift_left);
endmodule

module Shift_Reg (R1, data, SI_0, Shift_left, Load_regs, clock, reset_b);
 parameter R1_size = 8;
 output  [R1_size -1: 0]        R1;
 input   [R1_size -1: 0]        data;
 input                          SI_0, Shift_left, Load_regs;
 input                          clock, reset_b;
 reg     [R1_size -1: 0]        R1;
 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) R1 <= 0;
  else begin
   if (Load_regs) R1 <= data; else
    if (Shift_left) R1 <= {R1[R1_size -2: 0], SI_0}; end
endmodule
module Counter (R2, Load_regs, Incr_R2, clock, reset_b);
 parameter                      R2_size = 4;
 output  [R2_size -1: 0]        R2;
 input                          Load_regs, Incr_R2;
 input                          clock, reset_b;
 reg     [R2_size -1: 0]        R2;
 always @ (posedge clock, negedge reset_b)
  if (reset_b == 0) R2 <= 0;
  else if (Load_regs) R2 <= {R2_size {1'b1}};          // Fill with 1
   else if (Incr_R2 == 1) R2 <= R2 + 1;
endmodule
module D_flip_flop_AR (Q, D, CLK, RST_b);
 output          Q;
 input           D, CLK, RST_b;
```

a second door with a similar photocell that changes a signal y from 1 to 0 while the light is interrupted. The datapath circuit consists of an up–down counter with a display that shows how many people are in the room.

8.7* Draw a block diagram and an ASMD chart for a circuit with two eight-bit registers RA and RB that receive two unsigned binary numbers. The circuit performs the subtraction operation

$$RA \leftarrow RA - RB$$

Use the method for subtraction described in Section 1.5, and set a borrow flip-flop to 1 if the answer is negative. Write and verify an HDL model of the circuit.

8.8* Design a digital circuit with three 16-bit registers AR, BR, and CR that perform the following operations:
(a) Transfer two 16-bit signed numbers (in 2's-complement representation) to AR and BR.
(b) If the number in AR is negative, divide the number in AR by 2 and transfer the result to register CR.
(c) If the number in AR is positive but nonzero, multiply the number in BR by 2 and transfer the result to register CR.
(d) If the number in AR is zero, clear register CR to 0.
(e) Write and verify a behavioral model of the circuit.

8.9* Design the controller whose state diagram is given by Fig. 8.11(a). Use one flip-flop per state (a one-hot assignment). Write, simulate, verify, and compare RTL and structural models of the controller.

8.10 The state diagram of a control unit is shown in Fig. P8.10. It has four states and two inputs x and y. Draw the equivalent ASM chart. Write and verify a Verilog model of the controller.

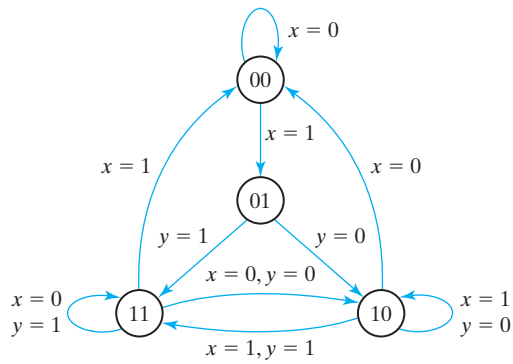


FIGURE P8.10
Control state diagram for Problems 8.10 and 8.11

8.11* Design the controller whose state diagram is shown in Fig. P8.10. Use D flip-flops.

8.12 Design the four-bit counter with synchronous clear specified in Fig. 8.10. Repeat for asynchronous clear.

8.13 Simulate *Design_Example_STR* (see HDL Example 8.4), and verify that its behavior matches that of the RTL description. Obtain state information by displaying *G0* and *G1* as a concatenated vector for the state.

8.14 What, if any, are the consequences of the machine in *Design_Example_RTL* (see HDL Example 8.2) entering an unused state?

8.15 *Simulate Design_Example_RTL* in HDL Example 8.2, and verify that it recovers from an unexpected reset condition during its operation, i.e., a "running reset" or a "reset on-the-fly."

8.16* Develop a block diagram and an ASMD chart for a digital circuit that multiplies two binary numbers by the repeated-addition method. For example, to multiply 5×4, the digital system evaluates the product by adding the multiplicand four times: $5 + 5 + 5 + 5 = 20$. Design the circuit. Let the multiplicand be in register *BR,* the multiplier in register *AR,* and the product in register *PR*. An adder circuit adds the contents of *BR* to *PR*. A zero-detection signal indicates whether *AR* is 0. Write and verify a Verilog behavioral model of the circuit.

8.17* Prove that the multiplication of two n-bit numbers gives a product of length less than or equal to $2n$ bits.

8.18* In Fig. 8.14, the *Q* register holds the multiplier and the *B* register holds the multiplicand. Assume that each number consists of 16 bits.
(a) How many bits can be expected in the product, and where is it available?
(b) How many bits are in the *P* counter, and what is the binary number loaded into it initially?
(c) Design the circuit that checks for zero in the *P* counter.

8.19 List the contents of registers *C, A, Q*, and *P* in a manner similar to Table 8.5 during the process of multiplying the two numbers 11011 (multiplicand) and 10111 (multiplier).

8.20* Determine the time it takes to process the multiplication operation in the binary multiplier described in Section 8.8. Assume that the *Q* register has n bits and the clock cycle is t ns.

8.21 Design the control circuit of the binary multiplier specified by the state diagram of Fig. 8.16, using multiplexers, a decoder, and a register.

8.22 Figure P8.22 shows an alternative ASMD chart for a sequential binary multiplier. Write and verify an RTL model of the system. Compare this design with that described by the ASMD chart in Fig. 8.15(b).

8.23 Figure P8.23 shows an alternative ASMD chart for a sequential binary multiplier. Write and verify an RTL model of the system. Compare this design with that described by the ASMD chart in Fig. 8.15(b).

8.24 The HDL description of a sequential binary multiplier given in HDL Example 8.5 encapsulates the descriptions of the controller and the datapath in a single Verilog module. Write and verify a model that encapsulates the controller and datapath in separate modules.

8.25 The sequential binary multiplier described by the ASMD chart in Fig. 8.15 does not consider whether the multiplicand or the shifted multiplier is 0. Therefore, it executes for a fixed number of clock cycles, independently of the data.
(a) Develop an ASMD chart for a more efficient multiplier that will terminate execution as soon as either word is found to be zero.

