Digital Design

FOURTH EDITION

M. Morris Mano
Emeritus Professor of Computer Engineering
California State University, Los Angeles

Michael D. Ciletti
Department of Electrical and Computer Engineering
University of Colorado at Colorado Springs

PEARSON
Prentice
Hall

Upper Saddle River, NJ 07458

Contents

Preface ix

1 Digital Systems and Binary Numbers 1

1.1 Digital Systems 1
1.2 Binary Numbers 3
1.3 Number-Base Conversions 5
1.4 Octal and Hexadecimal Numbers 8
1.5 Complements 9
1.6 Signed Binary Numbers 14
1.7 Binary Codes 17
1.8 Binary Storage and Registers 25
1.9 Binary Logic 28

2 Boolean Algebra and Logic Gates 36

2.1 Introduction 36
2.2 Basic Definitions 36
2.3 Axiomatic Definition of Boolean Algebra 38
2.4 Basic Theorems and Properties of Boolean Algebra 41
2.5 Boolean Functions 44
2.6 Canonical and Standard Forms 48
2.7 Other Logic Operations 55
2.8 Digital Logic Gates 57
2.9 Integrated Circuits 63

3 Gate-Level Minimization 70

3.1 Introduction 70
3.2 The Map Method 70
3.3 Four-Variable Map 76
3.4 Five-Variable Map 81
3.5 Product-of-Sums Simplification 83
3.6 Don't-Care Conditions 86
3.7 NAND and NOR Implementation 89
3.8 Other Two-Level Implementations 96
3.9 Exclusive-OR Function 101
3.10 Hardware Description Language 106

4 Combinational Logic 122

4.1 Introduction 122
4.2 Combinational Circuits 122
4.3 Analysis Procedure 123
4.4 Design Procedure 126
4.5 Binary Adder-Subtractor 130
4.6 Decimal Adder 139
4.7 Binary Multiplier 142
4.8 Magnitude Comparator 144
4.9 Decoders 146
4.10 Encoders 150
4.11 Multiplexers 152
4.12 HDL Models of Combinational Circuits 159

5 Synchronous Sequential Logic 182

5.1 Introduction 182
5.2 Sequential Circuits 182
5.3 Storage Elements: Latches 184
5.4 Storage Elements: Flip-Flops 188
5.5 Analysis of Clocked Sequential Circuits 195
5.6 Synthesizable HDL Models of Sequential Circuits 207
5.7 State Reduction and Assignment 221
5.8 Design Procedure 225

6 Registers and Counters 242

6.1 Registers 242
6.2 Shift Registers 245

6.3 Ripple Counters 253
6.4 Synchronous Counters 258
6.5 Other Counters 265
6.6 HDL for Registers and Counters 269

7 Memory and Programmable Logic 284

7.1 Introduction 284
7.2 Random-Access Memory 285
7.3 Memory Decoding 291
7.4 Error Detection and Correction 296
7.5 Read-Only Memory 299
7.6 Programmable Logic Array 305
7.7 Programmable Array Logic 309
7.8 Sequential Programmable Devices 311

8 Design at the Register Transfer Level 334

8.1 Introduction 334
8.2 Register Transfer Level (RTL) Notation 334
8.3 Register Transfer Level in HDL 336
8.4 Algorithmic State Machines (ASMs) 345
8.5 Design Example 352
8.6 HDL Description of Design Example 361
8.7 Sequential Binary Multiplier 371
8.8 Control Logic 376
8.9 HDL Description of Binary Multiplier 382
8.10 Design with Multiplexers 390
8.11 Race-Free Design 401
8.12 Latch-Free Design 403
8.13 Other Language Features 404

9 Asynchronous Sequential Logic 415

9.1 Introduction 415
9.2 Analysis Procedure 417
9.3 Circuits with Latches 425
9.4 Design Procedure 433
9.5 Reduction of State and Flow Tables 439
9.6 Race-Free State Assignment 446
9.7 Hazards 452
9.8 Design Example 457

10 Digital Integrated Circuits 471

10.1 Introduction 471
10.2 Special Characteristics 473
10.3 Bipolar-Transistor Characteristics 477
10.4 RTL and DTL Circuits 481
10.5 Transistor-Transistor Logic 484
10.6 Emitter-Coupled Logic 493
10.7 Metal-Oxide Semiconductor 495
10.8 Complementary MOS 498
10.9 CMOS Transmission Gate Circuits 501
10.10 Switch-Level Modeling with HDL 505

11 Laboratory Experiments with Standard ICs and FPGAs 511

11.1 Introduction to Experiments 511
11.2 Experiment 1: Binary and Decimal Numbers 516
11.3 Experiment 2: Digital Logic Gates 519
11.4 Experiment 3: Simplification of Boolean Functions 520
11.5 Experiment 4: Combinational Circuits 522
11.6 Experiment 5: Code Converters 524
11.7 Experiment 6: Design with Multiplexers 526
11.8 Experiment 7: Adders and Subtractors 527
11.9 Experiment 8: Flip-Flops 530
11.10 Experiment 9: Sequential Circuits 532
11.11 Experiment 10: Counters 534
11.12 Experiment 11: Shift Registers 535
11.13 Experiment 12: Serial Addition 538
11.14 Experiment 13: Memory Unit 539
11.15 Experiment 14: Lamp Handball 541
11.16 Experiment 15: Clock-Pulse Generator 545
11.17 Experiment 16: Parallel Adder and Accumulator 547
11.18 Experiment 17: Binary Multiplier 549
11.19 Experiment 18: Asynchronous Sequential Circuits 553
11.20 Verilog HDL Simulation Experiments and Rapid Prototyping with FPGAs 553

12 Standard Graphic Symbols 559

12.1 Rectangular-Shape Symbols 559
12.2 Qualifying Symbols 562
12.3 Dependency Notation 564

12.4 Symbols for Combinational Elements *566*
12.5 Symbols for Flip-Flops *568*
12.6 Symbols for Registers *570*
12.7 Symbols for Counters *573*
12.8 Symbol for RAM *575*

Answers to Selected Problems *577*

Index *597*

Preface

Digital electronic circuits are the engines of cell phones, MPEG players, digital cameras, computers, data servers, personal digital devices, GPS displays, and many other consumer products that process and use information in a digital format. This book presents a basic treatment of digital circuits and the fundamental concepts used in their design. It is suitable for use as a textbook in an introductory course in an electrical engineering, computer engineering, or computer science curriculum.

Each significant advance in industry practice ultimately works its way into the engineering curriculum. Since the mid-1980's, the use of computer-based design tools has transformed the electronics industry worldwide. Application specific integrated circuits (ASICs) are designed today by using a hardware description language (HDL), such as Verilog or VHDL, to write a behavioral model of the circuit's functionality, and then synthesizing that description into a hardware realization in a particular technology, e.g., CMOS integrated circuits or field-programmable gate arrays (FPGAs). No longer a novelty, these design tools are now readily available to universities, and are migrating in a strategic way from graduate level curricula into undergraduate courses. It is clear that HDLs have an essential, significant role in educating our future engineers. Learning to design with an HDL is as important to today's students, we think, as oscilloscopes, breadboards, and logic analyzers were to previous generations of engineers, so this edition of the text adds more weight to the use of hardware description languages in designing digital circuits.

We note that introducing HDLs in a first course in designing digital circuits is not intended to replace fundamental understanding of the building blocks of such circuits or to eliminate a discussion of manual methods of design. It is still essential for a student to understand *how hardware works*. Thus, we retain a thorough treatment of combinational and sequential logic devices. Manual design practices are presented, and their results are compared with those obtained with a HDL-based paradigm. What we are presenting, however, is a shift in emphasis on *how hardware is designed*, a shift that, we think, better prepares a student for a career in today's industry, where HDL-based design practices are prevalent.

FLEXIBILITY

The sequence of topics in the text can accommodate courses that adhere to traditional, manual-based, treatments of digital design, courses that treat design using an HDL, and courses that are in transition between or blend the two approaches. Because modern synthesis tools automatically perform logic minimization, Karnaugh maps and related topics in optimization can be presented at the beginning of a treatment of digital design, or they can be presented after circuits and their applications are examined, designed, and simulated with an HDL. The text includes both manual and HDL-based design examples. Our end-of-chapter problems further facilitate this flexibility by cross-referencing problems that address a traditional manual design task with a companion problem that uses an HDL to accomplish the task. Additionally, we link manual and HDL-based approaches by presenting annotated results of simulations in the text, in answers to selected problems at the end of the text, and in the solutions manual.

WHAT'S NEW?

The previous edition of this text recognized the importance of hardware description languages in the design of digital circuits, and incorporated new material and examples introducing students to the Verilog language, as defined by IEEE Standard 1364-1995. This revision updates and expands that treatment by:

- revising HDL-based examples to present the ANSI-C like syntax that was adopted in the standards IEEE 1364-2001 and IEEE 1364-2005
- ensuring that all HDL examples conform to industry-accepted practices for modelling digital circuits
- providing a systematic methodology for designing a datapath controller
- presenting selected exercises and solutions to end-of-chapter problems in Verilog 1995 and Verilog 2001/2005 syntax
- introducing an important design tool – the algorithmic state machine and datapath (ASMD) chart
- revising the end-of-chapter problems and expanding the set of problems by including over 75 additional problems
- providing students with fully developed answers to selected problems, including simulation results
- providing students with a CD-ROM containing simulator-ready HDL solutions of answers to selected problems
- expanding the treatment of programmable logic devices to include FPGAs

- revising the solutions manual and web-based materials and ensuring that solutions of HDL-based exercises conform to industry practices for modelling with an HDL
- discussing and demonstrating the importance of test plans for verifying HDL models of circuits
- providing instructors with verified, simulator-ready source code and test benches for all end-of chapter problems
- making all figures, tables, and HDL examples available to instructors for downloading in PDF format from the publisher
- including with the book a CD-ROM with tutorials and simulators for the IEEE-1995 and IEEE-2001 Standards of the Verilog language

In addition to the above enhancements, the text incorporates more graphical material to better serve learners who are oriented toward a graphical medium. Annotated graphical results and explanations of simulations are presented to help students understand digital circuits and to facilitate classroom discussions of them. Karnaugh maps are presented with additional graphics.

DESIGN METHODOLOGY

This edition of the text extends the previous edition's treatment of synchronous finite state machines by presenting a systematic methodology for designing a state machine to control the datapath of a digital system. Moreover, the framework in which this material is presented treats the realistic situation in which the controller uses signals from the datapath, i.e., the system has feedback. The methodology is applicable to manual and HDL-based approaches to design.

HDL-BASED APPROACH

It is not sufficient for an introduction to HDLs to dwell on language syntax. We present only those elements of the Verilog language that are matched to the level and scope of this text. Also, correct syntax does not guarantee that a model meets a functional specification or that it can be synthesized into physical hardware. We introduce students to a disciplined use of industry-based practices for writing models to ensure that a behavioral description can be synthesized into physical hardware, and that the behavior of the synthesized circuit will match that of the behavioral description. Failure to follow this discipline can lead to software race conditions in the HDL models of such machines, race conditions in the testbench used to verify them, and a mismatch between the results of simulating a behavioral model and its synthesized physical counterpart. Similarly, failure to abide by industry practices may lead to designs that simulate correctly, but which have hardware latches that are introduced into the design accidentally as a consequence of the modelling style used by the designer. The industry-based methodology we present leads to race-free and latch-free designs. It is important that students learn and follow industry practices in using HDL models, independent of whether a student's curriculum has access to synthesis tools.

VERIFICATION

In industry, significant effort is expended to verify that the functionality of a circuit is correct. Yet not much attention is given to verification in introductory texts on digital design, where the focus is on design itself, and testing is perhaps viewed as a secondary undertaking. Our experience is that this view can lead to premature declarations that "the circuit works beautifully." Likewise, industry gains repeated returns on its investment in an HDL model by ensuring that it is readable, portable and reusable. We demonstrate naming practices and the use of parameters. We also provide test benches for all of the solutions and exercises to (1) verify the functionality of the circuit, (2) underscore the importance of thorough testing, and (3) introduce students to important concepts, such as self-checking test benches. Advocating and illustrating the development of a *test plan* to guide the development of a test bench, we introduce them in the text and expand them in the solutions manual and in the answers to selected problems at the end of the text.

HDL CONTENT

This edition of the text updates and expands its treatment of the Verilog Hardware Description Language (HDL) and exploits key enhancements available in IEEE Standards 1364-2001 and 1364-2005. We have ensured that all examples in the text and all answers in the solution manual conform to accepted industry practices for modeling digital hardware. As in the previous edition, HDL material is inserted in separate sections so it can be covered or skipped as desired, does not diminish treatment of manual-based design, and does not dictate the sequence of presentation. The treatment is at a level suitable for beginning students that are learning digital circuits and a hardware description language at the same time. The text prepares students to work on significant independent design projects and to suceed in a later course in computer architecture.

- Digital circuits are introduced in Chapters 1 through 3 with an introduction to Verilog HDL in Section 3.10.
- Further discussion of modeling with HDLs occurs in Section 4.12 following the study of combinational circuits.
- Sequential circuits are covered in Chapters 5 and 6 with corresponding HDL examples in Sections 5.6 and 6.6.
- The HDL description of memory is presented in Section 7.2.
- The RTL symbols used in Verilog are introduced in Sections 8.3.
- Examples of RTL and structural models in Verilog are provided in Sections 8.6 and 8.9. Chapter 8 also presents a new, comprehensive treatment of HDL-based design of a datapath controller.
- Section 10.10 covers switch-level modeling corresponding to CMOS circuits.
- Section 11.20 supplements the hardware experiments of Chapter 11 with HDL experiments. Now the circuits designed in the laboratory can be checked by modeling them in Verilog and simulating their behavior. Then they can be synthesized and implemented with an FPGA on a prototyping board.

HDL SIMULATORS

The CD-ROM in the back of the book contains the Verilog HDL source code files for the examples in the book and 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.

INSTRUCTOR RESOURCES

Instructors can download the following classroom-ready resources from the publisher (www.prenhall.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 solution manual in typed hardcopy format with graphics, suitable for classroom presentation, will also be provided instructors.

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 BCD.

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 from that, the most useful logic gates used in the design of digital systems are determined. The characteristics of integrated circuit gates are mentioned in this chapter but a more detailed analysis of there the electronic circuits of the gates is done in Chapter 10.

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 gate-level modeling examples.

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 the gate-level, dataflow, and behavioral modeling 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 the analysis and design of 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 circuits 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 counter 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 are presented such as ROMs, PLAs, PALs, CPLDs, and FPGAs.

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 presents formal procedures for the analysis and design of asynchronous sequential circuits. Methods are outlined to show how an asynchronous sequential circuit can be implemented as a combinational circuit with feedback. An alternate implementation is also described that uses SR latches as the storage elements in asynchronous sequential circuits.

Chapter 10 presents the most common integrated circuit digital logic families. The electronic circuits of the common gate in each family are analyzed using electrical circuit theory. A basic knowledge of electronic circuits is necessary to fully understand the material in this chapter. Examples of Verilog switch-level descriptions demonstrate the ability to simulate circuits constructed with MOS and CMOS transistors.

Chapter 11 outlines experiments that can be performed in the laboratory with hardware that is readily available commercially. The operation of the integrated circuits 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 produce the circuit diagram and formulate a procedure for checking the operation of the circuit in the laboratory. The last section supplements the experiments with corresponding HDL experiments. Instead of, or in addition to, the hardware construction, the student can use the Verilog HDL software provided on the CD-ROM to simulate and verify the design.

Chapter 12 presents the standard graphic symbols for logic functions recommended by an ANSI/IEEE Standard. These graphic symbols have been developed for SSI and MSI components so that the user can recognize each function from the unique graphic symbol assigned. The chapter shows the standard graphic symbols of the integrated circuits used in the laboratory experiments. The various digital components that are represented throughout the book are similar to commercial integrated circuits. However, the text does not mention specific integrated circuits except in Chapters 11 and 12. Doing the suggested experiments in Chapter 11 while studying the theory presented in the text will enhance the practical application of digital design.

LAB EXPERIMENTS

The book may be used in a stand-alone course or with a companion lab based on the lab experiments included with the text. 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. Circuit boards for rapidly prototyping circuits with FPGAs are available at nominal cost, and typically include push buttons, switches, and seven-segment displays, LCDs, keypads and other I/O devices. With these resources, students can work prescribed lab exercises or their own projects and get results immediately.

The operation of the integrated circuits 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 produce the circuit diagram and formulate a procedure for verifying the operation of the circuit in the laboratory. The last section supplements the experiments with corresponding HDL experiments. Instead of, or in addition to, the hardware construction, the student can use the Verilog HDL software provided on the CD-ROM to simulate and check the design. Synthesis tools can then be used to implement the circuit in an FPGA on a prototyping board.

Our thanks go to the editorial team at Prentice Hall for committing to this timely revision of the text. Finally, we are grateful to our wives, Sandra and Jerilynn, for encouraging our pursuit of this project.

M. Morris Mano
Emeritus Professor of Computer Engineering
California State University, Los Angeles

Michael D. Ciletti
Department of Electrical and Computer Engineering
University of Colorado at Colorado Springs

Chapter 1

Digital Systems and Binary Numbers

1.1 DIGITAL SYSTEMS

Digital systems have such a prominent role in everyday life that we refer to the present technological period as the *digital age*. Digital systems are used in communication, business transactions, traffic control, space guidance, medical treatment, weather monitoring, the Internet, and many other commercial, industrial, and scientific enterprises. We have digital telephones, digital television, digital versatile discs, digital cameras, handheld devices, and, of course, digital computers. The most striking property of the digital computer is its generality. It can follow a sequence of instructions, called a program, that operates on given data. The user can specify and change the program or the data according to the specific need. Because of this flexibility, general-purpose digital computers can perform a variety of information-processing tasks that range over a wide spectrum of applications.

One characteristic of digital systems is their ability to represent and manipulate discrete elements of information. Any set that is restricted to a finite number of elements contains discrete information. Examples of discrete sets are the 10 decimal digits, the 26 letters of the alphabet, the 52 playing cards, and the 64 squares of a chessboard. Early digital computers were used for numeric computations. In this case, the discrete elements were the digits. From this application, the term *digital* computer emerged. Discrete elements of information are represented in a digital system by physical quantities called signals. Electrical signals such as voltages and currents are the most common. Electronic devices called transistors predominate in the circuitry that implements these signals. The signals in most present-day electronic digital systems use just two discrete values and are therefore said to be *binary*. A binary digit, called a *bit,* has two values: 0 and 1. Discrete elements of information are represented with groups of bits called *binary codes*. For example, the decimal digits 0 through 9 are represented in a digital system with a code of four bits (e.g., the number 7 is represented by 0111).

Through various techniques, groups of bits can be made to represent discrete symbols, which are then used to develop the system in a digital format. Thus, a digital system is a system that manipulates discrete elements of information represented internally in binary form.

Discrete quantities of information either emerge from the nature of the data being processed or may be quantized from a continuous process. On the one hand, a payroll schedule is an inherently discrete process that contains employee names, social security numbers, weekly salaries, income taxes, and so on. An employee's paycheck is processed by means of discrete data values such as letters of the alphabet (names), digits (salary), and special symbols (such as $). On the other hand, a research scientist may observe a continuous process, but record only specific quantities in tabular form. The scientist is thus quantizing continuous data, making each number in his or her table a discrete quantity. In many cases, the quantization of a process can be performed automatically by an analog-to-digital converter.

The general-purpose digital computer is the best-known example of a digital system. The major parts of a computer are a memory unit, a central processing unit, and input–output units. The memory unit stores programs as well as input, output, and intermediate data. The central processing unit performs arithmetic and other data-processing operations as specified by the program. The program and data prepared by a user are transferred into memory by means of an input device such as a keyboard. An output device, such as a printer, receives the results of the computations, and the printed results are presented to the user. A digital computer can accommodate many input and output devices. One very useful device is a communication unit that provides interaction with other users through the Internet. A digital computer is a powerful instrument that can perform not only arithmetic computations, but also logical operations. In addition, it can be programmed to make decisions based on internal and external conditions.

There are fundamental reasons that commercial products are made with digital circuits. Like a digital computer, most digital devices are programmable. By changing the program in a programmable device, the same underlying hardware can be used for many different applications. Dramatic cost reductions in digital devices have come about because of advances in digital integrated circuit technology. As the number of transistors that can be put on a piece of silicon increases to produce complex functions, the cost per unit decreases and digital devices can be bought at an increasingly reduced price. Equipment built with digital integrated circuits can perform at a speed of hundreds of millions of operations per second. Digital systems can be made to operate with extreme reliability by using error-correcting codes. An example of this strategy is the digital versatile disk (DVD), in which digital information representing video, audio, and other data is recorded without the loss of a single item. Digital information on a DVD is recorded in such a way that, by examining the code in each digital sample before it is played back, any error can be automatically identified and corrected.

A digital system is an interconnection of digital modules. To understand the operation of each digital module, it is necessary to have a basic knowledge of digital circuits and their logical function. The first seven chapters of this book present the basic tools of digital design, such as logic gate structures, combinational and sequential circuits, and programmable logic devices. Chapter 8 introduces digital design at the register transfer level (RTL). Chapters 9 and 10 deal with asynchronous sequential circuits and the various integrated digital logic families. Chapters 11 and 12 introduce commercial integrated circuits and show how they can be connected in the laboratory to perform experiments with digital circuits.

A major trend in digital design methodology is the use of a hardware description language (HDL) to describe and simulate the functionality of a digital circuit. An HDL resembles a programming language and is suitable for describing digital circuits in textual form. It is used to simulate a digital system to verify its operation before hardware is built in. It is also used in conjunction with logic synthesis tools to automate the design process. Because it is important that students become familiar with an HDL-based design methodology, HDL descriptions of digital circuits are presented throughout the book. While these examples help illustrate the features of an HDL, they also demonstrate the best practices used by industry to exploit HDLs. Ignorance of these practices will lead to cute, but worthless, HDL models that may simulate a phenomenon, but that cannot be synthesized by design tools, or to models that waste silicon area or synthesize to hardware that cannot operate correctly.

As previously stated, digital systems manipulate discrete quantities of information that are represented in binary form. Operands used for calculations may be expressed in the binary number system. Other discrete elements, including the decimal digits, are represented in binary codes. Digital circuits, also referred to as logic circuits, process data by means of binary logic elements (logic gates) using binary signals. Quantities are stored in binary (two-valued) storage elements (flip-flops). The purpose of this chapter is to introduce the various binary concepts as a frame of reference for further study in the succeeding chapters.

1.2 BINARY NUMBERS

A decimal number such as 7,392 represents a quantity equal to 7 thousands, plus 3 hundreds, plus 9 tens, plus 2 units. The thousands, hundreds, etc., are powers of 10 implied by the position of the coefficients in the number. To be more exact, 7,392 is a shorthand notation for what should be written as

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

However, the convention is to write only the coefficients and, from their position, deduce the necessary powers of 10. In general, a number with a decimal point is represented by a series of coefficients:

$$a_5a_4a_3a_2a_1a_0 \cdot a_{-1}a_{-2}a_{-3}$$

The coefficients a_j are any of the 10 digits $(0, 1, 2, \ldots, 9)$, and the subscript value j gives the place value and, hence, the power of 10 by which the coefficient must be multiplied. Thus, the preceding decimal number can be expressed as

$$10^5a_5 + 10^4a_4 + 10^3a_3 + 10^2a_2 + 10^1a_1 + 10^0a_0 + 10^{-1}a_{-1} + 10^{-2}a_{-2} + 10^{-3}a_{-3}$$

The decimal number system is said to be of *base*, or *radix*, 10 because it uses 10 digits and the coefficients are multiplied by powers of 10. The *binary* system is a different number system. The coefficients of the binary number system have only two possible values: 0 and 1. Each coefficient a_j is multiplied by 2^j, and the results are added to obtain the decimal equivalent of the number. The radix point (e.g., the decimal point when 10 is the radix) distinguishes positive powers of 10 from negative powers of 10. For example, the decimal equivalent of the binary number 11010.11 is 26.75, as shown from the multiplication of the coefficients by powers of 2:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 26.75$$

In general, a number expressed in a base-r system has coefficients multiplied by powers of r:

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

The coefficients a_j range in value from 0 to $r - 1$. To distinguish between numbers of different bases, we enclose the coefficients in parentheses and write a subscript equal to the base used (except sometimes for decimal numbers, where the content makes it obvious that the base is decimal). An example of a base-5 number is

$$(4021.2)_5 = 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 + 2 \times 5^{-1} = (511.4)_{10}$$

The coefficient values for base 5 can be only 0, 1, 2, 3, and 4. The octal number system is a base-8 system that has eight digits: 0, 1, 2, 3, 4, 5, 6, 7. An example of an octal number is 127.4. To determine its equivalent decimal value, we expand the number in a power series with a base of 8:

$$(127.4)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = (87.5)_{10}$$

Note that the digits 8 and 9 cannot appear in an octal number.

It is customary to borrow the needed r digits for the coefficients from the decimal system when the base of the number is less than 10. The letters of the alphabet are used to supplement the 10 decimal digits when the base of the number is greater than 10. For example, in the *hexadecimal* (base-16) number system, the first 10 digits are borrowed from the decimal system. The letters A, B, C, D, E, and F are used for the digits 10, 11, 12, 13, 14, and 15, respectively. An example of a hexadecimal number is

$$(\text{B65F})_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46{,}687)_{10}$$

As noted before, the digits in a binary number are called *bits*. When a bit is equal to 0, it does not contribute to the sum during the conversion. Therefore, the conversion from binary to decimal can be obtained by adding only the numbers with powers of two corresponding to the bits that are equal to 1. For example,

$$(110101)_2 = 32 + 16 + 4 + 1 = (53)_{10}$$

There are four 1's in the binary number. The corresponding decimal number is the sum of the four powers of two. The first 24 numbers obtained from 2 to the power of n are listed in Table 1.1. In computer work, 2^{10} is referred to as K (kilo), 2^{20} as M (mega), 2^{30} as G (giga), and 2^{40} as T (tera). Thus, $4\text{K} = 2^{12} = 4{,}096$ and $16\text{M} = 2^{24} = 16{,}777{,}216$. Computer capacity is usually given in bytes. A *byte* is equal to eight bits and can accommodate (i.e., represent the code of) one keyboard character. A computer hard disk with four gigabytes of storage has a capacity of $4\text{G} = 2^{32}$ bytes (approximately 4 billion bytes).

Table 1.1
Powers of Two

n	2^n	*n*	2^n	*n*	2^n
0	1	8	256	16	65,536
1	2	9	512	17	131,072
2	4	10	1,024	18	262,144
3	8	11	2,048	19	524,288
4	16	12	4,096	20	1,048,576
5	32	13	8,192	21	2,097,152
6	64	14	16,384	22	4,194,304
7	128	15	32,768	23	8,388,608

Arithmetic operations with numbers in base *r* follow the same rules as for decimal numbers. When a base other than the familiar base 10 is used, one must be careful to use only the *r*-allowable digits. Examples of addition, subtraction, and multiplication of two binary numbers are as follows:

$$\begin{array}{lr}
\text{augend:} & 101101 \\
\text{addend:} & +100111 \\ \hline
\text{sum:} & 1010100
\end{array}
\qquad
\begin{array}{lr}
\text{minuend:} & 101101 \\
\text{subtrahend:} & -100111 \\ \hline
\text{difference:} & 000110
\end{array}
\qquad
\begin{array}{lr}
\text{multiplicand:} & 1011 \\
\text{multiplier:} & \times\ 101 \\ \hline
 & 1011 \\
 & 0000 \\
 & 1011 \\ \hline
\text{product:} & 110111
\end{array}$$

The sum of two binary numbers is calculated by the same rules as in decimal, except that the digits of the sum in any significant position can be only 0 or 1. Any carry obtained in a given significant position is used by the pair of digits one significant position higher. Subtraction is slightly more complicated. The rules are still the same as in decimal, except that the borrow in a given significant position adds 2 to a minuend digit. (A borrow in the decimal system adds 10 to a minuend digit.) Multiplication is simple: The multiplier digits are always 1 or 0; therefore, the partial products are equal either to the multiplicand or to 0.

1.3 NUMBER-BASE CONVERSIONS

The conversion of a number in base *r* to decimal is done by expanding the number in a power series and adding all the terms as shown previously. We now present a general procedure for the reverse operation of converting a decimal number to a number in base *r*. If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since each part must be converted differently. The conversion of a decimal integer to a number in base *r* is done by dividing the number and all successive quotients by *r* and accumulating the remainders. This procedure is best illustrated by example.

EXAMPLE 1.1

Convert decimal 41 to binary. First, 41 is divided by 2 to give an integer quotient of 20 and a remainder of $\frac{1}{2}$. Then the quotient is again divided by 2 to give a new quotient and remainder. The process is continued until the integer quotient becomes 0. The *coefficients* of the desired binary number are obtained from the *remainders* as follows:

	Integer Quotient		**Remainder**	**Coefficient**
41/2 =	20	+	$\frac{1}{2}$	$a_0 = 1$
20/2 =	10	+	0	$a_1 = 0$
10/2 =	5	+	0	$a_2 = 0$
5/2 =	2	+	$\frac{1}{2}$	$a_3 = 1$
2/2 =	1	+	0	$a_4 = 0$
1/2 =	0	+	$\frac{1}{2}$	$a_5 = 1$

Therefore, the answer is $(41)_{10} = (a_5a_4a_3a_2a_1a_0)_2 = (101001)_2$.

The arithmetic process can be manipulated more conveniently as follows:

Integer	**Remainder**	
41		
20	1	
10	0	
5	0	
2	1	
1	0	
0	1	101001 = answer

Conversion from decimal integers to any base-*r* system is similar to this example, except that division is done by *r* instead of 2.

■

EXAMPLE 1.2

Convert decimal 153 to octal. The required base *r* is 8. First, 153 is divided by 8 to give an integer quotient of 19 and a remainder of 1. Then 19 is divided by 8 to give an integer quotient of 2 and a remainder of 3. Finally, 2 is divided by 8 to give a quotient of 0 and a remainder of 2. This process can be conveniently manipulated as follows:

153	
19	1
2	3
0	$2 = (231)_8$

The conversion of a decimal *fraction* to binary is accomplished by a method similar to that used for integers. However, multiplication is used instead of division, and integers instead of remainders are accumulated. Again, the method is best explained by example.

■

EXAMPLE 1.3

Convert $(0.6875)_{10}$ to binary. First, 0.6875 is multiplied by 2 to give an integer and a fraction. Then the new fraction is multiplied by 2 to give a new integer and a new fraction. The process is continued until the fraction becomes 0 or until the number of digits have sufficient accuracy. The coefficients of the binary number are obtained from the integers as follows:

	Integer		**Fraction**	**Coefficient**
$0.6875 \times 2 =$	1	+	0.3750	$a_{-1} = 1$
$0.3750 \times 2 =$	0	+	0.7500	$a_{-2} = 0$
$0.7500 \times 2 =$	1	+	0.5000	$a_{-3} = 1$
$0.5000 \times 2 =$	1	+	0.0000	$a_{-4} = 1$

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. 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:

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

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:

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

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

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 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 have either $1 - 0 = 1$ or $1 - 1 = 0$, which causes the bit to change from 0 to 1 or from 1 to 0, respectively. Therefore, the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's. The following are some numerical examples:

The 1's complement of 1011000 is 0100111.

The 1's complement of 0101101 is 1010010.

The $(r - 1)$'s complement of octal or hexadecimal numbers is obtained by subtracting each digit from 7 or F (decimal 15), respectively.

Radix Complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ and as 0 for $N = 0$. Comparing with the $(r - 1)$'s complement, we note that the r's complement is obtained by adding 1 to the $(r - 1)$'s complement, since $r^n - N = [(r^n - 1) - N] + 1$. Thus, the 10's complement of decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's-complement value. The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's-complement value.

Since 10 is a number represented by a 1 followed by n 0's, $10^n - N$, which is the 10's complement of N, can be formed also by leaving all least significant 0's unchanged, subtracting the first nonzero least significant digit from 10, and subtracting all higher significant digits from 9. Thus,

the 10's complement of 012398 is 987602

and

the 10's complement of 246700 is 753300

The 10's complement of the first number is obtained by subtracting 8 from 10 in the least significant position and subtracting all other digits from 9. The 10's complement of the second number is obtained by leaving the two least significant 0's unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.

Similarly, the 2's complement can be formed by leaving all least significant 0's and the first 1 unchanged and replacing 1's with 0's and 0's with 1's in all other higher significant digits. For example,

the 2's complement of 1101100 is 0010100

and

the 2's complement of 0110111 is 1001001

The 2's complement of the first number is obtained by leaving the two least significant 0's and the first 1 unchanged and then replacing 1's with 0's and 0's with 1's in the other four most significant digits. The 2's complement of the second number is obtained by leaving the least significant 1 unchanged and complementing all other digits.

In the previous definitions, it was assumed that the numbers did not have a radix point. If the original number N contains a radix point, the point should be removed temporarily in order to form the r's or $(r - 1)$'s complement. The radix point is then restored to the complemented number in the same relative position. It is also worth mentioning that the complement of the complement restores the number to its original value. To see this relationship, note that the r's complement of N is $r^n - N$, so that the complement of the complement is $r^n - (r^n - N) = N$ and is equal to the original number.

Subtraction with Complements

The direct method of subtraction taught in elementary schools uses the borrow concept. In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the subtrahend digit. The method works well when people perform subtraction with paper and pencil. However, when subtraction is implemented with digital hardware, the method is less efficient than the method that uses complements.

The subtraction of two n-digit unsigned numbers $M - N$ in base r can be done as follows:

1. Add the minuend M to the r's complement of the subtrahend N. Mathematically, $M + (r^n - N) = M - N + r^n$.
2. If $M \geq N$, the sum will produce an end carry r^n, which can be discarded; what is left is the result $M - N$.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is the r's complement of $(N - M)$. To obtain the answer in a familiar form, take the r's complement of the sum and place a negative sign in front.

The following examples illustrate the procedure:

EXAMPLE 1.5

Using 10's complement, subtract 72532 − 3250.

$$\begin{array}{rr} M = & 72532 \\ \text{10's complement of } N = & +\ \underline{96750} \\ \text{Sum} = & 169282 \\ \text{Discard end carry } 10^5 = & -\underline{100000} \\ \textit{Answer} = & 69282 \end{array}$$

Note that M has five digits and N has only four digits. Both numbers must have the same number of digits, so we write N as 03250. Taking the 10's complement of N produces a 9 in the most significant position. The occurrence of the end carry signifies that $M \geq N$ and that the result is therefore positive.

■

EXAMPLE 1.6

Using 10's complement, subtract 3250 − 72532.

$$\begin{array}{rr} M = & 03250 \\ \text{10's complement of } N = & +\underline{27468} \\ \text{Sum} = & 30718 \end{array}$$

There is no end carry. Therefore, the answer is −(10's complement of 30718) = −69282.

Note that since 3250 < 72532, the result is negative. Because we are dealing with unsigned numbers, there is really no way to get an unsigned result for this case. When subtracting with complements, we recognize the negative answer from the absence of the end carry and the complemented result. When working with paper and pencil, we can change the answer to a signed negative number in order to put it in a familiar form.

Subtraction with complements is done with binary numbers in a similar manner, using the procedure outlined previously.

■

EXAMPLE 1.7

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction **(a)** $X - Y$ and **(b)** $Y - X$ by using 2's complements.

(a)

$$\begin{array}{rr} X = & 1010100 \\ \text{2's complement of } Y = & +\ \underline{0111101} \\ \text{Sum} = & 10010001 \\ \text{Discard end carry } 2^7 = & -\underline{10000000} \\ \textit{Answer: } X - Y = & 0010001 \end{array}$$

(b)

$$\begin{array}{rr} Y = & 1000011 \\ \text{2's complement of } X = & +\ \underline{0101100} \\ \text{Sum} = & 1101111 \end{array}$$

There is no end carry. Therefore, the answer is $Y - X = -(\text{2's complement of } 1101111) = -0010001$.

■

Subtraction of unsigned numbers can also be done by means of the $(r - 1)$'s complement. Remember that the $(r - 1)$'s complement is one less than the r's complement. Because of this, the result of adding the minuend to the complement of the subtrahend produces a sum that is one less than the correct difference when an end carry occurs. Removing the end carry and adding 1 to the sum is referred to as an *end-around carry*.

EXAMPLE 1.8

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

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

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

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

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

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 identifying the bits if the type of representation for the number is known in advance.

The representation of the signed numbers in the last example is referred to as the *signed-magnitude* convention. In this notation, the number consists of a magnitude and a symbol (+ or −) or a bit (0 or 1) indicating the sign. This is the representation of signed numbers used in ordinary arithmetic. When arithmetic operations are implemented in a computer, it is more convenient to use a different system, referred to as the *signed-complement* system, for representing negative numbers. In this system, a negative number is indicated by its complement. Whereas the signed-magnitude system negates a number by changing its sign, the signed-complement system negates a number by taking its complement. Since positive numbers always start with 0 (plus) in the leftmost position, the complement will always start with a 1, indicating a negative number. The signed-complement system can use either the 1's or the 2's complement, but the 2's complement is the most common.

As an example, consider the number 9, represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent of 9, which gives 00001001. Note that all eight bits must have a value; therefore, 0's are inserted following the sign bit up to the first 1. Although there is only one way to represent +9, there are three different ways to represent −9 with eight bits:

signed-magnitude representation:	10001001
signed-1's-complement representation:	11110110
signed-2's-complement representation:	11110111

In signed-magnitude, −9 is obtained from +9 by changing the sign bit in the leftmost position from 0 to 1. In signed-1's complement, −9 is obtained by complementing all the bits of +9, including the sign bit. The signed-2's-complement representation of −9 is obtained by taking the 2's complement of the positive number, including the sign bit.

Table 1.3
Signed Binary Numbers

Decimal	Signed-2's Complement	Signed-1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

Table 1.3 lists all possible four-bit signed binary numbers in the three representations. The equivalent decimal number is also shown for reference. Note that the positive numbers in all three representations are identical and have 0 in the leftmost position. The signed-2's-complement system has only one representation for 0, which is always positive. The other two systems have either a positive 0 or a negative 0, something not encountered in ordinary arithmetic. Note that all negative numbers have a 1 in the leftmost bit position; that is the way we distinguish them from the positive numbers. With four bits, we can represent 16 binary numbers. In the signed-magnitude and the 1's-complement representations, there are eight positive numbers and eight negative numbers, including two zeros. In the 2's-complement representation, there are eight positive numbers, including one zero, and eight negative numbers.

The signed-magnitude system is used in ordinary arithmetic, but is awkward when employed in computer arithmetic because of the separate handling of the sign and the magnitude. Therefore, the signed-complement system is normally used. The 1's complement imposes some difficulties and is seldom used for arithmetic operations. It is useful as a logical operation, since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation, as will be shown in the next chapter. The discussion of signed binary arithmetic that follows deals exclusively with the signed-2's-complement representation of negative numbers. The same procedures can be applied to the signed-1's-complement system by including the end-around carry as is done with unsigned numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the difference the sign of the larger magnitude. For example, $(+25) + (-37) = -(37 - 25) = -12$ and is done by subtracting the smaller magnitude, 25, from the larger magnitude, 37, and appending the sign of 37 to the result. This is a process that requires a comparison of the signs and magnitudes and then performing either addition or subtraction. The same procedure applies to binary numbers in signed-magnitude representation. In contrast, the rule for adding numbers in the signed-complement system does not require a comparison or subtraction, but only addition. The procedure is very simple and can be stated as follows for binary numbers:

The addition of two signed binary numbers with negative numbers represented in signed-2's-complement form is obtained from the addition of the two numbers, including their sign bits. A carry out of the sign-bit position is discarded.

Numerical examples for addition follow:

$$\begin{array}{rlcrl}
+\ 6 & 00000110 & \qquad & -\ 6 & 11111010 \\
\underline{+13} & \underline{00001101} & & \underline{+13} & \underline{00001101} \\
+19 & 00010011 & & +\ 7 & 00000111 \\
+\ 6 & 00000110 & & -\ 6 & 11111010 \\
\underline{-13} & \underline{11110011} & & \underline{-13} & \underline{11110011} \\
-\ 7 & 11111001 & & -19 & 11101101
\end{array}$$

Note that negative numbers must be initially in 2's-complement form and that if the sum obtained after the addition is negative, it is in 2's-complement form.

In each of the four cases, the operation performed is addition with the sign bit included. Any carry out of the sign-bit position is discarded, and negative results are automatically in 2's-complement form.

In order to obtain a correct answer, we must ensure that the result has a sufficient number of bits to accommodate the sum. If we start with two n-bit numbers and the sum occupies $n + 1$ bits, we say that an overflow occurs. When one performs the addition with paper and pencil, an overflow is not a problem, because we are not limited by the width of the page. We just add another 0 to a positive number or another 1 to a negative number in the most significant position to extend the number to $n + 1$ bits and then perform the addition. Overflow is a problem in computers because the number of bits that hold a number is finite, and a result that exceeds the finite value by 1 cannot be accommodated.

The complement form of representing negative numbers is unfamiliar to those used to the signed-magnitude system. To determine the value of a negative number in signed-2's complement, it is necessary to convert the number to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to −7.

Arithmetic Subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's-complement form is simple and can be stated as follows:

> Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign-bit position is discarded.

This procedure is adopted because a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed, as is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B);$$
$$(\pm A) - (-B) = (\pm A) + (+B).$$

But changing a positive number to a negative number is easily done by taking the 2's complement of the positive number. The reverse is also true, because the complement of a negative number in complement form produces the equivalent positive number. To see this, consider the subtraction $(-6) - (-13) = +7$. In binary with eight bits, this operation is written as $(11111010 - 11110011)$. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13), giving $(+13)$. In binary, this is $11111010 + 00001101 = 100000111$. Removing the end carry, we obtain the correct answer: 00000111 $(+7)$.

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

1.7 BINARY CODES

Digital systems use signals that have two distinct values and circuit elements that have two stable states. There is a direct analogy among binary signals, binary circuit elements, and binary digits. A binary number of n digits, for example, may be represented by n binary circuit elements, each having an output signal equivalent to 0 or 1. Digital systems represent and manipulate not only binary numbers, but also many other discrete elements of information. Any discrete element of information that is distinct among a group of quantities can be represented with a binary code (i.e., a pattern of 0's and 1's). The codes must be in binary because, in today's technology, only circuits that represent and manipulate patterns of 0's and 1's can be manufactured economically for use in computers. However, it must be realized that binary codes merely change the symbols, not the meaning of the elements of information that they represent. If we inspect the bits of a computer at random, we will find that most of the time they represent some type of coded information rather than binary numbers.

An n-bit binary code is a group of n bits that assumes up to 2^n distinct combinations of 1's and 0's, with each combination representing one element of the set that is being coded. A set of four elements can be coded with two bits, with each element assigned one of the following bit combinations: 00, 01, 10, 11. A set of eight elements requires a three-bit code and a set of

16 elements requires a four-bit code. The bit combination of an n-bit code is determined from the count in binary from 0 to $2^n - 1$. Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment will be ambiguous.

Although the *minimum* number of bits required to code 2^n distinct quantities is n, there is no *maximum* number of bits that may be used for a binary code. For example, the 10 decimal digits can be coded with 10 bits, and each decimal digit can be assigned a bit combination of nine 0's and a 1. In this particular binary code, the digit 6 is assigned the bit combination 0001000000.

BCD Code

Although the binary number system is the most natural system for a computer, most people are more accustomed to the decimal system. One way to resolve this difference is to convert decimal numbers to binary, perform all arithmetic calculations in binary, and then convert the binary results back to decimal. This method requires that we store decimal numbers in the computer so that they can be converted to binary. Since the computer can accept only binary values, we must represent the decimal digits by means of a code that contains 1's and 0's. It is also possible to perform the arithmetic operations directly on decimal numbers when they are stored in the computer in coded form.

A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits form such a set. A binary code that distinguishes among 10 elements must contain at least four bits, but 6 out of the 16 possible combinations remain unassigned. Different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straight binary assignment listed in Table 1.4. This scheme is called *binary-coded decimal* and is commonly referred to as BCD. Other decimal codes are possible and a few of them are presented later in this section.

Table 1.4
Binary-Coded Decimal (BCD)

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

Table 1.4 gives the four-bit code for one decimal digit. A number with k decimal digits will require $4k$ bits in BCD. Decimal 396 is represented in BCD with 12 bits as 0011 1001 0110, with each group of 4 bits representing one decimal digit. A decimal number in BCD is the same as its equivalent binary number only when the number is between 0 and 9. A BCD number greater than 10 looks different from its equivalent binary number, even though both contain 1's and 0's. Moreover, the binary combinations 1010 through 1111 are not used and have no meaning in BCD. Consider decimal 185 and its corresponding value in BCD and binary:

$$(185)_{10} = (0001\ 1000\ 0101)_{\text{BCD}} = (10111001)_2$$

The BCD value has 12 bits to encode the characters of the decimal value, but the equivalent binary number needs only 8 bits. It is obvious that the representation of a BCD number needs more bits than its equivalent binary value. However, there is an advantage in the use of decimal numbers, because computer input and output data are generated by people who use the decimal system.

It is important to realize that BCD numbers are decimal numbers and not binary numbers, although they use bits in their representation. The only difference between a decimal number and BCD is that decimals are written with the symbols 0, 1, 2, . . . , 9 and BCD numbers use the binary code 0000, 0001, 0010, . . . , 1001. The decimal value is exactly the same. Decimal 10 is represented in BCD with eight bits as 0001 0000 and decimal 15 as 0001 0101. The corresponding binary values are 1010 and 1111 and have only four bits.

BCD Addition

Consider the addition of two decimal digits in BCD, together with a possible carry from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than $9 + 9 + 1 = 19$, with the 1 being a previous carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this range will be from 0000 to 10011, but in BCD, it is from 0000 to 1 1001, with the first (i.e., leftmost) 1 being a carry and the next four bits being the BCD sum. When the binary sum is equal to or less than 1001 (without a carry), the corresponding BCD digit is correct. However, when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of $6 = (0110)_2$ to the binary sum converts it to the correct digit and also produces a carry as required. This is because a carry in the most significant bit position of the binary sum and a decimal carry differ by $16 - 10 = 6$. Consider the following three BCD additions:

$$\begin{array}{rrrrrr}
4 & 0100 & 4 & 0100 & 8 & 1000\\
\underline{+5} & \underline{+0101} & \underline{+8} & \underline{+1000} & \underline{+9} & \underline{1001}\\
9 & 1001 & 12 & 1100 & 17 & 10001\\
 & & & \underline{+0110} & & \underline{+0110}\\
 & & & 10010 & & 10111
\end{array}$$

In each case, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than or equal to 1010, we add 0110 to obtain the correct BCD sum and a carry. In the first example, the sum is equal to 9 and is the correct BCD sum. In the second example, the binary sum produces an invalid BCD digit. The addition of 0110 produces the correct BCD sum, 0010 (i.e., the number 2), and a carry. In the third example, the binary sum produces a carry. This condition occurs when the sum is greater than or equal to 16. Although the other four bits are less than 1001, the binary sum requires a correction because of the carry. Adding 0110, we obtain the required BCD sum 0111 (i.e., the number 7) and a BCD carry.

The addition of two n-digit unsigned BCD numbers follows the same procedure. Consider the addition of $184 + 576 = 760$ in BCD:

BCD	1	1		
	0001	1000	0100	184
	+0101	0111	0110	+576
Binary sum	0111	10000	1010	
Add 6		0110	0110	
BCD sum	0111	0110	0000	760

The first, least significant pair of BCD digits produces a BCD digit sum of 0000 and a carry for the next pair of digits. The second pair of BCD digits plus a previous carry produces a digit sum of 0110 and a carry for the next pair of digits. The third pair of digits plus a carry produces a binary sum of 0111 and does not require a correction.

Decimal Arithmetic

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can use either the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform to the four-bit code of the decimal digits. It is customary to designate a plus with four 0's and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is seldom used in computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add 1 to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2's-complement system in the previous section also apply to the signed-10's-complement system for decimal numbers. Addition is done by summing all digits, including the sign digit, and discarding the end carry. This operation assumes that all negative numbers are in 10's-complement form. Consider the addition $(+375) + (-240) = +135$, done in the signed-complement system:

0	375
+9	760
0	135

The 9 in the leftmost position of the second number represents a minus, and 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer, including the sign digits, must be in BCD. The addition is done with BCD digits as described previously.

The subtraction of decimal numbers, either unsigned or in the signed-10's-complement system, is the same as in the binary case: Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify programmed instructions to perform the arithmetic operation with decimal numbers directly, without having to convert them to binary.

Other Decimal Codes

Binary codes for decimal digits require a minimum of four bits per digit. Many different codes can be formulated by arranging four bits into 10 distinct combinations. BCD and three other representative codes are shown in Table 1.5. Each code uses only 10 out of a possible 16 bit combinations that can be arranged with four bits. The other six unused combinations have no meaning and should be avoided.

BCD and the 2421 code are examples of weighted codes. In a weighted code, each bit position is assigned a weighting factor in such a way that each digit can be evaluated by adding the weights of all the 1's in the coded combination. The BCD code has weights of 8, 4, 2, and 1, which correspond to the power-of-two values of each bit. The bit assignment 0110, for example, is interpreted by the weights to represent decimal 6 because $8 \times 0 + 4 \times 1 + 2 \times 1 + 1 \times 0 = 6$. The bit combination 1101, when weighted by the respective digits 2421, gives the decimal equivalent of $2 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 1 = 7$. Note that some digits can be coded in two possible ways in the 2421 code. For instance, decimal 4 can be assigned to bit combination 0100 or 1010, since both combinations add up to a total weight of 4.

Table 1.5
Four Different Binary Codes for the Decimal Digits

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, −2, −1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
Unused bit combinations	1010	0101	0000	0001
	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

The 2421 and the excess-3 codes are examples of self-complementing codes. Such codes have the property that the 9's complement of a decimal number is obtained directly by changing 1's to 0's and 0's to 1's (i.e., by complementing each bit in the pattern). For example, decimal 395 is represented in the excess-3 code as 0110 1100 1000. The 9's complement of 604 is represented as 1001 0011 0111, which is obtained simply by complementing each bit of the code (as with the 1's complement of binary numbers).

The excess-3 code has been used in some older computers because of its self-complementing property. Excess-3 is an unweighted code in which each coded combination is obtained from the corresponding binary value plus 3. Note that the BCD code is not self-complementing.

The 8, 4, −2, −1 code is an example of assigning both positive and negative weights to a decimal code. In this case, the bit combination 0110 is interpreted as decimal 2 and is calculated from $8 \times 0 + 4 \times 1 + (-2) \times 1 + (-1) \times 0 = 2$.

Gray Code

The output data of many physical systems are quantities that are continuous. These data must be converted into digital form before they are applied to a digital system. Continuous or analog information is converted into digital form by means of an analog-to-digital converter. It is sometimes convenient to use the Gray code shown in Table 1.6 to represent digital data that have been converted from analog data. The advantage of the Gray code over the straight binary number sequence is that only one bit in the code group changes in going from one number to the next. For example, in going from 7 to 8, the Gray code changes from 0100 to 1100. Only the first bit changes, from 0 to 1; the other three bits remain the same. By contrast, with binary numbers the change from 7 to 8 will be from 0111 to 1000, which causes all four bits to change values.

Table 1.6
Gray Code

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

The Gray code is used in applications in which the normal sequence of binary numbers may produce an error or ambiguity during the transition from one number to the next. If binary numbers are used, a change, for example, from 0111 to 1000 may produce an intermediate erroneous number 1001 if the value of the rightmost bit takes longer to change than do the values of the other three bits. The Gray code eliminates this problem, since only one bit changes its value during any transition between two numbers.

A typical application of the Gray code is the representation of analog data by a continuous change in the angular position of a shaft. The shaft is partitioned into segments, and each segment is assigned a number. If adjacent segments are made to correspond with the Gray-code sequence, ambiguity is eliminated between the angle of the shaft and the value encoded by the sensor.

ASCII Character Code

Many applications of digital computers require the handling not only of numbers, but also of other characters or symbols, such as the letters of the alphabet. For instance, an insurance company with thousands of policyholders will use a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters (such as S). An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet, and a number of special characters. Such a set contains between 36 and 64 elements if only capital letters are included, or between 64 and 128 elements if both uppercase and lowercase letters are included. In the first case, we need a binary code of six bits, and in the second, we need a binary code of seven bits.

The standard binary code for the alphanumeric characters is the American Standard Code for Information Interchange (ASCII), which uses seven bits to code 128 characters, as shown in Table 1.7. The seven bits of the code are designated by b_1 through b_7, with b_7 the most significant bit. The letter *A*, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions. The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, *, and S.

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed again below the table with their functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication-control characters. Format effectors are characters that control the layout of printing. They include the familiar word processor and typewriter controls such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions such as paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication-control characters are useful during the transmission of text between remote terminals. Examples of communication-control characters are STX (start of text) and ETX (end of text), which are used to frame a text message transmitted through telephone wires.

Table 1.7
American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	∧	n	~
1111	SI	US	/	?	O	–	o	DEL

Control characters

NUL	Null	DLE	Data-link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End-of-transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

ASCII is a seven-bit code, but most computers manipulate an eight-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize eight-bit ASCII characters with the most significant bit set to 0. An additional

128 eight-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font.

Error-Detecting Code

To detect errors in data communication and processing, an eighth bit is sometimes added to the ASCII character to indicate its parity. A *parity bit* is an extra bit included with a message to make the total number of 1's either even or odd. Consider the following two characters and their even and odd parity:

	With even parity	With odd parity
ASCII A = 1000001	01000001	11000001
ASCII T = 1010100	11010100	01010100

In each case, we insert an extra bit in the leftmost position of the code to produce an even number of 1's in the character for even parity or an odd number of 1's in the character for odd parity. In general, one or the other parity is adopted, with even parity being more common.

The parity bit is helpful in detecting errors during the transmission of information from one location to another. This function is handled by generating an even parity bit at the sending end for each character. The eight-bit characters that include parity bits are transmitted to their destination. The parity of each character is then checked at the receiving end. If the parity of the received character is not even, then at least one bit has changed value during the transmission. This method detects one, three, or any odd combination of errors in each character that is transmitted. An even combination of errors, however, goes undetected, and additional error detection codes may be needed to take care of that possibility.

What is done after an error is detected depends on the particular application. One possibility is to request retransmission of the message on the assumption that the error was random and will not occur again. Thus, if the receiver detects a parity error, it sends back the ASCII NAK (negative acknowledge) control character consisting of an even-parity eight bits 10010101. If no error is detected, the receiver sends back an ACK (acknowledge) control character, namely, 00000110. The sending end will respond to an NAK by transmitting the message again until the correct parity is received. If, after a number of attempts, the transmission is still in error, a message can be sent to the operator to check for malfunctions in the transmission path.

1.8 BINARY STORAGE AND REGISTERS

The binary information in a digital computer must have a physical existence in some medium for storing individual bits. A *binary cell* is a device that possesses two stable states and is capable of storing one bit (0 or 1) of information. The input to the cell receives excitation signals that set it to one of the two states. The output of the cell is a physical quantity that distinguishes between the two states. The information stored in a cell is 1 when the cell is in one stable state and 0 when the cell is in the other stable state.

Registers

A *register* is a group of binary cells. A register with n cells can store any discrete quantity of information that contains n bits. The state of a register is an n-tuple of 1's and 0's, with each bit designating the state of one cell in the register. The content of a register is a function of the interpretation given to the information stored in it. Consider, for example, a 16-bit register with the following binary content:

$$1100001111001001$$

A register with 16 cells can be in one of 2^{16} possible states. If one assumes that the content of the register represents a binary integer, then the register can store any binary number from 0 to $2^{16} - 1$. For the particular example shown, the content of the register is the binary equivalent of the decimal number 50,121. If one assumes instead that the register stores alphanumeric characters of an eight-bit code, then the content of the register is any two meaningful characters. For the ASCII code with an even parity placed in the eighth most significant bit position, the register contains the two characters C (the leftmost eight bits) and I (the rightmost eight bits). If, however, one interprets the content of the register to be four decimal digits represented by a four-bit code, then the content of the register is a four-digit decimal number. In the excess-3 code, the register holds the decimal number 9,096. The content of the register is meaningless in BCD, because the bit combination 1100 is not assigned to any decimal digit. From this example, it is clear that a register can store discrete elements of information and that the same bit configuration may be interpreted differently for different types of data.

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 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.

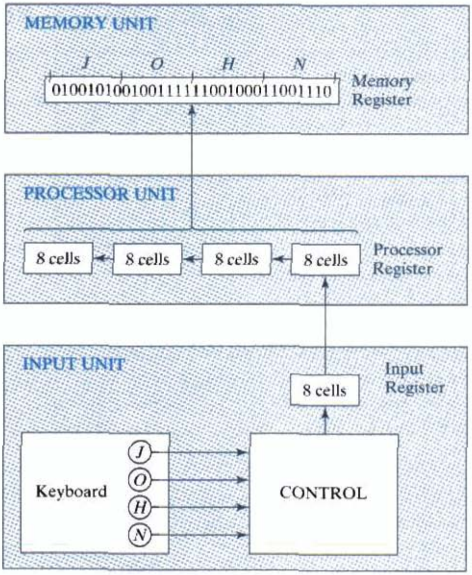


FIGURE 1.1
Transfer of information among registers

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 into *R1* and *R2*. The digital logic circuits produce the sum, which is transferred to register *R3*. The contents of *R3* can now be transferred back to one of the memory registers.

The last two examples demonstrated the information-flow capabilities of a digital system in a simple manner. The registers of the system are the basic elements for storing and holding the binary information. Digital logic circuits process the binary information stored in the

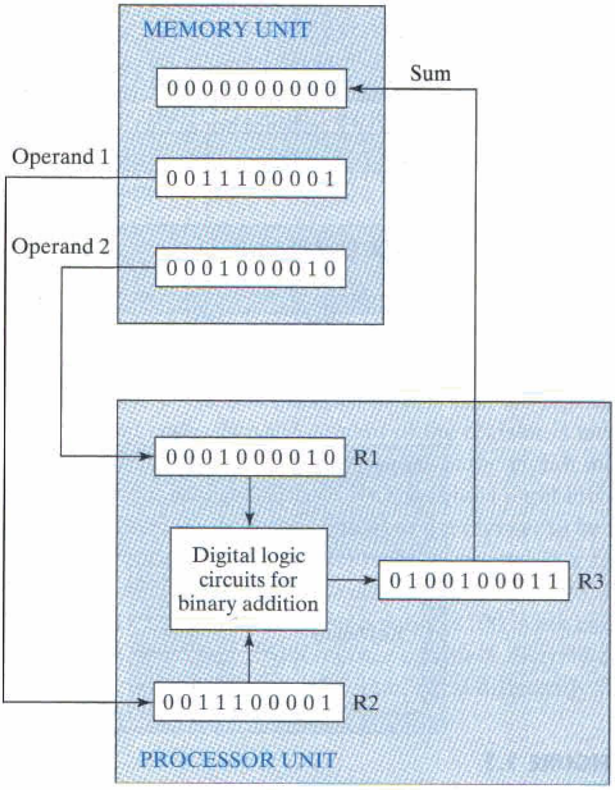


FIGURE 1.2
Example of binary information processing

registers. Digital logic circuits and registers are covered in Chapters 2 through 6. The memory unit is explained in Chapter 7. The description of register operations at the register transfer level and the design of digital systems are covered in Chapter 8.

1.9 BINARY LOGIC

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables assume may be called by different names (*true* and *false*, *yes* and *no*, etc.), but for our purpose, it is convenient to think in terms of bits and assign the values 1 and 0. The binary logic introduced in this section is equivalent to an algebra called Boolean algebra. The formal presentation of Boolean algebra is covered in more detail in Chapter 2. The purpose of this section is to introduce Boolean algebra in a heuristic manner and relate it to digital logic circuits and binary signals.

Definition of Binary Logic

Binary logic consists of binary variables and a set of logical operations. The variables are designated by letters of the alphabet, such as A, B, C, x, y, z, etc., with each variable having two and only two distinct possible values: 1 and 0. There are three basic logical operations: AND, OR, and NOT.

1. AND: This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read "x AND y is equal to z." The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$. (Remember that x, y, and z are binary variables and can be equal either to 1 or 0, and nothing else.)
2. OR: This operation is represented by a plus sign. For example, $x + y = z$ is read "x OR y is equal to z," meaning that $z = 1$ if $x = 1$ *or* if $y = 1$ *or* if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.
3. NOT: This operation is represented by a prime (sometimes by an overbar). For example, $x' = z$ (or $\overline{x} = z$) is read "not x is equal to z," meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$, but if $x = 0$, then $z = 1$. The NOT operation is also referred to as the complement operation, since it changes a 1 to 0 and a 0 to 1.

Binary logic resembles binary arithmetic, and the operations AND and OR have similarities to multiplication and addition, respectively. In fact, the symbols used for 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.

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		

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 range, say, 0 to 3 V, but in a digital system 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 volts and logic 1 as a signal equal to 3 volts. 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.

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

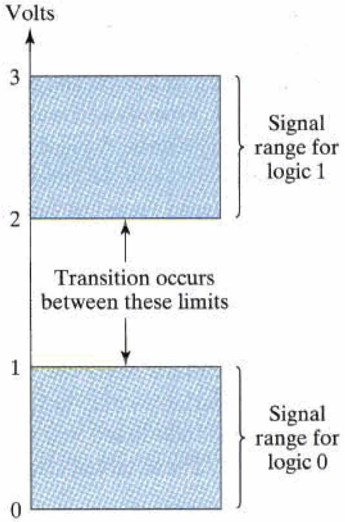
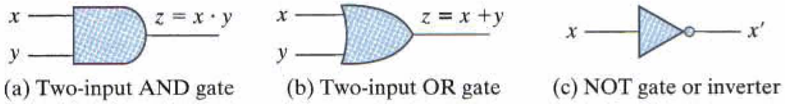


FIGURE 1.3
Example of binary signals



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

FIGURE 1.4
Symbols for digital logic circuits

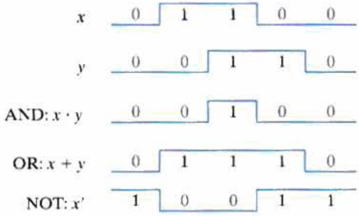
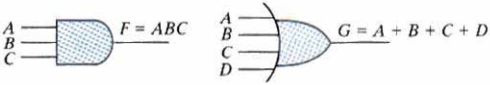


FIGURE 1.5
Input–output signals for gates



(a) Three-input AND gate (b) Four-input OR gate

FIGURE 1.6
Gates with multiple inputs

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 response of each gate to the four input signal combinations. The horizontal axis of the timing diagram represents time, and the vertical axis shows the signal as it changes between the two possible voltage levels. 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.

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with four inputs are shown in Fig. 1.6. The three-input AND gate responds with logic 1 output if all three inputs are logic 1. The output produces logic 0 if any input is logic 0. The four-input OR gate responds with logic 1 if any input is logic 1; its output becomes logic 0 only when all inputs are logic 0.

PROBLEMS

Answers to problems marked with * appear at the end of the book.

1.1 List the octal and hexadecimal numbers from 16 to 32. Using A, B, and C for the last three digits, list the numbers from 8 to 28 in base 13.

1.2* What is the exact number of bytes in a system that contains (a) 32K bytes, (b) 64M bytes, and (c) 6.4G bytes?

1.3 Convert the following numbers with the indicated bases to decimal:
(a)* $(4310)_5$ (b)* $(198)_{12}$
(c) $(735)_8$ (d) $(525)_6$

1.4 What is the largest binary number that can be expressed with 14 bits? What are the equivalent decimal and hexadecimal numbers?

1.5* Determine the base of the numbers in each case for the following operations to be correct:
(a) $14/2 = 5$, (b) $54/4 = 13$,
(c) $24 + 17 = 40$.

1.6* The solutions to the quadratic equation $x^2 - 11x + 22 = 0$ are $x = 3$ and $x = 6$. What is the base of the numbers?

1.7* Convert the hexadecimal number 68BE to binary, and then convert it from binary to octal.

1.8 Convert the decimal number 431 to binary in two ways: (a) Convert directly to binary; (b) convert first to hexadecimal and then from hexadecimal to binary. Which method is faster?

1.9 Express the following numbers in decimal:
(a)* $(10110.0101)_2$ (b)* $(16.5)_{16}$
(c)* $(26.24)_8$ (d) $(\mathrm{FAFA})_{16}$
(e) $(1010.1010)_2$

1.10 Convert the following binary numbers to hexadecimal and to decimal: (a) 1.10010, (b) 110.010. Explain why the decimal answer in (b) is 4 times that in (a).

1.11 Perform the following division in binary: 111011 ÷ 101.

1.12* Add and multiply the following numbers without converting them to decimal.
(a) Binary numbers 1011 and 101.
(b) Hexadecimal numbers 2E and 34.

1.13 Do the following conversion problems:
(a) Convert decimal 27.315 to binary.
(b) Calculate the binary equivalent of 2/3 out to eight places. Then convert from binary to decimal. How close is the result to 2/3?
(c) Convert the binary result in (b) into hexadecimal. Then convert the result to decimal. Is the answer the same?

1.14 Obtain the 1's and 2's complements of the following binary numbers:
(a) 10000000 (b) 00000000
(c) 11011010 (d) 01110110
(e) 10000101 (f) 11111111.

1.15 Find the 9's and the 10's complement of the following decimal numbers:
(a) 52,784,630 (b) 63,325,600
(c) 25,000,000 (d) 00,000,000.

1.16 (a) Find the 16's complement of B2FA.
(b) Convert B2FA to binary.
(c) Find the 2's complement of the result in (b).
(d) Convert the answer in (c) to hexadecimal and compare with the answer in (a).

1.17 Perform subtraction on the given unsigned numbers using the 10's complement of the subtrahend. Where the result should be negative, find its 10's complement and affix a minus sign. Verify your answers.

(a) 6,428 − 3,409 (b) 125 − 1,800

(c) 2,043 − 6,152 (d) 1,631 − 745

1.18 Perform subtraction on the given unsigned binary numbers using the 2's complement of the subtrahend. Where the result should be negative, find its 2's complement and affix a minus sign.

(a) 10011 − 10001 (b) 100010 − 100011

(c) 1001 − 101000 (d) 110000 − 10101

1.19* The following decimal numbers are shown in sign-magnitude form: +9,286 and +801. Convert them to signed-10's-complement form and perform the following operations (note that the sum is +10,627 and requires five digits and a sign).

(a) (+9,286) + (+801) (b) (+9,286) + (−801)

(c) (−9,286) + (+801) (d) (−9,286) + (−801)

1.20 Convert decimal +46 and +29 to binary, using the signed-2's-complement representation and enough digits to accommodate the numbers. Then perform the binary equivalent of (+29) + (−49), (−29) + (+49), and (−29) + (−49). Convert the answers back to decimal and verify that they are correct.

1.21 If the numbers $(+9{,}742)_{10}$ and $(+641)_{10}$ are in signed magnitude format, their sum is $(+10{,}383)_{10}$ and requires five digits and a sign. Convert the numbers to signed-10's-complement form and find the following sums:

(a) (+9,742) + (+641) (b) (+9,742) + (−641)

(c) (−9,742) + (+641) (d) (−9,742) + (−641)

1.22 Convert decimal 8,723 to both BCD and ASCII codes. For ASCII, an even parity bit is to be appended at the left.

1.23 Represent the unsigned decimal numbers 842 and 537 in BCD, and then show the steps necessary to form their sum.

1.24 Formulate a weighted binary code for the decimal digits, using weights

(a)* 6, 3, 1, 1

(b) 6, 4, 2, 1

1.25 Represent the decimal number 5,137 in (a) BCD, (b) excess-3 code, (c) 2421 code, and (d) a 6311 code.

1.26 Find the 9's complement of decimal 5,137 and express it in 2421 code. Show that the result is the 1's complement of the answer to (c) in Problem 1.25. This demonstrates that the 2421 code is self-complementing.

1.27 Assign a binary code in some orderly manner to the 52 playing cards. Use the minimum number of bits.

1.28 Write the expression "G. Boole" in ASCII, using an eight-bit code. Include the period and the space. Treat the leftmost bit of each character as a parity bit. Each eight-bit code should have even parity. (George Boole was a 19th century mathematician. Boolean algebra, introduced in the next chapter, bears his name.)

1.29* Decode the following ASCII code:

1000010 1101001 1101100 1101100 1000111 1100001 1110100 1100101 1110011.

1.30 The following is a string of ASCII characters whose bit patterns have been converted into hexadecimal for compactness: 73 F4 E5 76 E5 4A EF 62 73. Of the eight bits in each pair of digits, the leftmost is a parity bit. The remaining bits are the ASCII code.

(a) Convert the string to bit form and decode the ASCII.
(b) Determine the parity used: odd or even?

1.31* How many printing characters are there in ASCII? How many of them are special characters (not letters or numerals)?

1-32* What bit must be complemented to change an ASCII letter from capital to lowercase and vice versa?

1.33* The state of a 12-bit register is 100010010111. What is its content if it represents

(a) three decimal digits in BCD?
(b) three decimal digits in the excess-3 code?
(c) three decimal digits in the 84-2-1 code?
(d) a binary number?

1.34 List the ASCII code for the 10 decimal digits with an odd parity bit in the leftmost position.

1.35 By means of a timing diagram similar to Fig. 1.5, show the signals of the outputs f and g in Fig. P1.35 as functions of the three inputs a, b, and c. Use all eight possible combinations of a, b, and c.

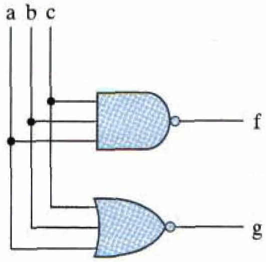


FIGURE P1.35

1.36 By means of a timing diagram similar to Fig. 1.5, show the signals of the outputs f and g in Fig. P1.36 as functions of the two inputs a and b. Use all four possible combinations of a and b.

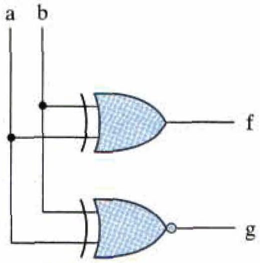


FIGURE P1.36

REFERENCES

1. Cavanagh, J. J. 1984. *Digital Computer Arithmetic.* New York: McGraw-Hill.
2. Mano, M. M. 1988. *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice-Hall.
3. Nelson, V. P., H. T. Nagle, J. D. Irwin, and B. D. Carroll. 1997. *Digital Logic Circuit Analysis and Design.* Upper Saddle River, NJ: Prentice Hall.
4. Schmid, H. 1974. *Decimal Computation.* New York: John Wiley.

Chapter 2

Boolean Algebra and Logic Gates

2.1 INTRODUCTION

Because binary logic is used in all of today's digital computers and devices, the cost of the circuits that implement it is an important factor addressed by designers. Finding simpler and cheaper, but equivalent, realizations of a circuit can reap huge payoffs in reducing the overall cost of the design. Mathematical methods that simplify circuits rely primarily on Boolean algebra. Therefore, this chapter provides a basic vocabulary and a brief foundation in Boolean algebra that will enable you to optimize simple circuits and to understand the purpose of algorithms used by software tools to optimize complex circuits involving millions of logic gates.

2.2 BASIC DEFINITIONS

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects, usually having a common property. If S is a set, and x and y are certain objects, then $x \in S$ means that x is a member of the set S and $y \notin S$ means that y is not an element of S. A set with a denumerable number of elements is specified by braces: $A = \{1, 2, 3, 4\}$ indicates that the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns, to each pair of elements from S, a unique element from S. As an example, consider the relation $a * b = c$. We say that $*$ is a binary operator if it specifies a rule for finding c from the pair (a, b) and also if $a, b, c \in S$. However, $*$ is not a binary operator if $a, b \in S$, if $c \notin S$.

The postulates of a mathematical system form the basic assumptions from which it is possible to deduce the rules, theorems, and properties of the system. The most common postulates used to formulate various algebraic structures are as follows:

1. *Closure*. A set S is closed with respect to a binary operator if, for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S. For example, the set of natural numbers $N = \{1, 2, 3, 4, \dots\}$ is closed with respect to the binary operator $+$ by the rules of arithmetic addition, since, for any $a, b \in N$, there is a unique $c \in N$ such that $a + b = c$. The set of natural numbers is *not* closed with respect to the binary operator $-$ by the rules of arithmetic subtraction, because $2 - 3 = -1$ and $2, 3 \in N$, but $(-1) \notin N$.
2. *Associative law*. A binary operator $*$ on a set S is said to be associative whenever
$$(x*y)*z = x*(y*z) \text{ for all } x, y, z, \in S$$
3. *Commutative law*. A binary operator $*$ on a set S is said to be commutative whenever
$$x*y = y*x \text{ for all } x, y \in S$$
4. *Identity element*. A set S is said to have an identity element with respect to a binary operation $*$ on S if there exists an element $e \in S$ with the property that
$$e*x = x*e = x \text{ for every } x \in S$$
Example: The element 0 is an identity element with respect to the binary operator $+$ on the set of integers $I = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$, since
$$x + 0 = 0 + x = x \text{ for any } x \in I$$
The set of natural numbers, N, has no identity element, since 0 is excluded from the set.
5. *Inverse*. A set S having the identity element e with respect to a binary operator $*$ is said to have an inverse whenever, for every $x \in S$, there exists an element $y \in S$ such that
$$x*y = e$$
Example: In the set of integers, I, and the operator $+$, with $e = 0$, the inverse of an element a is $(-a)$, since $a + (-a) = 0$.
6. *Distributive law*. If $*$ and $\cdot$ are two binary operators on a set S, $*$ is said to be distributive over $\cdot$ whenever
$$x*(y \cdot z) = (x*y) \cdot (x*z)$$

A *field* is an example of an algebraic structure. A field is a set of elements, together with two binary operators, each having properties 1 through 5 and both operators combining to give property 6. The set of real numbers, together with the binary operators $+$ and $\cdot$, forms the field of real numbers. The field of real numbers is the basis for arithmetic and ordinary algebra. The operators and postulates have the following meanings:

The binary operator $+$ defines addition.

The additive identity is 0.

The additive inverse defines subtraction.

The binary operator $\cdot$ defines multiplication.

The multiplicative identity is 1.

For $a \neq 0$, the multiplicative inverse of $a = 1/a$ defines division (i.e., $a \cdot 1/a = 1$).

The only distributive law applicable is that of $\cdot$ over $+$:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

2.3 AXIOMATIC DEFINITION OF BOOLEAN ALGEBRA

In 1854, George Boole developed an algebraic system now called *Boolean algebra*. In 1938, C. E. Shannon introduced a two-valued Boolean algebra called *switching algebra* that represented the properties of bistable electrical switching circuits. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

Boolean algebra is an algebraic structure defined by a set of elements, *B*, together with two binary operators, $+$ and $\cdot$, provided that the following (Huntington) postulates are satisfied:

1. (a) The structure is closed with respect to the operator $+$.
   (b) The structure is closed with respect to the operator $\cdot$.
2. (a) The element 0 is an identity element with respect to $+$; that is, $x + 0 = 0 + x = x$.
   (b) The element 1 is an identity element with respect to $\cdot$; that is, $x \cdot 1 = 1 \cdot x = x$.
3. (a) The structure is commutative with respect to $+$; that is, $x + y = y + x$.
   (b) The structure is commutative with respect to $\cdot$; that is, $x \cdot y = y \cdot x$.
4. (a) The operator $\cdot$ is distributive over $+$; that is, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.
   (b) The operator $+$ is distributive over $\cdot$; that is, $x + (y \cdot z) = (x + y) \cdot (x + z)$.
5. For every element $x \in B$, there exists an element $x' \in B$ (called the *complement* of x) such that (a) $x + x' = 1$ and (b) $x \cdot x' = 0$.
6. There exist at least two elements $x, y \in B$ such that $x \neq y$.

Comparing Boolean algebra with arithmetic and ordinary algebra (the field of real numbers), we note the following differences:

1. Huntington postulates do not include the associative law. However, this law holds for Boolean algebra and can be derived (for both operators) from the other postulates.
2. The distributive law of $+$ over $\cdot$ (i.e., $x + (y \cdot z) = (x + y) \cdot (x + z)$), is valid for Boolean algebra, but not for ordinary algebra.
3. Boolean algebra does not have additive or multiplicative inverses; therefore, there are no subtraction or division operations.

4. Postulate 5 defines an operator called the *complement* that is not available in ordinary algebra.
5. Ordinary algebra deals with the real numbers, which constitute an infinite set of elements. Boolean algebra deals with the as yet undefined set of elements, B, but in the two-valued Boolean algebra defined next (and of interest in our subsequent use of that algebra), B is defined as a set with only two elements, 0 and 1.

Boolean algebra resembles ordinary algebra in some respects. The choice of the symbols $+$ and $\cdot$ is intentional, to facilitate Boolean algebraic manipulations by persons already familiar with ordinary algebra. Although one can use some knowledge from ordinary algebra to deal with Boolean algebra, the beginner must be careful not to substitute the rules of ordinary algebra where they are not applicable.

It is important to distinguish between the elements of the set of an algebraic structure and the variables of an algebraic system. For example, the elements of the field of real numbers are numbers, whereas variables such as a, b, c, etc., used in ordinary algebra, are symbols that *stand for* real numbers. Similarly, in Boolean algebra, one defines the elements of the set B, and variables such as x, y, and z are merely symbols that *represent* the elements. At this point, it is important to realize that, in order to have a Boolean algebra, one must show that

1. the elements of the set B,
2. the rules of operation for the two binary operators, and
3. the set of elements, B, together with the two operators, satisfy the six Huntington postulates.

One can formulate many Boolean algebras, depending on the choice of elements of B and the rules of operation. In our subsequent work, we deal only with a two-valued Boolean algebra (i.e., a Boolean algebra with only two elements). Two-valued Boolean algebra has applications in set theory (the algebra of classes) and in propositional logic. Our interest here is in the application of Boolean algebra to gate-type circuits.

Two-Valued Boolean Algebra

A two-valued Boolean algebra is defined on a set of two elements, $B = \{0, 1\}$, with rules for the two binary operators $+$ and $\cdot$ as shown in the following operator tables (the rule for the complement operator is for verification of postulate 5):

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

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 presentation is necessary for developing the theorems and properties of the algebraic system. The two-valued Boolean algebra defined in this section is also called "switching algebra" by engineers. To emphasize the similarities between two-valued Boolean algebra and other binary systems, that algebra was called "binary logic" in Section 1.9. From here on, we shall drop the adjective "two-valued" from Boolean algebra in subsequent discussions.

2.4 BASIC THEOREMS AND PROPERTIES OF BOOLEAN ALGEBRA

Duality

In Section 2.3, the Huntington postulates were listed in pairs and designated by part (a) and part (b). One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called the *duality principle* and states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, the identity elements and the elements of the set B are the same: 1 and 0. The duality principle has many applications. If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

Basic Theorems

Table 2.1 lists six theorems of Boolean algebra and four of its postulates. The notation is simplified by omitting the binary operator whenever doing so does not lead to confusion. The theorems and postulates listed are the most basic relationships in Boolean algebra. The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. Proofs of the theorems with one variable are presented next. At the right is listed the number of the postulate which justifies that particular step of the proof.

Table 2.1
Postulates and Theorems of Boolean Algebra

Postulate 2	(a)	$x + 0 = x$	(b)	$x \cdot 1 = x$
Postulate 5	(a)	$x + x' = 1$	(b)	$x \cdot x' = 0$
Theorem 1	(a)	$x + x = x$	(b)	$x \cdot x = x$
Theorem 2	(a)	$x + 1 = 1$	(b)	$x \cdot 0 = 0$
Theorem 3, involution		$(x')' = x$		
Postulate 3, commutative	(a)	$x + y = y + x$	(b)	$xy = yx$
Theorem 4, associative	(a)	$x + (y + z) = (x + y) + z$	(b)	$x(yz) = (xy)z$
Postulate 4, distributive	(a)	$x(y + z) = xy + xz$	(b)	$x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a)	$(x + y)' = x'y'$	(b)	$(xy)' = x' + y'$
Theorem 6, absorption	(a)	$x + xy = x$	(b)	$x(x + y) = x$

THEOREM 1(a): $x + x = x$.

Statement	Justification
$x + x = (x + x) \cdot 1$	postulate 2(b)
$= (x + x)(x + x')$	5(a)
$= x + xx'$	4(b)
$= x + 0$	5(b)
$= x$	2(a)

THEOREM 1(b): $x \cdot x = x$.

Statement	Justification
$x \cdot x = xx + 0$	postulate 2(a)
$= xx + xx'$	5(b)
$= x(x + x')$	4(a)
$= x \cdot 1$	5(a)
$= x$	2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of its counterpart in part (a). Any dual theorem can be similarly derived from the proof of its corresponding theorem.

THEOREM 2(a): $x + 1 = 1$.

Statement	Justification
$x + 1 = 1 \cdot (x + 1)$	postulate 2(b)
$= (x + x')(x + 1)$	5(a)
$= x + x' \cdot 1$	4(b)
$= x + x'$	2(b)
$= 1$	5(a)

THEOREM 2(b): $x \cdot 0 = 0$ by duality.

THEOREM 3: $(x')' = x$. From postulate 5, we have $x + x' = 1$ and $x \cdot x' = 0$, which together define the complement of x. The complement of x' is x and is also $(x')'$. Therefore, since the complement is unique, we have $(x')' = x$. The theorems involving two or three variables may be proven algebraically from the postulates and the theorems that have already been proven. Take, for example, the absorption theorem:

THEOREM 6(a): $x + xy = x$.

Statement	Justification
$x + xy = x \cdot 1 + xy$	postulate 2(b)
$= x(1 + y)$	4(a)
$= x(y + 1)$	3(a)
$= x \cdot 1$	2(a)
$= x$	2(b)

THEOREM 6(b): $x(x + y) = x$ by duality.

The theorems of Boolean algebra can be proven by means of truth tables. In truth tables, both sides of the relation are checked to see whether they yield identical results for all possible combinations of the variables involved. The following truth table verifies the first absorption theorem:

x	y	xy	$x + xy$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

The algebraic proofs of the associative law and DeMorgan's theorem are long and will not be shown here. However, their validity is easily shown with truth tables. For example, the truth table for the first DeMorgan's theorem, $(x + y)' = x'y'$, is as follows:

x	y	$x + y$	$(x + y)'$	x'	y'	$x'y'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Operator Precedence

The operator precedence for evaluating Boolean expressions is (1) parentheses, (2) NOT, (3) AND, and (4) OR. In other words, expressions inside parentheses must be evaluated before all other operations. The next operation that holds precedence is the complement, and then follows the AND and, finally, the OR. As an example, consider the truth table for one of DeMorgan's theorems. The left side of the expression is $(x + y)'$. Therefore, the expression inside the parentheses is evaluated first and the result then complemented. The right side of the expression is $x'y'$, so the complement of x and the complement of y are both evaluated first and the result is then ANDed. Note that in ordinary arithmetic, the same precedence holds (except for the complement) when multiplication and addition are replaced by AND and OR, respectively.

2.5 BOOLEAN FUNCTIONS

Boolean algebra is an algebra that deals with binary variables and logic operations. A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols. For a given value of the binary variables, the function can be equal to either 1 or 0. As an example, consider the Boolean function

$$F_1 = x + y'z$$

The function F_1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. F_1 is equal to 0 otherwise. The complement operation dictates that when $y' = 1$, $y = 0$. Therefore, $F_1 = 1$ if $x = 1$ or if $y = 0$ and $z = 1$. A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n, where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$. Table 2.2 shows the truth table for the function F_1. There are eight possible binary combinations for assigning bits to the three variables x, y, and z. The column labeled F_1 contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when $x = 1$ or when $yz = 01$ and is equal to 0 otherwise.

A Boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates connected in a particular structure. The logic-circuit diagram (also called a schematic) for F_1 is shown in Fig. 2.1. There is an inverter for input y to generate its complement. There is an AND gate for the term $y'z$ and an OR gate 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.

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

Table 2.2
Truth Tables for F_1 and F_2

x	y	z	F_1	F_2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

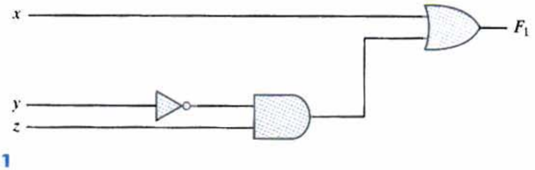
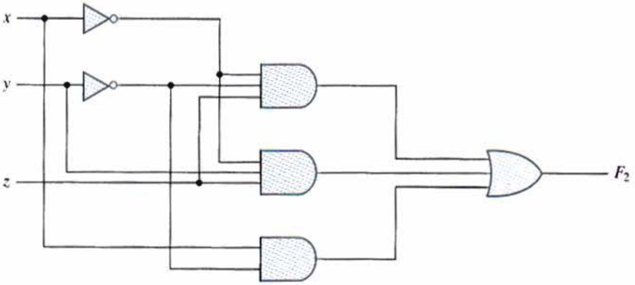
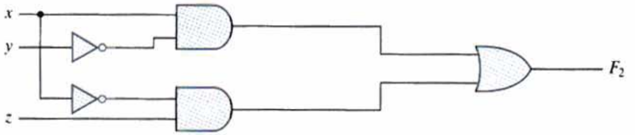


FIGURE 2.1
Gate implementation of $F_1 = x + y'z$



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



(b) $F_2 = xy' + x'z$

FIGURE 2.2
Implementation of Boolean function F_2 with gates

have equivalent logic. The particular expression used to represent the function will dictate the interconnection of gates in the logic-circuit diagram. 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 produce the same truth table, they are equivalent. Therefore, the two circuits have the same outputs for all possible binary combinations of inputs of the three variables. Each circuit implements the same identical function, but the one with fewer gates and fewer inputs to gates is preferable because it requires fewer wires and components. In general, there are many equivalent representations of a logic function.

Algebraic Manipulation

When a Boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to the gate. We define a *literal* to be a single variable within a term, in complemented or uncomplemented form. The function of Fig. 2.2(a) has three terms and eight literals, and the one in Fig. 2.2(b) has two terms and four literals. By reducing the number of terms, the number of literals, or both in a Boolean expression, it is often possible to obtain a simpler circuit. The manipulation of Boolean algebra consists mostly of reducing an expression for the purpose of obtaining a simpler circuit. Functions of up to five variables can be simplified by the map method described in the next chapter. For complex Boolean functions, designers of digital circuits use computer minimization programs that are capable of producing optimal circuits with millions of logic gates. The concepts introduced in this chapter provide the framework for those tools. The only manual method available is a cut-and-try procedure employing the basic relations and other manipulation techniques that become familiar with use, but remain, nevertheless, subject to human error. The examples that follow illustrate the algebraic manipulation of Boolean algebra.

EXAMPLE 2.1

Simplify the following Boolean functions to a minimum number of literals.

1. $x(x' + y) = xx' + xy = 0 + xy = xy.$
2. $x + x'y = (x + x')(x + y) = 1(x + y) = x + y.$

3. $(x + y)(x + y') = x + xy + xy' + yy' = x(1 + y + y') = x.$

4. $$\begin{aligned} xy + x'z + yz &= xy + x'z + yz(x + x') \\ &= xy + x'z + xyz + x'yz \\ &= xy(1 + z) + x'z(1 + y) \\ &= xy + x'z. \end{aligned}$$

5. $(x + y)(x' + z)(y + z) = (x + y)(x' + z)$, by duality from function 4.

■

Functions 1 and 2 are the dual of each other and use dual expressions in corresponding steps. An easier way to simplify function 3 is by means of postulate 4(b) from Table 2.1: $(x + y)(x + y') = x + yy' = x$. The fourth function illustrates the fact that an increase in the number of literals sometimes leads to a simpler final expression. Function 5 is not minimized directly, but can be derived from the dual of the steps used to derive function 4. Functions 4 and 5 are together known as the *consensus theorem.*

Complement of a Function

The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F. The complement of a function may be derived algebraically through DeMorgan's theorems, listed in Table 2.1 for two variables. DeMorgan's theorems can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived as follows, from postulates and theorems listed in Table 2.1:

$$\begin{aligned} (A + B + C)' &= (A + x)' && \text{let } B + C = x \\ &= A'x' && \text{by theorem 5(a) (DeMorgan)} \\ &= A'(B + C)' && \text{substitute } B + C = x \\ &= A'(B'C') && \text{by theorem 5(a) (DeMorgan)} \\ &= A'B'C' && \text{by theorem 4(b) (associative)} \end{aligned}$$

DeMorgan's theorems for any number of variables resemble the two-variable case in form and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as follows:

$$(A + B + C + D + \cdots + F)' = A'B'C'D'\ldots F'$$
$$(ABCD\ldots F)' = A' + B' + C' + D' + \cdots + F'$$

The generalized form of DeMorgan's theorems states that the complement of a function is obtained by interchanging AND and OR operators and complementing each literal.

EXAMPLE 2.2

Find the complement of the functions $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$. By applying DeMorgan's theorems as many times as necessary, the complements are obtained as follows:

$$\begin{aligned} F_1' &= (x'yz' + x'y'z)' = (x'yz')'(x'y'z)' = (x + y' + z)(x + y + z') \\ F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' = x' + (y'z')'(yz)' \\ &= x' + (y + z)(y' + z') \\ &= x' + yz' + y'z \end{aligned}$$

■

A simpler procedure for deriving the complement of a function is to take the dual of the function and complement each literal. This method follows from the generalized forms of DeMorgan's theorems. Remember that the dual of a function is obtained from the interchange of AND and OR operators and 1's and 0's.

EXAMPLE 2.3

Find the complement of the functions F_1 and F_2 of Example 2.2 by taking their duals and complementing each literal.

1. $F_1 = x'yz' + x'y'z$.
   The dual of F_1 is $(x' + y + z')(x' + y' + z)$.
   Complement each literal: $(x + y' + z)(x + y + z') = F_1'$.
2. $F_2 = x(y'z' + yz)$.
   The dual of F_2 is $x + (y' + z')(y + z)$.
   Complement each literal: $x' + (y + z)(y' + z') = F_2'$.

■

2.6 CANONICAL AND STANDARD FORMS

Minterms and Maxterms

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy', and xy. Each of these four AND terms is called a *minterm*, or a *standard product*. In a similar manner, n variables can be combined to form 2^n minterms. The 2^n different minterms may be determined by a method similar to the one shown in Table 2.3 for three variables. The binary numbers from 0 to $2^n - 1$ are listed under the n variables. Each minterm is obtained from an AND term of the n variables, with each variable being primed if the corresponding bit of the binary number is a 0 and unprimed if a 1. A symbol for each minterm is also shown in the table and is of

Table 2.3
Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	**y**	**z**	**Term**	**Designation**	**Term**	**Designation**
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Table 2.4
Functions of Three Variables

x	**y**	**z**	**Function f_1**	**Function f_2**
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

the form m_j, where the subscript j denotes the decimal equivalent of the binary number of the minterm designated.

In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called *maxterms*, or *standard sums*. The eight maxterms for three variables, together with their symbolic designations, are listed in Table 2.3. Any 2^n maxterms for n variables may be determined similarly. It is important to note that (1) each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is a 0 and primed if a 1, and (2) each maxterm is the complement of its corresponding minterm and vice versa.

A Boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces a 1 in the function and then taking the OR of all those terms. For example, the function f_1 in Table 2.4 is determined by expressing the combinations 001, 100, and 111 as $x'y'z$, $xy'z'$, and xyz, respectively. Since each one of these minterms results in $f_1 = 1$, we have

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7$$

Similarly, it may be easily verified that

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7$$

These examples demonstrate an important property of Boolean algebra: Any Boolean function can be expressed as a sum of minterms (with "sum" meaning the ORing of terms).

Now consider the complement of a Boolean function. It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms. The complement of f_1 is read as

$$f'_1 = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

If we take the complement of f'_1, we obtain the function f_1:

$$\begin{aligned} f_1 &= (x + y + z)(x + y' + z)(x' + y + z')(x' + y' + z) \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \end{aligned}$$

Similarly, it is possible to read the expression for f_2 from the table:

$$\begin{aligned} f_2 &= (x + y + z)(x + y + z')(x + y' + z)(x' + y + z) \\ &= M_0M_1M_2M_4 \end{aligned}$$

These examples demonstrate a second property of Boolean algebra: Any Boolean function can be expressed as a product of maxterms (with "product" meaning the ANDing of terms). The procedure for obtaining the product of maxterms directly from the truth table is as follows: Form a maxterm for each combination of the variables that produces a 0 in the function, and then form the AND of all those maxterms. Boolean functions expressed as a sum of minterms or product of maxterms are said to be in *canonical form*.

Sum of Minterms

Previously, we stated that, for n binary variables, one can obtain 2^n distinct minterms and that any Boolean function can be expressed as a sum of minterms. The minterms whose sum defines the Boolean function are those which give the 1's of the function in a truth table. Since the function can be either 1 or 0 for each minterm, and since there are 2^n minterms, one can calculate all the functions that can be formed with n variables to be 2^{2n}. It is sometimes convenient to express a Boolean function in its sum-of-minterms form. If the function is not in this form, it can be made so by first expanding the expression into a sum of AND terms. Each term is then inspected to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables. The next example clarifies this procedure.

EXAMPLE 2.4

Express the Boolean function $F = A + B'C$ as a sum of minterms. The function has three variables: A, B, and C. The first term A is missing two variables; therefore,

$$A = A(B + B') = AB + AB'$$

This function is still missing one variable, so

$$\begin{aligned} A &= AB(C + C') + AB'(C + C') \\ &= ABC + ABC' + AB'C + AB'C' \end{aligned}$$

The second term $B'C$ is missing one variable; hence,

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

Combining all terms, we have

$$\begin{aligned} F &= A + B'C \\ &= ABC + ABC' + AB'C + AB'C' + A'B'C \end{aligned}$$

But $AB'C$ appears twice, and according to theorem 1 ($x + x = x$), it is possible to remove one of those occurrences. Rearranging the minterms in ascending order, we finally obtain

$$\begin{aligned} F &= A'B'C + AB'C + AB'C + ABC' + ABC \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

■

When a Boolean function is in its sum-of-minterms form, it is sometimes convenient to express the function in the following brief notation:

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

The summation symbol Σ stands for the ORing of terms; the numbers following it are the minterms of the function. The letters in parentheses following F form a list of the variables in the order taken when the minterm is converted to an AND term.