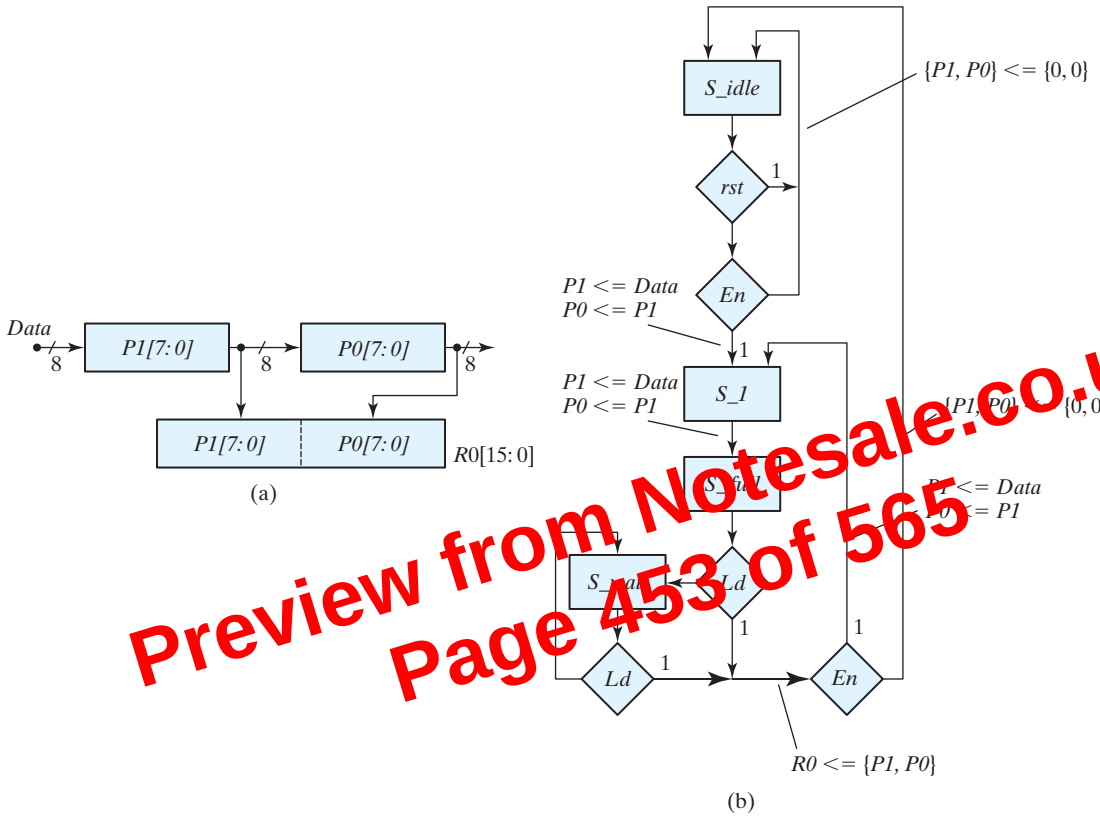


FIGURE P8.41
Two-stage pipeline register: Datapath unit and ASMD chart

8.41 The block diagram and partially completed ASMD chart in Fig. P8.41 describe the behavior of a two-stage pipeline that acts as a 2:1 decimator with a parallel input and output. Decimators are used in digital signal processors to move data from a datapath with a high clock rate to a datapath with a lower clock rate, converting data from a parallel format to a serial format in the process. In the datapath shown, entire words of data can be transferred into the pipeline at twice the rate at which the contents of the pipeline must be dumped into a holding register or consumed by some processor. The contents of the holding register *R0* can be shifted out serially, to accomplish an overall parallel-to-serial conversion of the data stream. The ASMD chart indicates that the machine has synchronous reset to *S_idle*, where it waits until *rst* is de-asserted and En is asserted. Note that synchronous transitions which would occur from the other states to *S_idle* under the action of *rst* are not shown. With *En* asserted, the machine transitions from *S_idle* to *S_1*, accompanied by concurrent register operations that load the MSByte of the pipe with *Data* and move the content of *P1* to the LSByte (*P0*). At the next clock, the state goes to *S_full*, and now the pipe is full. If *Ld* is asserted at the next clock, the machine moves to *S_1* while dumping the pipe into a holding register *R0*. If *Ld* is not asserted, the machine

a variable clock, a power supply, and IC socket strips. Some experiments may require additional switches, lamps, or IC socket strips. Extended breadboards with more solderless sockets and plug-in switches and lamps may be needed.

Additional equipment required is a dual-trace oscilloscope (for Experiments 1, 2, 8, and 15), a logic probe to be used for debugging, and a number of ICs. The ICs required for the experiments are of the TTL or CMOS series 7400.

The integrated circuits to be used in the experiments can be classified as small-scale integration (SSI) or medium-scale integration (MSI) circuits. SSI circuits contain individual gates or flip-flops, and MSI circuits perform specific digital functions. The eight SSI gate ICs needed for the experiments—two-input NAND, NOR, AND, OR, and XOR gates, inverters, and three-input and four-input NAND gates—are shown in Fig. 9.1. The pin assignments for the gates are indicated in the diagram. The pins are numbered from 1 to 14. Pin number 14 is marked V_{CC}, and pin number 7 is marked *GND* (ground). These are the supply terminals, which must be connected to a power supply of 5 V for proper operation of the circuit. Each IC is recognized by its identification number; for example, the two-input NAND gates are found inside the IC whose number is 7400.

Detailed descriptions of the MSI circuits can be found in data books published by the manufacturers. The best way to acquire experience with a commercial MSI circuit is to study its description in a data book that provides complete information on the internal, external, and electrical characteristics of integrated circuits. Various semiconductor companies publish data books for the 7400 series. The MSI circuits that are needed for the experiments are introduced and explained when they are used for the first time. The operation of the circuit is explained by referring to similar circuits in previous chapters. The information given in this chapter about the MSI circuits should be sufficient for performing the experiments adequately. Nevertheless, reference to a data book will always be preferable, as it gives more detailed description of the circuits.

We will now demonstrate the method of presentation of MSI circuits adopted here. To illustrate, we introduce the ripple counter IC, type 7493. This IC is used in Experiment 1 and in subsequent experiments to generate a sequence of binary numbers for verifying the operation of combinational circuits.

The information about the 7493 IC that is found in a data book is shown in Figs. 9.2(a) and (b). Part (a) shows a diagram of the internal logic circuit and its connection to external pins. All inputs and outputs are given symbolic letters and assigned to pin numbers. Part (b) shows the physical layout of the IC, together with its 14-pin assignment to signal names. Some of the pins are not used by the circuit and are marked as *NC* (no connection). The IC is inserted into a socket, and wires are connected to the various pins through the socket terminals. When drawing schematic diagrams in this chapter, we will show the IC in block diagram form, as in Fig. 9.2(c). The IC number (here, 7493) is written inside the block. All input terminals are placed on the left of the block and all output terminals on the right. The letter symbols of the signals, such as *A*, *R1*, and *QA*, are written inside the block, and the corresponding pin numbers, such as 14, 2, and 12, are written along the external lines. V_{CC}, and *GND* are the power terminals connected to pins 5 and 10. The size of the block may vary to accommodate all input

9.5 EXPERIMENT 4: COMBINATIONAL CIRCUITS

In this experiment, you will design, construct, and test four combinational logic circuits. The first two circuits are to be constructed with NAND gates, the third with XOR gates, and the fourth with a decoder and NAND gates. Reference to a parity generator can be found in Section 3.9. Implementation with a decoder is discussed in Section 4.9.

Design Example

Design a combinational circuit with four inputs—*A, B, C*, and *D*—and one output, *F*. *F* is to be equal to 1 when $A = 1$, provided that $B = 0$, or when $B = 1$, provided that either *C* or *D* is also equal to 1. Otherwise, the output is to be equal to 0.

1. Obtain the truth table of the circuit.
2. Simplify the output function.
3. Draw the logic diagram of the circuit, using NAND gates with a minimum number of ICs.
4. Construct the circuit and test it for proper operation by verifying the given conditions.

Majority Logic

A majority logic is a digital circuit whose output is equal to 1 if the majority of the inputs are 1's. The output is 0 otherwise. Design and test a three-input majority circuit using NAND gates with a minimum number of ICs.

Parity Generator

Design, construct, and test a circuit that generates an even parity bit from four message bits. Use XOR gates. Adding one more XOR gate, expand the circuit so that it generates an odd parity bit also.

Decoder Implementation

A combinational circuit has three inputs—*x, y*, and *z*—and three outputs—F_1, F_2, and F_3. The simplified Boolean functions for the circuit are

$$F_1 = xz + x'y'z'$$
$$F_2 = x'y + xy'z'$$
$$F_3 = xy + x'y'z$$

Implement and test the combinational circuit, using a 74155 decoder IC and external NAND gates.

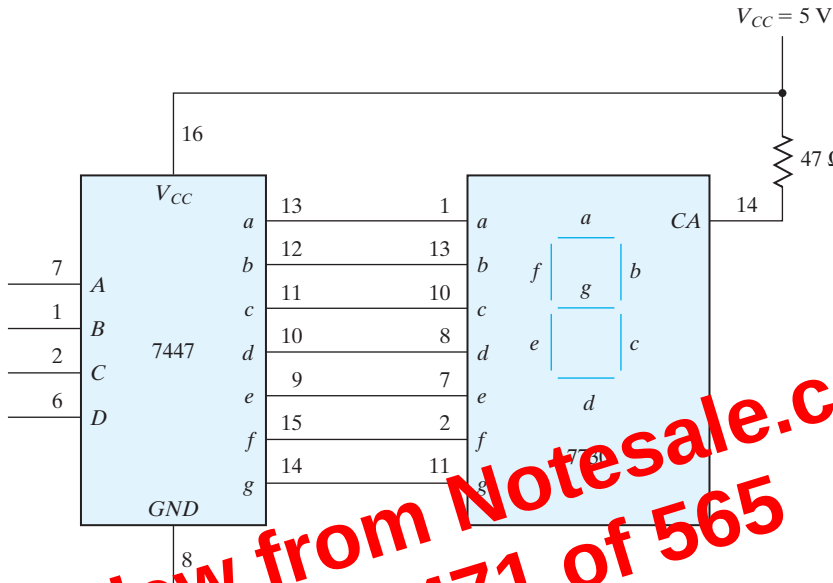


FIGURE 9.8
BCD-to-seven-segment decoder (7447) and seven-segment display (7730)

9.7 EXPERIMENT 6: DESIGN WITH MULTIPLEXERS

In this experiment, you will design a combinational circuit and implement it with multiplexers, as explained in Section 4.11. The multiplexer to be used is IC type 74151, shown in Fig. 9.9. The internal construction of the 74151 is similar to the diagram shown in Fig. 4.25, except that there are eight inputs instead of four. The eight inputs are designated *D0* through *D7*. The three selection lines—*C*, *B*, and *A*—select the particular input to be multiplexed and applied to the output. A strobe control *S* acts as an enable signal. The function table specifies the value of output *Y* as a function of the selection lines. Output *W* is the complement of *Y*. For proper operation, the strobe input *S* must be connected to ground.

Design Specifications

A small corporation has 10 shares of stock, and each share entitles its owner to one vote at a stockholder's meeting. The 10 shares of stock are owned by four people as follows:

Mr. W: 1 share

Mr. X: 2 shares

Mr. Y: 3 shares

Mrs. Z: 4 shares

preset and clear inputs. These inputs behave like a NAND *SR* latch and are independent of the clock or the *J* and *K* inputs. (The X's indicate don't-care conditions.) The last four entries in the function table specify the operation of the clock with both the preset and clear inputs maintained at logic 1. The clock value is shown as a single pulse. The positive transition of the pulse changes the master flip-flop, and the negative transition changes the slave flip-flop as well as the output of the circuit. With $J = K = 0$, the output does not change. The flip-flop toggles, or is complemented, when $J = K = 1$. Investigate the operation of one 7476 flip-flop and verify its function table.

IC type 7474 consists of two *D* positive-edge-triggered flip-flops with preset and clear. The pin assignment is shown in Fig. 9.13. The function table specifies the preset and clear operations and the clock's operation. The clock is shown with an upward arrow to indicate that it is a positive-edge-triggered flip-flop. Investigate the operation of one of the flip-flops and verify its function table.

9.10 EXPERIMENT 9: SEQUENTIAL CIRCUITS

In this experiment, you will design, construct, and test three synchronous sequential circuits. Use IC type 7476 (Fig. 9.12) or 7474 (Fig. 9.13). Choose any type of gate that will minimize the total number of ICs. The design of synchronous sequential circuits is covered in Section 5.7.

Up–Down Counter with Enable

Design, construct, and test a two-bit counter that counts up or down. An enable input *E* determines whether the counter is on or off. If $E = 0$, the counter is disabled and remains at its present count even though clock pulses are applied to the flip-flops. If $E = 1$, the counter is enabled and a second input, *x*, determines the direction of the count. If $x = 1$, the circuit counts upward with the sequence 00, 01, 10, 11, and the count repeats. If $x = 0$, the circuit counts downward with the sequence 11, 10, 01, 00, and the count repeats. Do not use *E* to disable the clock. Design the sequential circuit with *E* and *x* as inputs.

State Diagram

Design, construct, and test a sequential circuit whose state diagram is shown in Fig. 9.14. Designate the two flip-flops as *A* and *B*, the input as *x*, and the output as *y*.

Connect the output of the least significant flip-flop *B* to the input *x*, and predict the sequence of states and output that will occur with the application of clock pulses. Verify the state transition and output by testing the circuit.

Design of Counter

Design, construct, and test a counter that goes through the following sequence of binary states: 0, 1, 2, 3, 6, 7, 10, 11, 12, 13, 14, 15, and back to 0 to repeat. Note that binary states 4, 5, 8, and 9 are not used. The counter must be self-starting; that is, if the circuit starts from any one of the four invalid states, the count pulses must transfer the circuit to one of the valid states to continue the count correctly.

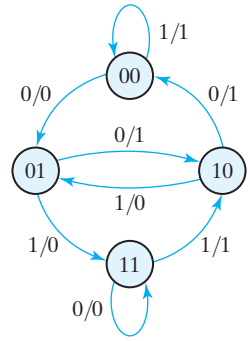


FIGURE 9.14
State diagram for Experiment 9

Check the circuit's operation for the required count sequence. Verify that the counter is self-starting. This is done by initializing the circuit to each unused state by means of the preset and clear inputs and then applying pulses to see whether the counter reaches one of the valid states.

9.11 EXPERIMENT 10: COUNTERS

In this experiment, you will construct and test various ripple and synchronous counter circuits. Ripple counters are discussed in Section 6.3 and synchronous counters are covered in Section 6.4.

Ripple Counter

Construct a four-bit binary ripple counter using two 7476 ICs (Fig. 9.12). Connect all asynchronous clear and preset inputs to logic 1. Connect the count-pulse input to a pulser and check the counter for proper operation.

Modify the counter so that it will count downward instead of upward. Check that each input pulse decrements the counter by 1.

Synchronous Counter

Construct a synchronous four-bit binary counter and check its operation. Use two 7476 ICs and one 7408 IC.

Decimal Counter

Design a synchronous BCD counter that counts from 0000 to 1001. Use two 7476 ICs and one 7408 IC. Test the counter for the proper sequence. Determine whether the counter is self-starting. This is done by initializing the counter to each of the six unused states by means of the preset and clear inputs. The application of pulses will transfer the counter to one of the valid states if the counter is self-starting.

the registers and the flip-flop. Another switch will be needed to specify whether register B is to accept parallel data or is to be shifted during the addition.

Testing the Adder

To test your serial adder, perform the binary addition $5 + 6 + 15 = 26$. This is done by first clearing the registers and the carry flip-flop. Parallel load the binary value 0101 into register B. Apply four pulses to add B to A serially, and check that the result in A is 0101. (Note that clock pulses for the 7476 must be as shown in Fig. 9.12.) Parallel load 0110 into B and add it to A serially. Check that A has the proper sum. Parallel load 1111 into B and add to A. Check that the value in A is 1010 and that the carry flip-flop is set.

Clear the registers and flip-flop and try a few other numbers to verify that your serial adder is functioning properly.

Serial Adder–Subtractor

If we follow the procedure used in Section 6.2 for the design of a serial subtractor (that subtracts $A - B$), we will find that the output difference is the same as the output sum, but that the input to the J and K of the borrow flip-flop needs the complement of QD (available in the 74195). Using the other two XOR gates from the 7486, convert the serial adder to a serial adder–subtractor with a mode control M. When $M = 0$, the circuit adds $A + B$. When $M = 1$, the circuit subtracts $A - B$ and the flip-flop holds the borrow instead of the carry.

Test the adder part of the circuit by repeating the operations recommended to ensure that the modification did not change the operation. Test the serial subtractor part by performing the subtraction $15 - 4 - 5 - 13 = -7$. Binary 15 can be transferred to register A by first clearing it to 0 and adding 15 from B. Check the intermediate results during the subtraction. Note that -7 will appear as the 2's complement of 7 with a borrow of 1 in the flip-flop.

9.14 EXPERIMENT 13: MEMORY UNIT

In this experiment, you will investigate the behavior of a random-access memory (RAM) unit and its storage capability. The RAM will be used to simulate a read-only memory (ROM). The ROM simulator will then be used to implement combinational circuits, as explained in Section 7.5. The memory unit is discussed in Sections 7.2 and 7.3.

IC RAM

IC type 74189 is a 16×4 RAM. The internal logic is similar to the circuit shown in Fig. 7.6 for a 4×4 RAM. The pin assignments to the inputs and outputs are shown in Fig. 9.18. The four address inputs select 1 of 16 words in the memory. The least significant bit of the address is A and the most significant is A_3. The chip select (CS) input must be equal to 0 to enable the memory. If CS is equal to 1, the memory is disabled and all four outputs are in a high-impedance state. The write enable (WE) input determines the type of operation, as indicated in the function table. The write operation is performed when $WE = 0$. This

data outputs to four 7404 inverters. Provide four indicator lamps for the address and four more for the outputs of the inverters. Connect input *CS* to ground and *WE* to a toggle switch (or a pulser that provides a negative pulse). Store a few words into the memory, and then read them to verify that the write and read operations are functioning properly. You must be careful when using the *WE* switch. Always leave the *WE* input in the read mode, unless you want to write into memory. The proper way to write is first to set the address in the counter and the inputs in the four toggle switches. Then, store the word in memory, flip the *WE* switch to the write position and return it to the read position. Be careful not to change the address or the inputs when *WE* is in the write mode.

ROM Simulator

A ROM simulator is obtained from a RAM operated in the read mode only. The pattern of 1's and 0's is first entered into the simulating RAM by placing the unit momentarily in the write mode. Simulation is achieved by placing the unit in the read mode and taking the address lines as inputs to the ROM. The ROM can then be used to implement any combinational circuit.