An alternative procedure for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table. Consider the Boolean function given in Example 2.4:

$$F = A + B'C$$

The truth table shown in Table 2.5 can be derived directly from the algebraic expression by listing the eight binary combinations under variables A, B, and C and inserting 1's under F for those

Table 2.5
Truth Table for $F = A + B'C$

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

combinations for which $A = 1$ and $BC = 01$. From the truth table, we can then read the five minterms of the function to be 1, 4, 5, 6, and 7.

Product of Maxterms

Each of the 2^{2^n} functions of n binary variables can be also expressed as a product of maxterms. To express a Boolean function as a product of maxterms, it must first be brought into a form of OR terms. This may be done by using the distributive law, $x + yz = (x + y)(x + z)$. Then any missing variable x in each OR term is ORed with xx'. The procedure is clarified in the following example.

EXAMPLE 2.5

Express the Boolean function $F = xy + x'z$ as a product of maxterms. First, convert the function into OR terms by using the distributive law:

$$\begin{aligned} F = xy + x'z &= (xy + x')(xy + z) \\ &= (x + x')(y + x')(x + z)(y + z) \\ &= (x' + y)(x + z)(y + z) \end{aligned}$$

The function has three variables: x, y, and z. Each OR term is missing one variable; therefore,

$$\begin{aligned} x' + y &= x' + y + zz' = (x' + y + z)(x' + y + z') \\ x + z &= x + z + yy' = (x + y + z)(x + y' + z) \\ y + z &= y + z + xx' = (x + y + z)(x' + y + z) \end{aligned}$$

Combining all the terms and removing those which appear more than once, we finally obtain

$$\begin{aligned} F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z') \\ &= M_0M_2M_4M_5 \end{aligned}$$

A convenient way to express this function is as follows:

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

The product symbol, Π, denotes the ANDing of maxterms; the numbers are the maxterms of the function.

■

Conversion between Canonical Forms

The complement of a function expressed as the sum of minterms equals the sum of minterms missing from the original function. This is because the original function is expressed by those minterms which make the function equal to 1, whereas its complement is a 1 for those minterms for which the function is a 0. As an example, consider the function

$$F(A, B, C) = \Sigma(1, 4, 5, 6, 7)$$

This function has a complement that can be expressed as

$$F'(A, B, C) = \Sigma(0, 2, 3) = m_0 + m_2 + m_3$$

Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:

$$F = (m_0 + m_2 + m_3)' = m_0' \cdot m_2' \cdot m_3' = M_0 M_2 M_3 = \Pi(0, 2, 3)$$

The last conversion follows from the definition of minterms and maxterms as shown in Table 2.3. From the table, it is clear that the following relation holds:

$$m_j' = M_j$$

That is, the maxterm with subscript j is a complement of the minterm with the same subscript j and vice versa.

The last example demonstrates the conversion between a function expressed in sum-of-minterms form and its equivalent in product-of-maxterms form. A similar argument will show that the conversion between the product of maxterms and the sum of minterms is similar. We now state a general conversion procedure: To convert from one canonical form to another, interchange the symbols Σ and Π and list those numbers missing from the original form. In order to find the missing terms, one must realize that the total number of minterms or maxterms is 2^n, where n is the number of binary variables in the function.

A Boolean function can be converted from an algebraic expression to a product of maxterms by means of a truth table and the canonical conversion procedure. Consider, for example, the Boolean expression

$$F = xy + x'z$$

First, we derive the truth table of the function, as shown in Table 2.6. The 1's under F in the table are determined from the combination of the variables for which $xy = 11$ or $xz = 01$. The minterms of the function are read from the truth table to be 1, 3, 6, and 7. The function expressed as a sum of minterms is

$$F(x, y, z) = \Sigma(1, 3, 6, 7)$$

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.

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

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

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

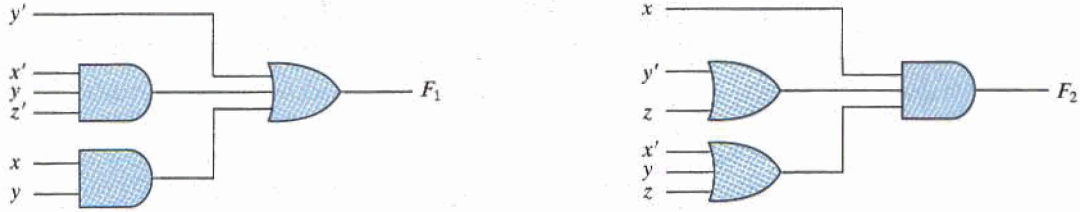
The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

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.*

A *product of sums* is a Boolean expression containing OR terms, called *sum terms.* Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed as a product of sums is

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

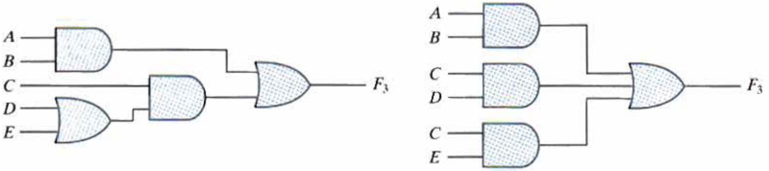
This expression has three sum terms, with one, two, and three literals. The product is an AND operation. The use of the words *product* and *sum* stems from the similarity of the AND operation to the arithmetic product (multiplication) and the similarity of the OR operation to the arithmetic sum (addition). The gate structure of the product-of-sums expression consists of a group of OR gates for the sum terms (except for a single literal), followed by an AND gate, as shown in Fig. 2.3(b). This standard type of expression results in a two-level gating structure.



(a) Sum of Products

(b) Product of Sums

FIGURE 2.3
Two-level implementation



(a) $AB + C(D + E)$ (b) $AB + CD + CE$

FIGURE 2.4
Three- and two-level implementation

A Boolean function may be expressed in a nonstandard form. For example, the function

$$F_3 = AB + C(D + E)$$

is neither in sum-of-products nor in product-of-sums form. The implementation of this expression is shown in Fig. 2.4(a) and requires two AND gates and two OR gates. There are three levels of gating in this circuit. It can be changed to a standard form by using the distributive law to remove the parentheses:

$$F_3 = AB + C(D + E) = AB + CD + CE$$

The sum-of-products expression is implemented in Fig. 2.4(b). In general, a two-level implementation is preferred because it produces the least amount of delay through the gates when the signal propagates from the inputs to the output. However, the number of inputs to a given gate might not be practical.

2.7 OTHER LOGIC OPERATIONS

When the binary operators AND and OR are placed between two variables, x and y, they form two Boolean functions, $x \cdot y$ and $x + y$, respectively. Previously we stated that there are 2^{2^n} functions for n binary variables. Thus, for two variables, $n = 2$, and the number of possible Boolean functions is 16. Therefore, the AND and OR functions are only 2 of a total of 16 possible functions formed with two binary variables. It would be instructive to find the other 14 functions and investigate their properties.

The truth tables for the 16 functions formed with two binary variables are listed in Table 2.7. Each of the 16 columns, F_0 to F_{15}, represents a truth table of one possible function for the two variables, x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F. The 16 functions can be expressed algebraically by means of Boolean functions, as is shown in the first column of Table 2.8. The Boolean expressions listed are simplified to their minimum number of literals.

Although each function can be expressed in terms of the Boolean operators AND, OR, and NOT, there is no reason one cannot assign special operator symbols for expressing the other functions. Such operator symbols are listed in the second column of Table 2.8. However, of all the new symbols shown, only the exclusive-OR symbol, $\oplus$, is in common use by digital designers.

Table 2.7
Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2.8
Boolean Expressions for the 16 Functions of Two Variables

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Each of the functions in Table 2.8 is listed with an accompanying name and a comment that explains the function in some way. The 16 functions listed can be subdivided into three categories:

1. Two functions that produce a constant 0 or 1.
2. Four functions with unary operations: complement and transfer.
3. Ten functions with binary operators that define eight different operations: AND, OR, NAND, NOR, exclusive-OR, equivalence, inhibition, and implication.

Constants for binary functions can be equal to only 1 or 0. The complement function produces the complement of each of the binary variables. A function that is equal to an input variable has been given the name *transfer*, because the variable x or y is transferred through the gate that forms the function without changing its value. Of the eight binary operators, two (inhibition and implication) are used by logicians, but are seldom used in computer logic. The AND and OR operators have been mentioned in conjunction with Boolean algebra. The other four functions are used extensively in the design of digital systems.

The NOR function is the complement of the OR function, and its name is an abbreviation of *not-OR*. Similarly, NAND is the complement of AND and is an abbreviation of *not-AND*. The exclusive-OR, abbreviated XOR, is similar to OR, but excludes the combination of *both* x and y being equal to 1; it holds only when x and y differ in value. (It is sometimes referred to as the binary difference operator.) Equivalence is a function that is 1 when the two binary variables are equal (i.e., when both are 0 or both are 1). The exclusive-OR and equivalence functions are the complements of each other. This can be easily verified by inspecting Table 2.7: The truth table for exclusive-OR is F_6 and for equivalence is F_9, and these two functions are the complements of each other. For this reason, the equivalence function is called exclusive-NOR, abbreviated XNOR.

Boolean algebra, as defined in Section 2.2, has two binary operators, which we have called AND and OR, and a unary operator, NOT (complement). From the definitions, we have deduced a number of properties of these operators and now have defined other binary operators in terms of them. There is nothing unique about this procedure. We could have just as well started with the operator NOR ($\downarrow$), for example, and later defined AND, OR, and NOT in terms of it. There are, nevertheless, good reasons for introducing Boolean algebra in the way it has been introduced. The concepts of "and," "or," and "not" are familiar and are used by people to express everyday logical ideas. Moreover, the Huntington postulates reflect the dual nature of the algebra, emphasizing the symmetry of $+$ and $\cdot$ with respect to each other.

2.8 DIGITAL LOGIC GATES

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates. Still, the possibility of constructing gates for the other logic operations is of practical interest. Factors to be weighed in considering the construction of other types of logic gates are (1) the feasibility and economy of producing the gate with physical components, (2) the possibility of extending the gate to more than two inputs, (3) the basic properties of the binary operator, such as commutativity and associativity, and (4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table 2.8, two are equal to a constant and four are repeated. There are only 10 functions left to be considered as candidates for logic gates. Two—inhibition and implication—are not commutative or associative and thus are impractical to use as standard logic gates. The other eight—complement, transfer, AND, OR, NAND, NOR, exclusive-OR, and equivalence—are used as standard gates in digital design.

The graphic symbols and truth tables of the eight gates are shown in Fig. 2.5. Each gate has one or two binary input variables, designated by x and y, and one binary output variable, designated by F. The AND, OR, and inverter circuits were defined in Fig. 1.6. The inverter circuit inverts the logic sense of a binary variable, producing the NOT, or complement, function. The small circle in the output of the graphic symbol of an inverter (referred to as a *bubble*) designates the logic complement. The triangle symbol by itself designates a buffer circuit. A buffer produces the *transfer* function, but does not produce a logic operation, since the binary value of the output is equal to the binary value of the input. This circuit is used for power amplification of the signal and is equivalent to two inverters connected in cascade.

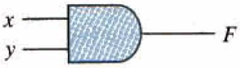
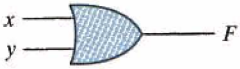
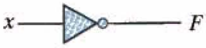
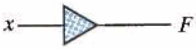
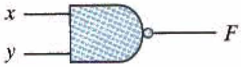
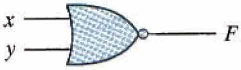
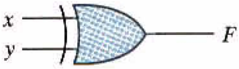
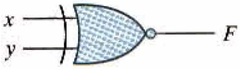
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = xy$	x y F 0 0 0 0 1 0 1 0 0 1 1 1
OR		$F = x + y$	x y F 0 0 0 0 1 1 1 0 1 1 1 1
Inverter		$F = x'$	x F 0 1 1 0
Buffer		$F = x$	x F 0 0 1 1
NAND		$F = (xy)'$	x y F 0 0 1 0 1 1 1 0 1 1 1 0
NOR		$F = (x + y)'$	x y F 0 0 1 0 1 0 1 0 0 1 1 0
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	x y F 0 0 0 0 1 1 1 0 1 1 1 0
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	x y F 0 0 1 0 1 0 1 0 0 1 1 1

FIGURE 2.5
Digital logic gates

The NAND function is the complement of the AND function, as indicated by a graphic symbol that consists of an AND graphic symbol followed by a small circle. The NOR function is the complement of the OR function and uses an OR graphic symbol followed by a small circle. NAND and NOR gates are used extensively as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because digital circuits can be easily implemented with them.

The exclusive-OR gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The equivalence, or exclusive-NOR, gate is the complement of the exclusive-OR, as indicated by the small circle on the output side of the graphic symbol.

Extension to Multiple Inputs

The gates shown in Fig. 2.5—except for the inverter and buffer—can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties. For the OR function, we have

$$x + y = y + x \quad \text{(commutative)}$$

and

$$(x + y) + z = x + (y + z) = x + y + z \quad \text{(associative)}$$

which indicates that the gate inputs can be interchanged and that the OR function can be extended to three or more variables.

The NAND and NOR functions are commutative, and their gates can be extended to have more than two inputs, provided that the definition of the operation is modified slightly. The difficulty is that the NAND and NOR operators are not associative (i.e., $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$), as shown in Fig. 2.6 and the following equations:

$$(x \downarrow y) \downarrow z = [(x + y)' + z]' = (x + y)z' = xz' + yz'$$
$$x \downarrow (y \downarrow z) = [x + (y + z)']' = x'(y + z) = x'y + x'z$$

To overcome this difficulty, we define the multiple NOR (or NAND) gate as a complemented OR (or AND) gate. Thus, by definition, we have

$$x \downarrow y \downarrow z = (x + y + z)'$$
$$x \uparrow y \uparrow z = (xyz)'$$

The graphic symbols for the three-input gates are shown in Fig. 2.7. In writing cascaded NOR and NAND operations, one must use the correct parentheses to signify the proper sequence of the gates. To demonstrate this principle, consider the circuit of Fig. 2.7(c). The Boolean function for the circuit must be written as

$$F = [(ABC)'(DE)']' = ABC + DE$$

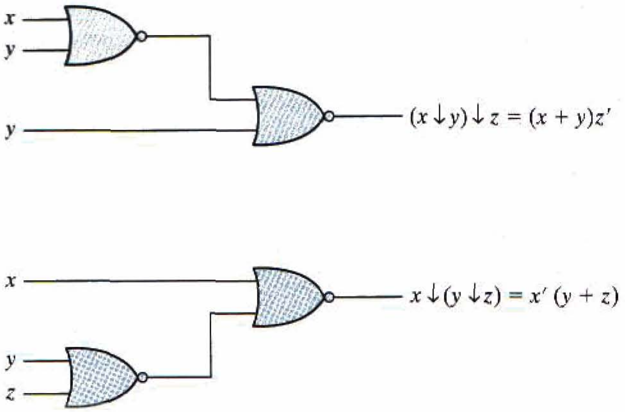
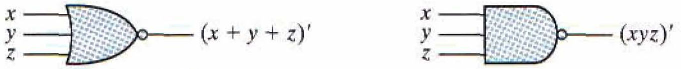
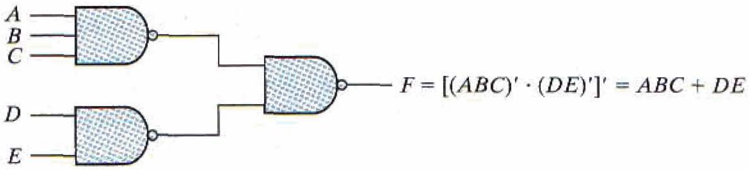


FIGURE 2.6
Demonstrating the nonassociativity of the NOR operator: $(x \downarrow y) \downarrow z \neq x \downarrow (y \downarrow z)$



(a) 3-input NOR gate (b) 3-input NAND gate

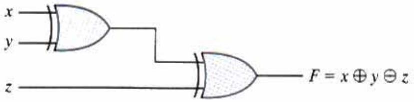


(c) Cascaded NAND gates

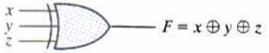
FIGURE 2.7
Multiple-input and cascaded NOR and NAND gates

The second expression is obtained from one of DeMorgan's theorems. It also shows that an expression in sum-of-products form can be implemented with NAND gates. (NAND and NOR gates are discussed further in Section 3.7.)

The exclusive-OR and equivalence gates are both commutative and associative and can be extended to more than two inputs. However, multiple-input exclusive-OR gates are uncommon from the hardware standpoint. In fact, even a two-input function is usually constructed with other types of gates. Moreover, the definition of the function must be modified when extended to more than two variables. Exclusive-OR is an *odd* function (i.e., it is equal to 1 if the input variables have an odd number of 1's). The construction of a three-input exclusive-OR function is shown in Fig. 2.8. This function is normally implemented by cascading two-input gates, as shown in (a). Graphically, it can be represented with a single three-input gate, as shown in (b). The truth table in (c) clearly indicates that the output F is equal to 1 if only one input is equal to 1 or if



(a) Using 2-input gates



(b) 3-input gate

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

(c) Truth table

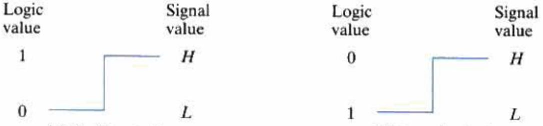
FIGURE 2.8
Three-input exclusive-OR gate

all three inputs are equal to 1 (i.e., when the total number of 1's in the input variables is *odd*). (Exclusive-OR gates are discussed further in Section 3.9.)

Positive and Negative Logic

The binary signal at the inputs and outputs of any gate has one of two values, except during transition. One signal value represents logic 1 and the other logic 0. Since two signal values are assigned to two logic values, there exist two different assignments of signal level to logic value, as shown in Fig. 2.9. The higher signal level is designated by H and the lower signal level by L. Choosing the high-level H to represent logic 1 defines a positive logic system. Choosing the low-level L to represent logic 1 defines a negative logic system. The terms *positive* and *negative* are somewhat misleading, since both signals may be positive or both may be negative. It is not the actual values of the signals that determine the type of logic, but rather the assignment of logic values to the relative amplitudes of the two signal levels.

Hardware digital gates are defined in terms of signal values such as H and L. It is up to the user to decide on a positive or negative logic polarity. Consider, for example, the electronic gate shown in Fig. 2.10(b). The truth table for this gate is listed in Fig. 2.10(a). It specifies the physical behavior of the gate when H is 3 volts and L is 0 volts. The truth table of Fig. 2.10(c) assumes a positive logic assignment, with $H = 1$ and $L = 0$. This truth table is the same as the one for the AND operation. The graphic symbol for a positive logic AND gate is shown in Fig. 2.10(d).



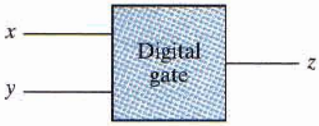
(a) Positive logic

(b) Negative logic

FIGURE 2.9
Signal assignment and logic polarity

x	y	z
L	L	L
L	H	L
H	L	L
H	H	H

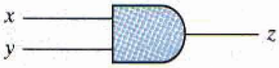
(a) Truth table with H and L



(b) Gate block diagram

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

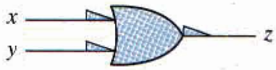
(c) Truth table for positive logic



(d) Positive logic AND gate

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

(e) Truth table for negative logic



(f) Negative logic OR gate

FIGURE 2.10
Demonstration of positive and negative logic

Now consider the negative logic assignment for the same physical gate with $L = 1$ and $H = 0$. The result is the truth table of Fig. 2.10(e). This table represents the OR operation, even though the entries are reversed. The graphic symbol for the negative-logic OR gate is shown in Fig. 2.10(f). The small triangles in the inputs and output designate a *polarity indicator*, the presence of which along a terminal signifies that negative logic is assumed for the signal. Thus, the same physical gate can operate either as a positive-logic AND gate or as a negative-logic OR gate.

The conversion from positive logic to negative logic and vice versa is essentially an operation that changes 1's to 0's and 0's to 1's in both the inputs and the output of a gate. Since this operation produces the dual of a function, the change of all terminals from one polarity to the other results in taking the dual of the function. The upshot is that all AND operations are converted to OR operations (or graphic symbols) and vice versa. In addition, one must not forget to include the polarity-indicator triangle in the graphic symbols when negative logic is assumed. In this book, we will not use negative logic gates and will assume that all gates operate with a positive logic assignment.

2.9 INTEGRATED CIRCUITS

An integrated circuit (abbreviated IC) is a silicon semiconductor crystal, called a *chip*, containing the electronic components for constructing digital gates. The various gates are interconnected inside the chip to form the required circuit. The chip is mounted in a ceramic or plastic container, and connections are welded to external pins to form the integrated circuit. The number of pins may range from 14 on a small IC package to several thousand on a larger package. Each IC has a numeric designation printed on the surface of the package for identification. Vendors provide data books, catalogs, and Internet websites that contain descriptions and information about the ICs that they manufacture.

Levels of Integration

Digital ICs are often categorized according to the complexity of their circuits, as measured by the number of logic gates in a single package. The differentiation between those chips which have a few internal gates and those having hundreds of thousands of gates is made by customary reference to a package as being either a small-, medium-, large-, or very large-scale integration device.

Small-scale integration (SSI) devices contain several independent gates in a single package. The inputs and outputs of the gates are connected directly to the pins in the package. The number of gates is usually fewer than 10 and is limited by the number of pins available in the IC.

Medium-scale integration (MSI) devices have a complexity of approximately 10 to 1,000 gates in a single package. They usually perform specific elementary digital operations. MSI digital functions are introduced in Chapter 4 as decoders, adders, and multiplexers and in Chapter 6 as registers and counters.

Large-scale integration (LSI) devices contain thousands of gates in a single package. They include digital systems such as processors, memory chips, and programmable logic devices. Some LSI components are presented in Chapter 7.

Very large-scale integration (VLSI) devices contain hundred of thousands of gates within a single package. Examples are large memory arrays and complex microcomputer chips. Because of their small size and low cost, VLSI devices have revolutionized the computer system design technology, giving the designer the capability to create structures that were previously uneconomical to build.

Digital Logic Families

Digital integrated circuits are classified not only by their complexity or logical operation, but also by the specific circuit technology to which they belong. The circuit technology is referred to as a *digital logic family*. Each logic family has its own basic electronic circuit upon which more complex digital circuits and components are developed. The basic circuit in each technology is a NAND, NOR, or inverter gate. The electronic components employed in the construction of the basic circuit are usually used to name the technology. Many different logic families of digital integrated circuits have been introduced commercially. The following are the most popular:

TTL	transistor–transistor logic;
ECL	emitter-coupled logic;
MOS	metal-oxide semiconductor;
CMOS	complementary metal-oxide semiconductor.

TTL is a logic family that has been in use for a long time and is considered to be standard. ECL has an advantage in systems requiring high-speed operation. MOS is suitable for circuits that need high component density, and CMOS is preferable in systems requiring low power consumption, such as digital cameras and other handheld portable devices. Low power consumption is essential for VLSI design; therefore, CMOS has become the dominant logic family, while TTL and ECL are declining in use. The basic electronic digital gate circuit in each logic family is analyzed in Chapter 10. The most important parameters that are evaluated and compared are discussed in Section 10.2 and are listed here for reference:

Fan-out specifies the number of standard loads that the output of a typical gate can drive without impairing its normal operation. A standard load is usually defined as the amount of current needed by an input of another similar gate in the same family.

Fan-in is the number of inputs available in a gate.

Power dissipation is the power consumed by the gate that must be available from the power supply.

Propagation delay is the average transition delay time for a signal to propagate from input to output. For example, if the input of an inverter switches from 0 to 1, the output will switch from 1 to 0, but after a time determined by the propagation delay of the device. The operating speed is inversely proportional to the propagation delay.

Noise margin is the maximum external noise voltage added to an input signal that does not cause an undesirable change in the circuit output.

Computer-Aided Design

Integrated circuits having submicron geometric features are manufactured by optically projecting patterns of light onto silicon wafers. Prior to exposure, the wafers are coated with a photoresistive material that either hardens or softens when exposed to light. Removing extraneous photoresist leaves patterns of exposed silicon. The exposed regions are then implanted with dopant atoms to create a semiconductor material having the electrical properties of transistors and the logical properties of gates. The design process translates a functional specification or description of the circuit (i.e., what it must do) into a physical specification or description (how it must be implemented in silicon).

The design of digital systems with VLSI circuits containing millions of transistors and gates is an enormous and formidable task. Systems of this complexity are usually impossible to develop and verify without the assistance of computer-aided design (CAD) tools, which consist of software programs that support computer-based representations of circuits and aid in the development of digital hardware by automating the design process. Electronic design automation (EDA) covers all phases of the design of integrated circuits. A typical design flow for creating VLSI circuits consists of a sequence of steps beginning with design entry (e.g., entering a schematic) and culminating with the generation of the database that contains the photomask used to fabricate the IC. There are a variety of options available for creating the physical realization of a digital circuit in silicon. The designer can choose between an application-specific integrated circuit (ASIC), a field-programmable gate array (FPGA), a programmable logic device (PLD), and a full-custom IC. With each of these devices comes a set of CAD tools that provide the necessary software to facilitate the hardware fabrication of the unit. Each of these technologies has a market niche determined by the size of the market and the unit cost of the devices that are required to implement a design.

Some CAD systems include an editing program for creating and modifying schematic diagrams on a computer screen. This process is called *schematic capture* or *schematic entry*. With the aid of menus, keyboard commands, and a mouse, a schematic editor can draw circuit diagrams of digital circuits on the computer screen. Components can be placed on the screen from a list in an internal library and can then be connected with lines that represent wires. The schematic entry software creates and manages a database containing the information produced with the schematic. Primitive gates and functional blocks have associated models that allow the functionality (i.e., logical behavior) and timing of the circuit to be verified. Verification is performed by applying inputs to the circuit and using a logic simulator to determine and display the outputs in text or waveform format.

An important development in the design of digital systems is the use of a hardware description language (HDL). Such a language resembles a computer programming language, but is specifically oriented to describing digital hardware. It represents logic diagrams and other digital information in textual form to describe the functionality and structure of a circuit. Moreover, the HDL description of a circuit's functionality can be abstract, without reference to specific hardware, thereby freeing a designer to devote attention to higher level functional detail (e.g., under certain conditions the circuit must detect a particular pattern of 1's and 0's in a serial bit stream of data) rather than transistor-level detail. HDL-based models of a circuit or system are simulated to check and verify its functionality before it is submitted to fabrication, thereby reducing the risk and waste of manufacturing a circuit that fails to operate correctly. In tandem with the emergence of HDL-based design languages, tools have been developed to automatically and optimally synthesize the logic described by an HDL model of a circuit. These two advances in technology have led to an almost total reliance by industry on HDL-based synthesis tools and methodologies for the design of the circuits of complex degital systems. Two hardware description languages—Verilog and VHDL—have been approved as standards by the Institute of Electronics and Electrical Engineers (IEEE) and are in use by design teams worldwide. The Verilog HDL is introduced in Section 3.10, and because of its importance, we include several exercises and design problems based on Verilog throughout the book.

PROBLEMS

Answers to problems marked with * appear at the end of the book.

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) $xyz' + x'yz + xyz + x'yz'$ (f) $(x + y + z')(x' + y' + z)$

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) $(x + y' + z')(x' + z')$

2.4 Reduce the following Boolean expressions to the indicated number of literals:

(a)* $A'C' + ABC + AC'$ to three literals

(b)* $(x'y' + z)' + z + xy + wz$ 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) $ABCD + A'BD + ABC'D$ 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) $(A'B + CD)E' + E$

(c) $(x' + y + z')(x + y')(x + z)$

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) $F = x'z' + yz$

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.

2.13 Draw logic diagrams to implement the following Boolean expressions:

(a) $Y = A + B + B'(A + C')$
(b) $Y = A(B \oplus D) + C'$
(c) $Y = A + CD + ABC$
(d) $Y = (A \oplus C)' + B$
(e) $Y = (A' + B')(C + D')$
(f) $Y = [(A + B')(C' + D)]$

2.14 Implement the Boolean function

$$F = xy + x'y' + y'z$$

(a) with AND, OR, and inverter gates,
(b)* with OR and inverter gates,
(c) with AND and inverter gates,
(d) with NAND and inverter gates, and
(e) with NOR and inverter gates.

2.15* Simplify the following Boolean functions T_1 and T_2 to a minimum number of literals:

A	*B*	*C*	T_1	T_2
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

2.16 The logical sum of all minterms of a Boolean function of n variables is 1.

(a) Prove the previous statement for $n = 3$.
(b) Suggest a procedure for a general proof.

2.17 Obtain the truth table of the following functions, and express each function in sum-of-minterms and product-of-maxterms form:

(a)* $(xy + z)(y + xz)$
(b) $(x + y')(y' + z)$
(c) $x'z + wx'y + wyz' + w'y'$
(d) $(xy + yz' + x'z)(x + z)$

2.18 For the Boolean function

$$F = xy'z + x'y'z + w'xy + wx'y + wxy$$

(a) Obtain the truth table of *F*.
(b) Draw the logic diagram, using the original Boolean expression.
(c)* Use Boolean algebra to simplify the function to a minimum number of literals.
(d) Obtain the truth table of the function from the simplified expression and show that it is the same as the one in part (a).
(e) Draw the logic diagram from the simplified expression, and compare the total number of gates with the diagram of part (b).

2.19* Express the following function as a sum of minterms and as a product of maxterms:

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

2.20 Express the complement of the following functions in sum-of-minterms form:
(a) $F(A, B, C, D) = \Sigma(3, 5, 9, 11, 15)$ (b) $F(x, y, z) = \Pi(2, 4, 5, 7)$

2.21 Convert each of the following to the other canonical form:
(a) $F(x, y, z) = \Sigma(2, 5, 6)$ (b) $F(A, B, C, D) = \Pi(0, 1, 2, 4, 7, 9, 12)$

2.22* Convert each of the following expressions into sum of products and product of sums:
(a) $(AB + C)(B + C'D)$ (b) $x' + x(x + y')(y + z')$

2.23 Draw the logic diagram corresponding to the following Boolean expressions without simplifying them:
(a) $BC' + AB + ACD$ (b) $(A + B)(C + D)(A' + B + D)$
(c) $(AB + A'B')(CD' + C'D)$ (d) $A + CD + (A + D')(C' + D)$

2.24 Show that the dual of the exclusive-OR is equal to its complement.

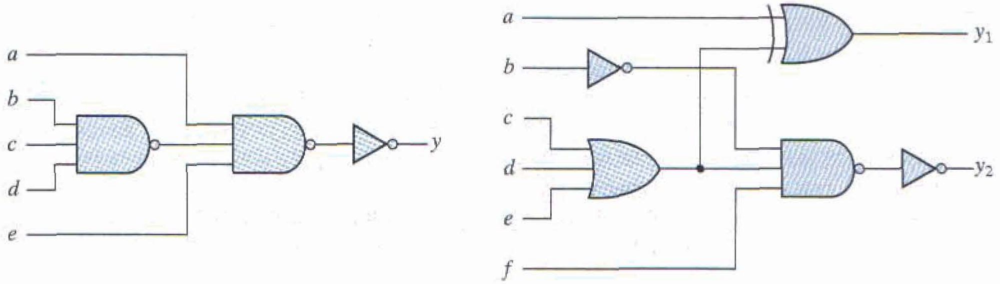
2.25 By substituting the Boolean expression equivalent of the binary operations as defined in Table 2.8, show the following:
(a) The inhibition operation is neither commutative nor associative.
(b) The exclusive-OR operation is commutative and associative.

2.26 Show that a positive logic NAND gate is a negative logic NOR gate and vice versa.

2.27 Write the Boolean equations and draw the logic diagram of the circuit whose outputs are defined by the following truth table:

f_1	f_2	a	b	c
1	0	0	0	0
0	0	0	0	1
0	1	0	1	0
1	1	0	1	1
0	1	1	0	0
0	1	1	0	1
1	1	1	1	0
1	0	1	1	1

2.28 Write Boolean expressions and construct the truth tables describing the outputs of the circuits described by the following logic diagrams:



(a) (b)

REFERENCES

1. Boole, G. 1854. *An Investigation of the Laws of Thought.* New York: Dover.
2. Dietmeyer, D. L. 1988. *Logic Design of Digital Systems*, 3d ed. Boston: Allyn and Bacon.
3. Huntington, E. V. Sets of independent postulates for the algebra of logic. *Trans. Am. Math. Soc.*, **5** (1904): 288–309.
4. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, Language Reference Manual (LRM), IEEE Std.1364-1995, 1996, 2001, 2005, The Institute of Electrical and Electronics Engineers, Piscataway, NJ.
5. *IEEE Standard VHDL Language Reference Manual* (LRM), IEEE Std, 1076-1987, 1988, The Institute of Electrical and Electronics Engineers, Piscataway, NJ.
6. Mano, M. M., and C. R. Kime. 2000. *Logic and Computer Design Fundamentals*, 2d ed. Upper Saddle River, NJ: Prentice Hall.
7. Shannon, C. E. A symbolic analysis of relay and switching circuits. *Trans. AIEE* 57 (1938): 713–723.

Chapter 3

Gate-Level Minimization

3.1 INTRODUCTION

Gate-level minimization refers to the design task of finding an optimal gate-level implementation of the Boolean functions describing a digital circuit. This task is well understood, but is difficult to execute by manual methods when the logic has more than a few inputs. Fortunately, computer-based logic synthesis tools can minimize a large set of Boolean equations efficiently and quickly. Nevertheless, it is important that a designer understand the underlying mathematical description and solution of the problem. This chapter serves as a foundation for your understanding of that important topic and will enable you to execute a manual design of simple circuits, preparing you for skilled use of modern design tools. The chapter will also introduce a hardware description language that is used by modern design tools.

3.2 THE MAP METHOD

The complexity of the digital logic gates that implement a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented. Although the truth table representation of a function is unique, when it is expressed algebraically it can appear in many different, but equivalent, forms. Boolean expressions may be simplified by algebraic means as discussed in Section 2.4. However, this procedure of minimization is awkward because it lacks specific rules to predict each succeeding step in the manipulative process. The map method presented here provides a simple, straightforward procedure for minimizing Boolean functions. This method may be regarded as a pictorial form of a truth table. The map method is also known as the *Karnaugh map* or *K-map*.

A K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. Since any Boolean function can be expressed as a sum of minterms, it follows that a Boolean function is recognized graphically in the map from the area enclosed by those squares whose minterms are included in the function. In fact, the map presents a visual diagram of all possible ways a function may be expressed in standard form. By recognizing various patterns, the user can derive alternative algebraic expressions for the same function, from which the simplest can be selected.

The simplified expressions produced by the map are always in one of the two standard forms: sum of products or product of sums. It will be assumed that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate. We will see subsequently that the simplest expression is not unique: It is sometimes possible to find two or more expressions that satisfy the minimization criteria. In that case, either solution is satisfactory.

Two-Variable Map

The two-variable map is shown in Fig. 3.1(a). There are four minterms for two variables; hence, the map consists of four squares, one for each minterm. The map is redrawn in (b) to show the relationship between the squares and the two variables x and y. The 0 and 1 marked in each row and column designate the values of variables. Variable x appears primed in row 0 and unprimed in row 1. Similarly, y appears primed in column 0 and unprimed in column 1.

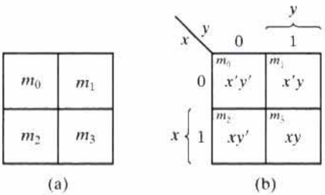
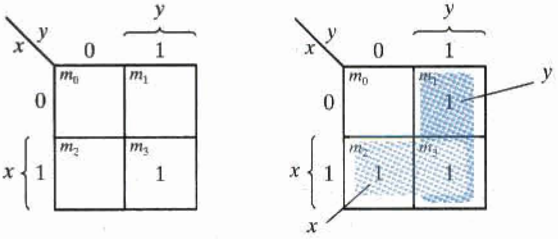


FIGURE 3.1
Two-variable map

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables. As an example, the function xy is shown in Fig. 3.2(a). Since xy is equal to m_3, a 1 is placed inside the square that belongs to m_3. Similarly, the function $x + y$ is represented in the map of Fig. 3.2(b) by three squares marked with 1's. These squares are found from the minterms of the function:

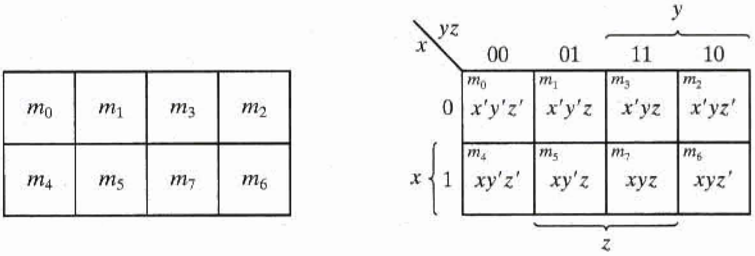
$$m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$$

The three squares could also have been determined from the intersection of variable x in the second row and variable y in the second column, which encloses the area belonging to x or y. In each example, the minterms at which the function is asserted are marked with a 1.



(a) xy (b) $x + y$

FIGURE 3.2
Representation of functions in the map



(a) (b)

FIGURE 3.3
Three-variable map

Three-Variable Map

A three-variable map is shown in Fig. 3.3. There are eight minterms for three binary variables; therefore, the map consists of eight squares. Note that the minterms are arranged, not in a binary sequence, but in a sequence similar to the Gray code (Table 1.6). The characteristic of this sequence is that only one bit changes in value from one adjacent column to the next. The map drawn in part (b) is marked with numbers in each row and each column to show the relationship between the squares and the three variables. For example, the square assigned to m_5 corresponds to row 1 and column 01. When these two numbers are concatenated, they give the binary number 101, whose decimal equivalent is 5. Each cell of the map corresponds to a unique minterm, so another way of looking at square $m_5 = xy'z$ is to consider it to be in the row marked x and the column belonging to $y'z$ (column 01). Note that there are four squares in which each variable is equal to 1 and four in which each is equal to 0. The variable appears unprimed in the former four squares and primed in the latter. For convenience, we write the variable with its letter symbol under the four squares in which it is unprimed.

To understand the usefulness of the map in simplifying Boolean functions, we must recognize the basic property possessed by adjacent squares: Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other. For example, m_5 and m_7 lie in two adjacent squares. Variable y is primed in m_5 and unprimed in m_7, whereas the other two variables are the same in both squares. From the postulates of Boolean algebra, it follows that the sum of two minterms in adjacent squares can be simplified to a single AND term consisting of only two literals. To clarify this concept, consider the sum of two adjacent squares such as m_5 and m_7:

$$m_5 + m_7 = xy'z + xyz = xz(y' + y) = xz$$

Here, the two squares differ by the variable y, which can be removed when the sum of the two minterms is formed. Thus, any two minterms in adjacent squares (vertically or horizontally, but not diagonally, adjacent) that are ORed together will cause a removal of the dissimilar variable. The next four examples explain the procedure for minimizing a Boolean function with a map.

EXAMPLE 3.1

Simplify the Boolean function

$$F(x, y, z) = \Sigma(2, 3, 4, 5)$$

First, a 1 is marked in each minterm that represents the function. This is shown in Fig. 3.4, in which the squares for minterms 010, 011, 100, and 101 are marked with 1's. The next step is to find possible adjacent squares. These are indicated in the map by two rectangles, each enclosing two 1's. The upper right rectangle represents the area enclosed by $x'y$. This area is determined by observing that the two-square area is in row 0, corresponding to x', and the last two columns, corresponding to y. Similarly, the lower left rectangle represents the product term xy'. (The second row represents x and the two left columns represent y'.) The logical sum of these two product terms gives the simplified expression

$$F = x'y + xy'$$

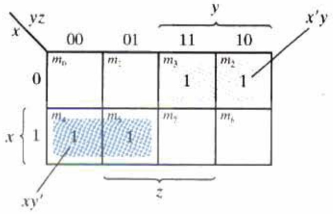


FIGURE 3.4
Map for Example 3.1, $F(x, y, z) = \Sigma(2, 3, 4, 5) = x'y + xy'$

■

In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other. In Fig. 3.3, m_0 is adjacent to m_2 and m_4 is adjacent to m_6 because the minterms differ by one variable. This difference can be readily verified algebraically:

$$m_0 + m_2 = x'y'z' + x'yz' = x'z'(y' + y) = x'z'$$
$$m_4 + m_6 = xy'z' + xyz' = xz' + (y' + y) = xz'$$

Consequently, we must modify the definition of adjacent squares to include this and other similar cases. We do so by considering the map as being drawn on a surface in which the right and left edges touch each other to form adjacent squares.

EXAMPLE 3.2

Simplify the Boolean function

$$F(x, y, z) = \Sigma(3, 4, 6, 7)$$

The map for this function is shown in Fig. 3.5. There are four squares marked with 1's, one for each minterm of the function. Two adjacent squares are combined in the third column to give a two-literal term yz. The remaining two squares with 1's are also adjacent by the new definition. These two squares, when combined, give the two-literal term xz'. The simplified function then becomes

$$F = yz + xz'$$

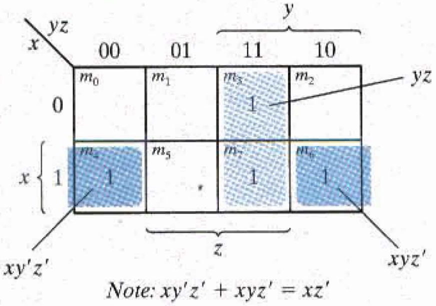


FIGURE 3.5
Map for Example 3.2, $F(x, y, z) = \Sigma(3, 4, 6, 7) = yz + xz'$

■

Consider now any combination of four adjacent squares in the three-variable map. Any such combination represents the logical sum of four minterms and results in an expression with only one literal. As an example, the logical sum of the four adjacent minterms 0, 2, 4, and 6 reduces to the single literal term z':

$$\begin{aligned} m_0 + m_2 + m_4 + m_6 &= x'y'z' + x'yz' + xy'z' + xyz' \\ &= x'z'(y' + y) + xz'(y' + y) \\ &= x'z' + xz' = z'(x' + x) = z' \end{aligned}$$

The number of adjacent squares that may be combined must always represent a number that is a power of two, such as 1, 2, 4, and 8. As more adjacent squares are combined, we obtain a product term with fewer literals.

One square represents one minterm, giving a term with three literals.

Two adjacent squares represent a term with two literals.

Four adjacent squares represent a term with one literal.

Eight adjacent squares encompass the entire map and produce a function that is always equal to 1.

EXAMPLE 3.3

Simplify the Boolean function

$$F(x, y, z) = \Sigma(0, 2, 4, 5, 6)$$

The map for F is shown in Fig. 3.6. First, we combine the four adjacent squares in the first and last columns to give the single literal term z'. The remaining single square, representing minterm 5, is combined with an adjacent square that has already been used once. This is not only permissible, but rather desirable, because the two adjacent squares give the two-literal term xy' and the single square represents the three-literal minterm $xy'z$. The simplified function is

$$F = z' + xy'$$

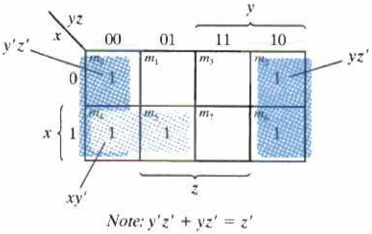


FIGURE 3.6
Map for Example 3.3, $F(x, y, z) = \Sigma(0, 2, 4, 5, 6) = z' + xy'$

■

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

Let 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.

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

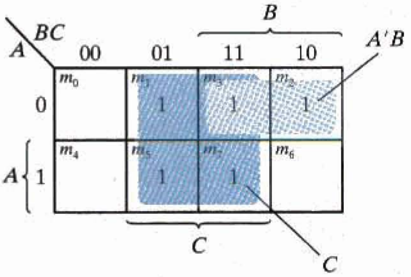


FIGURE 3.7
Map for Example 3.4, $A'C + A'B + AB'C + BC = C + A'B$

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:

$$F = C + A'B$$

■

3.3 FOUR-VARIABLE MAP

The map for Boolean functions of four binary variables is shown in Fig. 3.8. In (a) are listed the 16 minterms and the squares assigned to each. In (b), the map is redrawn to show the relationship between the squares and the four variables. The rows and columns are numbered in a Gray code sequence, with only one digit changing value between two adjacent rows or columns. The minterm corresponding to each square can be obtained from the concatenation of the row number with the column number. For example, the numbers of the third row (11) and the second column (01), when concatenated, give the binary number 1101, the binary equivalent of decimal 13. Thus, the square in the third row and second column represents minterm m_{13}.

The map minimization of four-variable Boolean functions is similar to the method used to minimize three-variable functions. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent squares. For example,

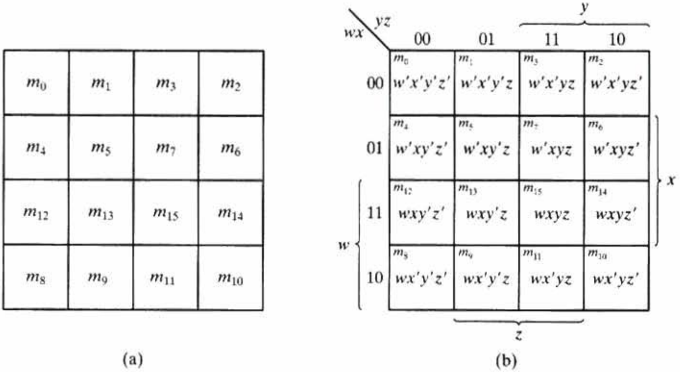


FIGURE 3.8
Four-variable map

m_0 and m_2 form adjacent squares, as do m_3 and m_{11}. The combination of adjacent squares that is useful during the simplification process is easily determined from inspection of the four-variable map:

One square represents one minterm, giving a term with four literals.

Two adjacent squares represent a term with three literals.

Four adjacent squares represent a term with two literals.

Eight adjacent squares represent a term with one literal.

Sixteen adjacent squares produce a function that is always equal to 1.

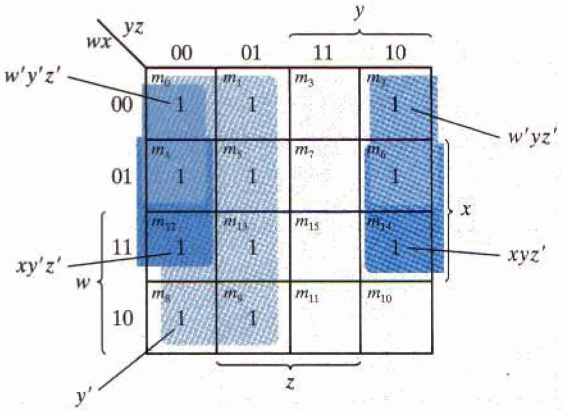
No other combination of squares can simplify the function. The next two examples show the procedure used to simplify four-variable Boolean functions.