Implement a combinational circuit using the ROM simulator that converts a four-bit binary number to its equivalent Gray code as listed in Table 1.6. This is done as follows: Obtain the truth table of the code converter. Store the truth table into the 74189 memory by setting the address inputs to the binary value and the data inputs to the corresponding Gray code value. After all 16 entries of the table are written into memory, the ROM simulator is set by permanently connecting the *WE* line to logic 1. Check the code converter by applying the inputs to the address lines and verifying the correct outputs in the data output lines.

Memory Expansion

Expand the memory unit to a 32 × 4 RAM using two 74189 ICs. Use the *CS* inputs to select between the two ICs. Note that since the data outputs are three-stated, you can tie pairs of terminals together to obtain a logic OR operation between the two ICs. Test your circuit by using it as a ROM simulator that adds a three-bit number to a two-bit number to produce a four-bit sum. For example, if the input of the ROM is 10110, then the output is calculated to be 101 + 10 = 0111. (The first three bits of the input represent 5, the last two bits represent 2, and the output sum is binary 7.) Use the counter to provide four bits of the address and a switch for the fifth bit of the address.

9.15 EXPERIMENT 14: LAMP HANDBALL

In this experiment, you will construct an electronic game of handball, using a single light to simulate the moving ball. The experiment demonstrates the application of a bidirectional shift register with parallel load. It also shows the operation of the asynchronous inputs of flip-flops. We will first introduce an IC that is needed for the experiment and then present the logic diagram of the simulated lamp handball game.

bidirectional shift register. The rate at which the light moves is determined by the frequency of the clock. The circuit is first initialized with the *reset* switch. The *start* switch starts the game by placing the ball (an indicator lamp) at the extreme right. The player must press the pulser push button to start the ball moving to the left. The single light shifts to the left until it reaches the leftmost position (the wall), at which time the ball returns to the player by reversing the direction of shift of the moving light. When the light is again at the rightmost position, the player must press the pulser again to reverse the direction of shift. If the player presses the pulser too soon or too late, the ball disappears and the light goes off. The game can be restarted by turning the start switch on and then off. The start switch must be open (logic 1) during the game.

Circuit Analysis

Prior to connecting the circuit, analyze the logic diagram to ensure that you understand how the circuit operates. In particular, try to answer the following questions:

1. What is the function of the reset switch?
2. How does the light in the rightmost position come on when the start switch is grounded? Why is it necessary to place the start switch in the logic-1 position before the game starts?
3. What happens to the two mode-control inputs, *S1* and *S0*, once the ball is set in motion?
4. What happens to the mode-control inputs and to the ball if the pulser is pressed while the ball is moving to the left? What happens if the ball is moving to the right, but has not yet reached the rightmost position?
5. If the ball has returned to the rightmost position, but the pulser has not yet been pressed, what is the state of the mode-control inputs if the pulser is pressed? What happens if it is not pressed?

Playing the Game

Wire the circuit of Fig. 9.20. Test the circuit for proper operation by playing the game. Note that the pulser must provide a positive-edge transition and that both the reset and start switches must be open (i.e., must be in the logic-1 state) during the game. Start with a low clock rate, and increase the clock frequency to make the handball game more challenging.

Counting the Number of Losses

Design a circuit that keeps score of the number of times the player loses while playing the game. Use a BCD-to-seven-segment decoder and a seven-segment display, as in Fig. 9.8, to display the count from 0 through 9. Counting is done with either the 7493 as a ripple decimal counter or the 74161 and a NAND gate as a synchronous decimal counter. The display should show 0 when the circuit is reset. Every time the ball disappears and the light goes off, the display should increase by 1. If the light stays on during the play, the number in the display should not change. The final design should be an

automatic scoring circuit, with the decimal display incremented automatically each time the player loses when the light disappears.

Lamp Ping-Pong™

Modify the circuit of Fig. 9.20 so as to obtain a lamp Ping-Pong game. Two players can participate in this game, with each player having his or her own pulser. The player with the right pulser returns the ball when it is in the extreme right position, and the player with the left pulser returns the ball when it is in the extreme left position. The only modification required for the Ping-Pong game is a second pulser and a change of a few wires.

With a second start circuit, the game can be made to start by either one of the two players (i.e., either one serves). This addition is optional.

9.16 EXPERIMENT 15: CLOCK-PULSE GENERATOR

In this experiment, you will use an IC timer unit and connect it to produce clock pulses at a given frequency. The circuit requires the connection of two external resistors and two external capacitors. The cathode-ray oscilloscope is used to observe the waveforms and to measure the frequency of the pulses.

IC Timer

IC type 72555 (or 555) is a precision timer circuit whose internal logic is shown in Fig. 9.21. (The resistors, R_A and R_B, and the two capacitors are not part of the IC.) The circuit consists of two voltage comparators, a flip-flop, and an internal transistor. The voltage division from $V_{CC} = 5$ V through the three internal resistors to ground produces $\frac{2}{3}$ and $\frac{1}{3}$ of V_{CC} (3.3 V and 1.7 V, respectively) into the fixed inputs of the comparators. When the threshold input at pin 6 goes above 3.3 V, the upper comparator resets the flip-flop and the output goes low to about 0 V. When the trigger input at pin 2 goes below 1.7 V, the lower comparator sets the flip-flop and the output goes high to about 5 V. When the output is low, Q' is high and the base–emitter junction of the transistor is forward biased. When the output is high, Q' is low and the transistor is cut off. (See Section 10.3.) The timer circuit is capable of producing accurate time delays controlled by an external RC circuit. In this experiment, the IC timer will be operated in the astable mode to produce clock pulses.

Circuit Operation

Figure 9.21 shows the external connections for astable operation of the circuit. Capacitor C charges through resistors R_A and R_B when the transistor is cut off and discharges through R_B when the transistor is forward biased and conducting. When the charging voltage across capacitor C reaches 3.3 V, the threshold input at pin 6 causes the flip-flop to reset and the transistor turns on. When the discharging voltage reaches 1.7 V, the trigger input at pin 2 causes the flip-flop to set and the transistor turns off. Thus, the output continually alternates

(c) Write a stimulus module (similar to HDL Example 3.3) and simulate the circuit to verify the answer in part (a).

(d) Implement the circuit with an FPGA and test its operation.

Supplement to Experiment 4 (Section 9.5)

The operation of a combinational circuit is verified by checking the output and comparing it with the truth table for the circuit. HDL Example 4.10 (Section 4.12) demonstrates the procedure for obtaining the truth table of a combinational circuit by simulating it.

(a) In order to get acquainted with this procedure, compile and simulate HDL Example 4.10 and check the output truth table.

(b) In Experiment 4, you designed a majority logic circuit. Write the HDL gate-level description of the majority logic circuit together with a stimulus for displaying the truth table. Compile and simulate the circuit and check the output response.

(c) Implement the majority logic circuit units in an FPGA and test its operation.

Supplement to Experiment 5 (Section 9.6)

This experiment deals with code conversion. A BCD-to-excess-3 converter was designed in Section 4.4. Use the results of the design to check it with an HDL simulator.

(a) Write an HDL gate-level description of the circuit shown in Fig. 4.4.

(b) Write a dataflow description using the Boolean expressions listed in Fig. 4.3.

(c) Write an HDL behavioral description of a BCD-to-excess-3 converter.

(d) Write a test bench to simulate and test the BCD-to-excess-3 converter circuit in order to verify the truth table. Check all three circuits.

(e) Implement the behavioral description with an FPGA and test the operation of the circuit.

Supplement to Experiment 7 (Section 9.8)

A four-bit adder–subtractor is developed in this experiment. An adder–subtractor circuit is also developed in Section 4.5.

(a) Write the HDL behavioral description of the 7483 four-bit adder.

(b) Write a behavioral description of the adder–subtractor circuit shown in Fig. 9.11.

(c) Write the HDL hierarchical description of the four-bit adder–subtractor shown in Fig. 4.13 (including V). This can be done by instantiating a modified version of the four-bit adder described in HDL Example 4.2 (Section 4.12).

(d) Write an HDL test bench to simulate and test the circuits of part (c). Check and verify the values that cause an overflow with $V = 1$.

(e) Implement the circuit of part (c) with an FPGA and test its operation.

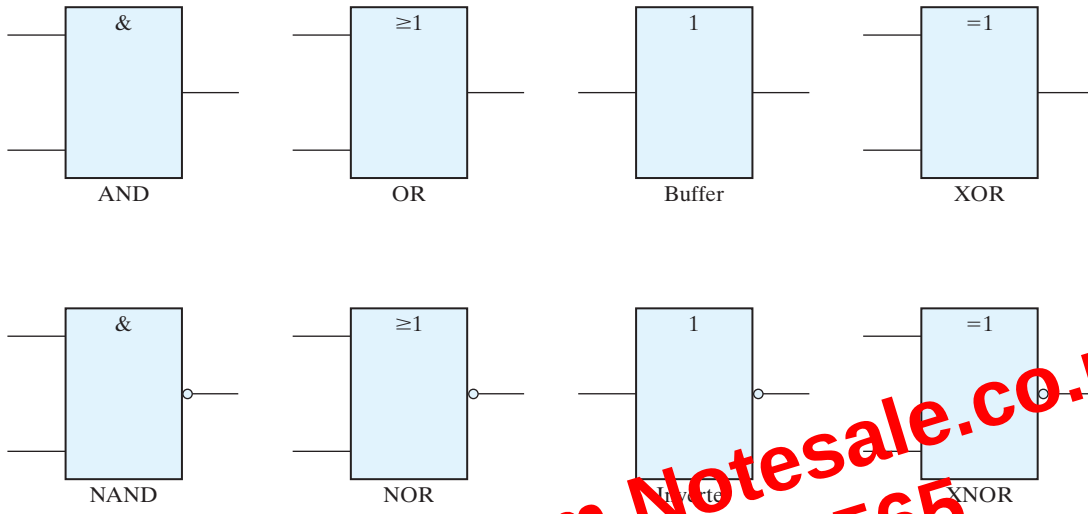


FIGURE 10.1
Rectangular-shape graphic symbols for gates

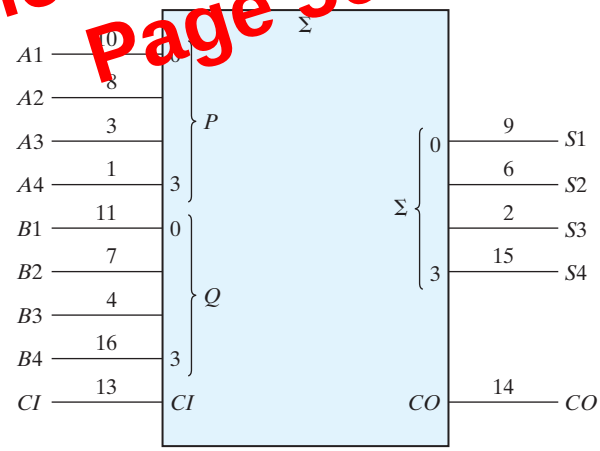


FIGURE 10.2
Standard graphic symbol for a four-bit parallel adder, IC type 7483

letters for the arithmetic operands are P and Q. The bit-grouping symbols in the two types of inputs and the sum output are the decimal equivalents of the weights of the bits to the power of 2. Thus, the input labeled 3 corresponds to the value of $2^3 = 8$. The input carry is designated by CI and the output carry by CO. When the digital component represented by the outline is also a commercial integrated circuit, it is customary to write the IC pin number along each input and output. Thus, IC type 7483 is a four-bit adder with look-ahead carry. It is enclosed in a package with 16 pins. The pin numbers

gate to enable the decoder. The output of the AND gate is labeled *EN* (enable) and is activated when E_1 is at a low-level state and E_2 at a high-level state.

10.2 QUALIFYING SYMBOLS

The IEEE standard graphic symbols for logic functions provide a list of qualifying symbols to be used in conjunction with the outline. A qualifying symbol is added to the basic outline to designate the overall logic characteristics of the element or the physical characteristics of an input or output. Table 10.1 lists some of the general qualifying symbols specified in the standard. A general qualifying symbol defines the basic function performed by the device represented in the diagram. It is placed near the top center position of the rectangular-shape outline. The general qualifying symbols for the gates, decoder, and adder were shown in previous diagrams. The other symbols are self-explanatory and will be used later in diagrams representing the corresponding digital elements.

Some of the qualifying symbols associated with inputs and outputs are shown in Fig. 10.4. Symbols associated with inputs are placed on the left side of the column labeled *symbol*. Symbols associated with outputs are placed on the right side of the column. The active-low input or output symbol is the polarity indicator. As mentioned

Table 10.1
General Qualifying Symbols

Symbol	Description
&	AND gate or function
≥ 1	OR gate or function
1	Buffer gate or inverter
=1	Exclusive-OR gate or function
2k	Even function or even parity element
2k + 1	Odd function or odd parity element
X/Y	Coder, decoder, or code converter
MUX	Multiplexer
DMUX	Demultiplexer
Σ	Adder
$\prod$	Multiplier
COMP	Magnitude comparator
ALU	Arithmetic logic unit
SRG	Shift register
CTR	Counter
RCTR	Ripple counter
ROM	Read-only memory
RAM	Random-access memory

previously, it is equivalent to the logic negation when positive logic is assumed. The dynamic input is associated with the clock input in flip-flop circuits. It indicates that the input is active on a transition from a low-to-high-level signal. The three-state output has a third high-impedance state, which has no logic significance. When the circuit is enabled, the output is in the normal 0 or 1 logic state, but when the circuit is disabled, the three-state output is in a high-impedance state. This state is equivalent to an open circuit.

The open-collector output has one state that exhibits a high-impedance condition. An externally connected resistor is sometimes required in order to produce the proper logic level. The diamond-shape symbol may have a bar on top (for high type) or on the bottom (for low type). The high or low type specifies the logic level when the output is not in the high-impedance state. For example, TTL-type integrated circuits have special outputs called open-collector outputs. These outputs are recognized by a diamond-shape symbol with a bar under it. This indicates that the output can be either in a high-impedance state or in a low-level state. When used as part of a distribution function, two or more open-collector NAND gates when connected to a common resistor perform a positive-logic AND function or a negative-logic OR function.

The output with special amplification is used in gates that provide special driving capabilities. Such gates are employed in components such as clock drivers or bus-oriented transmitters. The *EN* symbol designates an enable input. It has the effect of enabling all outputs when it is active. When the input marked with *EN* is inactive, all outputs are disabled. The symbols for flip-flop inputs have the usual meaning. The *D* input is also associated with other storage elements such as memory input.

The symbols for shift right and shift left are arrows pointing to the right or the left, respectively. The symbols for count-up and count-down counters are the plus and minus symbols, respectively. An output designated by $CT = 15$ will be active when the contents of the register reach the binary count of 15. When nonstandard information is shown inside the outline, it is enclosed in square brackets [like this].

10.3 DEPENDENCY NOTATION

The most important aspect of the standard logic symbols is the dependency notation. Dependency notation is used to provide the means of denoting the relationship between different inputs or outputs without actually showing all the elements and interconnections between them. We will first demonstrate the dependency notation with an example of the AND dependency and then define all the other symbols associated with this notation.

The AND dependency is represented with the letter *G* followed by a number. Any input or output in a diagram that is labeled with the number associated with *G* is considered to be ANDed with it. For example, if one input in the diagram has the label *G*1 and another input is labeled with the number 1, then the two inputs labeled *G*1 and 1 are considered to be ANDed together internally.

An example of AND dependency is shown in Fig. 10.5. In (a), we have a portion of a graphic symbol with two AND dependency labels, *G*1 and *G*2. There are two inputs labeled with the number 1 and one input labeled with the number 2. The equivalent

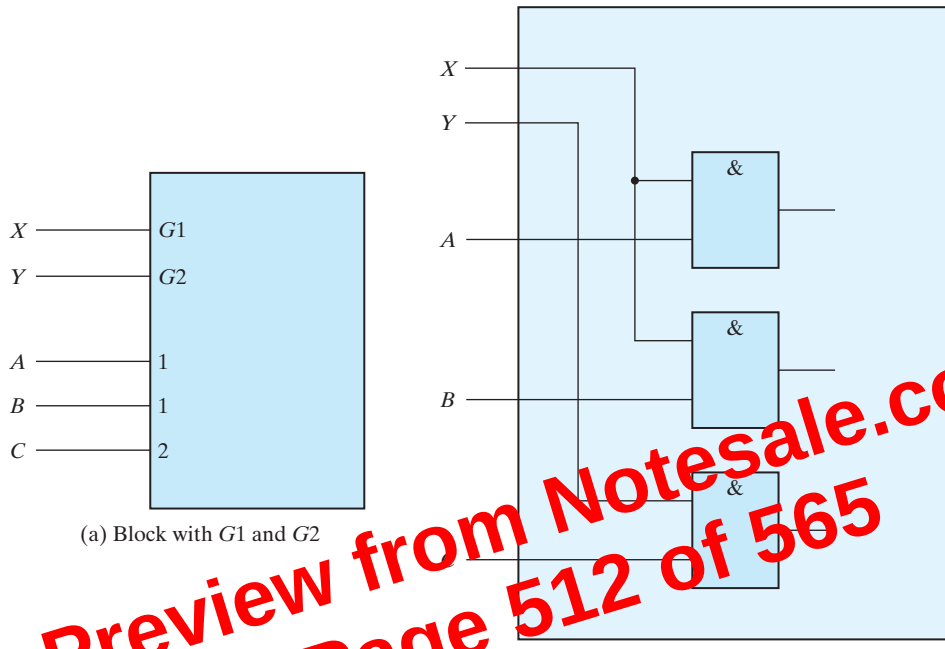


FIGURE 10.5
Example of *G* (AND) dependency

interpretation is shown in part (b) of the figure. Input *X* associated with *G*1 is considered to be ANDed with inputs *A* and *B*, which are labeled with a 1. Similarly, input *Y* is ANDed with input *C* to conform with the dependency between *G*2 and 2.