EXAMPLE 3.5

Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$$

Since the function has four variables, a four-variable map must be used. The minterms listed in the sum are marked by 1's in the map of Fig. 3.9. Eight adjacent squares marked with 1's can be combined to form the one literal term y'. The remaining three 1's on the right cannot be combined to give a simplified term; they must be combined as two or four adjacent squares. The larger the number of squares combined, the smaller is the number of literals in the term. In this example, the top two 1's on the right are combined with the top two 1's on the left to give the term $w'z'$. Note that it is permissible to use the same square more than once. We are



Note: $w'y'z' + w'yz' = w'z'$
$xy'z' + xyz' = xz'$

FIGURE 3.9
Map for Example 3.5, $F(w, x, y, z) = \Sigma(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14) = y' + w'z' + xz'$

now left with a square marked by 1 in the third row and fourth column (square 1110). Instead of taking this square alone (which will give a term with four literals), we combine it with squares already used to form an area of four adjacent squares. These squares make up the two middle rows and the two end columns, giving the term xz'. The simplified function is

$$F = y' + w'z' + xz'$$

■

EXAMPLE 3.6

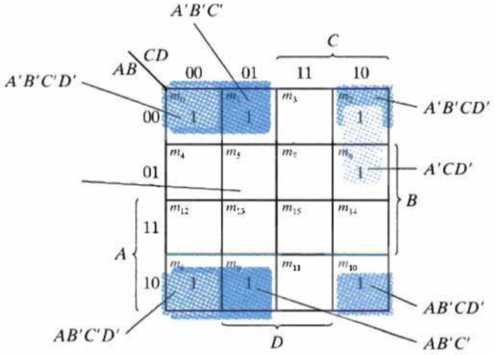
Simplify the Boolean function

$$F = A'B'C' + B'CD' + A'BCD' + AB'C'$$

The area in the map covered by this function consists of the squares marked with 1's in Fig. 3.10. The function has four variables and, as expressed, consists of three terms with three literals each and one term with four literals. Each term with three literals is represented in the map by two squares. For example, $A'B'C'$ is represented in squares 0000 and 0001. The function can be simplified in the map by taking the 1's in the four corners to give the term $B'D'$. This is possible because these four squares are adjacent when the map is drawn in a surface with top and bottom edges, as well as left and right edges, touching one another. The two left-hand 1's in the top row are combined with the two 1's in the bottom row to give the term $B'C'$. The remaining 1 may be combined in a two-square area to give the term $A'CD'$. The simplified function is

$$F = B'D' + B'C' + A'CD'$$

■



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$
$A'B'C' + AB'C' = B'C'$

FIGURE 3.10
Map for Example 3.6, $A'B'C' + B'CD' + A'BCD' + AB'C' = B'D' + B'C' + A'CD'$

Prime Implicants

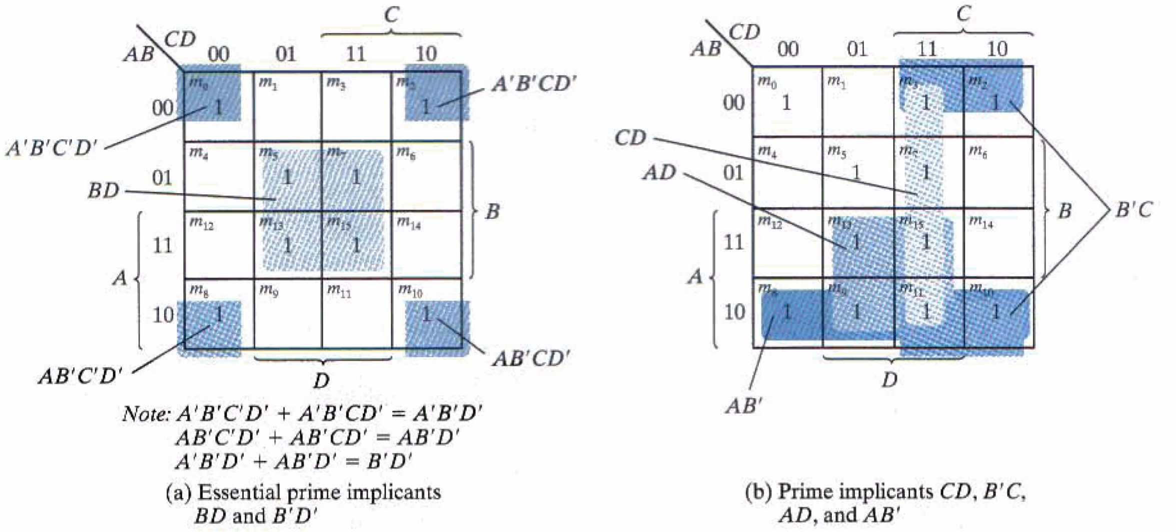
In choosing adjacent squares in a map, we must ensure that (1) all the minterms of the function are covered when we combine the squares, (2) the number of terms in the expression is minimized, and (3) there are no redundant terms (i.e., minterms already covered by other terms). Sometimes there may be two or more expressions that satisfy the simplification criteria. The procedure for combining squares in the map may be made more systematic if we understand the meaning of two special types of terms. A *prime implicant* is a product term obtained by combining the maximum possible number of adjacent squares in the map. If a minterm in a square is covered by only one prime implicant, that prime implicant is said to be *essential.*

The prime implicants of a function can be obtained from the map by combining all possible maximum numbers of squares. This means that a single 1 on a map represents a prime implicant if it is not adjacent to any other 1's. Two adjacent 1's form a prime implicant, provided that they are not within a group of four adjacent squares. Four adjacent 1's form a prime implicant if they are not within a group of eight adjacent squares, and so on. The essential prime implicants are found by looking at each square marked with a 1 and checking the number of prime implicants that cover it. The prime implicant is essential if it is the only prime implicant that covers the minterm.

Consider the following four-variable Boolean function:

$$F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

The minterms of the function are marked with 1's in the maps of Fig. 3.11. The partial map (part (a) of the figure) shows two essential prime implicants, each formed by collapsing four cells into a term having only two literals. One term is essential because there is only one way to include



Note: $A'B'C'D' + A'B'CD' = A'B'D'$
$AB'C'D' + AB'CD' = AB'D'$
$A'B'D' + AB'D' = B'D'$

(a) Essential prime implicants BD and $B'D'$

(b) Prime implicants CD, $B'C$, AD, and AB'

FIGURE 3.11
Simplification using prime implicants

minterm m_0 within four adjacent squares. These four squares define the term $B'D'$. Similarly, there is only one way that minterm m_5 can be combined with four adjacent squares, and this gives the second term BD. The two essential prime implicants cover eight minterms. The three minterms that were omitted from the partial map (m_3, m_9, and m_{11}) must be considered next.

Figure 3.11(b) shows all possible ways that the three minterms can be covered with prime implicants. Minterm m_3 can be covered with either prime implicant CD or prime implicant $B'C$. Minterm m_9 can be covered with either AD or AB'. Minterm m_{11} is covered with any one of the four prime implicants. The simplified expression is obtained from the logical sum of the two essential prime implicants and any two prime implicants 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.

3.4 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.

The five-variable map is shown in Fig. 3.12. It consists of 2 four-variable maps with variables A, B, C, D, and E. Variable A distinguishes between the two maps, as indicated at the top of the diagram. The left-hand four-variable map represents the 16 squares in which $A = 0$, and the other four-variable map represents the squares in which $A = 1$. Minterms 0 through 15 belong with $A = 0$ and minterms 16 through 31 with $A = 1$. Each four-variable map retains the previously defined adjacency when taken separately. In addition, each square in the $A = 0$ map is adjacent to the corresponding square in the $A = 1$ map. For example, minterm 4 is adjacent to minterm 20 and minterm 15 to 31. The best way to visualize this new rule for adjacent squares is to consider the two half maps as being one on top of the other. Any two squares that fall one over the other are considered adjacent.

By following the procedure used for the five-variable map, it is possible to construct a six-variable map with 4 four-variable maps to obtain the required 64 squares. Maps with six or more variables need too many squares and are impractical to use. The alternative is to employ computer programs specifically written to facilitate the simplification of Boolean functions with a large number of variables.

By inspection, and taking into account the new definition of adjacent squares, it is possible to show that any 2^k adjacent squares, for $k = (0, 1, 2, \ldots, n)$ in an n-variable map, will represent an area that gives a term of $n - k$ literals. For this statement to have any meaning, however, n must be larger than k. When $n = k$, the entire area of the map is combined to give the

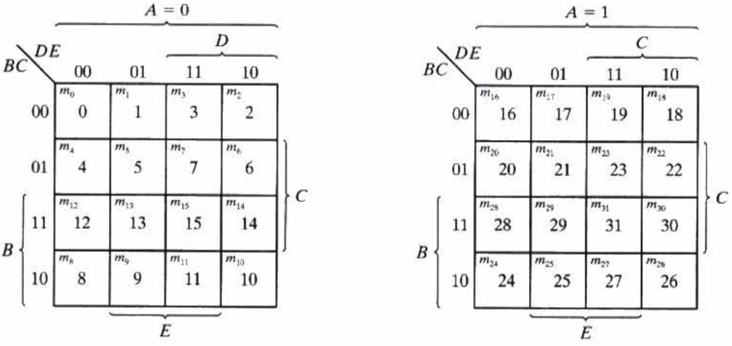

A = 0
DE
BC
D
00 01 11 10
00 01 11 10
B
C
E
m0 0, m1 1, m3 3, m2 2
m4 4, m5 5, m7 7, m6 6
m12 12, m13 13, m15 15, m14 14
m8 8, m9 9, m11 11, m10 10
A = 1
DE
BC
C
00 01 11 10
00 01 11 10
B
C
E
m16 16, m17 17, m19 19, m18 18
m20 20, m21 21, m23 23, m22 22
m28 28, m29 29, m31 31, m30 30
m24 24, m25 25, m27 27, m26 26

FIGURE 3.12
Five-variable map

Table 3.1
The Relationship between the Number of Adjacent Squares and the Number of Literals in the Term

	Number of Adjacent Squares	Number of Literals in a Term in an n-variable Map			
K	2^k	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0	1	2	3	4	5
1	2	1	2	3	4
2	4	0	1	2	3
3	8		0	1	2
4	16			0	1
5	32				0

identity function. Table 3.1 shows the relationship between the number of adjacent squares and the number of literals in the term. For example, eight adjacent squares combine an area in the five-variable map to give a term of two literals.

EXAMPLE 3.7

Simplify the Boolean function

$$F(A, B, C, D, E) = \Sigma(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$$

The five-variable map for this function is shown in Fig. 3.13. There are six minterms from 0 to 15 that belong to the part of the map with $A = 0$. The other five minterms belong with $A = 1$. Four adjacent squares in the $A = 0$ map are combined to give the three-literal term $A'B'E'$. Note that it is necessary to include A' with the term because all the squares are associated with $A = 0$. The two squares in column 01 and the last two rows are common to both parts of the map. Therefore, they constitute four adjacent squares and give the three-literal term $BD'E$. Variable A is not included here because the adjacent squares belong to both $A = 0$ and $A = 1$. The term ACE is obtained from the four adjacent squares that are entirely within the $A = 1$ map. The simplified function is the logical sum of the three terms:

$$F = A'B'E' + BD'E + ACE$$

■

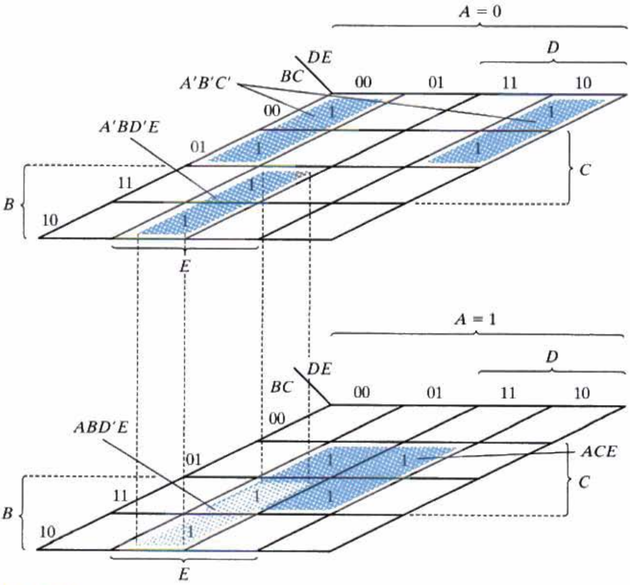


FIGURE 3.13
Map for Example 3.7, $F = A'B'E' + BD'E + ACE$

3.5 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 expression of the complement of the function (i.e., of F'). The complement of F' gives us back the function F. 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.

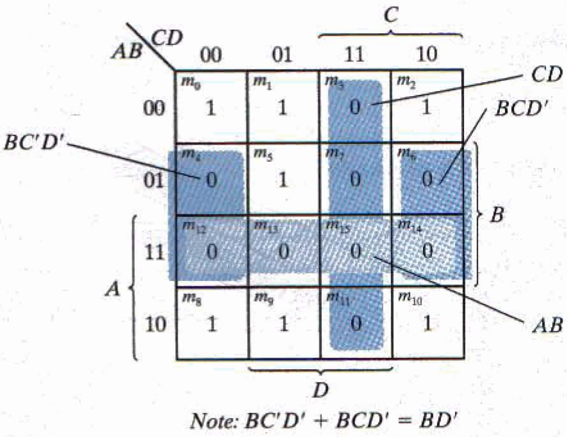


FIGURE 3.14
Map for Example 3.8, $F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$

EXAMPLE 3.8

Simplify the following Boolean function into (a) sum-of-products form and (b) product-of-sums form:

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10)$$

The 1's marked in the map of Fig. 3.14 represent all the minterms of the function. The squares marked with 0's represent the minterms not included in F and therefore denote the complement of F. Combining the squares with 1's gives the simplified function in sum-of-products form:

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

If the squares marked with 0's are combined, as shown in the diagram, we obtain the simplified complemented function:

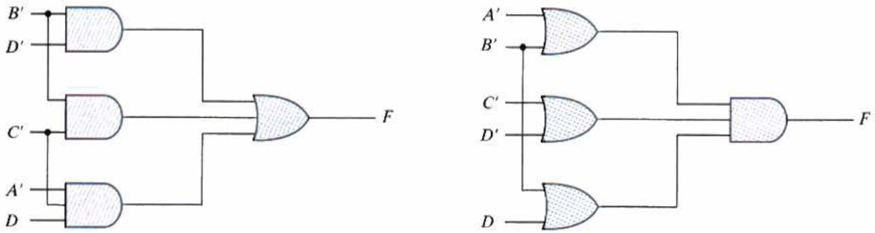
$$F' = AB + CD + BD'$$

Applying DeMorgan's theorem (by taking the dual and complementing each literal as described in Section 2.4), we obtain the simplified function in product-of-sums form:

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

■

The implementation of the simplified expressions obtained in Example 3.8 is shown in Fig. 3.15. The sum-of-products expression is implemented in (a) with a group of AND gates, one for each AND term. The outputs of the AND gates are connected to the inputs of a single OR gate. The same function is implemented in (b) in its product-of-sums form with a group of OR gates, one for each OR term. The outputs of the OR gates are connected to the inputs of a single AND gate. In each case, it is assumed that the input variables are directly



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

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

FIGURE 3.15
Gate implementations of the function of Example 3.8

Table 3.2
Truth Table of Function F

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

available in their complement, so inverters are not needed. The configuration pattern established in Fig. 3.15 is the general form by which any Boolean function is implemented when expressed in one of the standard forms. AND gates are connected to a single OR gate when in sum-of-products form; OR gates are connected to a single AND gate when in product-of-sums form. Either configuration forms two levels of gates. Thus, the implementation of a function in a standard form is said to be a two-level implementation.

Example 3.8 showed the procedure for obtaining the product-of-sums simplification when the function is originally expressed in the sum-of-minterms canonical form. The procedure is also valid when the function is originally expressed in the product-of-maxterms canonical form. Consider, for example, the truth table that defines the function F in Table 3.2. In sum-of-minterms form, this function is expressed as

$$F(x, y, z) = \Sigma(1, 3, 4, 6)$$

In product-of-maxterms form, it is expressed as

$$F(x, y, z) = \Pi(0, 2, 5, 7)$$

In other words, the 1's of the function represent the minterms and the 0's represent the maxterms. The map for this function is shown in Fig. 3.16. One can start simplifying the function by first marking the 1's for each minterm that the function is a 1. The remaining squares are

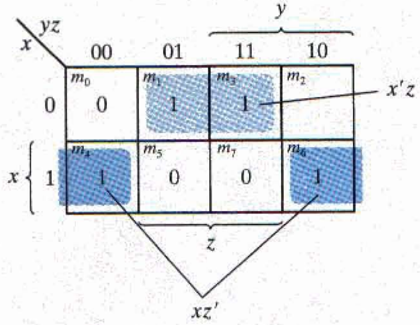


FIGURE 3.16
Map for the function of Table 3.2

marked by 0's. If, instead, the product of maxterms is initially given, one can start marking 0's in those squares listed in the function; the remaining squares are then marked by 1's. Once the 1's and 0's are marked, the function can be simplified in either one of the standard forms. For the sum of products, we combine the 1's to obtain

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

For the product of sums, we combine the 0's to obtain the simplified complemented function

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

which shows that the exclusive-OR function is the complement of the equivalence function (Section 2.6). Taking the complement of F', we obtain the simplified function in product-of-sums form:

$$F = (x' + z')(x + z)$$

To enter a function expressed in product-of-sums form into the map, use the complement of the function to find the squares that are to be marked by 0's. For example, the function

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

can be entered into the map by first taking its complement, namely,

$$F' = ABC + B'D'$$

and then marking 0's in the squares representing the minterms of F'. The remaining squares are marked with 1's.

3.6 DON'T-CARE CONDITIONS

The logical sum of the minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to 0 for the rest of the minterms. This pair of conditions assumes that all the combinations of the values for the variables of the function are valid. In practice, in some applications the function is not specified for certain combinations of the variables. As an example, the four-bit binary code for the decimal digits has six combinations that are not used and consequently are considered to be unspecified.

Functions that have unspecified outputs for some input combinations are called *incompletely specified functions*. In most applications, we simply don't care what value is assumed by the function for the unspecified minterms. For this reason, it is customary to call the unspecified minterms of a function *don't-care conditions*. These don't-care conditions can be used on a map to provide further simplification of the Boolean expression.

A don't-care minterm is a combination of variables whose logical value is not specified. Such a minterm cannot be marked with a 1 in the map, because it would require that the function always be a 1 for such a combination. Likewise, putting a 0 on the square requires the function to be 0. To distinguish the don't-care condition from 1's and 0's, an X is used. Thus, an X inside a square in the map indicates that we don't care whether the value of 0 or 1 is assigned to F for the particular minterm.

In choosing adjacent squares to simplify the function in a map, the don't-care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't-care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

EXAMPLE 3.9

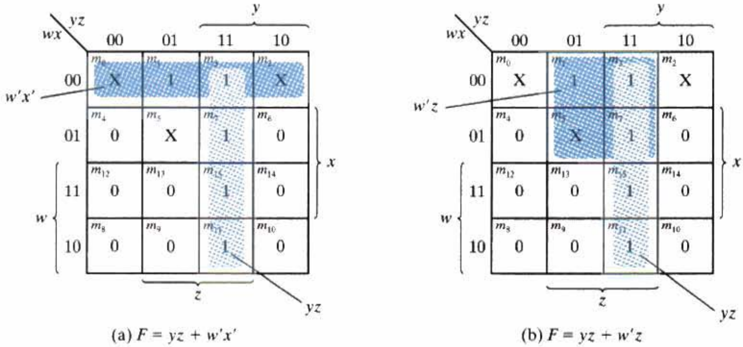
Simplify the Boolean function

$$F(w, x, y, z) = \Sigma(1, 3, 7, 11, 15)$$

which has the don't-care conditions

$$d(w, x, y, z) = \Sigma(0, 2, 5)$$

The minterms of F are the variable combinations that make the function equal to 1. The minterms of d are the don't-care minterms that may be assigned either 0 or 1. The map simplification is shown in Fig. 3.17. The minterms of F are marked by 1's, those of d are marked



(a) $F = yz + w'x'$

(b) $F = yz + w'z$

FIGURE 3.17
Example with don't-care conditions

by X's, and the remaining squares are filled with 0's. To get the simplified expression in sum-of-products form, we must include all five 1's in the map, but we may or may not include any of the X's, depending on the way the function is simplified. The term yz covers the four minterms in the third column. The remaining minterm, m_1, can be combined with minterm m_3 to give the three-literal term $w'x'z$. However, by including one or two adjacent X's we can combine four adjacent squares to give a two-literal term. In part (a) of the diagram, don't-care minterms 0 and 2 are included with the 1's, resulting in the simplified function

$$F = yz + w'x'$$

In part (b), don't-care minterm 5 is included with the 1's, and the simplified function is now

$$F = yz + w'z$$

Either one of the preceding two expressions satisfies the conditions stated for this example. ■

The previous example has shown that the don't-care minterms in the map are initially marked with X's and are considered as being either 0 or 1. The choice between 0 and 1 is made depending on the way the incompletely specified function is simplified. Once the choice is made, the simplified function obtained will consist of a sum of minterms that includes those minterms which were initially unspecified and have been chosen to be included with the 1's. Consider the two simplified expressions obtained in Example 3.9:

$$F(w, x, y, z) = yz + w'x' = \Sigma(0, 1, 2, 3, 7, 11, 15)$$
$$F(w, x, y, z) = yz + w'z = \Sigma(1, 3, 5, 7, 11, 15)$$

Both expressions include minterms 1, 3, 7, 11, and 15 that make the function F equal to 1. The don't-care minterms 0, 2, and 5 are treated differently in each expression. The first expression includes minterms 0 and 2 with the 1's and leaves minterm 5 with the 0's. The second expression includes minterm 5 with the 1's and leaves minterms 0 and 2 with the 0's. The two expressions represent two functions that are not algebraically equal. Both cover the specified minterms of the function, but each covers different don't-care minterms. As far as the incompletely specified function is concerned, either expression is acceptable because the only difference is in the value of F for the don't-care minterms.

It is also possible to obtain a simplified product-of-sums expression for the function of Fig. 3.17. In this case, the only way to combine the 0's is to include don't-care minterms 0 and 2 with the 0's to give a simplified complemented function:

$$F' = z' + wy'$$

Taking the complement of F' gives the simplified expression in product-of-sums form:

$$F(w, x, y, z) = z(w' + y) = \Sigma(1, 3, 5, 7, 11, 15)$$

In this case, we include minterms 0 and 2 with the 0's and minterm 5 with the 1's.

3.7 NAND AND NOR IMPLEMENTATION

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates used in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND Circuits

The NAND gate is said to be a universal gate because any digital system can be implemented with it. To show that any Boolean function can be implemented with NAND gates, we need only show that the logical operations of AND, OR, and complement can be obtained with NAND gates alone. This is indeed shown in Fig. 3.18. The complement operation is obtained from a one-input NAND gate that behaves exactly like an inverter. The AND operation requires two NAND gates. The first produces the NAND operation and the second inverts the logical sense of the signal. The OR operation is achieved through a NAND gate with additional inverters in each input.

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function in terms of Boolean operators and then convert the function to NAND logic. The conversion of an algebraic expression from AND, OR, and complement to NAND can be done by simple circuit manipulation techniques that change AND–OR diagrams to NAND diagrams.

To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate. Two equivalent graphic symbols for the NAND gate are shown in Fig. 3.19.

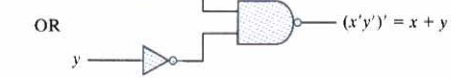
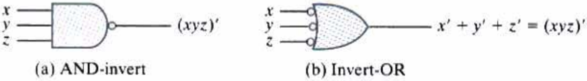


FIGURE 3.18
Logic operations with NAND gates



(a) AND-invert (b) Invert-OR

FIGURE 3.19
Two graphic symbols for the NAND gate

The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle negation indicator referred to as a bubble. Alternatively, it is possible to represent a NAND gate by an OR graphic symbol that is preceded by a bubble in each input. The invert-OR symbol for the NAND gate follows DeMorgan's theorem and the convention that the negation indicator denotes complementation. The two graphic symbols' representations are useful in the analysis and design of NAND circuits. When both symbols are mixed in the same diagram, the circuit is said to be in mixed notation.

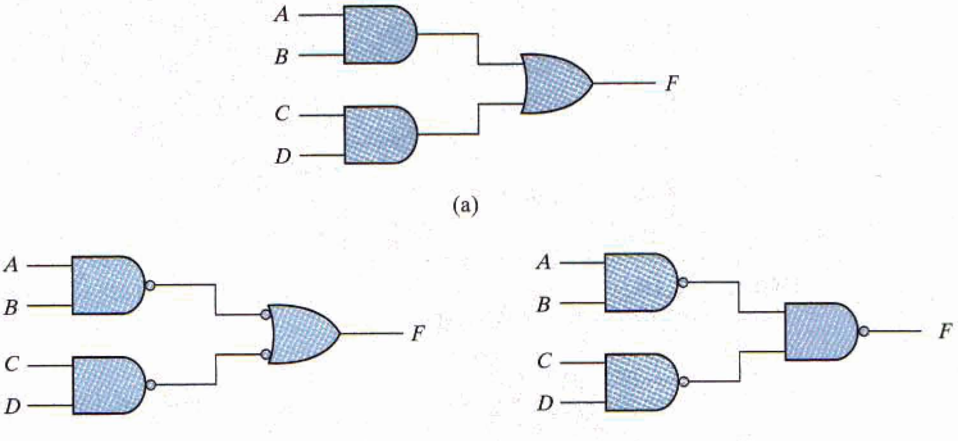
Two-Level Implementation

The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form. To see the relationship between a sum-of-product expression and its equivalent NAND implementation, consider the logic diagrams drawn in Fig. 3.20. All three diagrams are equivalent and implement the function

$$F = AB + CD$$

The function is implemented in (a) with AND and OR gates. In (b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an OR-invert graphic symbol. Remember that a bubble denotes complementation and two bubbles along the same line represent double complementation, so both can be removed. Removing the bubbles on the gates of (b) produces the circuit of (a). Therefore, the two diagrams implement the same function and are equivalent.

In Fig. 3.20(c), the output NAND gate is redrawn with the AND-invert graphic symbol. In drawing NAND logic diagrams, the circuit shown in either (b) or (c) is acceptable. The



(a)

(b)

(c)

FIGURE 3.20
Three ways to implement $F = AB + CD$

one in (b) is in mixed notation and represents a more direct relationship to the Boolean expression it implements. The NAND implementation in Fig. 3.20(c) can be verified algebraically. The function it implements can easily be converted to sum-of-products form by DeMorgan's theorem:

$$F = ((AB)'(CD)')' = AB + CD$$

EXAMPLE 3.10

Implement the following Boolean function with NAND gates:

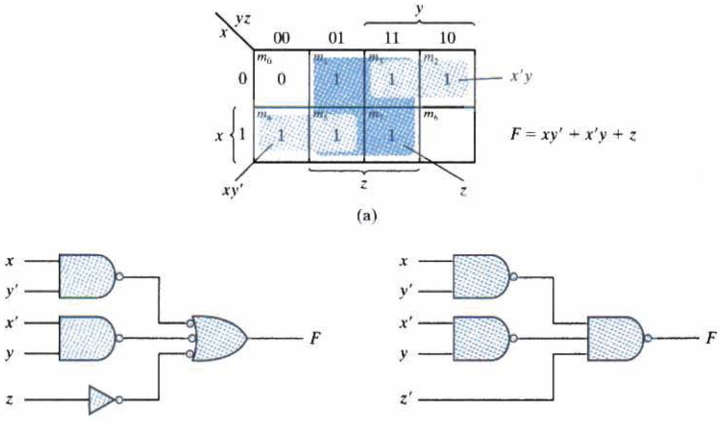
$$F(x, y, z) = (1, 2, 3, 4, 5, 7)$$

The first step is to simplify the function into sum-of-products form. This is done by means of the map of Fig. 3.21(a), from which the simplified function is obtained:

$$F = xy' + x'y + z$$

The two-level NAND implementation is shown in Fig. 3.21(b) in mixed notation. Note that input z must have a one-input NAND gate (an inverter) to compensate for the bubble in the second-level gate. An alternative way of drawing the logic diagram is given in Fig. 3.21(c). Here, all the NAND gates are drawn with the same graphic symbol. The inverter with input z has been removed, but the input variable is complemented and denoted by z'.

■



(a)

(b)

(c)

FIGURE 3.21
Solution to Example 3.10

The procedure described in the previous example indicates that a Boolean function can be implemented with two levels of NAND gates. The procedure for obtaining the logic diagram from a Boolean function is as follows:

1. Simplify the function and express it in sum-of-products form.
2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.
3. Draw a single gate using the AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from outputs of first-level gates.
4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

Multilevel NAND Circuits

The standard form of expressing Boolean functions results in a two-level implementation. There are occasions, however, when the design of digital systems results in gating structures with three or more levels. The most common procedure in the design of multilevel circuits is to express the Boolean function in terms of AND, OR, and complement operations. The function can then be implemented with AND and OR gates. After that, if necessary, it can be converted into an all-NAND circuit. Consider, for example, the Boolean function

$$F = A(CD + B) + BC'$$

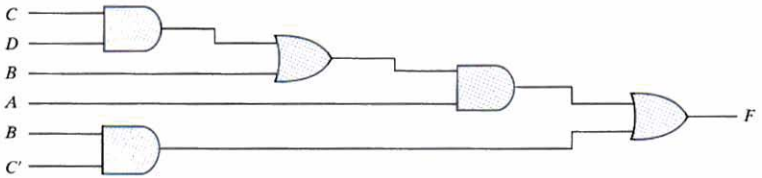
Although it is possible to remove the parentheses and reduce the expression into a standard sum-of-products form, we choose to implement it as a multilevel circuit for illustration. The AND–OR implementation is shown in Fig. 3.22(a). There are four levels of gating in the circuit. The first level has two AND gates. The second level has an OR gate followed by an AND gate in the third level and an OR gate in the fourth level. A logic diagram with a pattern of alternating levels of AND and OR gates can easily be converted into a NAND circuit with the use of mixed notation, shown in Fig. 3.22(b). The procedure is to change every AND gate to an AND-invert graphic symbol and every OR gate to an invert-OR graphic symbol. The NAND circuit performs the same logic as the AND–OR diagram as long as there are two bubbles along the same line. The bubble associated with input B causes an extra complementation, which must be compensated for by changing the input literal to B'.

The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:

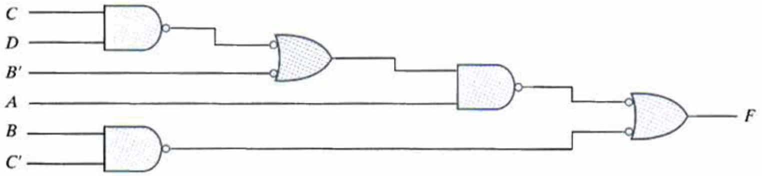
1. Convert all AND gates to NAND gates with AND-invert graphic symbols.
2. Convert all OR gates to NAND gates with invert-OR graphic symbols.
3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

As another example, consider the multilevel Boolean function

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



(a) AND–OR gates



(b) NAND gates

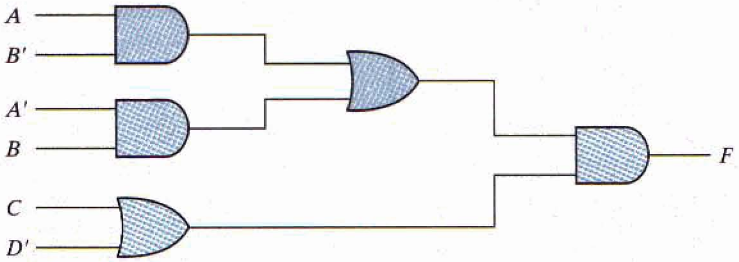
FIGURE 3.22
Implementing $F = A(CD + B) + BC'$

The AND–OR implementation of this function is shown in Fig. 3.23(a) with three levels of gating. The conversion to NAND with mixed notation is presented in part (b) of the diagram. The two additional bubbles associated with inputs C and D' cause these two literals to be complemented to C' and D. The bubble in the output NAND gate complements the output value, so we need to insert an inverter gate at the output in order to complement the signal again and get the original value back.

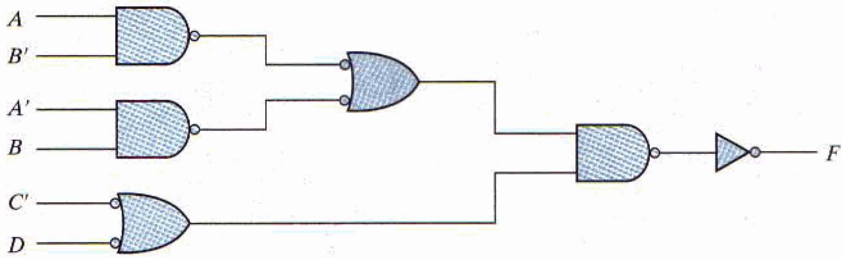
NOR Implementation

The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic. The NOR gate is another universal gate that can be used to implement any Boolean function. The implementation of the complement, OR, and AND operations with NOR gates is shown in Fig. 3.24. The complement operation is obtained from a one-input NOR gate that behaves exactly like an inverter. The OR operation requires two NOR gates, and the AND operation is obtained with a NOR gate that has inverters in each input.

The two graphic symbols for the mixed notation are shown in Fig. 3.25. The OR-invert symbol defines the NOR operation as an OR followed by a complement. The invert-AND symbol complements each input and then performs an AND operation. The two symbols designate the same NOR operation and are logically identical because of DeMorgan's theorem.



(a) AND–OR gates



(b) NAND gates

FIGURE 3.23
Implementing $F = (AB' + A'B)(C + D')$

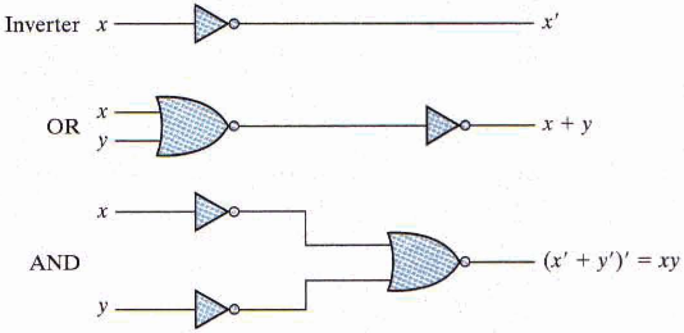
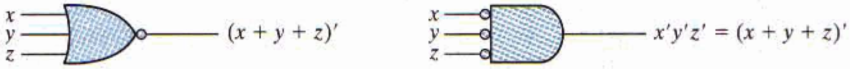


FIGURE 3.24
Logic operations with NOR gates



(a) OR-invert
(b) Invert-AND

FIGURE 3.25
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. Fig. 3.26 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. 3.23(a) into a NOR diagram is shown in Fig. 3.27. The Boolean function for this circuit is

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

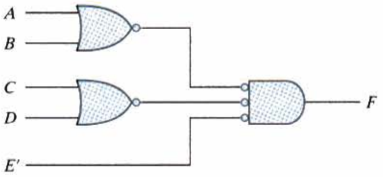


FIGURE 3.26
Implementing $F = (A + B)(C + D)E$

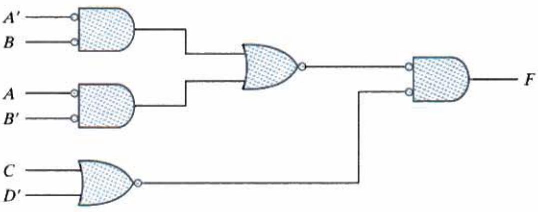


FIGURE 3.27
Implementing $F = (AB' + A'B)(C + D')$ with NOR gates

The equivalent AND–OR diagram can be recognized from the NOR diagram by removing all the bubbles. To compensate for the bubbles in four inputs, it is necessary to complement the corresponding input literals.

3.8 OTHER TWO-LEVEL IMPLEMENTATIONS

The types of gates most often found in integrated circuits are NAND and NOR gates. For this reason, NAND and NOR logic implementations are the most important from a practical point of view. Some (but not all) NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function. This type of logic is called *wired logic*. For example, open-collector TTL NAND gates, when tied together, perform wired-AND logic. (The open-collector TTL gate is shown in Chapter 10, Fig. 10.11.) The wired-AND logic performed with two NAND gates is depicted in Fig. 3.28(a). The AND gate is drawn with the lines going through the center of the gate to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated wired connection. The logic function implemented by the circuit of Fig. 3.28(a) is

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

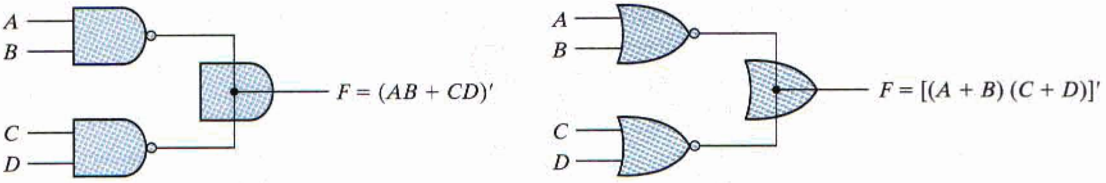
and is called an AND–OR–INVERT function.

Similarly, the NOR outputs of ECL gates (see Figure 10.17) can be tied together to perform a wired-OR function. The logic function implemented by the circuit of Fig. 3.28(b) is

$$F = (A + B)' + (C + D)' = [(A + B)(C + D)]'$$

and is called an OR–AND–INVERT function.

A wired-logic gate does not produce a physical second-level gate, since it is just a wire connection. Nevertheless, for discussion purposes, we will consider the circuits of Fig. 3.28 as two-level implementations. The first level consists of NAND (or NOR) gates and the second level has a single AND (or OR) gate. The wired connection in the graphic symbol will be omitted in subsequent discussions.



(a) Wired-AND in open-collector TTL NAND gates.

(AND–OR–INVERT)

(b) Wired-OR in ECL gates

(OR–AND–INVERT)

FIGURE 3.28
Wired logic
(a) Wired-AND logic with two NAND gates
(b) Wired-OR in emitter-coupled logic (ECL) gates

Nondegenerate Forms

It will be instructive from a theoretical point of view to find out how many two-level combinations of gates are possible. We consider four types of gates: AND, OR, NAND, and NOR. If we assign one type of gate for the first level and one type for the second level, we find that there are 16 possible combinations of two-level forms. (The same type of gate can be in the first and second levels, as in a NAND–NAND implementation.) Eight of these combinations are said to be *degenerate* forms because they degenerate to a single operation. This can be seen from a circuit with AND gates in the first level and an AND gate in the second level. The output of the circuit is merely the AND function of all input variables. The remaining eight *nondegenerate* forms produce an implementation in sum-of-products form or product-of-sums form. The eight nondegenerate forms are as follows:

AND–OR	OR–AND
NAND–NAND	NOR–NOR
NOR–OR	NAND–AND
OR–NAND	AND–NOR

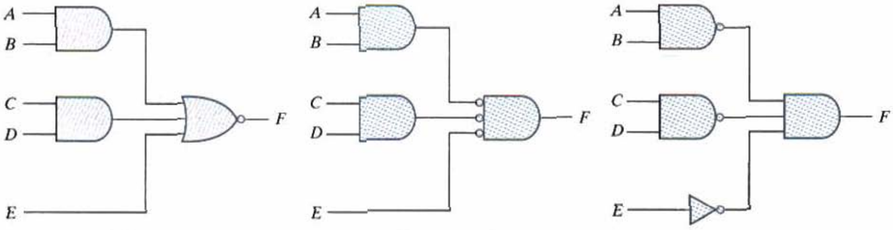
The first gate listed in each of the forms constitutes a first level in the implementation. The second gate listed is a single gate placed in the second level. Note that any two forms listed on the same line are duals of each other.

The AND–OR and OR–AND forms are the basic two-level forms discussed in Section 3.4. The NAND–NAND and NOR–NOR forms were presented in Section 3.6. The remaining four forms are investigated in this section.

AND–OR–INVERT Implementation

The two forms NAND–AND and AND–NOR are equivalent and can be treated together. Both perform the AND–OR–INVERT function, as shown in Fig. 3.29. The AND–NOR form resembles the AND–OR form, but with an inversion done by the bubble in the output of the NOR gate. It implements the function

$$F = (AB + CD + E)'$$



(a) AND–NOR (b) AND–NOR (c) NAND–AND

FIGURE 3.29
AND–OR–INVERT circuits, $F = (AB + CD + E)'$

By using the alternative graphic symbol for the NOR gate, we obtain the diagram of Fig. 3.29(b). Note that the single variable E is *not* complemented, because the only change made is in the graphic symbol of the NOR gate. Now we move the bubble from the input terminal of the second-level gate to the output terminals of the first-level gates. An inverter is needed for the single variable in order to compensate for the bubble. Alternatively, the inverter can be removed, provided that input E is complemented. The circuit of Fig. 3.29(c) is a NAND–AND form and was shown in Fig. 3.28 to implement the AND–OR–INVERT function.

An AND–OR implementation requires an expression in sum-of-products form. The AND–OR–INVERT implementation is similar, except for the inversion. Therefore, if the *complement* of the function is simplified into sum-of-products form (by combining the 0's in the map), it will be possible to implement F' with the AND–OR part of the function. When F' passes through the always present output inversion (the INVERT part), it will generate the output F of the function. An example for the AND–OR–INVERT implementation will be shown subsequently.

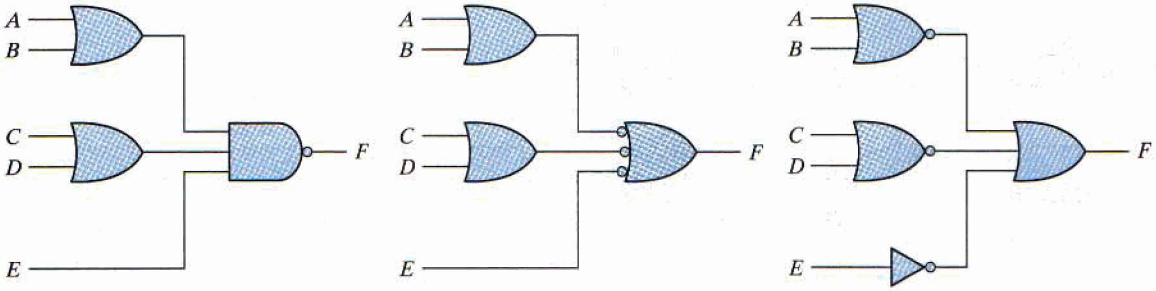
OR–AND–INVERT Implementation

The OR–NAND and NOR–OR forms perform the OR–AND–INVERT function, as shown in Fig. 3.30. The OR–NAND form resembles the OR–AND form, except for the inversion done by the bubble in the NAND gate. It implements the function

$$F = [(A + B)(C + D)E]'$$

By using the alternative graphic symbol for the NAND gate, we obtain the diagram of Fig. 3.30(b). The circuit in (c) is obtained by moving the small circles from the inputs of the second-level gate to the outputs of the first-level gates. The circuit of Fig. 3.30(c) is a NOR–OR form and was shown in Fig. 3.28 to implement the OR–AND–INVERT function.

The OR–AND–INVERT implementation requires an expression in product-of-sums form. If the complement of the function is simplified into that form, we can implement F' with the OR–AND part of the function. When F' passes through the INVERT part, we obtain the complement of F', or F, in the output.



(a) OR–NAND (b) OR–NAND (c) NOR–OR

FIGURE 3.30
OR–AND–INVERT circuits, $F = [(A + B)(C + D)E]'$

Table 3.3
Implementation with Other Two-Level Forms

Equivalent Nondegenerate Form		Implements the Function	Simplify F' into	To Get an Output of
(a)	(b)*			
AND–NOR	NAND–AND	AND–OR–INVERT	Sum-of-products form by combining 0's in the map.	F
OR–NAND	NOR–OR	OR–AND–INVERT	Product-of-sums form by combining 1's in the map and then complementing.	F

*Form (b) requires an inverter for a single literal term.

Tabular Summary and Example

Table 3.3 summarizes the procedures for implementing a Boolean function in any one of the four 2-level forms. Because of the INVERT part in each case, it is convenient to use the simplification of F' (the complement) of the function. When F' is implemented in one of these forms, we obtain the complement of the function in the AND–OR or OR–AND form. The four 2-level forms invert this function, giving an output that is the complement of F'. This is the normal output F.

EXAMPLE 3.11

Implement the function of Fig. 3.31(a) with the four 2-level forms listed in Table 3.3.
The complement of the function is simplified into sum-of-products form by combining the 0's in the map:

$$F' = x'y + xy' + z$$

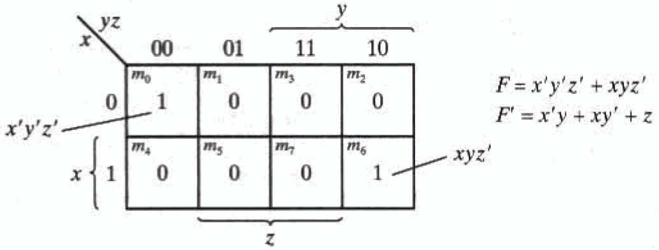
The normal output for this function can be expressed as

$$F = (x'y + xy' + z)'$$

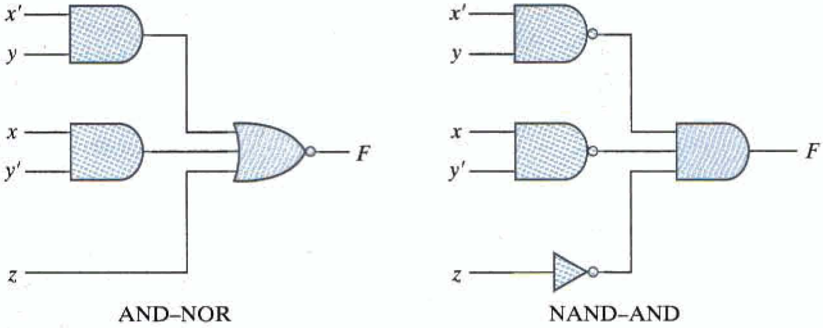
which is in the AND–OR–INVERT form. The AND–NOR and NAND–AND implementations are shown in Fig. 3.31(b). Note that a one-input NAND, or inverter, gate is needed in the NAND–AND implementation, but not in the AND–NOR case. The inverter can be removed if we apply the input variable z' instead of z.

The OR–AND–INVERT forms require a simplified expression of the complement of the function in product-of-sums form. To obtain this expression, we first combine the 1's in the map:

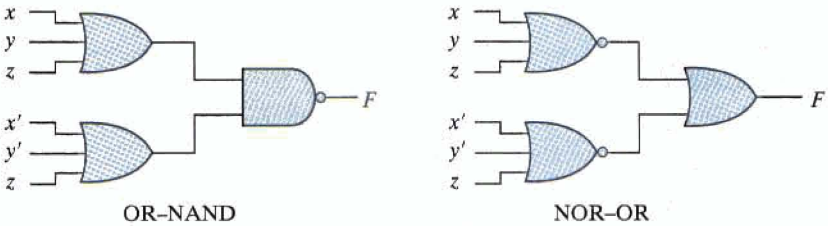
$$F = x'y'z' + xyz'$$



(a) Map simplification in sum of products



(b) $F = (x'y + xy' + z)'$



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

FIGURE 3.31
Other two-level implementations

Then we take the complement of the function:

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

The normal output F can now be expressed in the form

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

which is the OR–AND–INVERT form. From this expression, we can implement the function in the OR–NAND and NOR–OR forms, as shown in Fig. 3.31(c).

■

3.9 EXCLUSIVE-OR FUNCTION

The exclusive-OR (XOR), denoted by the symbol $\oplus$, is a logical operation that performs the following Boolean operation:

$$x \oplus y = xy' + x'y$$

The exclusive-OR is equal to 1 if only x is equal to 1 or if only y is equal to 1 (i.e., x and y differ in value), but not when both are equal to 1 or when both are equal to 0. The exclusive-NOR, also known as equivalence, performs the following Boolean operation:

$$(x \oplus y)' = xy + x'y'$$

The exclusive-NOR is equal to 1 if both x and y are equal to 1 or if both are equal to 0. The exclusive-NOR can be shown to be the complement of the exclusive-OR by means of a truth table or by algebraic manipulation:

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = xy + x'y'$$

The following identities apply to the exclusive-OR operation:

$$\begin{aligned} x \oplus 0 &= x \\ x \oplus 1 &= x' \\ x \oplus x &= 0 \\ x \oplus x' &= 1 \\ x \oplus y' &= x' \oplus y = (x \oplus y)' \end{aligned}$$

Any of these identities can be proven with a truth table or by replacing the $\oplus$ operation by its equivalent Boolean expression. Also, it can be shown that the exclusive-OR operation is both commutative and associative; that is,

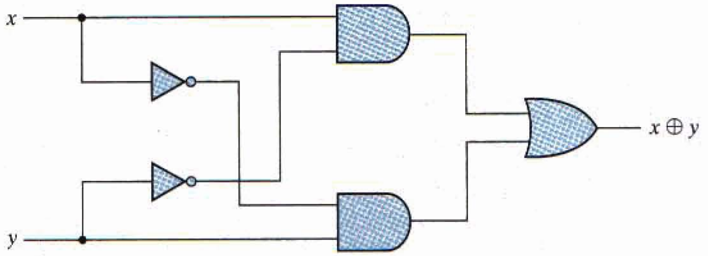
$$A \oplus B = B \oplus A$$

and

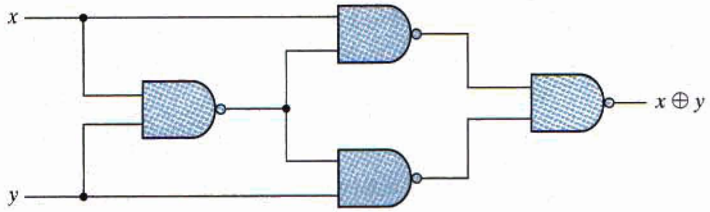
$$(A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation. It also means that we can evaluate a three-variable exclusive-OR operation in any order, and for this reason, three or more variables can be expressed without parentheses. This would imply the possibility of using exclusive-OR gates with three or more inputs. However, multiple-input exclusive-OR gates are difficult to fabricate with hardware. In fact, even a two-input function is usually constructed with other types of gates. A two-input exclusive-OR function is constructed with conventional gates using two inverters, two AND gates, and an OR gate, as shown in Fig. 3.32(a). Figure 3.32(b) shows the implementation of the exclusive-OR with four NAND gates. The first NAND gate performs the operation $(xy)' = (x' + y')$. The other two-level NAND circuit produces the sum of products of its inputs:

$$(x' + y')x + (x' + y')y = xy' + x'y = x \oplus y$$



(a) With AND–OR–NOT gates



(b) With NAND gates

FIGURE 3.32
Exclusive-OR implementations

Only a limited number of Boolean functions can be expressed in terms of exclusive-OR operations. Nevertheless, this function emerges quite often during the design of digital systems. It is particularly useful in arithmetic operations and error detection and correction circuits.

Odd Function

The exclusive-OR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ symbol with its equivalent Boolean expression. In particular, the three-variable case can be converted to a Boolean expression as follows:

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \Sigma(1, 2, 4, 7) \end{aligned}$$

The Boolean expression clearly indicates that the three-variable exclusive-OR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1. Contrary to the two-variable case, in which only one variable must be equal to 1, in the case of three or more variables the requirement is that an odd number of variables be equal to 1. As a consequence, the multiple-variable exclusive-OR operation is defined as an *odd function.*

The Boolean function derived from the three-variable exclusive-OR operation is expressed as the logical sum of four minterms whose binary numerical values are 001, 010, 100, and 111. Each of these binary numbers has an odd number of 1's. The remaining four minterms not included in the function are 000, 011, 101, and 110, and they have an even number of 1's in their binary numerical values. In general, an n-variable exclusive-OR function is an odd function defined as the logical sum of the $2^n/2$ minterms whose binary numerical values have an odd number of 1's.

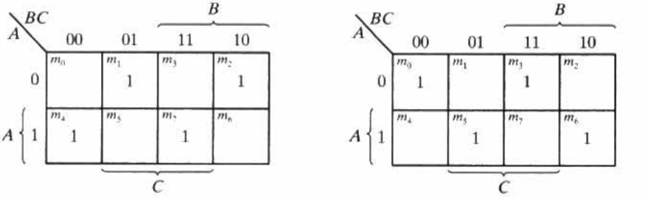
The definition of an odd function can be clarified by plotting it in a map. Figure 3.33(a) shows the map for the three-variable exclusive-OR function. The four minterms of the function are a unit distance apart from each other. The odd function is identified from the four minterms whose binary values have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 3.33(b), the three-variable even function is equal to 1 when an even number of its variables is equal to 1 (including the condition that none of the variables is equal to 1).

The three-input odd function is implemented by means of two-input exclusive-OR gates, as shown in Fig. 3.34(a). The complement of an odd function is obtained by replacing the output gate with an exclusive-NOR gate, as shown in Fig. 3.34(b).

Consider now the four-variable exclusive-OR operation. By algebraic manipulation, we can obtain the sum of minterms for this function:

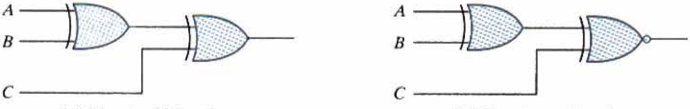
$$\begin{aligned} A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D) \\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D) \\ &= \Sigma(1, 2, 4, 7, 8, 11, 13, 14) \end{aligned}$$

There are 16 minterms for a four-variable Boolean function. Half of the minterms have binary numerical values with an odd number of 1's; the other half of the minterms have binary numerical



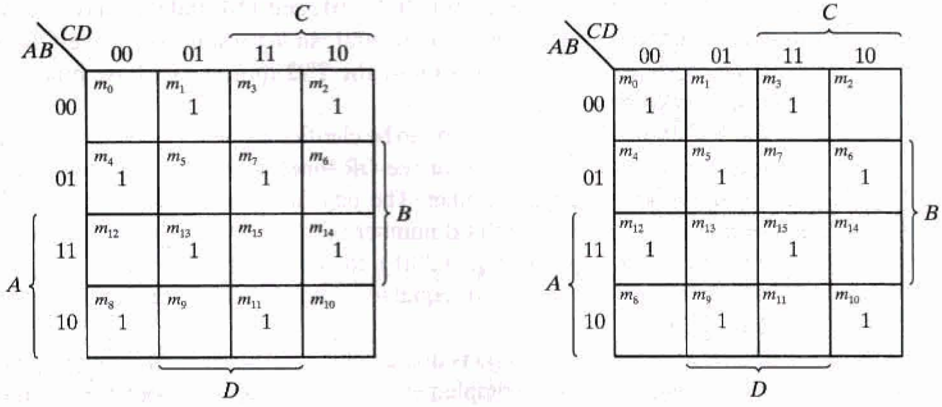
(a) Odd function $F = A \oplus B \oplus C$ (b) Even function $F = (A \oplus B \oplus C)'$

FIGURE 3.33
Map for a three-variable exclusive-OR function



(a) 3-input odd function (b) 3-input even function

FIGURE 3.34
Logic diagram of odd and even functions


CD
AB
00 01 11 10
C
B
A
D
m_0 m_1 1 m_3 m_2 1
m_4 1 m_5 m_7 1 m_6
m_{12} m_{13} 1 m_{15} m_{14} 1
m_8 1 m_9 m_{11} 1 m_{10}
m_0 1 m_1 m_3 1 m_2
m_4 m_5 1 m_7 m_6 1
m_{12} 1 m_{13} m_{15} 1 m_{14}
m_8 m_9 1 m_{11} m_{10} 1

(a) Odd function $F = A \oplus B \oplus C \oplus D$ (b) Even function $F = (A \oplus B \oplus C \oplus D)'$

FIGURE 3.35
Map for a four-variable exclusive-OR function

values with an even number of 1's. In plotting the function in the map, the binary numerical value for a minterm is determined from the row and column numbers of the square that represents the minterm. The map of Fig. 3.35(a) is a plot of the four-variable exclusive-OR function. This is an odd function because the binary values of all the minterms have an odd number of 1's. The complement of an odd function is an even function. As shown in Fig. 3.35(b), the four-variable even function is equal to 1 when an even number of its variables is equal to 1.

Parity Generation and Checking

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. As discussed in Section 1.7, a parity bit is used for the purpose of detecting errors during the transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even. The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator.* The circuit that checks the parity in the receiver is called a *parity checker.*

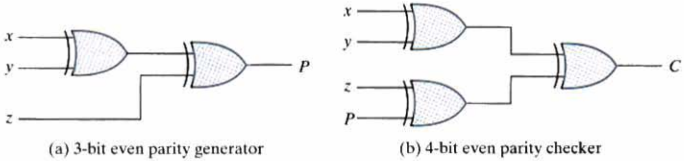
As an example, consider a three-bit message to be transmitted together with an even parity bit. Table 3.4 shows the truth table for the parity generator. The three bits—x, y, and z—constitute the message and are the inputs to the circuit. The parity bit P is the output. For even parity, the bit P must be generated to make the total number of 1's (including P) even. From the truth table, we see that P constitutes an odd function because it is equal to 1 for those minterms whose numerical values have an odd number of 1's. Therefore, P can be expressed as a three-variable exclusive-OR function:

$$P = x \oplus y \oplus z$$

The logic diagram for the parity generator is shown in Fig. 3.36(a).

Table 3.4
Even-Parity-Generator Truth Table

Three-Bit Message			Parity Bit
x	*y*	*z*	*P*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



(a) 3-bit even parity generator

(b) 4-bit even parity checker

FIGURE 3.36
Logic diagram of a parity generator and checker

The three bits in the message, together with the parity bit, are transmitted to their destination, where they are applied to a parity-checker circuit to check for possible errors in the transmission. Since the information was transmitted with even parity, the four bits received must have an even number of 1's. An error occurs during the transmission if the four bits received have an odd number of 1's, indicating that one bit has changed in value during transmission. The output of the parity checker, denoted by C, will be equal to 1 if an error occurs—that is, if the four bits received have an odd number of 1's. Table 3.5 is the truth table for the even-parity checker. From it, we see that the function C consists of the eight minterms with binary numerical values having an odd number of 1's. The table corresponds to the map of Fig. 3.35(a), which represents an odd function. The parity checker can be implemented with exclusive-OR gates:

$$C = x \oplus y \oplus z \oplus P$$

The logic diagram of the parity checker is shown in Fig. 3.36(b).

It is worth noting that the parity generator can be implemented with the circuit of Fig. 3.36(b) if the input P is connected to logic 0 and the output is marked with P. This is because $z \oplus 0 = z$, causing the value of z to pass through the gate unchanged. The advantage of this strategy is that the same circuit can be used for both parity generation and checking.

Table 3.5
Even-Parity-Checker Truth Table

Four Bits Received				Parity Error Check
x	y	z	P	C
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

It is obvious from the foregoing example that parity generation and checking circuits always have an output function that includes half of the minterms whose numerical values have either an odd or even number of 1's. As a consequence, they can be implemented with exclusive-OR gates. A function with an even number of 1's is the complement of an odd function. It is implemented with exclusive-OR gates, except that the gate associated with the output must be an exclusive-NOR to provide the required complementation.

3.10 HARDWARE DESCRIPTION LANGUAGE

Manual methods for designing logic circuits are feasible only when the circuit is small. For anything else (i.e., a practical circuit), designers use computer-based design tools. Coupled with a correct-by-construction methodology, computer-based design tools leverage the creativity and effort of a designer and reduce the risk of producing a flawed design. Prototype integrated circuits are too expensive and time consuming to build, so all modern design tools rely on a hardware description language to describe, design, and test a circuit in software before it is ever manufactured.

A *hardware description language* (HDL) is a computer-based language that describes the hardware of digital systems in a textual form. It resembles an ordinary computer programming language, such as C, but is specifically oriented to describing hardware structures and the behavior of logic circuits. It can be used to represent logic diagrams, truth tables, Boolean expressions, and complex abstractions of the behavior of a digital system. One way to view an HDL is to observe that it describes a relationship between signals that are the inputs to a circuit and the signals that are outputs of the circuit. For example, an HDL description of an AND gate describes how the logic value of the gate's output is determined by the logic values of its inputs.

As a *documentation* language, an HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers. The language content can be stored, retrieved, edited, and transmitted easily and processed by computer software in an efficient manner.

HDLs are used in several major steps in the design flow of an integrated circuit: design entry, functional simulation or verification, logic synthesis, timing verification, and fault simulation.

Design entry creates an HDL-based description of the functionality that is to be implemented in hardware. Depending on the HDL, the description can be in a variety of forms: Boolean logic equations, truth tables, a netlist of interconnected gates, or an abstract behavioral model. The HDL model may also represent a partition of a larger circuit into smaller interconnected and interacting functional units.

Logic simulation displays the behavior of a digital system through the use of a computer. A simulator interprets the HDL description and either produces readable output, such as a time-ordered sequence of input and output signal values, or displays waveforms of the signals. The simulation of a circuit predicts how the hardware will behave before it is actually fabricated. Simulation allows the detection of functional errors in a design without having to physically create and operate the circuit. Errors that are detected during a simulation can be corrected by modifying the appropriate HDL statements. The stimulus (i.e., the logic values of the inputs to a circuit) that tests the functionality of the design is called a *test bench.* Thus, to simulate a digital system, the design is first described in an HDL and then verified by simulating the design and checking it with a test bench, which is also written in the HDL. An alternative and more complex approach relies on formal mathematical methods to prove that a circuit is functionally correct. We will focus exclusively on simulation.

Logic synthesis is the process of deriving a list of physical components and their interconnections (called a *netlist*) from the model of a digital system described in an HDL. The netlist can be used to fabricate an integrated circuit or to lay out a printed circuit board with the hardware counterparts of the gates in the list. Logic synthesis is similar to compiling a program in a conventional high-level language. The difference is that, instead of producing an object code, logic synthesis produces a database describing the elements and structure of a circuit. The database specifies how to fabricate a physical integrated circuit that implements in silicon the functionality described by statements made in an HDL. Logic synthesis is based on formal exact procedures that implement digital circuits and addresses that part of a digital design which can be automated with computer software. The design of today's large, complex circuits is made possible by logic synthesis software.

Timing verification confirms that the fabricated integrated circuit will operate at a specified speed. Because each logic gate in a circuit has a propagation delay, a signal transition at the input of a circuit cannot immediately cause a change in the logic value of the output of a circuit. Propagation delays ultimately limit the speed at which a circuit can operate. Timing verification checks each signal path to verify that it is not compromised by propagation delay. This step is done after logic synthesis specifies the actual devices that will compose a circuit and before the circuit is released for production.

In VLSI circuit design, *fault simulation* compares the behavior of an ideal circuit with the behavior of a circuit that contains a process-induced flaw. Dust and other particulates in the atmosphere of the clean room can cause a circuit to be fabricated with a fault. A circuit with a fault will not exhibit the same functionality as a fault-free circuit. Fault simulation is used to identify input stimuli that can be used to reveal the difference between the faulty circuit and the fault-free circuit. These test patterns will be used to test fabricated devices to ensure that only good devices are shipped to the customer. Test generation and fault simulation may occur at different steps in the design process, but they are always done before production in order to avoid the disaster of producing a circuit whose internal logic cannot be tested.

Companies that design integrated circuits use proprietary and public HDLs. In the public domain, there are two standard HDLs that are supported by the IEEE: VHDL and Verilog. VHDL is a Department of Defense–mandated language. (The *V* in VHDL stands for the first letter in VHSIC, an acronym for very high speed integrated circuit.) Verilog began as a proprietary HDL of Cadence Design Systems, but Cadence transferred control of Verilog to a consortium of companies and universities known as Open Verilog International (OVI) as a step leading to its adoption as an IEEE standard. VHDL is more difficult to learn than Verilog. Because Verilog is an easier language than VHDL to describe, learn, and use, we have chosen it for this book. However, the Verilog HDL descriptions listed throughout the book are not just about Verilog, but also serve to introduce a design methodology based on the concept of computer-aided modeling of digital systems by means of a typical hardware description language. Our emphasis will be on the modeling, verification, and synthesis (both manual and automated) of Verilog models of circuits having specified behavior. The Verilog HDL was initially approved as a standard HDL in 1995; revised and enhanced versions of the language were approved in 2001 and 2005. We will address only those features of Verilog, including the latest standard, that support our discussion of HDL-based design methodology for integrated circuits.

Module Declaration

The language reference manual for the Verilog HDL presents a syntax that describes precisely the constructs that can be used in the language. In particular, a Verilog model is composed of text using keywords, of which there are about 100. Keywords are predefined lowercase identifiers that define the language constructs. Examples of keywords are **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, and **not**. For clarity, keywords will be displayed in boldface in the text in all examples of code and wherever it is appropriate to call attention to their use. Any text between two forward slashes (//) and the end of the line is interpreted as a comment and will have no effect on a simulation using the model. Multiline comments begin with /* and terminate with */. Blank spaces are ignored, but they may not appear within the text of a keyword, a user-specified identifier, an operator, or the representation of a number. Verilog is case sensitive, which means that uppercase and lowercase letters are distinguishable (e.g., **not** is not the same as NOT). The term *module* refers to the text enclosed

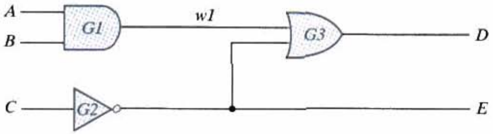


FIGURE 3.37
Circuit to demonstrate an HDL

by the keyword pair **module** ... **endmodule**. A module is the fundamental descriptive unit in the Verilog language. It is declared by the keyword **module** and must always be terminated by the keyword **endmodule**.

Combinational logic can be described by a schematic connection of gates, by a set of Boolean equations, or by a truth table. Each type of description can be developed in Verilog. We will demonstrate each style, beginning with a simple example of a Verilog gate-level description to illustrate some aspects of the language.

The HDL description of the circuit of Fig. 3.37 is shown in HDL Example 3.1. The first line of text is a comment (optional) providing useful information to the reader. The second line begins with the keyword **module** and starts the declaration (description) of the module; the last line completes the declaration with the keyword **endmodule**. The keyword **module** is followed by a name and a list of ports. The name (*Simple_Circuit* in this example) is an identifier. Identifiers are names given to modules, variables (e.g., a signal), and other elements of the language so that they can be referenced in the design. In general, we choose meaningful names for modules. Identifiers are composed of alphanumeric characters and the underscore (_), and are case sensitive. Identifiers must start with an alphabetic character or an underscore, but they cannot start with a number.

HDL Example 3.1 (Combinational logic modeled with primitives)

```
// Verilog model of circuit of Figure 3.37. IEEE 1364–1995 Syntax

module Simple_Circuit (A, B, C, D, E);
 output       D, E;
 input        A, B, C;
 wire         w1;

 and          G1 (w1, A, B); // Optional gate instance name
 not          G2 (E, C);
 or           G3 (D, w1, E);
endmodule
```

The port list of a module is the interface between the module and its environment. In this example, the ports are the inputs and outputs of the circuit. The logic values of the inputs to a circuit are determined by the environment; the logic values of the outputs are determined within the circuit and result from the action of the inputs on the circuit. The port list is enclosed in parentheses, and commas are used to separate elements of the list. The statement is terminated with a semicolon (;). In our examples, all keywords (which must be in lowercase) are printed in bold for clarity, but that is not a requirement of the language. Next, the keywords **input** and **output** specify which of the ports are inputs and which are outputs. Internal connections are declared as wires. The circuit in this example has one internal connection, at terminal *w1*, and is declared with the keyword **wire**. The structure of the circuit is specified by a list of (predefined) *primitive* gates, each identified by a descriptive keyword (**and**, **not**, **or**). The elements of the list are referred to as *instantiations* of a gate, each of which is referred to as a *gate instance*. Each *gate instantiation* consists of an optional name (such as *G1*, *G2*, etc.) followed by the gate output and inputs separated by commas and enclosed in parentheses. The output of a primitive gate is always listed first, followed by the inputs. For example, the OR gate of the schematic is represented by the **or** primitive, is named *G3*, and has output *D* and inputs *w1* and *E*. (*Note*: The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.) The module description ends with the keyword **endmodule**. Each statement must be terminated with a semicolon, but there is no semicolon after **endmodule**.

It is important to understand the distinction between the terms *declaration* and *instantiation*. A Verilog module is declared. Its declaration specifies the input–output behavior of the hardware that it represents. Predefined primitives are not declared, because their definition is specified by the language and is not subject to change by the user. Primitives are used (i.e., instantiated), just as gates are used to populate a printed circuit board. We'll see that once a module has been declared, it may be used (instantiated) within a design. Note that *Simple_Circuit* is not a computational model like those developed in an ordinary programming language: The sequential ordering of the statements in the model does not specify a sequence of computations. A Verilog model is a *descriptive* model. *Simple_Circuit* describes what primitives form a circuit and how they are connected. The input–output behavior of the circuit is implicitly specified by the description because the behavior of each logic gate is defined. Thus, an HDL-based model can be used to simulate the circuit that it represents.

Gate Delays

All physical circuits exhibit a propagation delay between the transition of an input and a resulting transition of an output. When an HDL model of a circuit is simulated, it is sometimes necessary to specify the amount of delay from the input to the output of its gates. In Verilog, the propagation delay of a gate is specified in terms of *time units* and is specified by the symbol #. The numbers associated with time delays in Verilog are dimensionless. The association of a time unit with physical time is made with the ‘**timescale** compiler directive. (Compiler directives start with the (‘) back quote, or grave accent, symbol.) Such a directive is specified before the declaration of a module and applies to all numerical values of time in the code that follows. An example of a timescale directive is

timescale 1ns/100ps

The first number specifies the unit of measurement for time delays. The second number specifies the precision for which the delays are rounded off, in this case to 0.1 ns. If no timescale is specified, a simulator may display dimensionless values or default to a certain time unit, usually 1 ns ($= 10^{-9}$ sec). Our examples will use only the default time unit.

Table 3.6
Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	000	1 0 1
Change	—	111	1 0 1
	10	111	0 0 1
	20	111	0 0 1
	30	111	0 1 0
	40	111	0 1 0
	50	111	0 1 1

HDL Example 3.2 repeats the description of the simple circuit of Example 3.1, but with propagation delays specified for each gate. The **and**, **or**, and **not** gates have a time delay of 30, 20, and 10 ns, respectively. If the circuit is simulated and the inputs change from $A, B, C = 0$ to $A, B, C = 1$, the outputs change as shown in Table 3.6 (calculated by hand or generated by a simulator). The output of the inverter at E changes from 1 to 0 after a 10-ns delay. The output of the AND gate at $w1$ changes from 0 to 1 after a 30-ns delay. The output of the OR gate at D changes from 1 to 0 at $t = 30$ ns and then changes back to 1 at $t = 50$ ns. In both cases, the change in the output of the OR gate results from a change in its inputs 20 ns earlier. It is clear from this result that although output D eventually returns to a final value of 1 after the input changes, the gate delays produce a negative spike that lasts 20 ns before the final value is reached.

HDL Example 3.2 (Gate-level model with propagation delays)

```
// Verilog model of simple circuit with propagation delay

module Simple_Circuit_prop_delay (A, B, C, D, E);
 output  D, E;
 input   A, B, C;
 wire    w1;

 and           #(30) G1 (w1, A, B);
 not           #(10) G2 (E, C);
 or            #(20) G3 (D, w1, E);
endmodule
```

In order to simulate a circuit with an HDL, it is necessary to apply inputs to the circuit so that the simulator will generate an output response. An HDL description that provides the stimulus to a design is called a *test bench.* The writing of test benches is explained in more detail at the end of Section 4.12. Here, we demonstrate the procedure with a simple example without dwelling on too many details. HDL Example 3.3 shows a test bench for simulating the circuit with delay. (Note the distinguishing name *Simple_Circuit_prop_delay.*) In its simplest form, a test bench is a module containing a signal generator and an instantiation of the model that is to be verified. Note that the test bench (*t_Simple_Circuit_prop_delay*) has no input or output ports, because it does not interact with its environment. In general, we prefer to name the test bench with the prefix *t_* concatenated with the name of the module that is to be tested by the test bench, but that choice is left to the designer. Within the test bench, the inputs to the circuit are declared with keyword **reg** and the outputs are declared with the keyword **wire**. The module *Simple_Circuit_prop_delay* is instantiated with the instance name M1. Every instantiation of a module must include a unique instance name. Note that using a test bench is similar to testing actual hardware by attaching signal generators to the inputs of a circuit and attaching probes (wires) to the outputs of the circuit. (The interaction between the signal generators of the stimulus module and the instantiated circuit module is illustrated in Fig. 4.33.)

HDL Example 3.3

```
// Test bench for Simple_Circuit_prop_delay

module t_Simple_Circuit_prop_delay;
 wire      D, E;
 reg       A, B, C;

 Simple_Circuit_prop_delay M1 (A, B, C, D, E); // Instance name required

 initial
  begin
   A = 1'b0; B = 1'b0; C = 1'b0;
   #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

Hardware signal generators are not used to verify an HDL model: The entire simulation exercise is done with software models executing on a digital computer. The waveforms of the input signals are abstractly modeled (generated) by Verilog statements specifying waveform values and transitions. The **initial** keyword is used with a set of statements that begin executing when the simulation is initialized; **initial** terminates execution when the last statement has finished executing. **initial** statements are commonly used to describe waveforms in a test bench. The set of statements to be executed is called a *block statement* and consists of several statements enclosed by the keywords **begin** and **end**. The action specified by the statements begins when the simulation is launched, and the statements are executed in sequence, from top to bottom, by a simulator in order to provide the input to the circuit. Initially, $A, B, C = 0$. (A, B, and C are each set to $1'b0$, which signifies one binary digit with a value of 0.) After 100 ns, the inputs change to $A, B, C = 1$. After another 100 ns, the simulation terminates at time 200 ns. A second **initial** statement uses the **$finish** system task to specify termination of the simulation. If a statement is preceded by a delay value (e.g., #100), the simulator postpones executing the statement until the specified time delay has elapsed. The timing diagram of waveforms that result

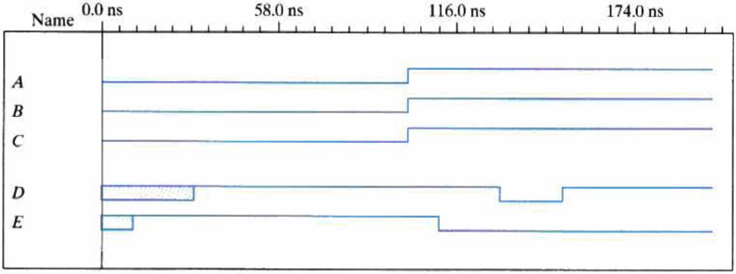


FIGURE 3.38
Simulation output of HDL Example 3.3

from the simulation is shown in Figure 3.38. The total simulation takes 200 ns. The inputs *A*, *B*, and *C* change from 0 to 1 after 100 ns. Output *E* is unknown for the first 10 ns (denoted by shading), and output *D* is unknown for the first 30 ns. Output *E* goes from 1 to 0 at 110 ns. Output *D* goes from 1 to 0 at 130 ns and back to 1 at 150 ns, just as we predicted in Table 3.6.

Boolean Expressions

Boolean equations describing combinational logic are specified in Verilog with a continuous assignment statement consisting of the keyword **assign** followed by a Boolean expression. To distinguish arithmetic operators from logical operators, Verilog uses the symbols (&), (/), and (~) for AND, OR, and NOT (complement), respectively. Thus, to describe the simple circuit of Fig. 3.37 with a Boolean expression, we use the statement

```
assign D = (A & B)|~C;
```

HDL Example 3.4 describes a circuit that is specified with the following two Boolean expressions:

$$E = A + BC + B'D$$

$$F = B'C + BC'D'$$

The equations specify how the logic values *E* and *F* are determined by the values of *A*, *B*, *C*, and *D*.

HDL Example 3.4 (Combinational logic modeled with Boolean equations)

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
 output     E, F;
 input      A, B, C, D;

 assign E = A | (B & C) | (~B & D);
 assign F = (~B & C) | (B & ~C & ~D);
endmodule
```

The circuit has two outputs E and F and four inputs A, B, C, and D. The two **assign** statements describe the Boolean equations. The values of E and F during simulation are determined dynamically by the values of A, B, C, and D. The simulator detects when the test bench changes a value of one or more of the inputs. When this happens, the simulator updates the values of E and F. The continuous assignment mechanism is so named because the relationship between the assigned value and the variables is permanent. The mechanism acts just like combinational logic, has a gate-level equivalent circuit, and is referred to as *implicit combinational logic.*

We have shown that a digital circuit can be described with HDL statements, just as it can be drawn in a circuit diagram or specified with a Boolean expression. A third alternative is to describe combinational logic with a truth table.

User-Defined Primitives

The logic gates used in Verilog descriptions with keywords **and**, **or**, etc., are defined by the system and are referred to as *system primitives. (Caution: Other languages may use these words differently.)* The user can create additional primitives by defining them in tabular form. These types of circuits are referred to as *user-defined primitives* (UDPs). One way of specifying a digital circuit in tabular form is by means of a truth table. UDP descriptions do not use the keyword pair **module** … **endmodule**. Instead, they are declared with the keyword pair **primitive** … **endprimitive**. The best way to demonstrate a UDP declaration is by means of an example.

HDL Example 3.5 defines a UDP with a truth table. It proceeds according to the following general rules:

- It is declared with the keyword **primitive**, followed by a name and port list.
- There can be only one output, and it must be listed first in the port list and declared with keyword **output**.
- There can be any number of inputs. The order in which they are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords **table** and **endtable**.
- The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
- The declaration of a UDP ends with the keyword **endprimitive**.

Note that the variables listed on top of the table are part of a comment and are shown only for clarity. The system recognizes the variables by the order in which they are listed in the input declaration. A user-defined primitive can be instantiated in the construction of other modules (digital circuits), just as the system primitives are used. For example, the declaration

```
Circuit_with_UDP_02467 (E, F, A, B, C, D);
```

will produce a circuit that implements the hardware shown in Figure 3.39.

Although Verilog HDL uses this kind of description for UDPs only, other HDLs and computer-aided design (CAD) systems use other procedures to specify digital circuits in tabular form. The tables can be processed by CAD software to derive an efficient gate structure of the design. None of Verilog's predefined primitives describes sequential logic. The

HDL Example 3.5

```
// Verilog model: User-defined Primitive

primitive UDP_02467 (D, A, B, C);
 output D;
 input   A, B, C;

// Truth table for D = f (A, B, C) = Σ (0, 2, 4, 6, 7);
 table
//     A      B      C      :      D       // Column header comment
       0      0      0      :      1;
       0      0      1      :      0;
       0      1      0      :      1;
       0      1      1      :      0;
       1      0      0      :      1;
       1      0      1      :      0;
       1      1      0      :      1;
       1      1      1      :      1;
 endtable
endprimitive

// Instantiate primitive

// Verilog model: Circuit instantiation of Circuit_UDP_02467

module Circuit_with_UDP_02467 (e, f, a, b, c, d);
 output       e, f;
 input        a, b, c, d;

 UDP_02467          (e, a, b, c);
 and                (f, e, d);          // Option gate instance name omitted
endmodule
```

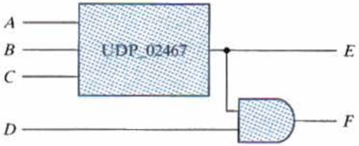


FIGURE 3.39
Schematic for *Circuit with_UDP_02467*

model of a sequential UDP requires that its output be declared as a **reg** data type, and that a column be added to the truth table to describe the next state. So the columns are organizes as inputs : state : next state.

In this section, we introduced the Verilog HDL and presented simple examples to illustrate alternatives for modeling combinational logic. A more detailed presentation of Verilog HDL can be found in the next chapter. The reader familiar with combinational circuits can go directly to Section 4.12 to continue with this subject.

PROBLEMS

Answers to problems marked with * appear at the end of the book.

3.1* Simplify the following Boolean functions, using three-variable maps:
(a) $F(x, y, z) = \Sigma(0, 2, 6, 7)$
(b) $F(x, y, z) = \Sigma(0, 2, 3, 4, 6)$
(c) $F(x, y, z) = \Sigma(0, 1, 2, 3, 7)$
(d) $F(x, y, z) = \Sigma(3, 5, 6, 7)$

3.2 Simplify the following Boolean functions, using three-variable maps:
(a)* $F(x, y, z) = \Sigma(0, 1, 5, 7)$
(b)* $F(x, y, z) = \Sigma(1, 2, 3, 6, 7)$
(c) $F(x, y, z) = \Sigma(0, 1, 6, 7)$
(d) $F(x, y, z) = \Sigma(0, 1, 3, 4, 5)$
(e) $F(x, y, z) = \Sigma(1, 3, 5, 7)$
(f) $F(x, y, z) = \Sigma(1, 4, 5, 6, 7)$

3.3* Simplify the following Boolean expressions, using three-variable maps:
(a)* $F(x,y,z) = xy + x'y'z' + x'yz'$
(b)* $F(x,y,z) = x'y' + yz + x'yz'$
(c)* $F(x, y, z) = x'y + yz' + y'z'$
(d) $F(x, y, z) = xyz + x'y'z + xy'z'$

3.4 Simplify the following Boolean functions, using *Karnaugh* maps:
(a)* $F(x, y, z) = \Sigma(2, 3, 6, 7)$
(b)* $F(A, B, C, D) = \Sigma(4, 6, 7, 15)$
(c)* $F(A, B, C, D) = \Sigma(3, 7, 11, 13, 14, 15)$
(d)* $F(w, x, y, z) = \Sigma(2, 3, 12, 13, 14, 15)$
(e) $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 7, 13)$
(f) $F(w, x, y, z) = \Sigma(0, 1, 5, 8, 9)$

3.5 Simplify the following Boolean functions, using four-variable maps:
(a)* $F(w, x, y, z) = \Sigma(1, 4, 5, 6, 12, 14, 15)$
(b) $F(A, B, C, D) = \Sigma(1, 5, 9, 10, 11, 14, 15)$
(c) $F(w, x, y, z) = \Sigma(0, 1, 4, 5, 6, 7, 8, 9)$
(d)* $F(A, B, C, D) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$

3.6 Simplify the following Boolean expressions, using four-variable maps:
(a)* $A'B'C'D' + AC'D' + B'CD' + A'BCD + BC'D$
(b)* $x'z + w'xy' + w(x'y + xy')$
(c) $A'B'C'D' + A'CD' + AB'D' + ABCD + A'BD$
(d) $A'B'C'D' + AB'C + B'CD' + ABCD' + BC'D$

3.7 Simplify the following Boolean expressions, using four-variable maps:
(a)* $w'z + xz + x'y + wx'z$
(b) $C'D + A'B'C + ABC' + AB'C$
(c)* $AB'C + B'C'D' + BCD + ACD' + A'B'C + A'BC'D$
(d) $xyz + wy + wxy' + x'y$

3.8 Find the minterms of the following Boolean expressions by first plotting each function in a map:
(a)* $xy + yz + xy'z$
(b)* $C'D + ABC' + ABD' + A'B'D$
(c) $wyz + w'x' + wxz'$
(d) $A'B + A'CD + B'CD + BC'D'$

3.9 Find all the prime implicants for the following Boolean functions, and determine which are essential:

(a)* $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
(b)* $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c) $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$
(d) $F(w, x, y, z) = \Sigma(1, 3, 6, 7, 8, 9, 12, 13, 14, 15)$
(e) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$
(f) $F(w, x, y, z) = \Sigma(0, 2, 7, 8, 9, 10, 12, 13, 14, 15)$

3.10 Simplify the following Boolean functions by first finding the essential prime implicants:

(a) $F(w, x, y, z) = \Sigma(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
(b) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$
(c)* $F(A, B, C, D) = \Sigma(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$
(d) $F(w, x, y, z) = \Sigma(1, 3, 6, 7, 8, 9, 12, 13, 14, 15)$
(e) $F(A, B, C, D) = \Sigma(0, 2, 3, 5, 7, 8, 10, 11, 13, 15)$
(f) $F(w, x, y, z) = \Sigma(0, 2, 7, 8, 9, 10, 12, 13, 14, 15)$

3.11 Simplify the following Boolean functions, using five-variable maps:

(a)* $F(A, B, C, D, E) = \Sigma(0, 1, 4, 5, 16, 17, 21, 25, 29)$
(b) $F(A, B, C, D) = A'B'CE' + B'C'D'E' + A'B'D' + B'CD' + A'CD + A'BD$

3.12 Simplify the following Boolean functions to product-of-sums form:

(a) $F(w, x, y, z) = \Sigma(0, 1, 2, 5, 8, 10, 13)$
(b)* $\mathrm{F(A, B, C, D)} = \Pi\ (1, 3, 5, 7, 13, 15)$
(c) $F(A, B, C, D) = \Pi\ (1, 3, 6, 9, 11, 12, 14)$

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) $(A + C' + D')(A' + B' + D')(A' + B + D')(A' + B + C')$
(d) $ABC' + AB'D + BCD$

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

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

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

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

(c) $F(A, B, C, D) = \Sigma(4, 5, 7, 12, 13, 14)$
$d(A, B, C, D) = \Sigma(1, 9, 11, 15)$

(d) $F(A, B, C, D) = \Sigma(1, 3, 8, 10, 15)$
$d(A, B, C, D) = \Sigma(0, 2, 9)$

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

(a) $F(A, B, C, D) = A'B'C + AC' + ACD + ACD' + A'B'D'$
(b) $F(A, B, C, D) = AB + A'BC + A'B'C'D$
(c) $F(A, B, C) = (A' + B' + C')(A' + B')(A' + C')$
(d) $F(A, B, C, D) = A'B + A + C' + D'$

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

$$F(A, B, C, D) = \Sigma(0, 1, 2, 3, 4, 8, 9, 12)$$

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) $F(w, x, y, z) = \Sigma(1, 2, 13, 14)$
(c) $F(x, y, z) = [(x + y)(x' + z)]'$

3.20 Draw the multi-level NOR and multi-level NAND circuits for the following expression:

$$(AB' + CD')E + BC(A + B)$$

3.21 Draw the multi-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:

$$F(A, B, C, D) = \Sigma(2, 4, 6, 10, 12)$$
$$d(A, B, C, D) = \Sigma(0, 8, 9, 13)$$

Assume that both the normal and complement inputs are available.

3.24 Implement the following Boolean function F, using the two-level forms of logic (a) NAND-AND, (b) AND-NOR, (c) OR-NAND, and (d) NOR-OR:

$$F(A, B, C, D) = \Sigma(0, 4, 8, 9, 10, 11, 12, 14)$$

3.25 List the eight degenerate two-level forms and show that they reduce to a single operation. Explain how the degenerate two-level forms can be used to extend the number of inputs to a gate.

3.26 With the use of maps, find the simplest sum-of-products form of the function $F = fg$, where

$$f = abc' + c'd + a'cd' + b'cd'$$

and

$$g = (a + b + c' + d')(b' + c' + d)(a' + c + d')$$

3.27 Show that the dual of the exclusive-OR is also its complement.

3.28 Derive the circuits for a three-bit parity generator and four-bit parity checker using an odd parity bit.

3.29 Implement the following four Boolean expressions with three half adders

$$D = A \oplus B \oplus C$$
$$E = A'BC + AB'C$$
$$F = ABC' + (A' + B')C$$
$$G = ABC$$

3.30* Implement the following Boolean expression with exclusive-OR and AND gates:

$$F = AB'CD' + A'BCD' + AB'C'D + A'BC'D$$

3.31 Write a Verilog gate-level description of the circuit shown in

(a) Fig. 3.22(a) (b) Fig. 3.22(b) (c) Fig. 3.23(a)
(d) Fig. 3.23(b) (e) Fig. 3.26 (f) Fig. 3.27

3.32 Using continuous assignment statements, write a Verilog description of the circuit shown in

(a) Fig. 3.22(a) (b) Fig. 3.22(b) (c) Fig. 3.23(a)
(d) Fig. 3.23(b) (e) Fig. 3.26 (f) Fig. 3.27

3.33 The exclusive-OR circuit of Fig. 3.32(a) has gates with a delay of 4 ns for an inverter, a 8 ns delay for an AND gate, and a 10 ns delay for an OR gate. The input of the circuit goes from $xy = 00$ to $xy = 01$.

(a) Determine the signals at the output of each gate from $t = 0$ to $t = 50$ ns.
(b) Write a Verilog gate-level description of the circuit, including the delays.
(c) Write a stimulus module (i.e., a test bench similar to HDL Example 3.3), and simulate the circuit to verify the answer in part (a).

3.34 Using continuous assignments, write a Verilog description of the circuit specified by the following Boolean functions:

$$Out\_1 = (C + B)(A' + D)B'$$
$$Out\_2 = (CB' + ABC + C'B)(A + D')$$
$$Out\_3 = C(AD + B) + BA'$$

Write a test bench and simulate the circuit's behavior.

3.35* Find the syntax errors in the following declarations (note that names for primitive gates are optional):

```
module Exmpl-3(A, B, C, D, F)              // Line 1
  inputs        A, B, C, Output D, F,      // Line 2
  output        B                          // Line 3
  and           g1(A, B, D);               // Line 4
  not           (D, A, C),                 // Line 5
  OR            (F, B; C);                 // Line 6
endofmodule;                               // Line 7
```

3.36 Draw the logic diagram of the digital circuit specified by the following Verilog description:

```
(a) module Circuit_A (A, B, C, D, F);
      input   A, B, C, D;
      output          F;
      wire    w, x, y, z, a, d;
      and     (x, B, C, d);
      and     (y, a ,C);
      and     (w, z ,B);
      or      (z, y, A);
      or      (F, x, w);
      not     (a, A);
      not     (d, D);
    endmodule
```

(b)

```
module Circuit_B (A_gtB, A_ltB, A_eqB, A0, A1, B0, B1);
  output A_gtB, A_ltB, A_eqB;
  input  A0, A1, B0, B1;
  nor    (A_gtB, A_ltB, A_eqB);
  or     (A_ltB, w1, w2, w3);
  and    (A_eqB, w4, w5);
  and    (w1, w6, B1);
  and    (w2, w6, w7, B0);
  and    (w3, w7, B0, B1);
  not    (w6, A1);
  not    (w7, A0);
  xnor   (w4, A1, B1);
  xnor   (w5, A0, B0);
endmodule
```

(c)

```
module Circuit_C (output y1, input a, b, output y2);
  assign y1 = a & b;
  or (y2, a, b);
endmodule
```

3.37 A majority logic function is a Boolean function that is equal to 1 if the majority of the variables are equal to 1, equal to 0 otherwise. Write a Verilog user-defined primitive for a four-bit majority function.

3.38 Simulate the behavior of *Circuit_with_UDP_02467*, using the stimulus waveforms shown in Fig. P3.38.

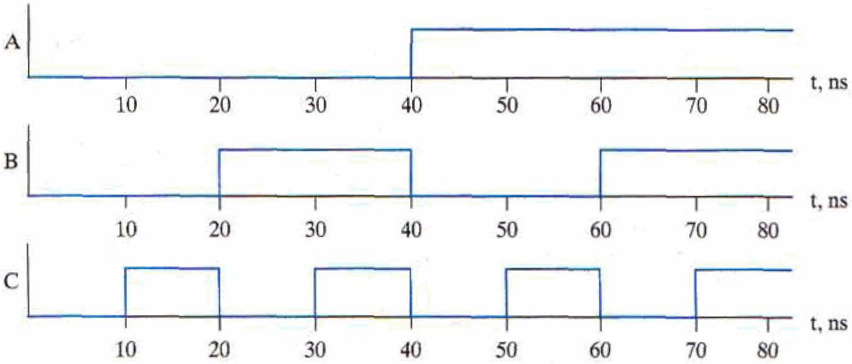


FIGURE P3.38
Stimulus waveforms for Problem 3.38

REFERENCES

1. BHASKER, J. 1997. *A Verilog HDL Primer.* Allentown, PA: Star Galaxy Press.
2. CILETTI, M.D. 1999. *Modeling, Synthesis and Rapid Prototyping with the Verilog HDL.* Upper Saddle River, NJ: Prentice Hall.
3. HILL, F. J., and G. R. PETERSON. 1981. *Introduction to Switching Theory and Logical Design*, 3d ed. New York: John Wiley.
4. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-1995). 1995. New York: The Institute of Electrical and Electronics Engineers.
5. KARNAUGH, M. A Map Method for Synthesis of Combinational Logic Circuits. *Transactions of AIEE, Communication and Electronics.* 72, part I (Nov. 1953): 593–99.
6. KOHAVI, Z. 1978. *Switching and Automata Theory*, 2d ed. New York: McGraw-Hill.
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).

Chapter 4

Combinational Logic

4.1 INTRODUCTION

Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs. A combinational circuit performs an operation that can be specified logically by a set of Boolean functions. In contrast, sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states. Sequential circuits are the building blocks of digital systems and are discussed in Chapters 5, 8, and 9.

4.2 COMBINATIONAL CIRCUITS

A combinational circuit consists of input variables, logic gates, and output variables. Combinational logic gates react to the values of the signals at their inputs and produce the value of the output signal, transforming binary information from the given input data to a required output data. A block diagram of a combinational circuit is shown in Fig. 4.1. The n input binary variables come from an external source; the m output variables are produced by the internal combinational logic circuit and go to an external destination. Each input and output variable exists physically as an analog signal whose values are interpreted to be a binary signal that represents logic 1 and logic 0. (*Note*: Logic simulators show only 0's and 1's, not the actual analog signals.) In many applications, the source and destination are storage registers. If the registers are included with the combinational gates, then the total circuit must be considered to be a sequential circuit.

FIGURE 4.1
Block diagram of combinational circuit

For n input variables, there are 2^n possible binary input combinations. For each possible input combination, there is one possible output value. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables. A combinational circuit also can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

In Chapter 1, we learned about binary numbers and binary codes that represent discrete quantities of information. The binary variables are represented physically by electric voltages or some other type of signal. The signals can be manipulated in digital logic gates to perform required functions. In Chapter 2, we introduced Boolean algebra as a way to express logic functions algebraically. In Chapter 3, we learned how to simplify Boolean functions to achieve economical (simpler) gate implementations. The purpose of the current chapter is to use the knowledge acquired in previous chapters to formulate systematic analysis and design procedures for combinational circuits. The solution of some typical examples will provide a useful catalog of elementary functions that are important for the understanding of digital systems. We'll address three tasks: (1) Analyze the behavior of a given logic circuit, (2) synthesize a circuit that will have a given behavior, and (3) write HDL models for some common circuits.