The standard defines 10 other dependencies. Each dependency is denoted by a letter symbol (except *EN*). The letter appears at the input or output and is followed by a number. Each input or output affected by that dependency is labeled with that same number. The 11 dependencies and their corresponding letter designation are as follows:

G	Denotes an AND (gate) relationship
V	Denotes an OR relationship
N	Denotes a negate (exclusive-OR) relationship
EN	Specifies an enable action
C	Identifies a control dependency
S	Specifies a setting action
R	Specifies a resetting action
M	Identifies a mode dependency
A	Identifies an address dependency

Z Indicates an internal interconnection

X Indicates a controlled transmission

The *V* and *N* dependencies are used to denote the Boolean relationships of OR and exclusive-OR similar to the *G* that denotes the Boolean AND. The *EN* dependency is similar to the qualifying symbol *EN* except that a number follows it (for example, *EN* 2). Only the outputs marked with that number are disabled when the input associated with *EN* is active.

The control dependency *C* is used to identify a clock input in a sequential element and to indicate which input is controlled by it. The set *S* and reset *R* dependencies are used to specify internal logic states of an *SR* flip-flop. The *C, S*, and *R* dependencies are explained in Section 10.5 in conjunction with the flip-flop circuit. The mode *M* dependency is used to identify inputs that select the mode of operation of the unit. The mode dependency is presented in Section 10.6 in conjunction with registers and counters. The address *A* dependency is used to identify the address input of a memory. It is introduced in Section 10.8 in conjunction with the memory unit.

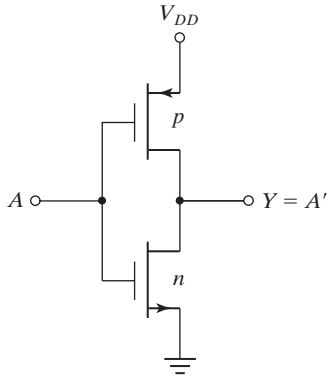
The *Z* dependency is used to indicate interconnections inside the unit. It signifies the existence of internal logic connections between inputs, outputs, internal inputs, and internal outputs, in any combination. The *X* dependency is used to indicate the controlled transmission path in a CMOS transmission gate.

10.4 SYMBOLS FOR COMBINATIONAL ELEMENTS

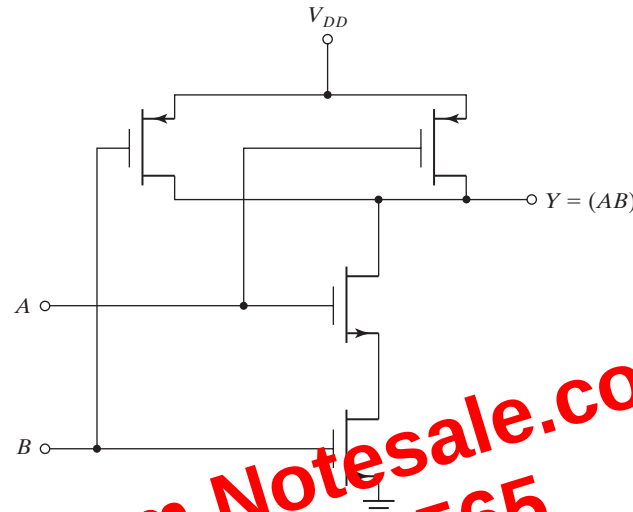
The examples in this section and the rest of this chapter illustrate the use of the standard in representing various digital components with graphic symbols. The examples demonstrate actual commercial integrated circuits with the pin numbers included in the inputs and outputs. Most of the ICs presented in this chapter are included with the suggested experiments outlined in Chapter 9.

The graphic symbols for the adder and decoder were shown in Section 10.2. IC type 74155 can be connected as a 3×8 decoder, as shown in Fig. 10.6. (The truth table of this decoder is shown in Fig. 9.7.) There are two *C* and two *G* inputs in the IC. Each pair must be connected together as shown in the diagram. The enable input is active when in the low-level state. The outputs are all active low. The inputs are assigned binary weights 1, 2, and 4, equivalent to 2^0, 2^1, and 2^2, respectively. The outputs are assigned numbers from 0 to 7. The sum of the weights of the inputs determines the output that is active. Thus, if the two input lines with weights 1 and 4 are activated, the total weight is $1 + 4 = 5$ and output 5 is activated. Of course, the *EN* input must be activated for any output to be active.

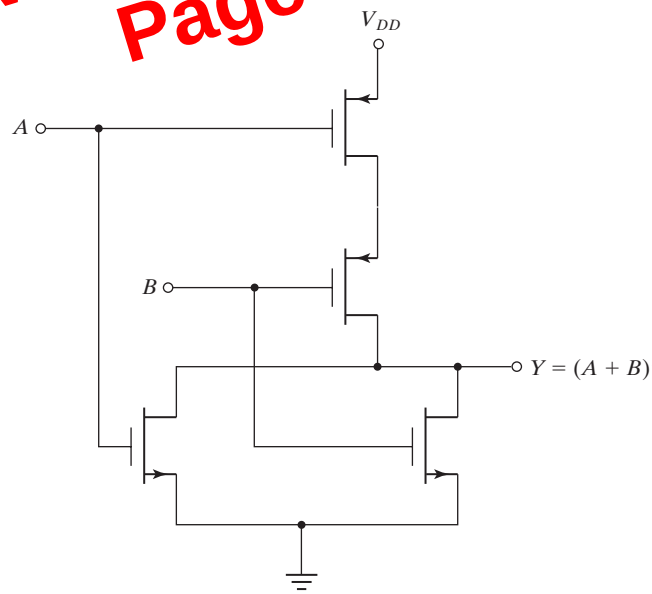
The decoder is a special case of a more general component referred to as a *coder*. A coder is a device that receives an input binary code on a number of inputs and produces a different binary code on a number of outputs. Instead of using the qualifying symbol *X/Y*, the coder can be specified by the code name. For example, the 3-to-8-line decoder of Fig. 10.6 can be symbolized with the name *BIN/OCT* since the circuit converts a 3-bit binary number into 8 octal values, 0 through 7.