There are several combinational circuits that are employed extensively in the design of digital systems. These circuits are available in integrated circuits and are classified as standard components. They perform specific digital functions commonly needed in the design of digital systems. In this chapter, we introduce the most important standard combinational circuits, such as adders, subtractors, comparators, decoders, encoders, and multiplexers. These components are available in integrated circuits as medium-scale integration (MSI) circuits. They are also used as *standard cells* in complex very large-scale integrated (VLSI) circuits such as application-specific integrated circuits (ASICs). The standard cell functions are interconnected within the VLSI circuit in the same way that they are used in multiple-IC MSI design.

4.3 ANALYSIS PROCEDURE

The analysis of a combinational circuit requires that we determine the function that the circuit implements. This task starts with a given logic diagram and culminates with a set of Boolean functions, a truth table, or, possibly, an explanation of the circuit operation. If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function. The analysis can be performed manually by finding the Boolean functions or truth table or by using a computer simulation program.

The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate. Feedback paths in a digital circuit define a sequential circuit and must be analyzed according to procedures outlined in Chapter 9.

Once the logic diagram is verified to be that of a combinational circuit, one can proceed to obtain the output Boolean functions or the truth table. If the function of the circuit is under investigation, then it is necessary to interpret the operation of the circuit from the derived Boolean functions or truth table. The success of such an investigation is enhanced if one has previous experience and familiarity with a wide variety of digital circuits.

To obtain the output Boolean functions from a logic diagram, we proceed as follows:

1. Label all gate outputs that are a function of input variables with arbitrary symbols—but with meaningful names. Determine the Boolean functions for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the Boolean functions for these gates.
3. Repeat the process outlined in step 2 until the outputs of the circuit are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

The analysis of the combinational circuit of Fig. 4.2 illustrates the proposed procedure. We note that the circuit has three binary inputs—A, B, and C—and two binary outputs—F_1 and F_2.

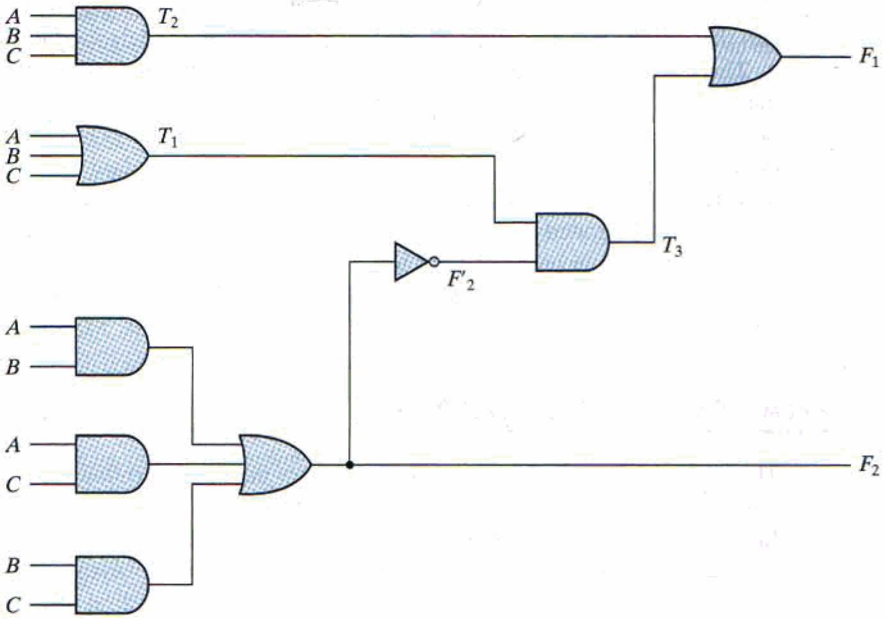


FIGURE 4.2
Logic diagram for analysis example

The outputs of various gates are labeled with intermediate symbols. The outputs of gates that are a function only of input variables are T_1 and T_2. Output F_2 can easily be derived from the input variables. The Boolean functions for these three outputs are

$$F_2 = AB + AC + BC$$
$$T_1 = A + B + C$$
$$T_2 = ABC$$

Next, we consider outputs of gates that are a function of already defined symbols:

$$T_3 = F_2'T_1$$
$$F_1 = T_3 + T_2$$

To obtain F_1 as a function of A, B, and C, we form a series of substitutions as follows:

$$\begin{aligned} F_1 = T_3 + T_2 = F_2'T_1 + ABC &= (AB + AC + BC)'(A + B + C) + ABC \\ &= (A' + B')(A' + C')(B' + C')(A + B + C) + ABC \\ &= (A' + B'C')(AB' + AC' + BC' + B'C) + ABC \\ &= A'BC' + A'B'C + AB'C' + ABC \end{aligned}$$

If we want to pursue the investigation and determine the information transformation task achieved by this circuit, we can draw the circuit from the derived Boolean expressions and try to recognize a familiar operation. The Boolean functions for F_1 and F_2 implement a circuit discussed in Section 4.5. Merely finding a Boolean representation of a circuit doesn't provide insight into its behavior, but in this example we will observe that the Boolean equations and truth table for F_1 and F_2 match those describing the functionality of what we call a full adder.

The derivation of the truth table for a circuit is a straightforward process once the output Boolean functions are known. To obtain the truth table directly from the logic diagram without going through the derivations of the Boolean functions, we proceed as follows:

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from 0 to $2^n - 1$ in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

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 that 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

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

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.
3. Obtain the simplified Boolean functions for each output as a function of the input variables.
4. Draw the logic diagram and verify the correctness of the design (manually or by simulation).

A truth table for a combinational circuit consists of input columns and output columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the outputs are determined from the stated specifications. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly in the truth table, as they are often incomplete, and any wrong interpretation may result in an incorrect truth table.

The output binary functions listed in the truth table are simplified by any available method, such as algebraic manipulation, the map method, or a computer-based simplification program. Frequently, there is a variety of simplified expressions from which to choose. In a particular application, certain criteria will serve as a guide in the process of choosing an implementation. A practical design must consider such constraints as the number of gates, number of inputs to a gate, propagation time of the signal through the gates, number of interconnections, limitations of the driving capability of each gate (i.e., the number of gates to which the output of the circuit may be connected), and various other criteria that must be taken into consideration when designing integrated circuits. Since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement about what constitutes an acceptable implementation. In most cases, the simplification begins by satisfying an elementary objective, such as producing the simplified Boolean functions in a standard form. Then the simplification proceeds with further steps to meet other performance criteria.

Code Conversion Example

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be inserted between the two systems if each uses different codes for the same information. Thus, a code converter is a circuit that makes the two systems compatible even though each uses a different binary code.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. The design procedure will be illustrated by an example that converts binary coded decimal (BCD) to the excess-3 code for the decimal digits.

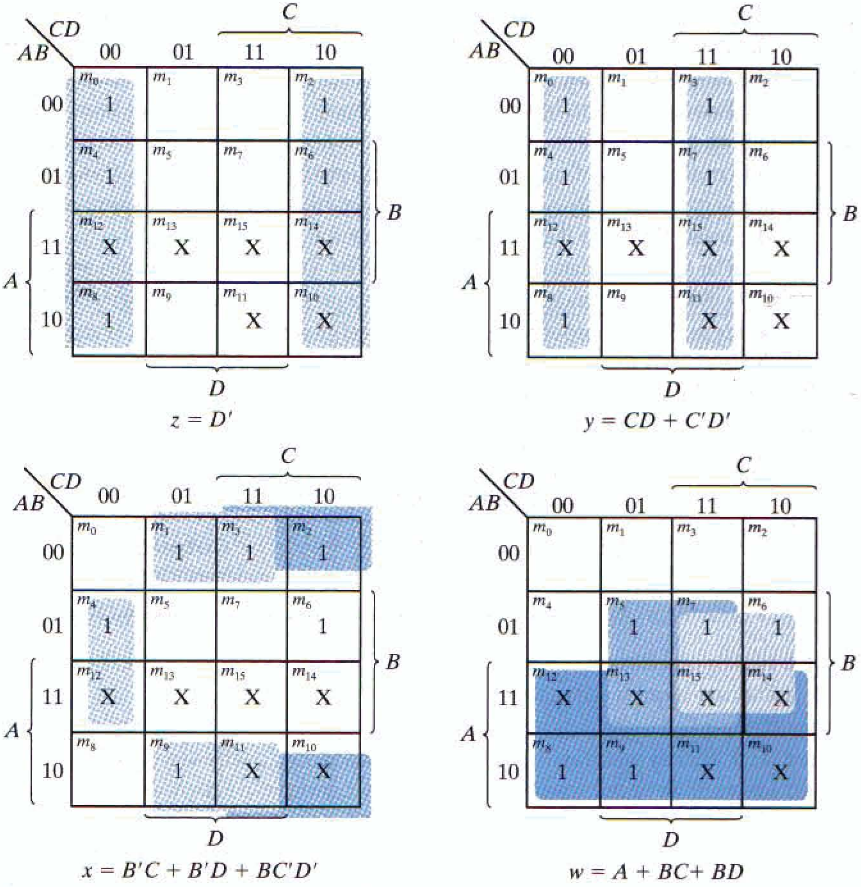
The bit combinations assigned to the BCD and excess-3 codes are listed in Table 1.5 (Section 1.7). Since each code uses four bits to represent a decimal digit, there must be four input variables and four output variables. We designate the four input binary variables by the symbols A, B, C, and D, and the four output variables by w, x, y, and z. The truth table relating the input and output variables is shown in Table 4.2. The bit combinations for the inputs and their

Table 4.2
Truth Table for Code-Conversion Example

Input BCD				Output Excess-3 Code			
A	***B***	***C***	***D***	***w***	***x***	***y***	***z***
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

corresponding outputs are obtained directly from Section 1.7. Note that four binary variables may have 16 bit combinations, but only 10 are listed in the truth table. The six bit combinations not listed for the input variables are don't-care combinations. These values have no meaning in BCD and we assume that they will never occur. Therefore, we are at liberty to assign to the output variables either a 1 or a 0, whichever gives a simpler circuit.

The maps in Fig. 4.3 are plotted to obtain simplified Boolean functions for the outputs. Each one of the four maps represents one of the four outputs of the circuit as a function of the four input variables. The 1's marked inside the squares are obtained from the minterms that make the output equal to 1. The 1's are obtained from the truth table by going over the output columns one at a time. For example, the column under output z has five 1's; therefore, the map for z has five 1's, each being in a square corresponding to the minterm that makes z equal to 1. The six don't-care minterms 10 through 15 are marked with an X. One possible way to simplify the functions into sum-of-products form is listed under the map of each variable. (See Chapter 3.)



$z = D'$

$y = CD + C'D'$

$x = B'C + B'D + BC'D'$

$w = A + BC + BD$

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

A two-level logic diagram may be obtained directly from the Boolean expressions derived from the maps. There are various other possibilities for a logic diagram that implements this circuit. The expressions obtained in Fig. 4.3 may be manipulated algebraically for the purpose of using common gates for two or more outputs. This manipulation, shown next, illustrates the flexibility obtained with multiple-output systems when implemented with three or more levels of gates:

$$\begin{aligned} z &= D' \\ y &= CD + C'D' = CD + (C + D)' \\ x &= B'C + B'D + BC'D' = B'(C + D) + BC'D' \\ &= B'(C + D) + B(C + D)' \\ w &= A + BC + BD = A + B(C + D) \end{aligned}$$

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 implementation will require inverters for variables B, C, and D, and the second implementation will require inverters for variables B and D. Thus, the three-level logic circuit requires fewer gates, all of which in turn require no more than two inputs.

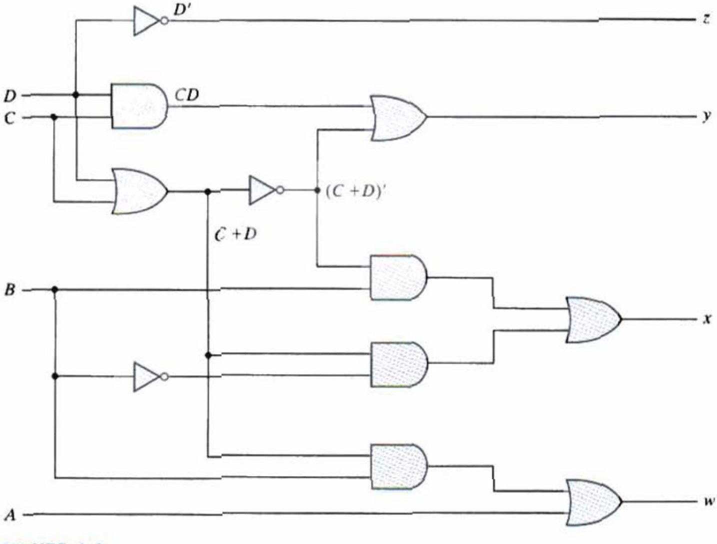


FIGURE 4.4
Logic diagram for BCD-to-excess-3 code converter

4.5 BINARY ADDER–SUBTRACTOR

Digital computers perform a variety of information-processing tasks. Among the functions encountered are the various arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, and $1 + 1 = 10$. The first three operations produce a sum of one digit, but when both augend and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant digits, the carry obtained from the addition of two bits is added to the next higher order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half adder*. One that performs the addition of three bits (two significant bits and a previous carry) is a *full adder*. The names of the circuits stem from the fact that two half adders can be employed to implement a full adder.

A binary adder–subtractor is a combinational circuit that performs the arithmetic operations of addition and subtraction with binary numbers. We will develop this circuit by means of a hierarchical design. The half adder design is carried out first, from which we develop the full adder. Connecting n full adders in cascade produces a binary adder for two n-bit numbers. The subtraction circuit is included in a complementing circuit.

Half Adder

From the verbal explanation of a half adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs. The truth table for the half adder is listed in Table 4.3. The C output is 1 only when both inputs are 1. The S output represents the least significant bit of the sum.

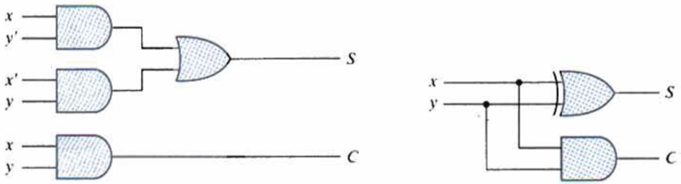
The simplified Boolean functions for the two outputs can be obtained directly from the truth table. The simplified sum-of-products expressions are

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

The logic diagram of the half adder implemented in sum of products is shown in Fig. 4.5(a). It can be also implemented with an exclusive-OR and an AND gate as shown in Fig. 4.5(b). This form is used to show that two half adders can be used to construct a full adder.

Table 4.3
Half Adder

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



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

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

FIGURE 4.5
Implementation of half adder

Full Adder

A full adder is a combinational circuit that forms the arithmetic sum of three bits. It consists of three inputs and two outputs. Two of the input variables, denoted by x and y, represent the two significant bits to be added. The third input, z, represents the carry from the previous lower significant position. Two outputs are necessary because the arithmetic sum of three binary digits ranges in value from 0 to 3, and binary 2 or 3 needs two digits. The two outputs are designated by the symbols S for sum and C for carry. The binary variable S gives the value of the least significant bit of the sum. The binary variable C gives the output carry. The truth table of the full adder is listed in Table 4.4. The eight rows under the input variables designate all possible combinations of the three variables. The output variables are determined from the arithmetic sum of the input bits. When all input bits are 0, the output is 0. The S output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The C output has a carry of 1 if two or three inputs are equal to 1.

The input and output bits of the combinational circuit have different interpretations at various stages of the problem. On the one hand, physically, the binary signals of the inputs are considered binary digits to be added arithmetically to form a two-digit sum at the output. On the other hand, the same binary values are considered as variables of Boolean functions when expressed in the truth table or when the circuit is implemented with logic gates. The maps for the outputs of the full adder are shown in Fig. 4.6. The simplified expressions are

Table 4.4
Full Adder

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

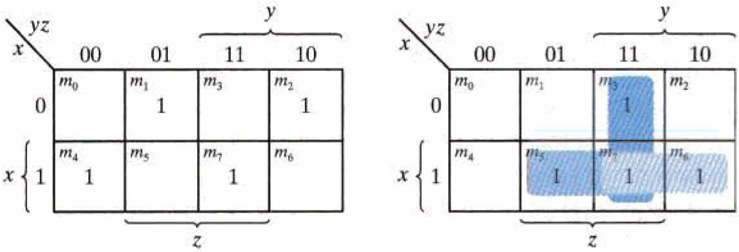


FIGURE 4.6
Maps for full adder

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

The logic diagram for the full adder implemented in sum-of-products form is shown in Fig. 4.7. It can also be implemented with two half adders and one OR gate, as shown 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

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

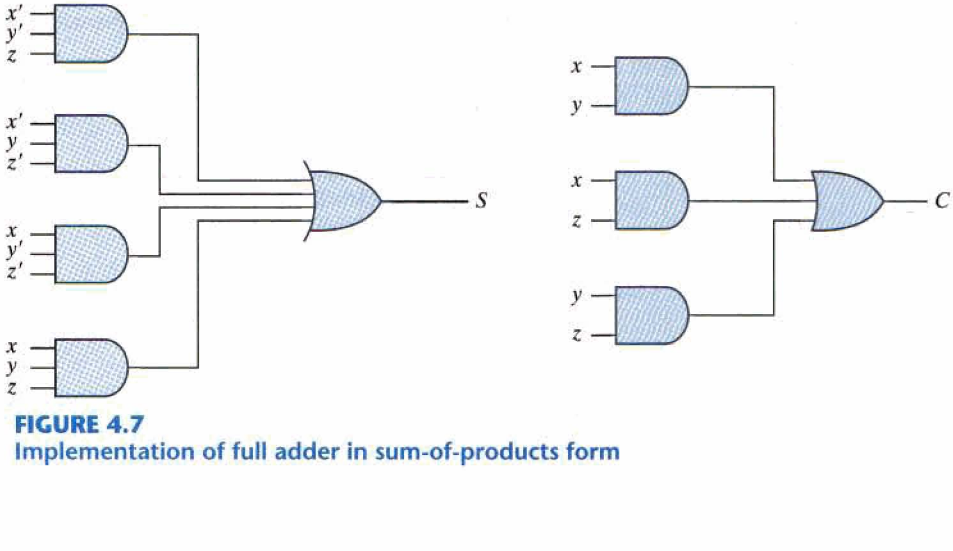


FIGURE 4.7
Implementation of full adder in sum-of-products form

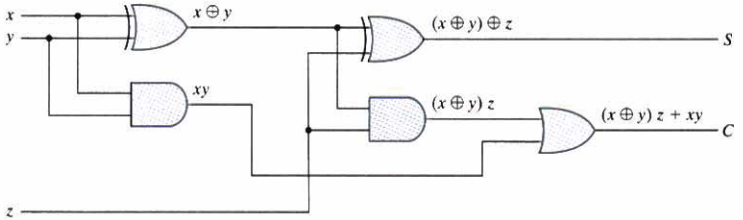


FIGURE 4.8
Implementation of full adder with two half adders and an OR gate

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. Figure 4.9 shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the least significant bit. The carries are connected in a chain through the full adders. The input carry to the adder is C_0, and it ripples through the full adders to the output carry C_4. The S outputs generate the required sum bits. An n-bit adder requires n full adders, with each output carry connected to the input carry of the next higher order full adder.

To demonstrate with a specific example, consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum $S = 1110$ is formed with the four-bit adder as follows:

Subscript i:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}

The bits are added with full adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The value of C_{i+1} in a given significant position is the output carry of the full adder. This value is transferred into the input carry of the full adder that adds the bits one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated. All the carries must be generated for the correct sum bits to appear at the outputs.

The four-bit adder is a typical example of a standard component. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to

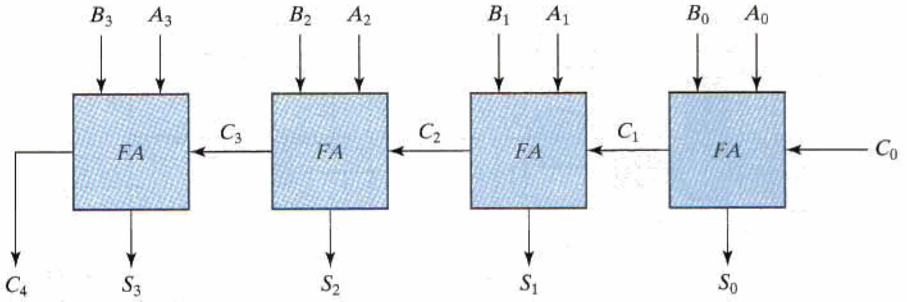


FIGURE 4.9
Four-bit adder

the circuit. By using an iterative method of cascading a standard function, it is possible to obtain a simple and straightforward implementation.

Carry Propagation

The addition of two binary numbers in parallel implies that all the bits of the augend and addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate, times the number of gate levels in the circuit. The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i at any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. In this regard, consider output S_3 in Fig. 4.9. Inputs A_3 and B_3 are available as soon as input signals are applied to the adder. However, input carry C_3 does not settle to its final value until C_2 is available from the previous stage. Similarly, C_2 has to wait for C_1 and so on down to C_0. Thus, only after the carry propagates and ripples through all stages will the last output S_3 and carry C_4 settle to their final correct value.

The number of gate levels for the carry propagation can be found from the circuit of the full adder. The circuit is redrawn with different labels in Fig. 4.10 for convenience. The input and

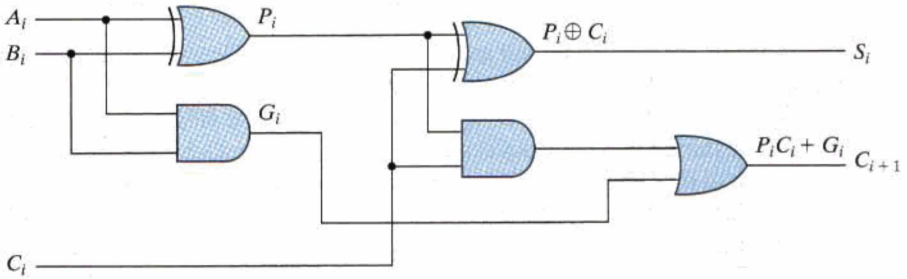


FIGURE 4.10
Full adder with *P* and *G* shown

output variables use the subscript i to denote a typical stage of the adder. The signals at P_i and G_i settle to their steady-state values after they propagate through their respective gates. These two signals are common to all full adders and depend only on the input augend and addend bits. The signal from the input carry C_i to the output carry C_{i+1} propagates through an AND gate and an OR gate, which constitute two gate levels. If there are four full adders in the adder, the output carry C_4 would have $2 \times 4 = 8$ gate levels from C_0 to C_4. For an n-bit adder, there are $2n$ gate levels for the carry to propagate from input to output.

The carry propagation time is an important attribute of the adder because it limits the speed with which two numbers are added. Although the adder—or, for that matter, any combinational circuit—will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is critical. An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays. However, physical circuits have a limit to their capability. Another solution is to increase the complexity of the equipment in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *carry lookahead logic*.

Consider the circuit of the full adder shown in Fig. 4.10. If we define two new binary variables

$$
\begin{aligned}
P_i &= A_i \oplus B_i \\
G_i &= A_iB_i
\end{aligned}
$$

the output sum and carry can respectively be expressed as

$$
\begin{aligned}
S_i &= P_i \oplus C_i \\
C_{i+1} &= G_i + P_iC_i
\end{aligned}
$$

G_i is called a *carry generate*, and it produces a carry of 1 when both A_i and B_i are 1, regardless of the input carry C_i. P_i is called a *carry propagate*, because it determines whether a carry into stage i will propagate into stage $i + 1$ (i.e., whether an assertion of C_i will propagate to an assertion of C_{i+1}).

We now write the Boolean functions for the carry outputs of each stage and substitute the value of each C_i from the previous equations:

$$
\begin{aligned}
C_0 &= \text{input carry} \\
C_1 &= G_0 + P_0C_0 \\
C_2 &= G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0 \\
C_3 &= G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 = P_2P_1P_0C_0
\end{aligned}
$$

Since the Boolean function for each output carry is expressed in sum-of-products form, each function can be implemented with one level of AND gates followed by an OR gate (or by a two-level NAND). The three Boolean functions for C_1, C_2, and C_3 are implemented in the carry lookahead generator shown in Fig. 4.11. Note that this circuit can add in less time because C_3 does not have to wait for C_2 and C_1 to propagate; in fact, C_3 is propagated at the same time as C_1 and C_2. This gain in speed of operation is achieved at the expense of additional complexity (hardware).

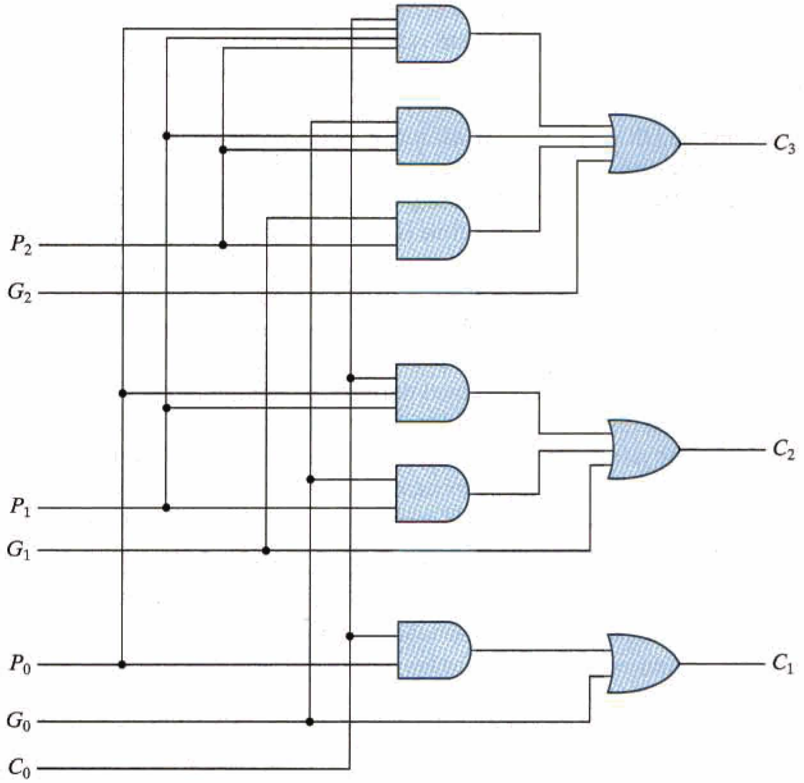


FIGURE 4.11
Logic diagram of carry lookahead generator

The construction of a four-bit adder with a carry lookahead scheme is shown in Fig. 4.12. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry lookahead generator (similar to that in Fig. 4.11) and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times. The two-level circuit for the output carry C_4 is not shown. This circuit can easily be derived by the equation-substitution method.

Binary Subtractor

The subtraction of unsigned binary numbers can be done most conveniently by means of complements, as discussed in Section 1.5. Remember that the subtraction $A - B$ can be done by taking the 2's complement of B and adding it to A. The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits. The 1's complement can be implemented with inverters, and a 1 can be added to the sum through the input carry.

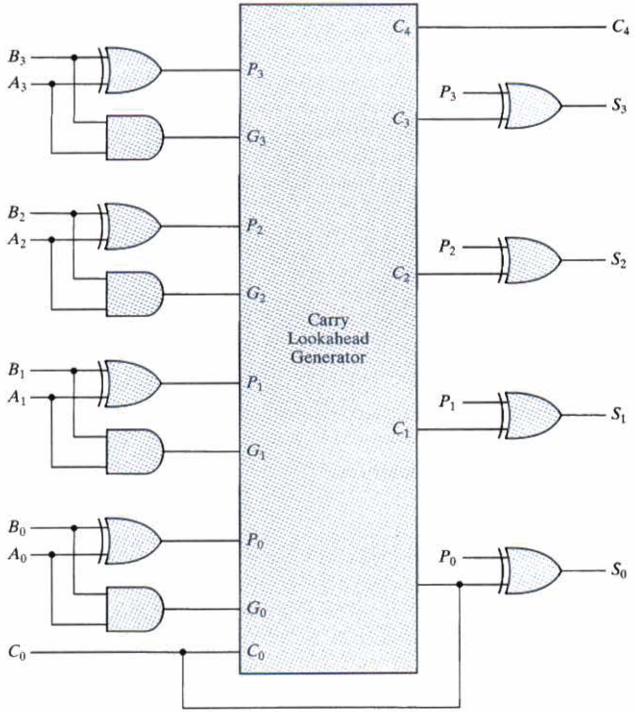


FIGURE 4.12
Four-bit adder with carry lookahead

The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when subtraction is performed. The operation thus performed becomes A, plus the 1's complement of B, plus 1. This is equal to A plus the 2's complement of B. For unsigned numbers, that gives $A - B$ if $A \geq B$ or the 2's complement of $(B - A)$ if $A < B$. For signed numbers, the result is $A - B$, provided that there is no overflow. (See Section 1.6.)

The addition and subtraction operations can be combined into one circuit with one common binary adder by including an exclusive-OR gate with each full adder. A four-bit adder–subtractor circuit is shown in Fig. 4.13. The mode input M controls the operation. When $M = 0$, the circuit is an adder, and when $M = 1$, the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When $M = 0$, we have $B \oplus 0 = B$. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When $M = 1$,

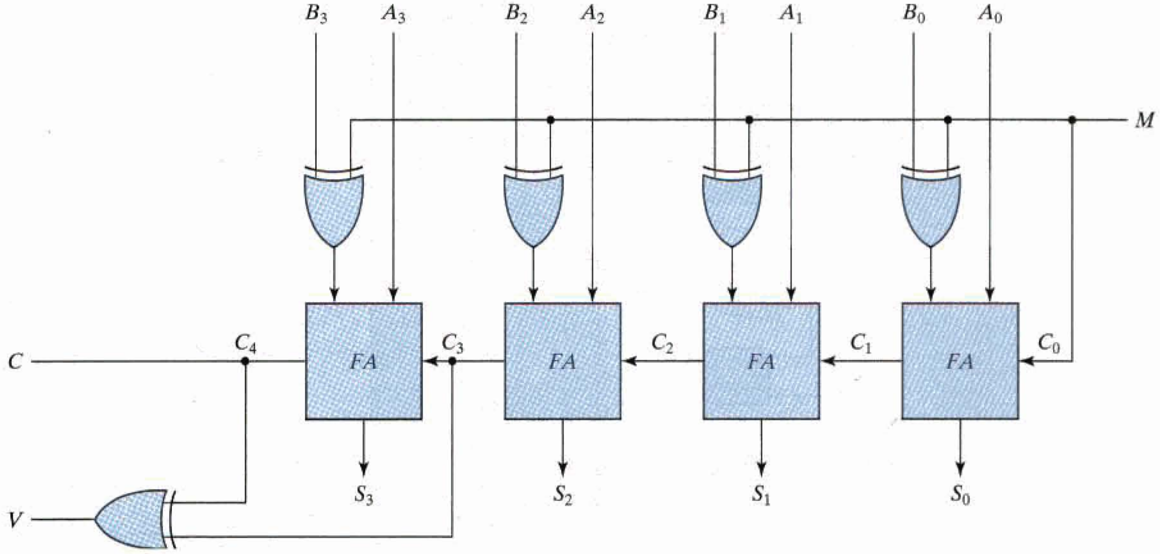


FIGURE 4.13
Four-bit adder–subtractor

we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B. (The exclusive-OR with output V is for detecting an overflow.)

It is worth noting that binary numbers in the signed-complement system are added and subtracted by the same basic addition and subtraction rules as are unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently, depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

When two numbers with n digits each are added and the sum is a number occupying $n + 1$ digits, we say that an overflow occurred. This is true for binary or decimal numbers, signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit by the width of the page to write down the sum. Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains $n + 1$ bits cannot be accommodated by an n-bit word. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set that can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, two details are important: the leftmost bit always represents the sign, and negative numbers are in 2's-complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example: Two signed binary numbers, +70 and +80, are stored in two eight-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary −128. Since the sum of the two numbers is +150, it exceeds the capacity of an eight-bit register. This is also true for −70 and −80. The two additions in binary are shown next, together with the last two carries:

$$\begin{array}{rlcrl} \text{carries:} & 0\quad 1 & \qquad\qquad & \text{carries:} & 1\quad 0 \\ +70 & 0\ 1000110 & & -70 & 1\ 0111010 \\ \underline{+80} & \underline{0\ 1010000} & & \underline{-80} & \underline{1\ 0110000} \\ +150 & 1\ 0010110 & & -150 & 0\ 1101010 \end{array}$$

Note that the eight-bit result that should have been positive has a negative sign bit (i.e., the 8-th bit) and the eight-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, then the nine-bit answer so obtained will be correct. But since the answer cannot be accommodated within eight bits, we say that an overflow has occurred.

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow has occurred. This is indicated in the examples in which the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1. For this method to work correctly, the 2's complement of a negative number must be computed by taking the 1's complement and adding 1. This takes care of the condition when the maximum negative number is complemented.

The binary adder–subtractor circuit with outputs C and V is shown in Fig. 4.13. If the two binary numbers are considered to be unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow. If $V = 0$ after an addition or subtraction, then no overflow occurred and the n-bit result is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits, but only the rightmost n bits of the number fit in the space available, so an overflow has occurred. The $(n + 1)$th bit is the actual sign and has been shifted out of position.

4.6 DECIMAL ADDER

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and present results in the same code. For binary addition, it is sufficient to consider a pair of significant bits together with a previous carry. A decimal adder requires a minimum of nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input and output carry. There is a wide variety of possible decimal adder circuits, depending upon the code used to represent the decimal digits. Here we examine a decimal adder for the BCD code. (See Section 1.7.)

BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than $9 + 9 + 1 = 19$, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a four-bit binary adder. The adder will form the sum in *binary* and produce a result that ranges from 0 through 19. These binary numbers are listed in Table 4.5 and are labeled by symbols K, Z_8, Z_4, Z_2, and Z_1. K is the carry, and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The columns under the binary sum list the binary value that appears in the outputs of the four-bit binary adder. The output sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD Sum." The problem is to find a rule by which the binary sum is converted to the correct BCD digit representation of the number in the BCD sum.

In examining the contents of the table, it becomes apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation.

Table 4.5
Derivation of BCD Adder

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

The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the entries in the table. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 through 1111 that need a correction have a 1 in position Z_8. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_8, we specify further that either Z_4 or Z_2 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_8Z_4 + Z_8Z_2$$

When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

A BCD adder that adds two BCD digits and produces a sum digit in BCD is shown in Fig. 4.14. The two decimal digits, together with the input carry, are first added in the top four-bit adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum.

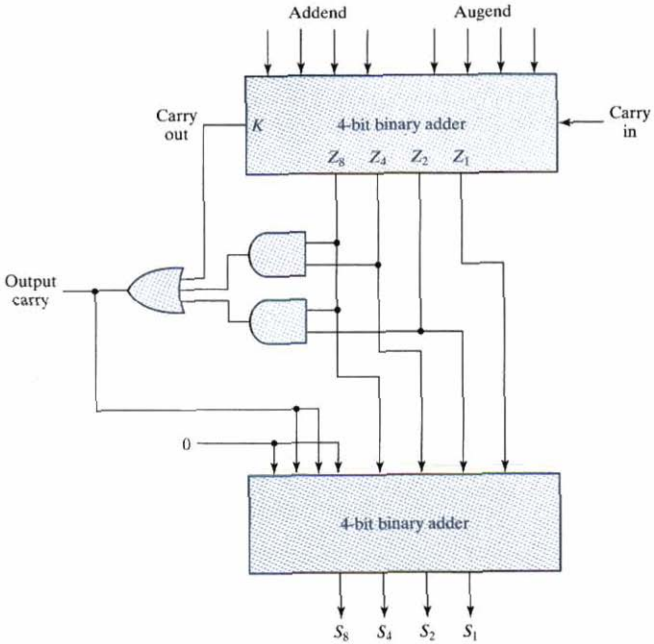


FIGURE 4.14
Block diagram of a BCD adder

When it is equal to 1, binary 0110 is added to the binary sum through the bottom four-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. A decimal parallel adder that adds n decimal digits needs n BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher order stage.

4.7 BINARY MULTIPLIER

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

To see how a binary multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 4.15. The multiplicand bits are B_1 and B_0, the multiplier bits are A_1 and A_0, and the product is $C_3C_2C_1C_0$. The first partial product is formed by multiplying B_1B_0 by A_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in the diagram. The second partial product is formed by multiplying B_1B_0 by A_1 and shifting one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial

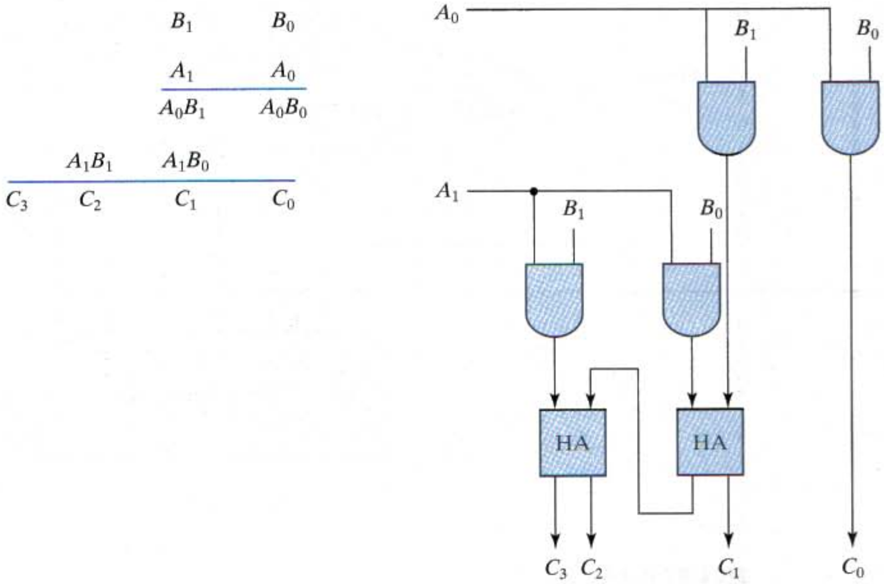


FIGURE 4.15
Two-bit by two-bit binary multiplier

products. Note that the least significant bit of the product does not have to go through an adder, since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates is added with the partial product of the previous level to form a new partial product. The last level produces the product. For J multiplier bits and K multiplicand bits, we need $(J \times K)$ AND gates and $(J - 1)$ K-bit adders to produce a product of $J + K$ bits.

As a second example, consider a multiplier circuit that multiplies a binary number represented by four bits by a number represented by three bits. Let the multiplicand be represented by $B_3B_2B_1B_0$ and the multiplier by $A_2A_1A_0$. Since $K = 4$ and $J = 3$, we need 12 AND gates and 2 four-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig. 4.16.

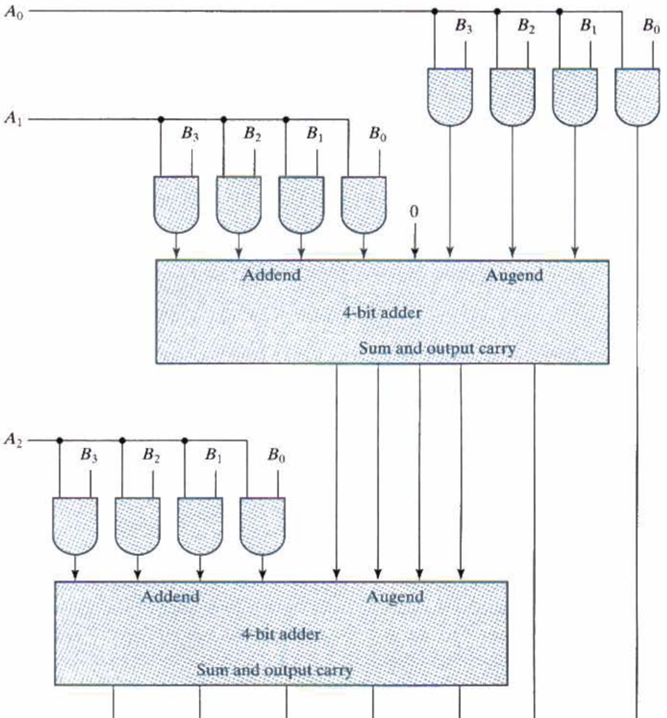


FIGURE 4.16
Four-bit by three-bit binary multiplier

4.8 MAGNITUDE COMPARATOR

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

On the one hand, the circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome, even with $n = 3$. On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions that possess an inherent well-defined regularity can usually be designed by means of an algorithm—a procedure which specifies a finite set of steps that, if followed, give the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a four-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitudes of two numbers. Consider two numbers, A and B, with four digits each. Write the coefficients of the numbers in descending order of significance:

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

Each subscripted letter represents one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal: $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$, and $A_0 = B_0$. When the numbers are binary, the digits are either 1 or 0, and the equality of each pair of bits can be expressed logically with an exclusive-NOR function as

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

where $x_i = 1$ only if the pair of bits in position i are equal (i.e., if both are 1 or both are 0).

The equality of the two numbers A and B is displayed in a combinational circuit by an output binary variable that we designate by the symbol $(A = B)$. This binary variable is equal to 1 if the input numbers, A and B, are equal, and is equal to 0 otherwise. For equality to exist, all x_i variables must be equal to 1, a condition that dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

The *binary* variable $(A = B)$ is equal to 1 only if all pairs of digits of the two numbers are equal.

To determine whether A is greater or less than B, we inspect the relative magnitudes of pairs of significant digits, starting from the most significant position. If the two digits of a pair are equal, we compare the next lower significant pair of digits. The comparison continues until a pair of unequal digits is reached. If the corresponding digit of A is 1 and that of B is 0, we conclude that $A > B$. If the corresponding digit of A is 0 and that of B is 1, we have $A < B$. The sequential comparison can be expressed logically by the two Boolean functions

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

The symbols $(A > B)$ and $(A < B)$ are *binary* output variables that are equal to 1 when $A > B$ and $A < B$, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The unequal outputs can use the same gates that are needed to generate the equal output. The logic diagram of the four-bit magnitude comparator is shown in Fig. 4.17. The four x outputs are generated with exclusive-NOR circuits and are applied to an AND gate to give the output binary variable $(A = B)$. The other two outputs use the x variables to generate the Boolean functions listed previously. This is a multilevel implementation and has a regular pattern. The procedure for obtaining magnitude comparator circuits for binary numbers with more than four bits is obvious from this example.

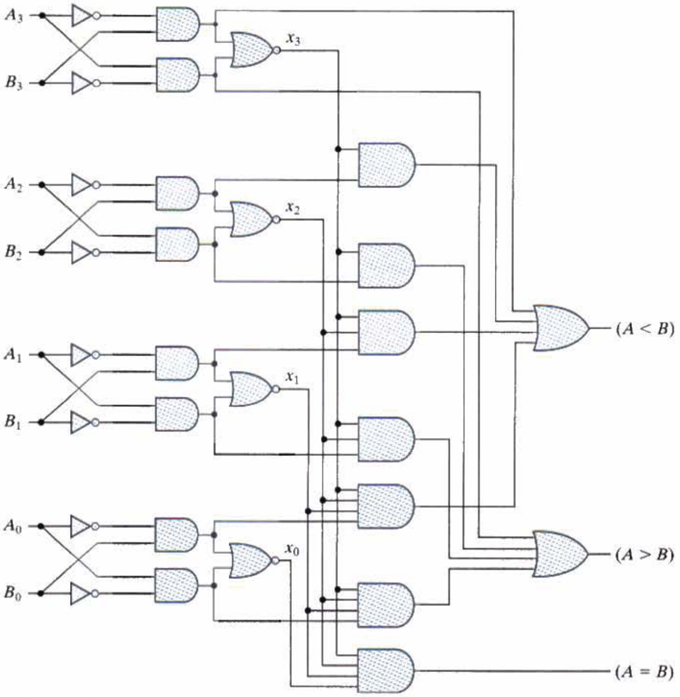


FIGURE 4.17
Four-bit magnitude comparator

4.9 DECODERS

Discrete quantities of information are represented in digital systems by binary codes. A binary code of n bits is capable of representing up to 2^n distinct elements of coded information. A *decoder* is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n-bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

The decoders presented here are called n-to-m-line decoders, where $m \leq 2^n$. Their purpose is to generate the 2^n (or fewer) minterms of n input variables. The name *decoder* is also used in conjunction with other code converters, such as a BCD-to-seven-segment decoder.

As an example, consider the three-to-eight-line decoder circuit of Fig. 4.18. The three inputs are decoded into eight outputs, each representing one of the minterms of the three input variables. The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms. A particular application of this decoder is binary-to-octal

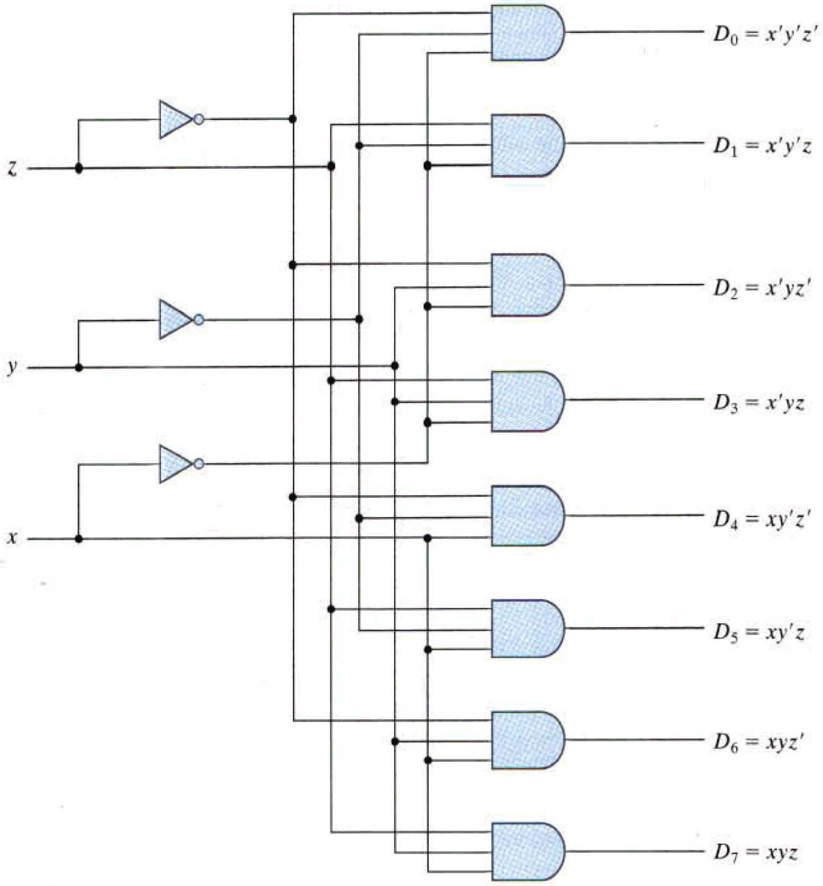


FIGURE 4.18
Three-to-eight-line decoder

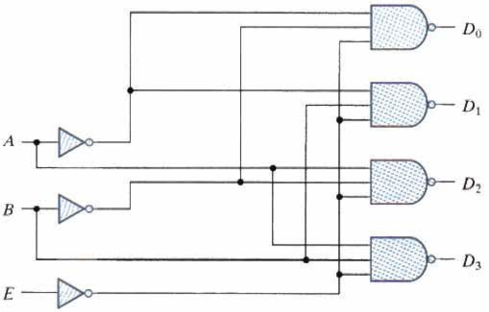
Table 4.6
Truth Table of a Three-to-Eight-Line Decoder

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

conversion. The input variables represent a binary number, and the outputs represent the eight digits of a number in the octal number system. However, a three-to-eight-line decoder can be used for decoding *any* three-bit code to provide eight outputs, one for each element of the code.

The operation of the decoder may be clarified by the truth table listed in Table 4.6. For each possible input combination, there are seven outputs that are equal to 0 and only one that is equal to 1. The output whose value is equal to 1 represents the minterm equivalent of the binary number currently available in the input lines.

Some decoders are constructed with NAND gates. Since a NAND gate produces the AND operation with an inverted output, it becomes more economical to generate the decoder minterms in their complemented form. Furthermore, decoders include one or more *enable* inputs to control the circuit operation. A two-to-four-line decoder with an enable input constructed with NAND gates is shown in Fig. 4.19. The circuit operates with complemented outputs and a complement



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

FIGURE 4.19
Two-to-four-line decoder with enable input

enable input. The decoder is enabled when E is equal to 0 (i.e., active-low enable). As indicated by the truth table, only one 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 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 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.

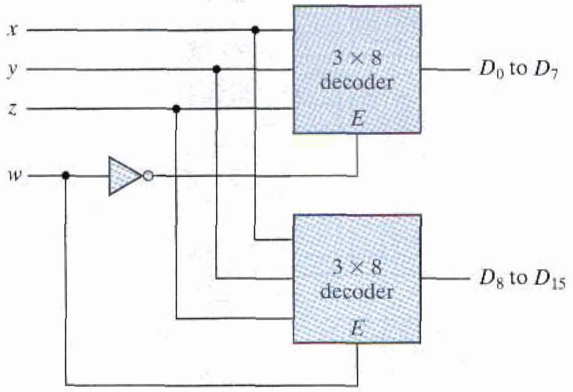


FIGURE 4.20
4 × 16 decoder constructed with two 3 × 8 decoders

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.

A function with a long list of minterms requires an OR gate with a large number of inputs. A function having a list of k minterms can be expressed in its complemented form F' with $2^n - k$ minterms. If the number of minterms in the function is greater than $2^n/2$, then F' can be expressed with fewer minterms. In such a case, it is advantageous to use a NOR gate to sum the minterms of F'. The output of the NOR gate complements this sum and generates the normal output F. If NAND gates are used for the decoder, as in Fig. 4.19, then the external gates must be NAND gates instead of OR gates. This is because a two-level NAND gate circuit implements a sum-of-minterms function and is equivalent to a two-level AND–OR circuit.

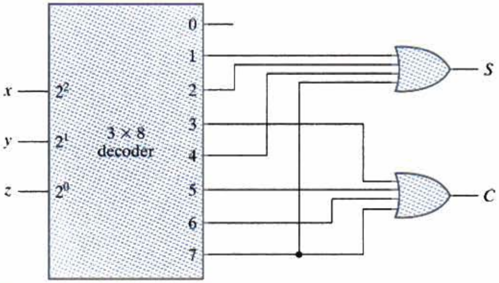


FIGURE 4.21
Implementation of a full adder with a decoder

4.10 ENCODERS

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines, as an aggregate, generate the binary code corresponding to the input value. An example of an encoder is the octal-to-binary encoder whose truth table is given in Table 4.7. It has eight inputs (one for each of the octal digits) and three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1 when the input octal digit is 1, 3, 5, or 7. Output y is 1 for octal digits 2, 3, 6, or 7, and output x is 1 for digits 4, 5, 6, or 7. These conditions can be expressed by the following Boolean output functions:

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

The encoder can be implemented with three OR gates.

The encoder defined in Table 4.7 has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111 because all three outputs are equal to 1. The output 111 does not represent either binary 3 or binary 6. To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.

Another ambiguity in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; but this output is the same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate whether at least one input is equal to 1.

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

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

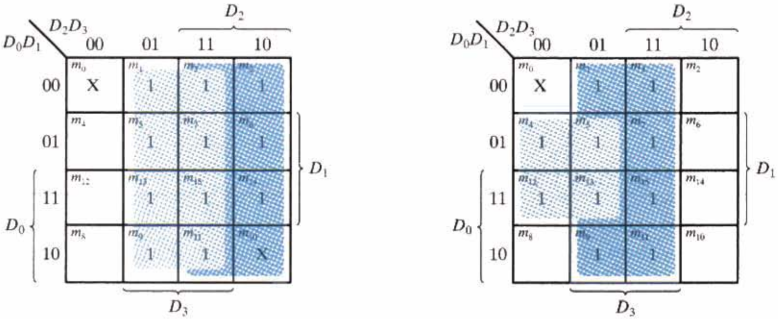
Table 4.8
Truth Table of a Priority Encoder

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

Priority Encoder

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 4.8. In addition to the two outputs x and y, the circuit has a third output designated by V; this is a *valid* bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0. The other two outputs are not inspected when V equals 0 and are specified as don't-care conditions. Note that whereas X's in output columns represent don't-care conditions, the X's in the input columns are useful for representing a truth table in condensed form. Instead of listing all 16 minterms of four variables, the truth table uses an X to represent either 1 or 0. For example, X100 represents the two minterms 0100 and 1100.

According to Table 4.8, the higher the subscript number, the higher the priority of the input. Input D_3 has the highest priority, so, regardless of the values of the other inputs, when this



$x = D_2 + D_3$

$y = D_3 + D_1D'_2$

FIGURE 4.22
Maps for a priority encoder

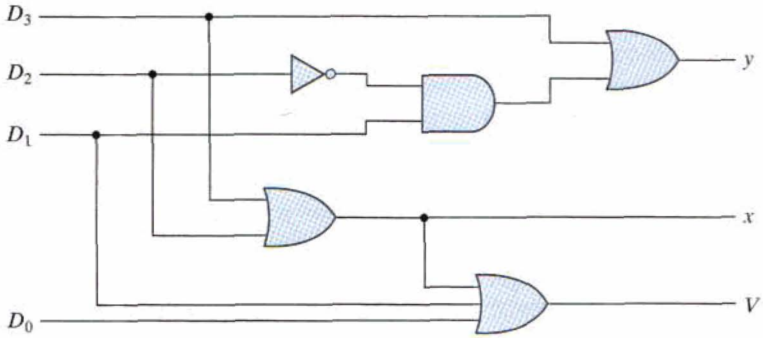


FIGURE 4.23
Four-input priority encoder

input is 1, the output for xy is 11 (binary 3). D_2 has the next priority level. The output is 10 if $D_2 = 1$, provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

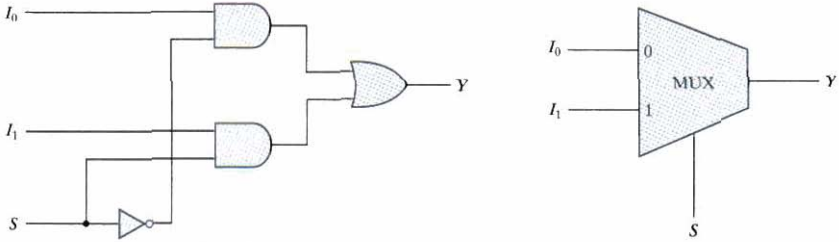
The maps for simplifying outputs x and y are shown in Fig. 4.22. The minterms for the two functions are derived from Table 4.8. Although the table has only five rows, when each X in a row is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the fourth row in the table, with inputs XX10, represents the four minterms 0010, 0110, 1010, and 1110. The simplified Boolean expressions for the priority encoder are obtained from the maps. The condition for output V is an OR function of all the input variables. The priority encoder is implemented in Fig. 4.23 according to the following Boolean functions:

$$x = D_2 + D_3$$
$$y = D_3 + D_1D_2'$$
$$V = D_0 + D_1 + D_2 + D_3$$

4.11 MULTIPLEXERS

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

A two-to-one-line multiplexer connects one of two 1-bit sources to a common destination, as shown in Fig. 4.24. The circuit has two data input lines, one output line, and one selection line S. When $S = 0$, the upper AND gate is enabled and I_0 has a path to the output. When $S = 1$, the lower AND gate is enabled and I_1 has a path to the output. The multiplexer acts like



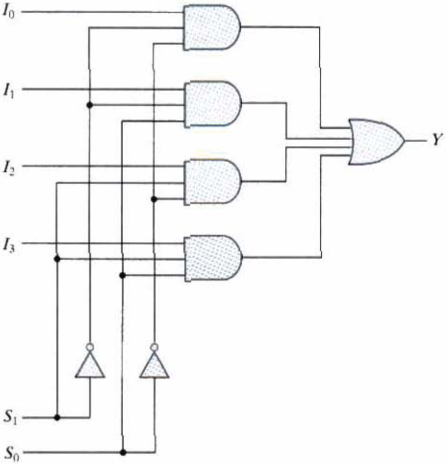
(a) Logic diagram

(b) Block diagram

FIGURE 4.24
Two-to-one-line multiplexer

an electronic switch that selects one of two sources. The block diagram of a multiplexer is sometimes depicted by a wedge-shaped symbol, as shown in Fig. 4.24(b). It suggests visually how a selected one of multiple data sources is directed into a single destination. The multiplexer is often labeled "MUX" in block diagrams.

A four-to-one-line multiplexer is shown in Fig. 4.25. Each of the four inputs, I_0 through I_3, is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a



(a) Logic diagram

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

(b) Function table

FIGURE 4.25
Four-to-one-line multiplexer

particular AND gate. The outputs of the AND gates are applied to a single OR gate that provides the one-line output. The function table lists the input that is passed to the output for each combination of the binary selection values. To demonstrate the operation of the circuit, consider the case when $S_1S_0 = 10$. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have at least one input equal to 0, which makes their outputs equal to 0. The output of the OR gate is now equal to the value of I_2, providing a path from the selected input to the output. A multiplexer is also called a *data selector*, since it selects one of many inputs and steers the binary information to the output line.

The AND gates and inverters in the multiplexer resemble a decoder circuit, and indeed, they decode the selection input lines. In general, a 2^n-to-1-line multiplexer is constructed from an n-to-2^n decoder by adding 2^n input lines to it, one to each AND gate. The outputs of the AND gates are applied to a single OR gate. The size of a multiplexer is specified by the number 2^n of its data input lines and the single output line. The n selection lines are implied from the 2^n data lines. As in decoders, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

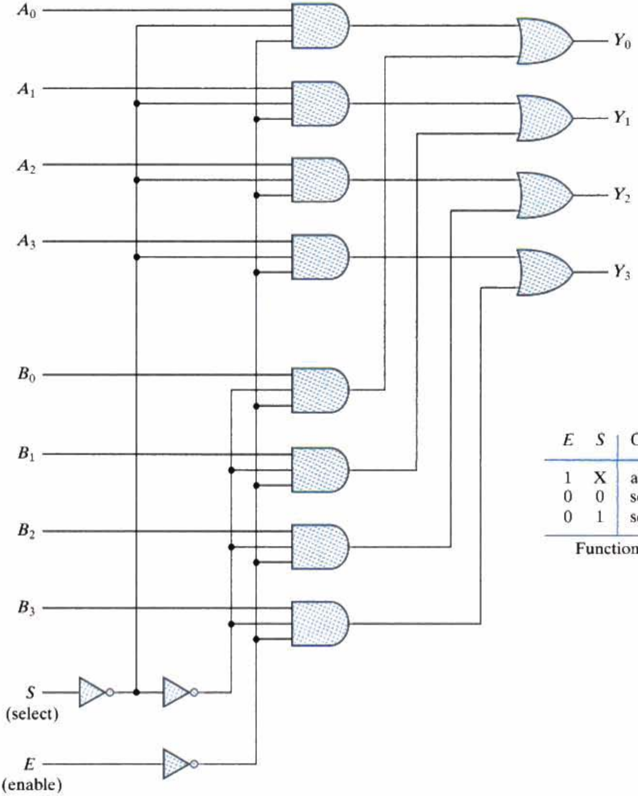
Multiplexer circuits can be combined with common selection inputs to provide multiple-bit selection logic. As an illustration, a quadruple 2-to-1-line multiplexer is shown in Fig. 4.26. The circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either input A_0 or input B_0. Similarly, output Y_1 may have the value of A_1 or B_1, and so on. Input selection line S selects one of the lines in each of the four multiplexers. The enable input E must be active (i.e., asserted) for normal operation. Although the circuit contains four 2-to-1-line multiplexers, we are more likely to view it as a circuit that selects one of two 4-bit sets of data lines. As shown in the function table, the unit is enabled when $E = 0$. Then, if $S = 0$, the four A inputs have a path to the four outputs. If, by contrast, $S = 1$, the four B inputs are applied to the outputs. The outputs have all 0's when $E = 1$, regardless of the value of S.

Boolean Function Implementation

In Section 4.9, it was shown that a decoder can be used to implement Boolean functions by employing external OR gates. An examination of the logic diagram of a multiplexer reveals that it is essentially a decoder that includes the OR gate within the unit. The minterms of a function are generated in a multiplexer by the circuit associated with the selection inputs. The individual minterms can be selected by the data inputs, thereby providing a method of implementing a Boolean function of n variables with a multiplexer that has n selection inputs and 2^n data inputs, one for each minterm.

We will now show a more efficient method for implementing a Boolean function of n variables with a multiplexer that has $n - 1$ selection inputs. The first $n - 1$ variables of the function are connected to the selection inputs of the multiplexer. The remaining single variable of the function is used for the data inputs. If the single variable is denoted by z, each data input of the multiplexer will be z, z', 1, or 0. To demonstrate this procedure, consider the Boolean function

$$F(x, y, z) = \Sigma(1, 2, 6, 7)$$



E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

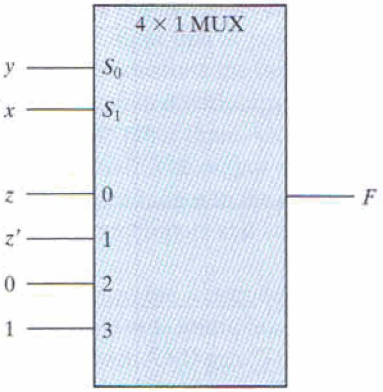
Function table

FIGURE 4.26
Quadruple two-to-one-line multiplexer

This function of three variables can be implemented with a four-to-one-line multiplexer as shown in Fig. 4.27. The two variables x and y are applied to the selection lines in that order; x is connected to the S_1 input and y to the S_0 input. The values for the data input lines are determined from the truth table of the function. When $xy = 00$, output F is equal to z because $F = 0$ when $z = 0$ and $F = 1$ when $z = 1$. This requires that variable z be applied to data input 0. The operation of the multiplexer is such that when $xy = 00$, data input 0 has a path to the output, and that makes F equal to z. In a similar fashion, we can determine the required input to data lines 1, 2, and 3 from the value of F when $xy = 01$, 10, and 11, respectively. This particular example shows all four possibilities that can be obtained for the data inputs.

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

FIGURE 4.27
Implementing a Boolean function with a multiplexer

The general procedure for implementing any Boolean function of n variables with a multiplexer with $n - 1$ selection inputs and 2^{n-1} data inputs follows from the previous example. To begin with, Boolean function is listed in a truth table. Then first $n - 1$ variables in the table are applied to the selection inputs of the multiplexer. For each combination of the selection variables, we evaluate the output as a function of the last variable. This function can be 0, 1, the variable, or the complement of the variable. These values are then applied to the data inputs in the proper order.

As a second example, consider the implementation of the Boolean function

$$F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$$

This function is implemented with a multiplexer with three selection inputs as shown in - Fig. 4.28. Note that the first variable A must be connected to selection input S_2 so that A, B, and C correspond to selection inputs S_2, S_1, and S_0, respectively. The values for the data inputs are determined from the truth table listed in the figure. The corresponding data line number is determined from the binary combination of ABC. For example, the table shows that when $ABC = 101$, $F = D$, so the input variable D is applied to data input 5. The binary constants 0 and 1 correspond to two fixed signal values. When integrated circuits are used, logic 0 corresponds to signal ground and logic 1 is equivalent to the power signal, depending on the technology (e.g., 5 volts).

Three-State Gates

A multiplexer can be constructed with three-state gates—digital circuits that exhibit three states. Two of the states are signals equivalent to logic 1 and logic 0 as in a conventional gate. The third state is a *high-impedance* state in which (1) the logic behaves like an open circuit, which means that the output appears to be disconnected, (2) the circuit has no logic significance, and

A	B	C	D	F	
0	0	0	0	0	$F = D$
0	0	0	1	1	
0	0	1	0	0	$F = D$
0	0	1	1	1	
0	1	0	0	1	$F = D'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = D$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	

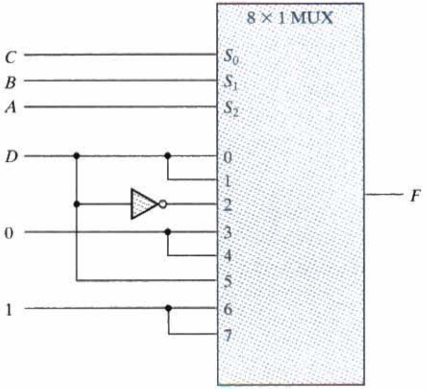


FIGURE 4.28
Implementing a four-input function with a multiplexer

(3) the circuit connected to the output of the three-state gate is not affected by the inputs to the gate. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used is the buffer gate.

The graphic symbol for a three-state buffer gate is shown in Fig. 4.29. It is distinguished from a normal buffer by an input control line entering the bottom of the symbol. The buffer has a normal input, an output, and a control input that determines the state of the output. When the control input is equal to 1, the output is enabled and the gate behaves like a conventional buffer, with the output equal to the normal input. When the control 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.

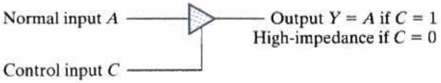
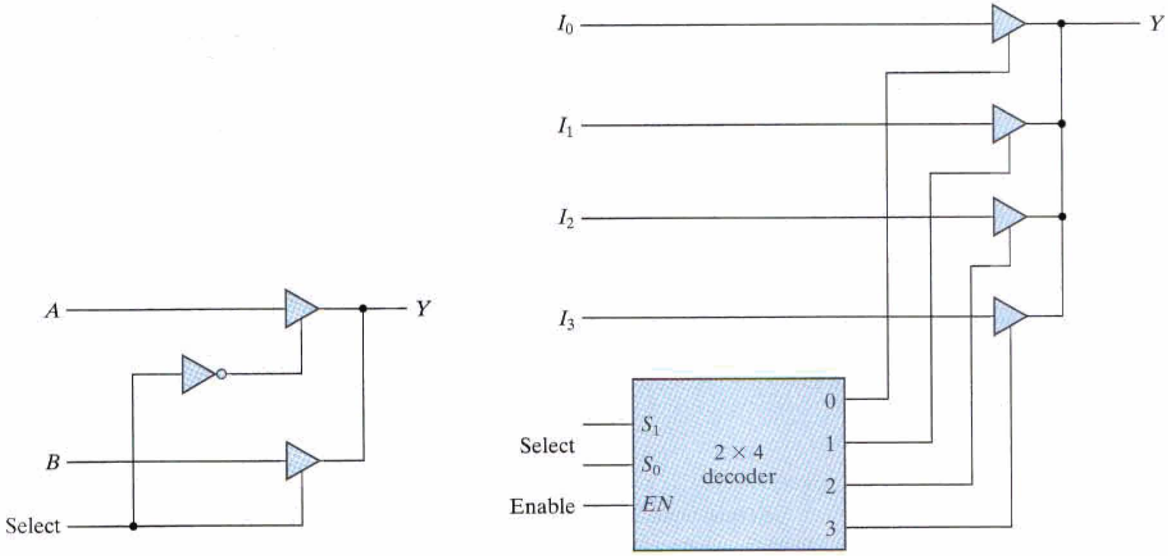


FIGURE 4.29
Graphic symbol for a three-state buffer

The construction of multiplexers with three-state buffers is demonstrated in Fig. 4.30. Part (a) of the figure 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 I_3 will be connected to the output line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only 1 three-state buffer has access to the output while all other buffers are maintained in a high-impedance state. One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0 and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation reveals that this circuit is another way of constructing a four-to-one-line multiplexer.



(a) 2-to-1-line mux

(b) 4-to-1-line mux

FIGURE 4.30
Multiplexers with three-state gates

4.12 HDL MODELS OF COMBINATIONAL CIRCUITS

The Verilog hardware description language (HDL) was introduced in Section 3.10. In the current section, we present more elaborate examples and compare alternative descriptions of combinational circuits in Verilog. Sequential circuits are presented in the next chapter. As mentioned previously, the module is the basic building block for modeling hardware with the Verilog HDL. The logic of a module can be described in any one (or a combination) of the following modeling styles:

- Gate-level modeling using instantiations of predefined and user-defined primitive gates.
- Dataflow modeling using continuous assignment statements with the keyword **assign**.
- Behavioral modeling using procedural assignment statements with the keyword **always**.

Gate-level (structural) modeling describes a circuit by specifying its gates and how they are connected with each other. Dataflow modeling is used mostly for describing the Boolean equations of combinational logic. We'll also consider here behavioral modeling that is used to describe combinational and sequential circuits at a higher level of abstraction. There is one other modeling style, called switch-level modeling. It is sometimes used in the simulation of MOS transistor circuit models, but not in logic synthesis. We consider switch-level modeling briefly in Section 10.10.

Gate-Level Modeling

Gate-level modeling was introduced in Section 3.10 with a simple example. In this type of representation, a circuit is specified by its logic gates and their interconnections. Gate-level modeling provides a textual description of a schematic diagram. The Verilog HDL 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

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

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 *instantiate* the gate. Think of 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 (leftmost) 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

```
// 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 two half adders. The third module describes the four-bit adder by instantiating 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*.

HDL Example 4.2

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder (Fig. 4.5b)

// module half_adder (S, C, x, y);                    // Verilog 1995 syntax
// output S, C;
// input   x, y;

module half_adder (output S, C, input x, y);          // Verilog 2001, 2005 syntax
 // Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
endmodule

// Description of full adder (Fig. 4.8)               // Verilog 1995 syntax
// module full_adder (S, C, x, y, z);
// output        S, C;
// input         x, y, z;

module full_adder (output S, C, input x, y, z);       // Verilog 2001, 2005 syntax
  wire S1, C1, C2;

// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder (Fig. 4.9)           // Verilog 1995 syntax
// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
// output [3: 0] Sum;
// output        C4;
// input   [3: 0] A, B;
// input         C0;
// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder ( output [3: 0] Sum, output C4,
  input [3: 0]  A, B, input C0);
  wire          C1, C2, C3;              // Intermediate carries
// Instantiate chain of full adders
  full_adder    FA0 (Sum[0], C1, A[0], B[0], C0),
                FA1 (Sum[1], C2, A[1], B[1], C1),
                FA2 (Sum[2], C3, A[2], B[2], C2),
                FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

HDL Example 4.2 illustrates Verilog 2001, 2005 syntax, which eliminates extra typing of identifiers declaring the mode (e.g., **output**), type (**reg**), and declaration of a vector range (e.g., [3: 0]). The first version of the standard (1995) uses separate statements for these declarations.

Note that modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration. In other words, a module definition cannot be inserted into the text between the **module** and **endmodule** keywords of another module. The only way one module definition can be incorporated into another module is by instantiating it. Instantiating modules within other modules creates a hierarchical decomposition of a design. A description of a module is said to be a *structural* description if it is composed of instantiations of other modules. Note also that *instance names* must be specified when defined modules are instantiated (such as *FA0* for the first full adder in the third module), but using a name is optional when instantiating primitive gates. Module *ripple_carry_4_bit_adder* is composed of instantiated and interconnected full adders, each of which is itself composed of half adders and some *glue logic*. The top level, or parent module, of the design hierarchy is the module *ripple_carry_4_bit_adder*. Four copies of *full_adder* are its child modules, etc. *C0* is an input of the cell forming the least significant bit of the chain, and *C4* is the output of the cell forming the most significant bit.

Three-State Gates

As mentioned in Section 4.11, a three-state gate has a control input that can place the gate into a high-impedance state. The high-impedance state is symbolized by **z** in Verilog. There are four types of three-state gates, as shown in Fig. 4.31. The **bufif1** gate behaves like a normal buffer if *control* = 1. The output goes to a high-impedance state **z** when *control* = 0. The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when *control* = 1. The two **not** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state. The gates are instantiated with the statement

gate name (output, input, control);

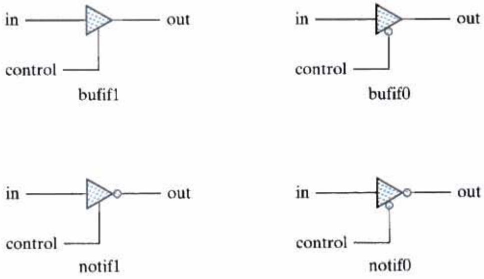


FIGURE 4.31
Three-state gates

The gate name can be that of any 1 of the 4 three-state gates. In simulation, the output can result in 0, 1, **x**, or **z**. Two examples of gate instantiation are

```
bufif1 (OUT, A, control);
notif0 (Y, B, enable);
```

In the first example, input A is transferred to OUT when $control = 1$. OUT goes to **z** when $control = 0$. In the second example, output $Y =$ **z** when $enable = 1$ and output $Y = B'$ when $enable = 0$.

The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers. As an example, consider the two-to-one-line multiplexer with three-state gates shown in Fig. 4.32.

The HDL description must use a **tri** data type for the output:

```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
 output m_out;
 input  A, B, select;
 tri    m_out;

 bufif1 (m_out, A, select);
 bufif0 (m_out, B, select);
endmodule
```

The 2 three-state buffers have the same output. In order to show that they have a common connection, it is necessary to declare *m_out* with the keyword **tri.**

Keywords **wire** and **tri** are examples of a set of data types called *nets*, which represent connections between hardware elements. In simulation, their value is determined by a continuous assignment statement or by the device whose output they represent. The word *net* is not a keyword, but represents a class of data types, such as **wire**, **wor**, **wand**, **tri**, **supply1**, and **supply0**. The **wire** declaration is used most frequently. In fact, if an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire**. The net **wor** models the hardware implementation of the wired-OR configuration (emitter-coupled logic). The **wand** models the wired-AND configuration (open-collector technology; see Fig. 3.28). The nets **supply1** and **supply0** represent power supply and ground, respectively. They are used to hardwire an input of a device to either 1 or 0.

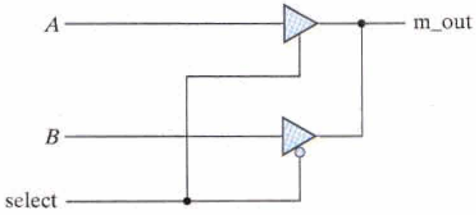


FIGURE 4.32
Two-to-one-line multiplexer with three-state buffers

Dataflow Modeling

Dataflow modeling of combinational logic uses a number of operators that act on operands to produce desired results. Verilog HDL provides about 30 different operators. Table 4.10 lists some of these operators, their symbols, and the operation that they perform. (A complete list of operators supported by Verilog 2001, 2005 can be found in Table 8.1 in Section 8.2.) It is necessary to distinguish between arithmetic and logic operations, so different symbols are used for each. The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation (conjunction) uses the symbol &. There are special symbols for bitwise logical OR (disjunction), NOT, and XOR. The equality symbol uses two equals signs (without spaces between them) to distinguish it from the equals sign used with the **assign** statement. The bitwise operators operate bit by bit on a pair of vector operands. The concatenation operator provides a mechanism for appending multiple operands. For example, two operands with two bits each can be concatenated to form an operand with four bits. The conditional operator acts like a multiplexer and is explained later, in conjunction with HDL Example 4.6.

Dataflow modeling uses continuous assignments and the keyword **assign**. A continuous assignment is a statement that assigns a value to a net. The data type family *net* is used in Verilog HDL to represent a physical connection between circuit elements. A net is declared explicitly by a net keyword (e.g., **wire**) or by declaring an identifier to be an output port. The logic value associated with a net is determined by what the net is connected to. If the net is connected to an output of a gate, the net is said to be *driven* by the gate, and the logic value of the net is determined by the logic values of the inputs to the gate and the truth table of the gate. If the identifier of a net is the left-hand side of a continuous assignment statement or a procedural assignment statement, the value assigned to the net is specified by an expression that uses operands and operators. As an example, assuming that the variables were declared, a two-to-one-line multiplexer with data inputs *A* and *B*, select input *S*, and output *Y* is described with the continuous assignment

assign Y = (A & S)|(B & ~S);

Table 4.10
Some Verilog HDL Operators

Symbol	Operation
+	binary addition
−	binary subtraction
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise NOT
==	equality
>	greater than
<	less than
{ }	concatenation
? :	conditional

The relationship between Y, A, B, and S is declared by the keyword **assign**, followed by the target output Y and an equals sign. Following the equals sign is a Boolean expression. In hardware terms, this assignment would be equivalent to connecting the output of the OR (|) gate to wire Y.

The next two examples show the dataflow models of the two previous gate-level examples. The dataflow description of a two-to-four-line decoder is shown in HDL Example 4.3. The circuit is defined with four continuous assignment statements using Boolean expressions, one for each output. The dataflow description of the four-bit adder is shown in HDL Example 4.4. The addition logic is described by a single statement using the operators of addition and concatenation. The plus symbol (+) specifies the binary addition of the four bits of A with the four bits of B and the one bit of *C_in*. The target output is the *concatenation* of the output carry *C_out* and the four bits of *Sum*. Concatenation of operands is expressed within braces and a comma separating the operands. Thus, *{C_out, Sum}* represents the five-bit result of the addition operation.

HDL Example 4.3

```
// Dataflow description of two-to-four-line decoder

// See Fig. 4.19. Note: The figure uses symbol E, but the
// Verilog model uses enable to clearly indicate functionality.

module decoder_2x4_df (                    // Verilog 2001, 2005 syntax
  output      [0: 3]     D,
  input                  A, B,
                         enable
);

  assign     D[0] = ~(~A & ~B & ~enable),
             D[1] = ~(~A & B & ~enable),
             D[2] = ~(A & ~B & ~enable),
             D[3] = ~(A & B & ~enable);
endmodule
```

HDL Example 4.4

```
// Dataflow description of four-bit adder

// Verilog 2001, 2005 module port syntax

module binary_adder (
  output [3: 0]          Sum,
  output                 C_out,
  input   [3: 0]         A, B,
  input                  C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Dataflow HDL models describe combinational circuits by their *function* rather than by their gate structure. To show how dataflow descriptions facilitate digital design, consider the 4-bit magnitude comparator described in HDL Example 4.5. The module specifies two 4-bit inputs *A* and *B* and three outputs. One output (*A_lt_B*) is logic 1 if *A* is less than *B*, a second output (*A_gt_B*) is logic 1 if *A* is greater than *B*, and a third output (*A_eq_B*) is logic 1 if *A* is equal to *B*. Note that equality (identity) is symbolized with two equals signs (==) to distinguish the operation from that of the assignment operator (=). A Verilog HDL synthesis compiler can accept this module description as input, execute synthesis algorithms, and provide an output netlist and a schematic of a circuit equivalent to the one in Fig. 4.17, all without manual intervention!

HDL Example 4.5

```
// Dataflow description of a four-bit comparator      //V2001, 2005 syntax

module mag_compare
( output              A_lt_B, A_eq_B, A_gt_B,
  input [3: 0]        A, B
);
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```

The next example uses the conditional operator (? :). This operator takes three operands:

condition ? true-expression : false-expression;

The condition is evaluated. If the result is logic 1, the true expression is evaluated. If the result is logic 0, the false expression is evaluated. The two conditions together are equivalent to an if-else condition. HDL Example 4.6 describes a two-to-one-line multiplexer using the conditional operator. The continuous assignment

```
assign OUT = select ? A : B;
```

specifies the condition that $OUT = A$ if $select = 1$, else $OUT = B$ if $select = 0$.

HDL Example 4.6

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output       m_out;
  input        A, B;
  input        select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits. Here, we give two simple combinational circuit examples to introduce the subject. Behavioral modeling is presented in more detail in Section 5.6, after the study of sequential circuits.

Behavioral descriptions use the keyword **always**, followed by an optional event control expression and a list of procedural assignment statements. The event control expression specifies when the statements will execute. The target output of procedural assignment statements must be of the **reg** data type. Contrary to the **wire** data type, whereby the target output of an assignment may be continuously updated, a **reg** data type retains its value until a new value is assigned.

HDL Example 4.7 shows the behavioral description of a two-to-one-line multiplexer. (Compare it with HDL Example 4.6.) Since variable *m_out* is a target output, it must be declared as **reg** data (in addition to the **output** declaration). The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the @ symbol. (Note that there is no semicolon (;) at the end of the **always** statement.) In this case, these variables are the input variables *A*, *B*, and *select*. The statements execute if *A*, *B*, or *select* changes value. Note that the keyword **or**, instead of the bitwise logical OR operator "|", is used between variables. The conditional statement **if-else** provides a decision based upon the value of the *select* input. The **if** statement can be written without the equality symbol:

```
if (select) OUT = A;
```

The statement implies that *select* is checked for logic 1.

HDL Example 4.7

```
// Behavioral description of two-to-one-line multiplexer

module mux_2x1_beh (m_out, A, B, select);
 output      m_out;
 input       A, B, select;
 reg         m_out;

 always      @(A or B or select)
  if (select == 1) m_out = A;
  else m_out = B;
 endmodule
```

HDL Example 4.8 describes the function of a four-to-one-line multiplexer. The *select* input is defined as a two-bit vector, and output *y* is declared to have type **reg**. The **always** statement, in this example, has a sequential block enclosed between the keywords **case** and **endcase**. The block is executed whenever any of the inputs listed after the @ symbol changes in value. The **case** statement is a multiway conditional branch construct. Whenever *in_0*, *in_1*, *in_2*, *in_3* or *select* change, the case expression (*select*) is evaluated and its value compared, from top to bottom, with the values in the list of statements that follow, the so-called **case** items. The statement associated with the first **case** item that matches the **case** expression is executed. In the absence of a match, no statement is executed. Since *select* is a two-bit number, it can be equal to 00, 01, 10, or 11. The **case** items have an implied priority because the list is evaluated from top to bottom.

The list is called a *sensitivity list* (Verilog 2001, 2005) and is equivalent to the *event control expression* (Verilog 1995) formed by "ORing" the signals.

HDL Example 4.8

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg  m_out,
 input        in_0, in_1, in_2, in_3,
 input [1: 0] select
);
 always @ (in_0, in_1, in_2, in_3, select)       // Verilog 2001, 2005 syntax
  case (select)
    2'b00:          m_out = in_0;
    2'b01:          m_out = in_1;
    2'b10:          m_out = in_2;
    2'b11:          m_out = in_3;
   endcase
 endmodule
```

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. 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.

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 executes repeatedly in a loop. The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #. For example, consider the **initial** block

```
initial
 begin
      A = 0; B = 0;
  #10 A = 1;
  #20 A = 0; B = 1;
 end
```

The block is enclosed between the keywords **begin** and **end**. At time 0, A and B are set to 0. Ten time units later, A is changed to 1. twenty time units after that (at $t = 30$) A is changed to 0 and B to 1. Inputs specified by a three-bit truth table can be generated with the **initial** block:

```
initial
 begin
  D = 3'b000;
  repeat (7)
   #10 D = D + 3'b001;
 end
```

When the simulator runs, the three-bit vector D is initialized to 000 at time $= 0$. The keyword **repeat** specifies a looping statement: D is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

A stimulus module has the following form:

```
module test_module_name;
 // Declare local reg and wire identifiers.
 // Instantiate the design module under test.
 // Specify a stopwatch, using $finish to terminate the simulation.
```

```
  // Generate stimulus, using initial and always statements.
  // Display the output response (text or graphics (or both)).
endmodule
```

A test module is written like any other module, but it typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type. The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type. The module under test is then instantiated, using the local identifiers in its port list. Figure 4.33 clarifies this relationship. The stimulus module generates inputs for the design module by declaring local identifiers *t_A* and *t_B* as **reg** type and checks the output of the design unit with the **wire** identifier *t_C*. The local identifiers are then used to instantiate the design module being tested. The simulator associates the (actual) local identifiers within the test bench, *t_A*, *t_B*, and *t_C*, with the formal identifiers of the module (*A*, *B*, *C*). The association shown here is based on position in the port list, which is adequate for the examples that we will consider. The reader should note, however, that Verilog provides a more flexible name association mechanism for connecting ports in larger circuits.

The response to the stimulus generated by the **initial** and **always** blocks will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability. Numerical outputs are displayed by using Verilog *system tasks*. These are built-in system functions that are recognized by keywords that begin with the symbol **$**. Some of the system tasks that are useful for display are

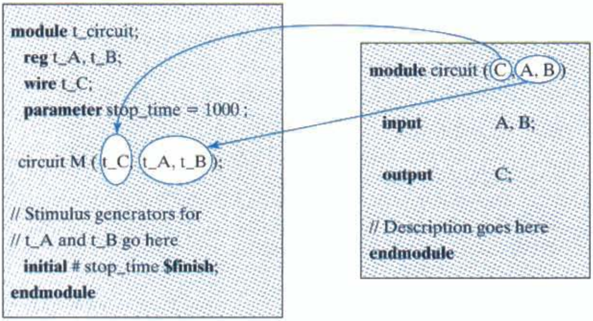
$display—display a one-time value of variables or strings with an end-of-line return,

$write**—same as **$display, but without going to next line,

$monitor—display variables whenever a value changes during a simulation run,

$time—display the simulation time,

$finish—terminate the simulation.



```
module t_circuit;
  reg t_A, t_B;
  wire t_C;
  parameter stop_time = 1000;

 circuit M (t_C, t_A, t_B);

// Stimulus generators for
// t_A and t_B go here
  initial # stop_time $finish;
endmodule
```

```
module circuit (C, A, B)

  input        A, B;

  output       C;

// Description goes here
endmodule
```

FIGURE 4.33
Interaction between stimulus and design modules

The syntax for **$display**, **$write**, and **$monitor** is of the form

Task-name (format specification, argument list);

The format specification uses the symbol % to specify the radix of the numbers that are displayed and may have a string enclosed in quotes ("). The base may be binary, decimal, hexadecimal, or octal, identified with the symbols %b, %d, %h, and %o, respectively (%B, %D, %H, and %O are valid too). For example, the statement

```
$display ("%d %b %b", C, A, B);
```

specifies the display of *C* in decimal and of *A* and *B* in binary. Note that there are no commas in the format specification, that the format specification and argument list are separated by a comma, and that the argument list has commas between the variables. An example that specifies a string enclosed in quotes may look like the statement

```
$display ("time = %0d A = %b B = %b", $time, A, B);
```

and will produce the display

```
time = 3  A = 10  B = 1
```

where (*time* =), (*A* =), and (*B* =) are part of the string to be displayed. The format specifiers %0d, %b, and %b specify the base for **$time**, *A*, and *B*, respectively. In displaying time values, it is better to use the format %0d instead of %d. This provides a display of the significant digits without the leading spaces that %d will include. (%d will display about 10 leading spaces because time is calculated as a 32-bit number.)

An example of a stimulus module is shown in HDL Example 4.9. The circuit to be tested is the two-to-one-line multiplexer described in Example 4.6. The module *t_mux_2x1_df* has no ports. The inputs for the mux are declared with a **reg** keyword and the outputs with a **wire** keyword. The mux is instantiated with the local variables. The **initial** block specifies a sequence of binary values to be applied during the simulation. The output response is checked with the **$monitor** system task. Every time a variable in its argument changes value, the simulator displays the inputs, output, and time. The result of the simulation is listed under the simulation log in the example. It shows that $m\_out = A$ when $select = 1$ and $m\_out = B$ when $select = 0$, verifying the operation of the multiplexer.

HDL Example 4.9

```
// Test bench with stimulus for mux_2x1_df

module t_mux_2x1_df;
 wire          t_mux_out;
 reg           t_A, t_B;
 reg           t_select;
 parameter  stop_time = 50;

 mux_2x1_df  M1 (t_mux_out, t_A, t_B, t_select);     // Instantiation of circuit to be tested
```

```
  initial # stop_time $finish;

  initial begin                                        // Stimulus generator
         t_select = 1; t_A = 0; t_B = 1;
   #10   t_A = 1; t_B = 0;
   #10   t_select = 0;
   #10   t_A = 0; t_B = 1;
  end

  initial begin                                        // Response monitor
   // $display ("    time   Select  A    B   m_out");
   // $monitor ($time,, "    %b    %b    %b    %b", t_select, t_A, t_B, t_m_out);
   $monitor ("time=", $time,, "select = %b A = %b B = %b OUT = %b",
   t_select, t_A, t_B, t_mux_out);
  end
endmodule

// Dataflow description of two-to-one-line multiplexer

// from Example 4.6
module mux_2x1_df (m_out, A, B, select);
  output        m_out;
  input         A, B;
  input         select;

  assign m_out = (select)? A : B;
endmodule
```

```
Simulation log:
select = 1 A = 0 B = 1 OUT = 0 time = 0
select = 1 A = 1 B = 0 OUT = 1 time = 10
select = 0 A = 1 B = 0 OUT = 0 time = 20
select = 0 A = 0 B = 1 OUT = 1 time = 30
```

Logic simulation is a fast, accurate method of analyzing combinational circuits to verify that they operate properly. There are two types of verification: functional and timing. In *functional* verification, we study the circuit logical operation independently of timing considerations. This can be done by deriving the truth table of the combinational circuit. In *timing* verification, we study the circuit's operation by including the effect of delays through the gates. This can be done by observing the waveforms at the outputs of the gates when they respond to a given input. An example of a circuit with gate delays was presented in Section 3.10 in HDL Example 3.3. We next show an HDL example that produces the truth table of a combinational circuit. A **$monitor** system task displays the output caused by the given stimulus. A commented alternative statement having a **$display** task would create a header that could be used with a **$monitor** statement to eliminate the repetition of names on each line of output.

The analysis of combinational circuits was covered in Section 4.3. A multilevel circuit of a full adder was analyzed, and its truth table was derived by inspection. The gate-level description of this circuit is shown in HDL Example 4.10. The circuit has three inputs, two outputs, and nine gates. The description of the circuit follows the interconnections between the gates according to the schematic diagram of Fig. 4.2. The stimulus for the circuit is listed in the second module. The inputs for simulating the circuit are specified with a three-bit **reg** vector *D*. *D[2]* is equivalent to input *A*, *D[1]* to input *B*, and *D[0]* to input *C*. The outputs of the circuit F_1 and F_2 are declared as **wire**. The complement of *F2* is named *F2_b* to illustrate a common industry practice for designating the complement of a signal (instead of appending *_not*). This procedure follows the steps outlined in Fig. 4.33. The **repeat** loop provides the seven binary numbers after 000 for the truth table. The result of the simulation generates the output truth table displayed with the example. The truth table listed shows that the circuit is a full adder.