(a) Inverter



(b) NAND gate



(c) NOR gate

FIGURE A.4
CMOS logic circuits

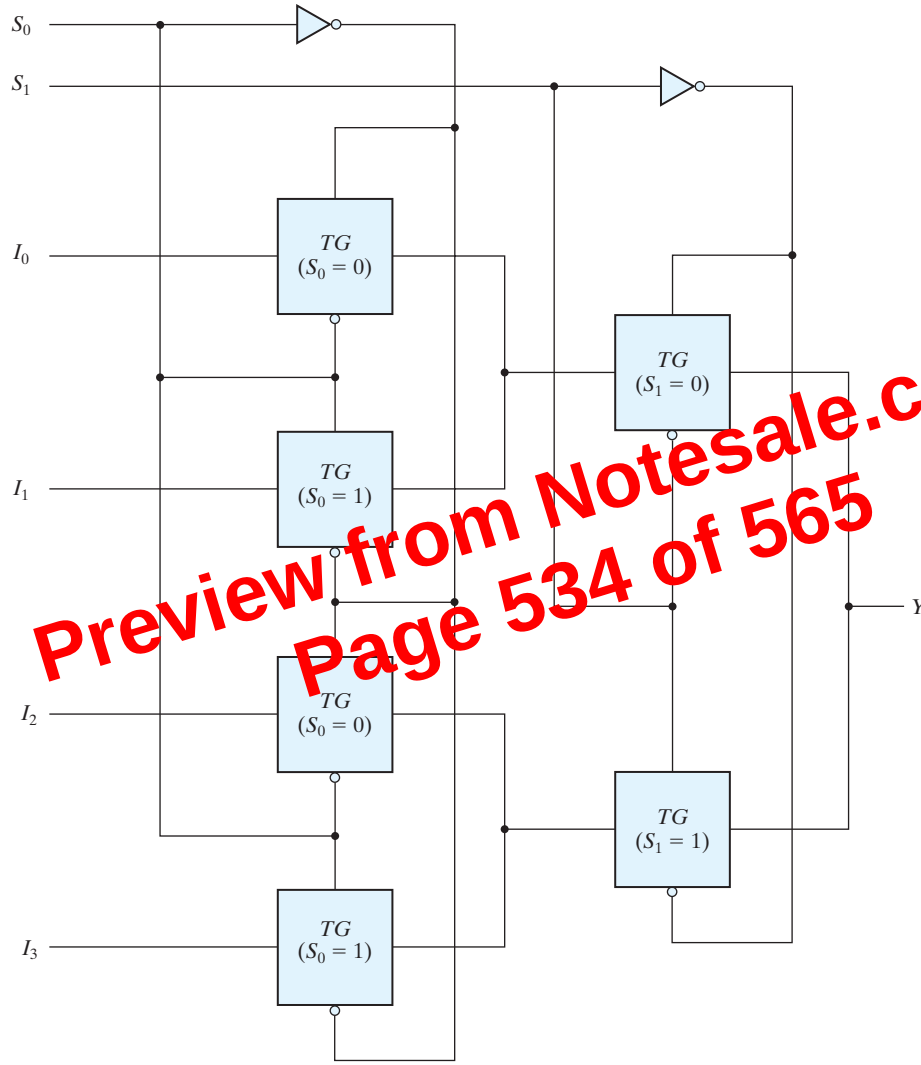


FIGURE A.9
Multiplexer with transmission gates

output lines when the two vertical control inputs have the value of 1 in the uncircled terminal and 0 in the circled terminal. With an opposite polarity in the control inputs, the path disconnects and the circuit behaves like an open switch. The two selection inputs, S_1 and S_0, control the transmission path in the *TG* circuits. Inside each box is marked the condition for the transmission gate switch to be closed. Thus, if $S_0 = 0$ and $S_1 = 0$, there is a closed path from input I_0 to output Y through the two *TG*s marked with $S_0 = 0$ and $S_1 = 0$. The other three inputs are disconnected from the output by one of the other *TG* circuits.

Answers to Selected Problems

CHAPTER 1

1.2 **(a)** 32,768 **(b)** 67,108,864 **(c)** 6,871,947,674

1.3 **(a)** $(4310)_5 = 580$ **(b)** $(198)_{12} = 260$

1.5 **(a)** 6 **(b)** 8 **(c)** 11

1.6 8

1.7 $(62315)_8$

1.9 22.3125 (all three)

1.12 **(a)** 10000 and 110111 **(b)** 62 and 958

1.19 **(a)** 010087 **(b)** 008485 **(c)** 991515 **(d)** 989913

1.24 **(a)**

6	**3**	**1**	**1**	**Decimal**
0	0	0	0	0
0	0	0	1	1
0	0	1	1	2
0	1	0	0	3
0	1	1	0	4 (*or* 0101)
0	1	1	1	5
1	0	0	0	6
1	0	1	0	7 (*or* 1001)
1	0	1	1	8
1	1	0	0	9

5.16 $D_A = Ax' + Bx$

$D_B = A'x + Bx'$

5.18 $J_A = K_A = (BF + B'F')E$

$J_B = K_B = E$

5.19 **(a)** $D_A = A'B'x\_in$

$D_B = A + C'x\_in' + BCx\_in$

$D_C = Cx\_in' + Ax\_in + A'B'x\_in'$

$y\_out = A'x\_in$

5.23 **(a)** $RegA = 125, \quad RegB = 125$

(b) $RegA = 125, \quad RegB = 30$

5.26 **(a)**

$Q(t + 1) = JQ' + K'Q$

When $Q = 0, Q(t + 1) = J$

When $Q = 1, Q(t + 1) = K'$

```
module JK_Behavior (output reg Q, input J, K, CLK);
  always @ (posedge CLK)
      if (Q == 0)                Q <= J;
      else                       Q <= ~K;
endmodule
```

5.31 The HDL description is available on the Companion Website.
Note: The statements must be written in an order that produces the effect of concurrent assignments.

CHAPTER 6

6.4 0110; 0011; 0001; 1000; 1100; 1110; 0111; 1011

6.8 $A = 0010, 0001, 1000, 1100$. Carry $= 1, 1, 1, 0$

6.9 **(b)** $J_Q = x'y; K_Q = (x' + y)'$

6.14 **(a)** 4

6.15 30 ns; 33.3 MHz

6.16 $1010 \rightarrow 1011 \rightarrow 0100$

$1100 \rightarrow 1101 \rightarrow 0100$

$1110 \rightarrow 1111 \rightarrow 0000$

6.17 $D_{A0} = A_0 \oplus E$

$D_{A1} = A_1 \oplus (A_0E)$

$D_{A2} = A_2 \oplus (A_1A_0E)$

$D_{A3} = A_3 \oplus (A_2A_1A_0E)$

6.19 **(b)** $D_{Q1} = Q_1'$

$D_{Q2} = Q_2Q_1' + Q_8'Q_2'Q_1$

$D_{Q4} = Q_4Q_1' + Q_4Q_2' + Q_4'Q_2'Q_1$

$D_{Q8} = Q_8Q_1' + Q_4Q_2Q_1$

6.21 $J_{A0} = LI_0 + L'C$

$K_{A0} = LI_0' + L'C$

6.24 $T_A = A \oplus B$

$T_B = B \oplus C$

$T_C = AC + A'C'$ (not self-starting)

$= AC + A'B'C$ (self-starting)

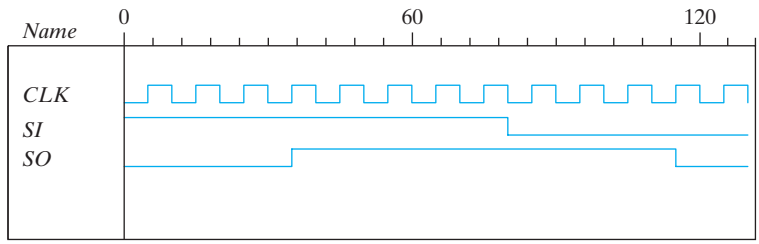
6.26 The clock generator has a period of 12.5 ns. Use a 2-bit counter to count four pulses.

6.28 $D_A = A \oplus B$

$D_B = AB' + C$

$D_C = A'B'C'$

6.34 The HDL description is available on the Companion Website. Simulations results for Problem 6.34 follow:

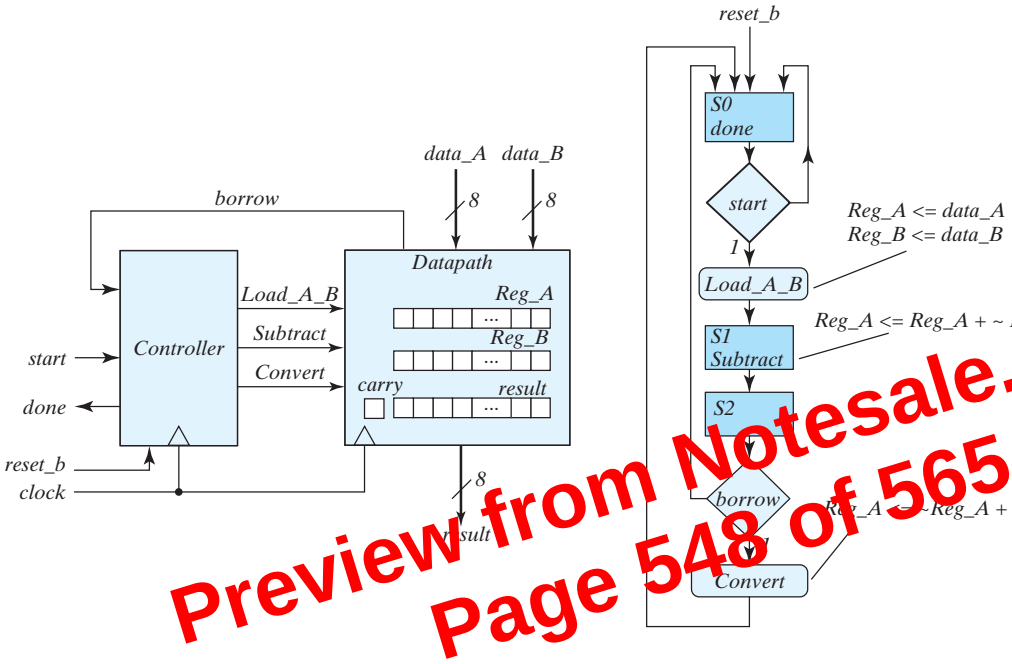


6.35 **(b)** The HDL description is available on the Companion Website.

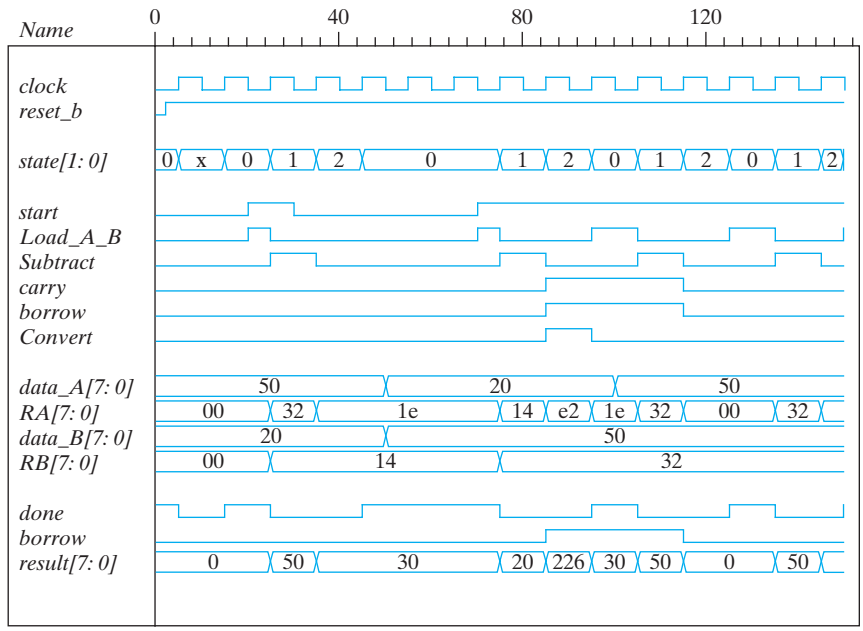
6.37 The HDL description is available on the Companion Website.

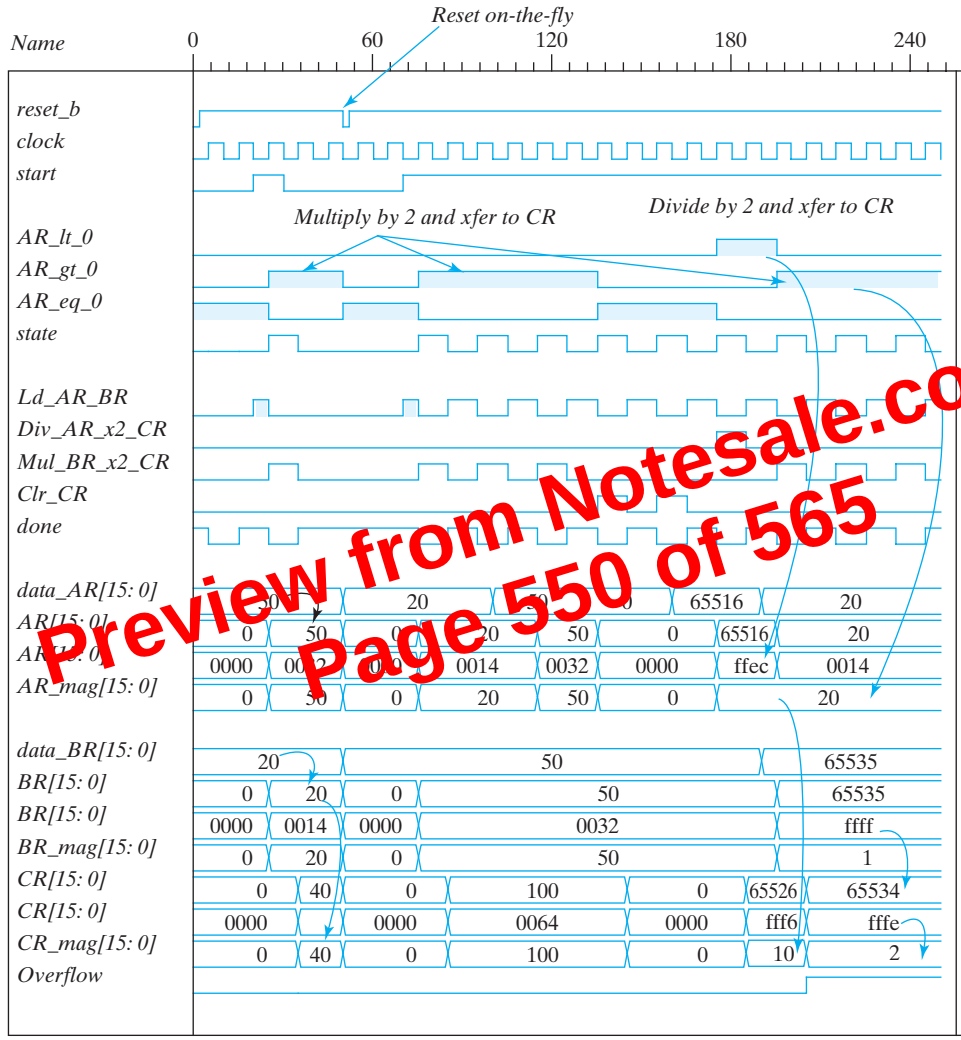
6.38 **(a)** The HDL description is available on the Companion Website.

Block diagram and ASMD chart:



The HDL description is available on the Companion Website. Simulations results for Problems 8.7 follow:





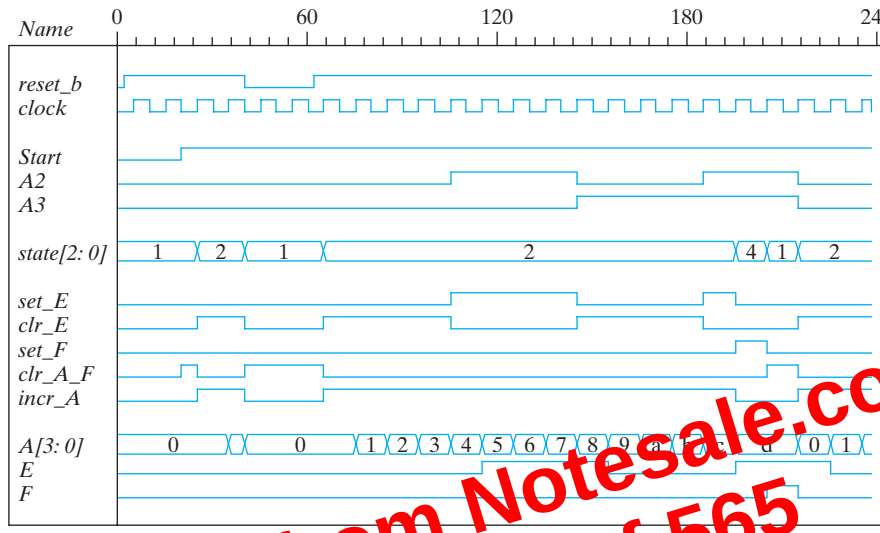
8.9 Design equations:

$$D_{S\_idle} = S\_2 + S\_idle\ Start'$$

$$D_{S\_1} = S\_idle\ Start + S\_1(A2\ A3)'$$

$$D_{S\_2} = A2\ A3\ S\_1$$

The HDL description is available on the Companion Website. Simulations results for Problem 8.9 follow:



8.11 $D_A = AB + Ax$

$D_B = A'B'x + AB'y + xy$

8.16 RTL notation:

s0: (initial state) If *start* = 0 go back to state *s0*, If(*start* = 1) then $BR \leftarrow multiplicand$, $AR \leftarrow multiplier$, $PR \leftarrow 0$, go to *s*1.

s1: (check *AR* for Zero) *Zero* = 1 if *AR* = 0, if (*Zero* = 1) then go back to *s0* (*done*) If (*Zero* = 0) then go to *s1*, $PR \leftarrow PR + BR$, $AR \leftarrow AR - 1$.

The internal architecture of the datapath consists of a double-width register to hold the product (PR), a register to hold the multiplier (AR), a register to hold the multiplicand (BR), a double-width parallel adder, and single-width parallel adder. The single-width adder is used to implement the operation of decrementing the multiplier unit. Adding a word consisting entirely of 1s to the multiplier accomplishes the 2's complelment subtraction of 1 from the multiplier. Figure 8.16 (a) below shows the ASMD chart, block diagram, and controller of othe circuit. Figure 8.16 (b) shows the internal architecture of the datapath. Figure 8.16 (c) shows the results of simulating the circuit.