HDL Example 4.10

```
// Gate-level description of circuit of Fig. 4.2
module Circuit_of_Fig_4_2 (A, B, C, F1, F2);
 input   A, B, C;
 output  F1, F2;
 wire    T1, T2, T3, F2_b, E1, E2, E3;
 or   g1 (T1, A, B, C);
 and g2 (T2, A, B, C);
 and g3 (E1, A, B);
 and g4 (E2, A, C);
 and g5 (E3, B, C);
 or   g6 (F2, E1, E2, E3);
 not  g7 (F2_b, F2);
 and g8 (T3, T1, F2_b);
 or   g9 (F1, T2, T3);
endmodule

// Stimulus to analyze the circuit

module test_circuit;
 reg [2: 0] D;
 wire F1, F2;
 Circuit_of_Fig_4_2  M_F4_32 (D[2], D[1], D[0], F1, F2);
 initial
  begin
   D = 3'b000;
   repeat (7) #10 D = D + 1'b1;
  end
 initial
 $monitor ("ABC = %b F1 = %b F2 =%b ", D, F1, F2);

endmodule
```

```
Simulation log:  ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0 ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1 ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1 ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```

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.

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

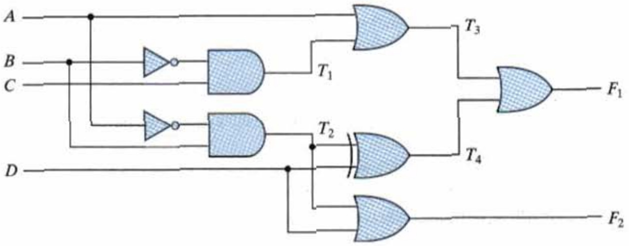


FIGURE P4.1

(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 16 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 Boolean output 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 outputs F and G in terms of the input variables in the circuit of Fig. P4.2.

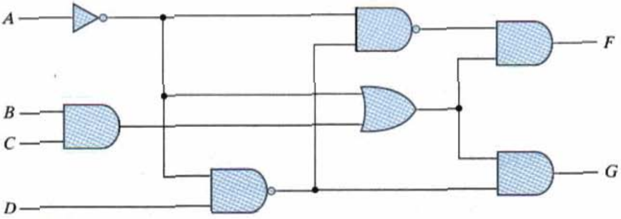


FIGURE P4.2

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 listed 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 odd number.

4.5 Design a combinational circuit with three inputs, x, y, and z, and three outputs, A, B, and C. When the binary input is 0, 1, 2, or 3, the binary output is two greater than the input. When the binary input is 4, 5, 6, or 7, the binary output is three less than the input.

4.6 A majority circuit is a combinational circuit whose output is equal to 1 if the input variables have more 1's than 0's. The output is 0 otherwise.

(a)* Design a three-input majority circuit by finding the circuits truth table, Boolean equation, and a logic diagram.

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

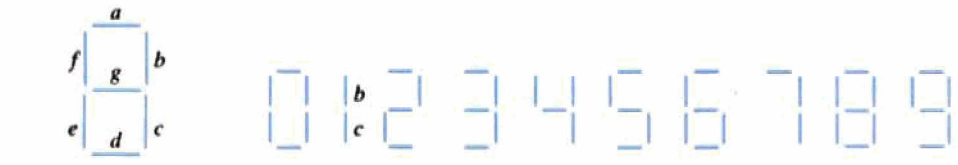
4.7 Design a combinational circuit that converts a four-bit Gray code (Table 1-6) to a four-bit binary number.

(a)* Implement the circuit with exclusive-OR gates.

(b) Using a case statement, write and verify a Verilog model of the circuit.

4.8* Design a code converter that converts a decimal digit from the 8, 4, −2, −1 code to BCD (see Table 1.5). (HDL — see Problem 4.50.)

4.9 An ABCD-to-seven-segment decoder is a combinational circuit that converts a decimal digit in BCD to an appropriate code for the selection of segments in an indicator used to display the decimal digit in a familiar form. The seven outputs of the decoder *(a, b, c, d, e, f, g)* select the corresponding segments in the display, as shown in Fig. P4.9(a). The numeric display chosen to represent the decimal digit is shown in Fig. P4.9(b). Using a truth table and Karnaugh maps, design the BCD-to-seven-segment decoder, using a minimum number of gates. The six invalid combinations should result in a blank display. (HDL—see Problem 4.51.)



(a) Segment designation

(b) Numerical designation for display

FIGURE P4.9

4.10* Design a four-bit combinational circuit 2's complementer. (The output generates the 2's complement of the input binary number.) Show that the circuit can be constructed with exclusive-OR gates. Can you predict what the output functions are for a five-bit 2's complementer?

4.11 Using four half-adders (HDL — see Problem 4.52).

(a) Design a four-bit combinational circuit incrementer (a circuit that adds 1 to a four-bit binary number).

(b) Design a four-bit combinational circuit decrementer (a circuit that subtracts 1 from a four-bit binary number).

4.12 (a) Design a half-subtractor circuit with inputs x and y and outputs *Diff* and B_{out}. The circuit subtracts the bits $x - y$ and places the difference in *Diff* and the borrow in B_{out}.

(b)* Design a full-subtractor circuit with three inputs, x, y, B_{in}, and two outputs *Diff* and B_{out}. The circuit subtracts $x - y - B_{in}$, where B_{in} is the input borrow, B_{out} is the output borrow, and *Diff* is the difference.

4.13* The adder-subtractor circuit of Fig. 4.13 has the following values for mode input M and data inputs A and B:

	M	A	B
(a)	0	0111	0110
(b)	0	1000	1001
(c)	1	1100	1000
(d)	1	0101	1010
(e)	1	0000	0001

In each case, determine the values of the four SUM outputs, the carry C, and overflow V. (HDL—see Problems 4.37 and 4.40.)

4.14* Assume that the exclusive-OR gate has a propagation delay of 10 ns and that the AND or OR gates have a propagation delay of 5 ns. What is the total propagation delay time in the four-bit adder of Fig. 4.12?

4.15 Derive the two-level Boolean expression for the output carry C_4 shown in the lookahead carry generator of Fig. 4.12.

4.16 Define the carry propagate and carry generate as

$$P_i = A_i + B_i$$
$$G_i = A_i B_i$$

respectively. Show that the output carry and output sum of a full adder becomes

$$C_{i+1} = (C_i' G_i' + P_i')'$$
$$S_i = (P_i G_i') \oplus C_i$$

The logic diagram of the first stage of a four-bit parallel adder as implemented in IC type 74283 is shown in Fig. P4.16. Identify the P_i' and G_i' terminals and show that the circuit implements a full adder.

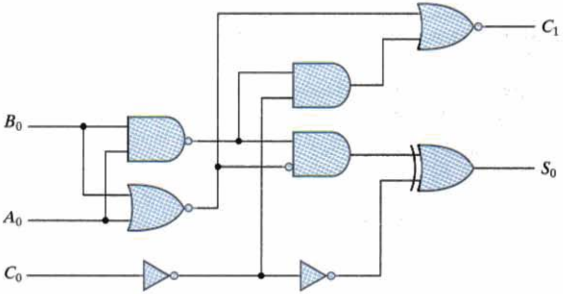


FIGURE P4.16
First stage of a parallel adder

4.17 Show that the output carry in a full-adder circuit can be expressed in the AND-OR-INVERT form

$$C_{i+1} = G_i + P_iC_i = (G_i'P_i' + G_i'C_i')'$$

IC type 74182 is a lookahead carry generator circuit that generates the carries with AND-OR-INVERT (AOI) gates (see Section 3-8.) The circuit assumes that the input terminals have the complements of the G's, the P's, and of C_1. Derive the Boolean functions for the lookahead carries C_2, C_3, and C_4 in this IC. (*Hint:* Use the equation-substitution method to derive the carries in terms of C_i')

4.18* Design a combinational circuit that generates the 9's complement of a BCD digit. (HDL — see Problem 4.54.)

4.19 Construct a BCD adder-subtractor circuit. Use the BCD adder of Fig. 4.14 and the 9's complementer of Problem 4.18. Use block diagrams for the components. (HDL — see Problem 4.55.)

4.20 A binary multiplier multiplies two unsigned four-bit numbers.

(a) Using AND gates and binary adders (see Fig. 4.16), design the circuit.

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

4.21 Design a combinational circuit that compares two four-bit numbers to check if they are equal. The circuit output is equal to 1 if the two numbers are equal and 0 otherwise.

4.22* Design an excess-3-to-binary decoder using the unused combinations of the code as don't-care conditions. (HDL — see Problem 4.42.)

4.23 Draw the logic diagram of a two-to-four-line decoder using (a) NOR gates only, and (b) NAND gates only. Include an enable input.

4.24 Design a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions. (HDL — see Problem 4.60.)

4.25 Construct a 5-to-32-line decoder with four 3-to-8-line decoders with enable and a 2-to-4-line decoder. Use block diagrams for the components.

4-26 Construct a 4-to-16-line decoder with five 2-to-4-line decoders with enable.

4.27 A combinational circuit is specified by the following three Boolean functions:

$$F_1(A, B, C) = \Sigma(3, 5, 6)$$
$$F_2(A, B, C) = \Sigma(1, 4)$$
$$F_3(A, B, C) = \Sigma(2, 3, 5, 6, 7)$$

Implement the circuit with a decoder constructed with NAND gates (similar to Fig. 4.19) and NAND or AND gates connected to the decoder outputs. Use a block diagram for the decoder. Minimize the number of inputs in the external gates.

4.28 Using a decoder and external gates, design the combinational circuit defined by the following three Boolean functions:

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

(b) $F_1 = (y' + x)z$
$F_2 = y'z' + xy' + yz'$
$F_3 = (x' + y)z$

4.29* Design a four-input priority encoder with inputs as in Table 4.8, but with input D_0 having the highest priority and input D_3 the lowest priority. (HDL—see Problem 4.57.)

4.30 Specify the truth table of an octal-to-binary priority encoder. Provide an output V to indicate that at least one of the inputs is present. The input with the highest subscript number has the highest priority. What will be the value of the four outputs if inputs D_2 and D_6 are 1 at the same time?

4.31 Construct a 16 × 1 multiplexer with two 8 × 1 and one 2 × 1 multiplexers. Use block diagrams.

4.32 Implement the following Boolean function with a multiplexer (HDL—see Problem 4.46):

(a) $F(A, B, C, D) = \Sigma(0, 2, 5, 7, 11, 14)$
(b) $F(A, B, C, D) = \Pi(3, 8, 12)$

4.33 Implement a full adder with two 4 × 1 multiplexers.

4.34 An 8 × 1 multiplexer has inputs A, B, and C connected to the selection inputs S_2, S_1, and S_0, respectively. The data inputs I_0 through I_7 are as follows:

(a)* $I_1 = I_2 = I_7 = 0$; $I_3 = I_5 = 1$; $I_0 = I_4 = D$; and $I_6 = D'$.
(b) $I_1 = I_2 = 0$; $I_3 = I_7 = 1$; $I_4 = I_5 = D$; and $I_0 = I_6 = D'$.

Determine the Boolean function that the multiplexer implements.

4.35 Implement the following Boolean function with a 4 × 1 multiplexer and external gates.

(a)* $F(A, B, C, D) = \Sigma(1, 3, 4, 11, 12, 13, 14, 15)$
(b) $F(A, B, C, D) = \Sigma(1, 2, 4, 7, 8, 9, 10, 11, 13, 15)$

Connect inputs A and B to the selection lines. The input requirements for the four data lines will be a function of variables C and D. These values are obtained by expressing F as a function of C and D for each of the four cases when $AB = 00, 01, 10$, and 11. The functions may have to be implemented with external gates and with connections to power and ground.

4.36 Write the HDL gate-level description of the priority encoder circuit shown in Fig. 4.23. (HDL—see Problem 4.45.)

4.37 Write the HDL gate-level hierarchical description of a four-bit adder-subtractor for unsigned binary numbers. The circuit is similar to Fig. 4.13 but without output V. You can instantiate the four-bit full adder described in HDL Example 4.2. (See Problems 4.13 and 4.40.)

4.38 Write the HDL dataflow description of a quadruple two-to-one-line multiplexer with enable. (See Fig. 4.26.)

4.39* Write an HDL behavioral description of a four-bit comparator with a six-bit output $Y[5:0]$. Bit 5 of Y is for "equals," bit 4 is for "not equal to," bit 3 is for "greater than," bit 2 is for "less than," bit 1 for "greater than or equal to," and bit 0 for "less than or equal to."

4.40 Using the conditional operator (? :), write an HDL dataflow description of a four-bit adder subtractor of unsigned numbers. (See Problems 4.13 and 4.37.)

4.41 Repeat Problem 4.40, using a cyclic behavior.

4.42 (a) Write an HDL gate-level description of the BCD-to-excess-3 converter circuit shown in Fig. 4.4 (see Problem 4.22).
(b) Write a dataflow description of the BCD-to-excess-3 converter, 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.

4.43 Explain the function of the circuit specified by the following HDL description:

```
module Prob4_43 (A, B, S, E, Q);
 input  [1: 0] A, B;
 input         S, E;
 output [1: 0] Q;
 assign Q = E ? (S ? A : B) : 'bz;
endmodule
```

4.44 Using a case statement, write an HDL behavioral description of a eight-bit arithmetic-logic unit (ALU). The circuit has a three-bit select bus (*Sel*), eight-bit input datapaths (*A[7: 0]* and *B[7: 0]*, an eight-bit output datapath (*y[7: 0]*), and performs the arithmetic and logical operations listed below

Sel	Operation	Description
000	y = 8'b0	
001	y = A & B	Bitwise and
010	y = A | B	Bitwise or
011	y = A ^ B	Bitwise exclusive or
100	y = A + B	Add (Assume A and B are unsigned)
101	y = A − B	Subtract
110	y = ~A	Bitwise complement
111	y = 8'hFF	

4.45 Write an HDL behavioral description of a four-input priority encoder. Use a four-bit vector for the *D* inputs and an **always** block with if-else statements. Assume that input *D[3]* has the highest priority (see Problem 4.36).

4.46 Repeat Problem 4.32, using a dataflow description.

4.47 Repeat Problem 4.37, using a dataflow description.

4.48 Develop and modify the eight-bit ALU specified in Problem 4.44 so that it has three-state output controlled by an enable input, *En*. Write a test bench and simulate the circuit.

4.49 For the circuit shown in Fig. P4.1,

(a) write and verify and verify a gate-level HDL model of the circuit

(b) compare your results with those obtained in Problem 4.1.

4.50* Using a case statement, develop and simulate a behavioral model of the 84-2-1 to BCD code converter described in Problem 4.8.

4.51 Develop and simulate a behavioral model of the ABCD-to-seven-segment decoder described in Problem 4.9.

4.52 Using a continuous assignment, develop and simulate a dataflow model of

(a) the four-bit incrementer described in Problem 4.11(a)

(b) the four-bit decrementer described in Problem 4.11(b).

4.53 Develop and simulate a structural model of the decimal adder shown in Fig. 4.14.

4.54 Develop and simulate a behavioral model of a circuit that generates the 9's complement of a BCD digit (see Problem 4.18).

4.55 Construct a hierarchical model of the BCD adder–subtractor described in Problem 4.19. The BCD adder and the 9's complementer are to be described as behavioral models in separate modules, and they are to be instantiated in a top-level module.

4.56* Write a continuous assignment statement that compares two four-bit numbers to check if their bit patterns match. The variable to which the assignment is made is to equal 1 if the numbers match and 0 otherwise.

4.57* Develop and verify a behavioral model of the four-bit priority encoder described in Problem 4.29.

4.58 Write a Verilog model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the right and filling the vacated positions with the bit that was in the MSB before the shift occurred (shift arithmetic right).

4.59 Write a Verilog model of a circuit whose 32-bit output is formed by shifting its 32-bit input three positions to the left and filling the vacated positions with 0s (shift logical left).

4.60 Write a Verilog model of a BCD-to-decimal decoder using the unused combinations of the BCD code as don't-care conditions (see Problem 4.24).

4.61 Using the port syntax of the IEEE 1364-2001 standard, write and verify a gate-level model of the 4-bit even parity checker shown in Fig. 3.36.

4.62 Using continuous assignment statements and the port syntax of the IEEE 1364-2001 standard, write and verify an HDL model of the 4-bit even parity checker shown in Fig. 3.36.

REFERENCES

1. Bhasker, J. 1997. *A Verilog HDL Primer.* Allentown, PA: Star Galaxy Press.
2. Bhasker, J. 1998. *Verilog HDL Synthesis.* Allentown, PA: Star Galaxy Press.
3. Ciletti, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL.* Upper Saddle River, NJ: Prentice Hall.
4. Dietmeyer, D. L. 1988. *Logic Design of Digital Systems*, 3d ed. Boston: Allyn Bacon.
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. 2000. *Logic and Computer Design Fundamentals*, 2d 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. 1992. *Fundamentals of Logic Design*, 4th ed. St. Paul, MN: West.
12. Thomas, D. E., and P. R. Moorby. 1998. *The Verilog Hardware Description Language*, 4th ed. Boston: Kluwer Academic Publishers.
13. Wakerly, J. F. 2000. *Digital Design: Principles and Practices*, 3d ed. Upper Saddle River, NJ: Prentice Hall.

Chapter 5

Synchronous Sequential Logic

5.1 INTRODUCTION

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

5.2 SEQUENTIAL CIRCUITS

A block diagram of a sequential circuit is shown in Fig. 5.1. It consists of a combinational circuit to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the *state* of the sequential circuit at that time. The sequential circuit receives binary information from external inputs that, together with the present state of the

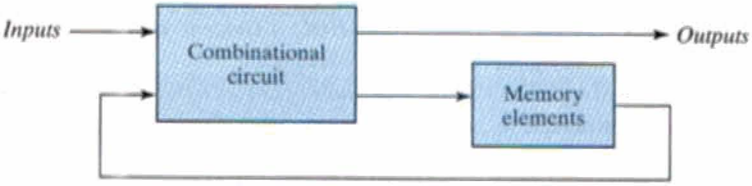


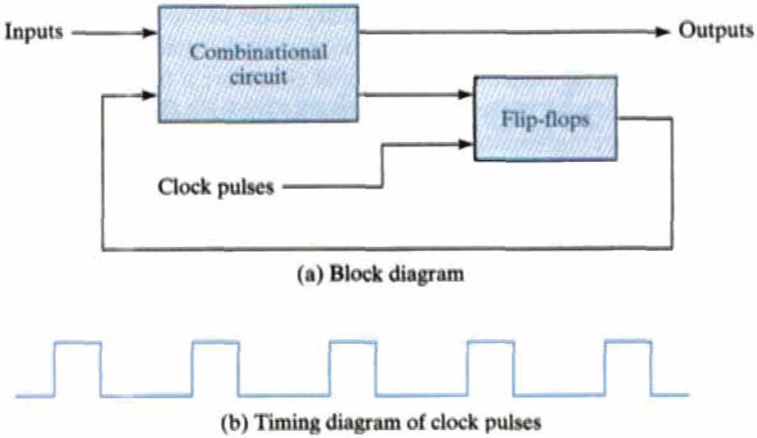
FIGURE 5.1
Block diagram of sequential circuit

storage elements, determine the binary value of the outputs. These external inputs also determine the condition for changing the state in the storage elements. The block diagram demonstrates that the outputs in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements. The next state of the storage elements is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a time sequence of inputs, outputs, and internal states. In contrast, the outputs of combinational logic depend only on the present values of the inputs.

There are two main types of sequential circuits, and their classification is a function of the timing of their signals. A *synchronous* sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous* sequential circuit depends upon the input signals at any instant of time *and* the order in which the inputs change. The storage elements commonly used in asynchronous sequential circuits are time-delay devices. The storage capability of a time-delay device varies with the time it takes for the signal to propagate through the device. In practice, the internal propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary. In gate-type asynchronous systems, the storage elements consist of logic gates whose propagation delay provides the required storage. Thus, an asynchronous sequential circuit may be regarded as a combinational circuit with feedback. Because of the feedback among logic gates, an asynchronous sequential circuit may become unstable at times. The instability problem imposes many difficulties on the designer. Asynchronous sequential circuits are presented in Chapter 9.

A synchronous sequential circuit employs signals that affect the storage elements at only discrete instants of time. Synchronization is achieved by a timing device called a *clock generator*, which provides a clock signal having the form of a periodic train of *clock pulses*. The clock signal is commonly denoted by the identifiers *clock* and *clk*. The clock pulses are distributed throughout the system in such a way that storage elements are affected only with the arrival of each pulse. In practice, the clock pulses determine *when* computational activity will occur within the circuit, and other signals (external inputs and otherwise) determine *what* changes will take place affecting the storage elements and the outputs. For example, a circuit that is to add and store two binary numbers would compute their sum from the values of the numbers and store the sum at the occurrence of a clock pulse. Synchronous sequential circuits that use clock pulses to control storage elements are called *clocked sequential circuits* and are the type most frequently encountered in practice. They are called *synchronous circuits* because the activity within the circuit and the resulting updating of stored values is synchronized to the occurrence of clock pulses. The design of synchronous circuits is feasible because they seldom manifest instability problems and their timing is easily broken down into independent discrete steps, each of which can be considered separately.

The storage elements (memory) used in clocked sequential circuits are called *flip-flops*. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The block diagram of a synchronous clocked sequential circuit is shown in Fig. 5.2. The *outputs* are formed by a combinational logic function of the inputs to the circuit or the values stored in the flip-flops (or both). The value that is stored in a flip-flop when the clock pulse occurs is also determined by the inputs to the circuit or the values presently



(a) Block diagram

(b) Timing diagram of clock pulses

FIGURE 5.2
Synchronous clocked sequential circuit

stored in the flip-flop (or both). The new value is stored (i.e., the flip-flop is updated) when a pulse of the clock signal occurs. Prior to the occurrence of the clock pulse, the combinational logic forming the next value of the flop-flop must have reached a stable value. Consequently, the speed at which the combinational logic circuits operate is critical. If the clock (synchronizing) pulses arrive at a regular interval, as shown in the timing diagram in Fig. 5.2, the combinational logic must respond to a change in the state of the flip-flop in time to be updated before the next pulse arrives. Propagation delays play an important role in determining the minimum interval between clock pulses that will allow the circuit to operate correctly. The state of the flip-flops can change only during a clock pulse transition—for example, when the value of the clock signals changes from 0 to 1. When a clock pulse is not active, the feedback loop between the value stored in the flip-flop and the value formed at the input to the flip-flop is effectively broken because the flip-flop outputs cannot change even if the outputs of the combinational circuit driving their inputs change in value. Thus, the transition from one state to the next occurs only at predetermined intervals dictated by the clock pulses.

5.3 STORAGE ELEMENTS: LATCHES

A storage element in a digital circuit can maintain a binary state indefinitely (as long as power is delivered to the circuit), until directed by an input signal to switch states. The major differences among various types of storage elements are in the number of inputs they possess and in the manner in which the inputs affect the binary state. *Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches*; *those controlled by a clock transition are flip-flops*. Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices. The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed. Although latches are useful for storing binary information and for the design of asynchronous sequential circuits (see Section 9.3), they are not practical for use in synchronous sequential circuits. Because they are the building blocks of flip-flops, however, we will consider the fundamental storage mechanism used in latches before considering flip-flops in the next section.

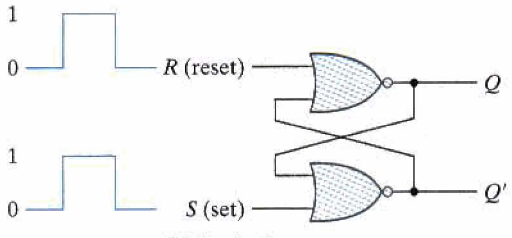
SR Latch

The *SR* latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates, and two inputs labeled *S* for set and *R* for reset. The *SR* latch constructed with two cross-coupled NOR gates is shown in Fig. 5.3. The latch has two useful states. When output $Q = 1$ and $Q' = 0$, the latch is said to be in the *set state*. When $Q = 0$ and $Q' = 1$, it is in the *reset state*. Outputs Q and Q' are normally the complement of each other. However, when both inputs are equal to 1 at the same time, a condition in which both outputs are equal to 0 (rather than be mutually complementary) occurs. If both inputs are then switched to 0 simultaneously, the device will enter an unpredictable or undefined state or a metastable state. Consequently, in practical applications, setting both inputs to 1 is forbidden.

Under normal conditions, both inputs of the latch remain at 0 unless the state has to be changed. The application of a momentary 1 to the S input causes the latch to go to the set state. The S input must go back to 0 before any other changes take place, in order to avoid the occurrence of an undefined next state that results from the forbidden input condition. As shown in the function table of Fig. 5.3(b), two input conditions cause the circuit to be in the set state. The first condition ($S = 1, R = 0$) is the action that must be taken by input S to bring the circuit to the set state. Removing the active input from S leaves the circuit in the same state. After both inputs return to 0, it is then possible to shift to the reset state by momentary applying a 1 to the R input. The 1 can then be removed from R, whereupon the circuit remains in the reset state. Thus, when both inputs S and R are equal to 0, the latch can be in either the set or the reset state, depending on which input was most recently a 1.

If a 1 is applied to both the S and R inputs of the latch, both outputs go to 0. This action produces an undefined next state, because the state that results from the input transitions depends on the order in which they return to 0. It also violates the requirement that outputs be the complement of each other. In normal operation, this condition is avoided by making sure that 1's are not applied to both inputs simultaneously.

The *SR* latch with two cross-coupled NAND gates is shown in Fig. 5.4. It operates with both inputs normally at 1, unless the state of the latch has to be changed. The application of 0

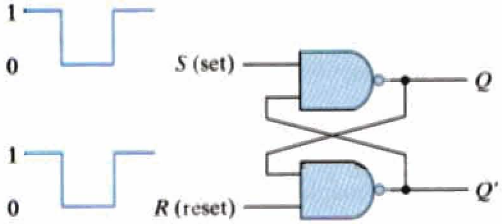


(a) Logic diagram

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

(b) Function table

FIGURE 5.3
SR latch with NOR gates



(a) Logic diagram

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

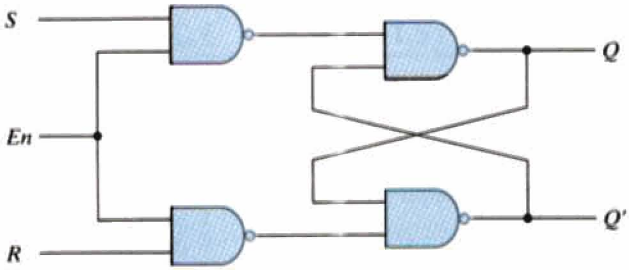
(b) Function table

FIGURE 5.4
SR latch with NAND gates

to the *S* input causes output *Q* to go to 1, putting the latch in the set state. When the *S* input goes back to 1, the circuit remains in the set state. After both inputs go back to 1, we are allowed to change the state of the latch by placing a 0 in the *R* input. This action causes the circuit to go to the reset state and stay there even after both inputs return to 1. The condition that is forbidden for the NAND latch is both inputs being equal to 0 at the same time, an input combination that should be avoided.

In comparing the NAND with the NOR latch, note that the input signals for the NAND require the complement of those values used for the NOR latch. Because the NAND latch requires a 0 signal to change its state, it is sometimes referred to as an $S'R'$ latch. The primes (or, sometimes, bars over the letters) designate the fact that the inputs must be in their complement form to activate the circuit.

The operation of the basic *SR* latch can be modified by providing an additional input signal that determines (controls) *when* the state of the latch can be changed. An *SR* latch with a control input is shown in Fig. 5.5. It consists of the basic *SR* latch and two additional NAND gates. The control input *En* acts as an *enable* signal for the other two inputs. The outputs of the NAND gates stay at the logic-1 level as long as the enable signal remains at 0. This is the quiescent condition for the *SR* latch. When the enable input goes to 1, information from the *S* or *R* input is allowed to affect the latch. The set state is reached with $S = 1$, $R = 0$, and $En = 1$ (active-high enabled). To change to the reset state, the inputs must be $S = 0$, $R = 1$, and

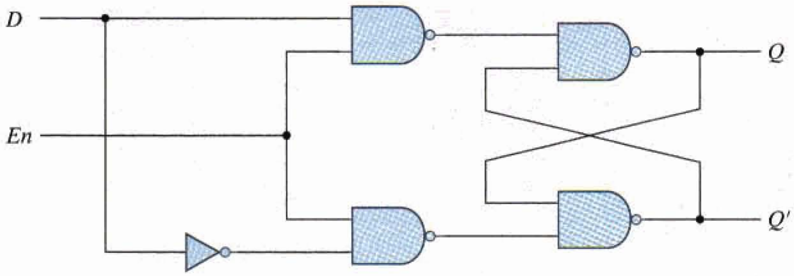


(a) Logic diagram

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

(b) Function table

FIGURE 5.5
SR latch with control input



(a) Logic diagram

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

(b) Function table

FIGURE 5.6
***D* latch**

$En = 1$. In either case, when En returns to 0, the circuit remains in its current state. The control input disables the circuit by applying 0 to En, so that the state of the output does not change regardless of the values of S and R. Moreover, when $En = 1$ and both the S and R inputs are equal to 0, the state of the circuit does not change. These conditions are listed in the function table accompanying the diagram.

An indeterminate condition occurs when all three inputs are equal to 1. This condition places 0's on both inputs of the basic SR latch, which puts it in the undefined state. When the enable input goes back to 0, one cannot conclusively determine the next state, because it depends on whether the S or R input goes to 0 first. This indeterminate condition makes this circuit difficult to manage, and it is seldom used in practice. Nevertheless, it is an important circuit because other useful latches and flip-flops are constructed from it.

D Latch (Transparent Latch)

One way to eliminate the undesirable condition of the indeterminate state in the SR latch is to ensure that inputs S and R are never equal to 1 at the same time. This is done in the D latch, shown in Fig. 5.6. This latch has only two inputs: D (data) and En (enable). The D input goes directly to the S input, and its complement is applied to the R input. As long as the enable input is at 0, the cross-coupled SR latch has both inputs at the 1 level and the circuit cannot change state regardless of the value of D. The D input is sampled when $En = 1$. If $D = 1$, the Q output goes to 1, placing the circuit in the set state. If $D = 0$, output Q goes to 0, placing the circuit in the reset state.

The D latch receives that designation from its ability to hold *data* in its internal storage. It is suited for use as a temporary storage for binary information between a unit and its environment. The binary information present at the data input of the D latch is transferred to the Q output when the enable input is asserted. The output follows changes in the data input as long as the enable input is asserted. This situation provides a path from input D to the output, and for this reason, the circuit is often called a *transparent* latch. When the enable input signal is deasserted, the binary information that was present at the data input at the time the transition occurred is retained (i.e., stored) at the Q output until the enable input is asserted again. Note that

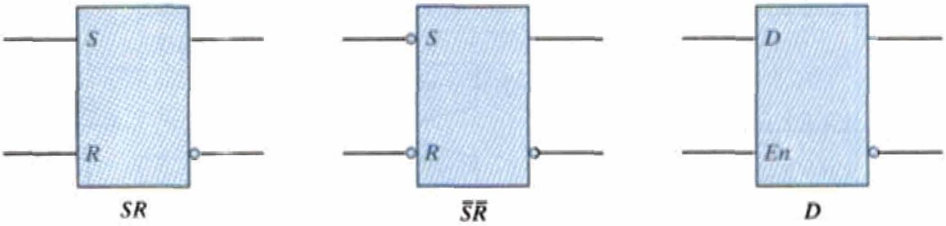


FIGURE 5.7
Graphic symbols for latches

an inverter could be placed at the enable input. Then, depending on the physical circuit, the external enabling signal will be a value of 0 (active low) or 1 (active high).

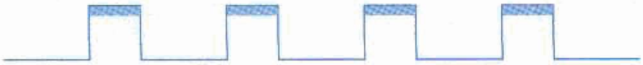
The graphic symbols for the various latches are shown in Fig. 5.7. A latch is designated by a rectangular block with inputs on the left and outputs on the right. One output designates the normal output, and the other (with the bubble designation) designates the complement output. The graphic symbol for the *SR* latch has inputs *S* and *R* indicated inside the block. In the case of a NAND gate latch, bubbles are added to the inputs to indicate that setting and resetting occur with a logic-0 signal. The graphic symbol for the *D* latch has inputs *D* and *En* indicated inside the block.

5.4 STORAGE ELEMENTS: FLIP-FLOPS

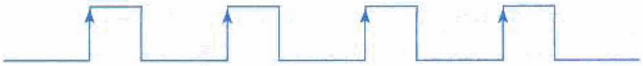
The state of a latch or flip-flop is switched by a change in the control input. This momentary change is called a *trigger*, and the transition it causes is said to trigger the flip-flop. The *D* latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level. As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

As seen from the block diagram of Fig. 5.2, a sequential circuit has a feedback path from the outputs of the flip-flops to the input of the combinational circuit. Consequently, the inputs of the flip-flops are derived in part from the outputs of the same and other flip-flops. When latches are used for the storage elements, a serious difficulty arises. The state transitions of the latches start as soon as the clock pulse changes to the logic-1 level. The new state of a latch appears at the output while the pulse is still active. This output is connected to the inputs of the latches through the combinational circuit. If the inputs applied to the latches change while the clock pulse is still at the logic-1 level, the latches will respond to new values and a new output state may occur. The result is an unpredictable situation, since the state of the latches may keep changing for as long as the clock pulse stays at the active level. Because of this unreliable operation, the output of a latch cannot be applied directly or through combinational logic to the input of the same or another latch when all the latches are triggered by a common clock source.

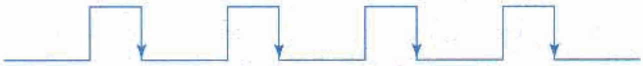
Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock. The problem with the latch is that it responds to a change in the *level* of a clock pulse. As shown in Fig. 5.8(a), a positive level response in the enable input allows changes in the output when the *D* input changes while the



(a) Response to positive level



(b) Positive-edge response



(c) Negative-edge response

FIGURE 5.8
Clock response in latch and flip-flop

clock pulse stays at logic 1. The key to the proper operation of a flip-flop is to trigger it only during a signal *transition*. This can be accomplished by eliminating the feedback path that is inherent in the operation of the sequential circuit using latches. A clock pulse goes through two transitions: from 0 to 1 and the return from 1 to 0. As shown in Fig. 5.8, the positive transition is defined as the positive edge and the negative transition as the negative edge. There are two ways that a latch can be modified to form a flip-flop. One way is to employ two latches in a special configuration that isolates the output of the flip-flop and prevents it from being affected while the input to the flip-flop is changing. Another way is to produce a flip-flop that triggers only during a signal transition (from 0 to 1 or from 1 to 0) of the synchronizing signal (clock) and is disabled during the rest of the clock pulse. We will now proceed to show the implementation of both types of flip-flops.

Edge-Triggered *D* Flip-Flop

The construction of a *D* flip-flop with two *D* latches and an inverter is shown in Fig. 5.9. The first latch is called the master and the second the slave. The circuit samples the *D* input and changes its output *Q* only at the negative edge of the synchronizing or controlling clock (designated as

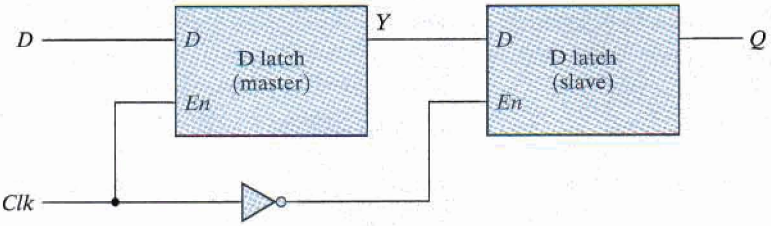


FIGURE 5.9
Master–slave *D* flip-flop

Clk). When the clock is 0, the output of the inverter is 1. The slave latch is enabled, and its output Q is equal to the master output Y. The master latch is disabled because $Clk = 0$. When the input pulse changes to the logic-1 level, the data from the external D input are transferred to the master. The slave, however, is disabled as long as the clock remains at the 1 level, because its *enable* input is equal to 0. Any change in the input changes the master output at Y, but cannot affect the slave output. When the clock pulse returns to 0, the master is disabled and is isolated from the D input. At the same time, the slave is enabled and the value of Y is transferred to the output of the flip-flop at Q. Thus, a change in the output of the flip-flop can be triggered only by and during the transition of the clock from 1 to 0.

The behavior of the master–slave flip-flop just described dictates that (1) the output may change only once, (2) a change in the output is triggered by the negative edge of the clock, and (3) the change may occur only during the clock's negative level. The value that is produced at the output of the flip-flop is the value that was stored in the master stage immediately before the negative edge occurred. It is also possible to design the circuit so that the flip-flop output changes on the positive edge of the clock. This happens in a flip-flop that has an additional inverter between the *Clk* terminal and the junction between the other inverter and input *En* of the master latch. Such a flip-flop is triggered with a negative pulse, so that the negative edge of the clock affects the master and the positive edge affects the slave and the output terminal.

Another construction of an edge-triggered D flip-flop uses three SR latches as shown in Fig. 5.10. Two latches respond to the external D (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop. The S and R inputs of the output latch are maintained at the logic-1 level when $Clk = 0$. This causes the output to remain in its present state. Input D

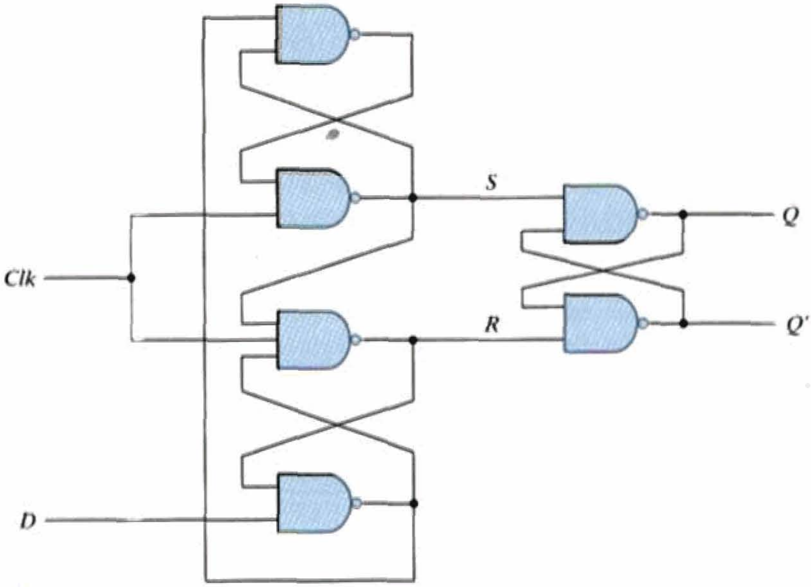
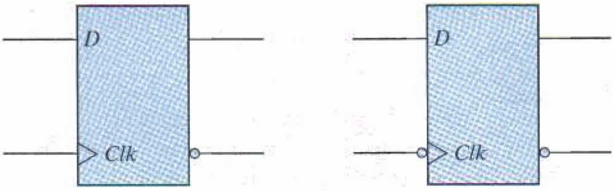


FIGURE 5.10
D-type positive-edge-triggered flip-flop



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

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

may be equal to 0 or 1. If $D = 0$ when Clk becomes 1, R changes to 0. This causes the flip-flop to go to the reset state, making $Q = 0$. If there is a change in the D input while $Clk = 1$, terminal R remains at 0 because Q is 0. Thus, the flip-flop is locked out and is unresponsive to further changes in the input. When the clock returns to 0, R goes to 1, placing the output latch in the quiescent condition without changing the output. Similarly, if $D = 1$ when Clk goes from 0 to 1, S changes to 0. This causes the circuit to go to the set state, making $Q = 1$. Any change in D while $Clk = 1$ does not affect the output.

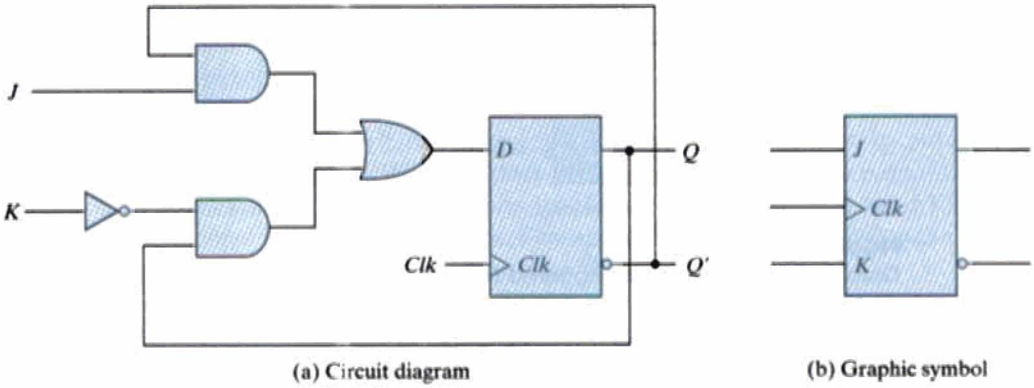
In sum, when the input clock in the positive-edge-triggered flip-flop makes a positive transition, the value of D is transferred to Q. A negative transition of the clock (i.e., from 1 to 0) does not affect the output, nor is the output affected by changes in D when Clk is in the steady logic-1 level or the logic-0 level. Hence, this type of flip-flop responds to the transition from 0 to 1 and nothing else.

The timing of the response of a flip-flop to input data and to the clock must be taken into consideration when one is using edge-triggered flip-flops. There is a minimum time called the *setup time* during which the D input must be maintained at a constant value prior to the occurrence of the clock transition. Similarly, there is a minimum time called the *hold time* during which the D input must not change after the application of the positive transition of the clock. The propagation delay time of the flip-flop is defined as the interval between the trigger edge and the stabilization of the output to a new state. These and other parameters are specified in manufacturers' data books for specific logic families.

The graphic symbol for the edge-triggered D flip-flop is shown in Fig. 5.11. It is similar to the symbol used for the D latch, except for the arrowheadlike symbol in front of the letter Clk, designating a *dynamic* input. The *dynamic indicator* denotes the fact that the flip-flop responds to the edge transition of the clock. A bubble outside the block adjacent to the dynamic indicator designates a negative edge for triggering the circuit. The absence of a bubble designates a positive-edge response.

Other Flip-Flops

Very large-scale integration circuits contain thousands of gates within one package. Circuits are constructed by interconnecting the various gates to provide a digital system. Each flip-flop is constructed from an interconnection of gates. The most economical and efficient flip-flop constructed in this manner is the edge-triggered D flip-flop, because it requires the smallest number



(a) Circuit diagram

(b) Graphic symbol

FIGURE 5.12
JK flip-flop

of gates. Other types of flip-flops can be constructed by using the *D* flip-flop and external logic. Two flip-flops less widely used in the design of digital systems are the *JK* and *T* flip-flops.

There are three operations that can be performed with a flip-flop: Set it to 1, reset it to 0, or complement its output. With only a single input, the *D* flip-flop can set or reset the output, depending on the value of the *D* input immediately before the clock transition. Synchronized by a clock signal, the *JK* flip-flop has two inputs and performs all three operations. The circuit diagram of a *JK* flip-flop constructed with a *D* flip-flop and gates is shown in Fig. 5.12(a). The *J* input sets the flip-flop to 1, the *K* input resets it to 0, and when both inputs are enabled, the output is complemented. This can be verified by investigating the circuit applied to the *D* input:

$$D = JQ' + K'Q$$

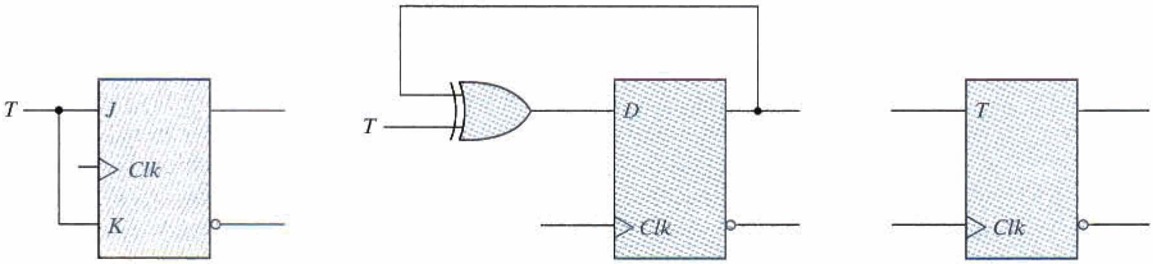
When $J = 1$ and $K = 0$, $D = Q' + Q = 1$, so the next clock edge sets the output to 1. When $J = 0$ and $K = 1$, $D = 0$, so the next clock edge resets the output to 0. When both $J = K = 1$ and $D = Q'$, the next clock edge complements the output. When both $J = K = 0$ and $D = Q$, the clock edge leaves the output unchanged. The graphic symbol for the *JK* flip-flop is shown in Fig. 5.12(b). It is similar to the graphic symbol of the *D* flip-flop, except that now the inputs are marked *J* and *K*.

The *T* (toggle) flip-flop is a complementing flip-flop and can be obtained from a *JK* flip-flop when inputs *J* and *K* are tied together. This is shown in Fig. 5.13(a). When $T = 0$ $(J = K = 0)$, a clock edge does not change the output. When $T = 1$ $(J = K = 1)$, a clock edge complements the output. The complementing flip-flop is useful for designing binary counters.

The *T* flip-flop can be constructed with a *D* flip-flop and an exclusive-OR gate as shown in Fig. 5.13(b). The expression for the *D* input is

$$D = T \oplus Q = TQ' + T'Q$$

When $T = 0$, $D = Q$ and there is no change in the output. When $T = 1$, $D = Q'$ and the output complements. The graphic symbol for this flip-flop has a *T* symbol in the input.



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

FIGURE 5.13
T flip-flop

Characteristic Tables

A characteristic table defines the logical properties of a flip-flop by describing its operation in tabular form. The characteristic tables of three types of flip-flops are presented in Table 5.1. They define the next state (i.e., the state that results from a clock transition) as a function of the inputs and the present state. $Q(t)$ refers to the present state (i.e., the state present prior to the application of a clock edge). $Q(t + 1)$ is the next state one clock period later. Note that the clock edge input is not included in the characteristic table, but is implied to occur between times t and $t + 1$. Thus, $Q(t)$ denotes the state of the flip-flop immediately before the clock edge, and $Q(t + 1)$ denotes the state that results from the clock transition.

The characteristic table for the *JK* flip-flop shows that the next state is equal to the present state when inputs J and K are both equal to 0. This condition can be expressed as $Q(t + 1) = Q(t)$, indicating that the clock produces no change of state. When $K = 1$ and

Table 5.1
Flip-Flop Characteristic Tables

***JK* Flip-Flop**

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

***D* Flip-Flop**

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

***T* Flip-Flop**

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

$J = 0$, the clock resets the flip-flop and $Q(t + 1) = 0$. With $J = 1$ and $K = 0$, the flip-flop sets and $Q(t + 1) = 1$. When both J and K are equal to 1, the next state changes to the complement of the present state, a transition that can be expressed as $Q(t + 1) = Q'(t)$.

The next state of a D flip-flop is dependent only on the D input and is independent of the present state. This can be expressed as $Q(t + 1) = D$. It means that the next-state value is equa to the value of D. Note that the D flip-flop does not have a "no-change" condition. Such a condition can be accomplished either by disabling the clock or by operating the clock by having the output of the flip-flop connected into the D input. Either method effectively circulates the output of the flip-flop when the state of the flip-flop must remain unchanged.

The characteristic table of the T flip-flop has only two conditions: When $T = 0$, the clock edge does not change the state; when $T = 1$, the clock edge complements the state of the flip-flop.

Characteristic Equations

The logical properties of a flip-flop, as described in the characteristic table, can be expressed algebraically with a characteristic equation. For the D flip-flop, we have the characteristic equation

$$Q(t + 1) = D$$

which states that the next state of the output will be equal to the value of input D in the present state. The characteristic equation for the JK flip-flop can be derived from the characteristic table or from the circuit of Fig. 5.12. We obtain

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

where Q is the value of the flip-flop output prior to the application of a clock edge. The characteristic equation for the T flip-flop is obtained from the circuit of Fig. 5.13:

$$Q(t + 1) = T \oplus Q = TQ' + T'Q$$

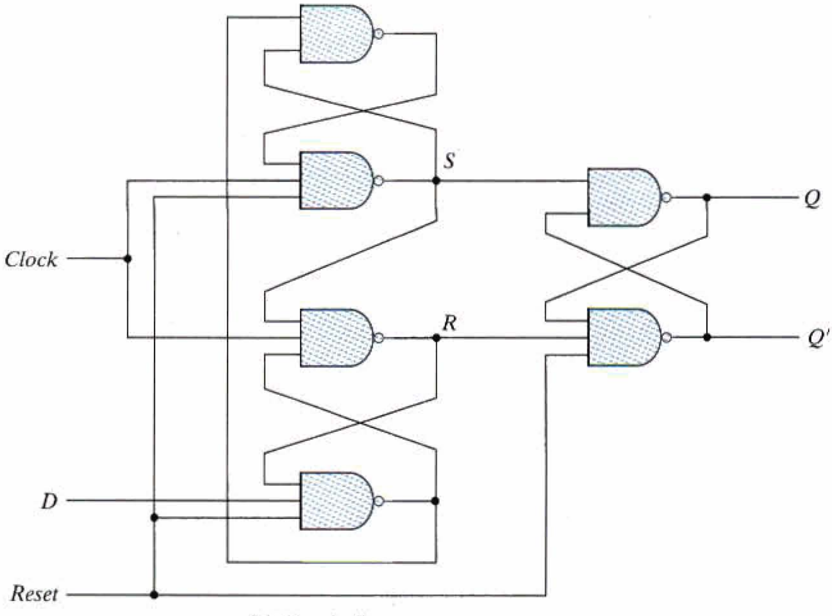
Direct Inputs

Some flip-flops have asynchronous inputs that are used to force the flip-flop to a particular state independently of the clock. The input that sets the flip-flop to 1 is called *preset* or *direct set*. The input that clears the flip-flop to 0 is called *clear* or *direct reset*. When power is turned on in a digital system, the state of the flip-flops is unknown. The direct inputs are useful for bringing all flip-flops in the system to a known starting state prior to the clocked operation.

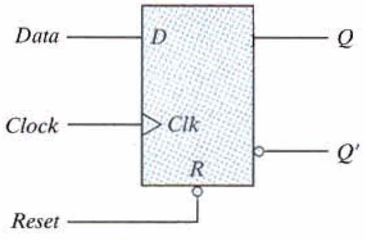
A positive-edge-triggered D flip-flop with active-low asynchronous reset is shown in Fig. 5.14. The circuit diagram is the same as the one in Fig. 5.10, except for the additional reset input connections to three NAND gates. When the reset input is 0, it forces output Q' to stay at 1, which, in turn, clears output Q to 0, thus resetting the flip-flop. Two other connections from the reset input ensure that the S input of the third SR latch stays at logic 1 while the reset input is at 0, regardless of the values of D and Clk.

The graphic symbol for the D flip-flop with a direct reset has an additional input marked with R. The bubble along the input indicates that the reset is active at the logic-0 level. Flip-flops with a direct set use the symbol S for the asynchronous set input.

The function table specifies the operation of the circuit. When $R = 0$, the output is reset to 0. This state is independent of the values of D or Clk. Normal clock operation can proceed only



(a) Circuit diagram



(b) Graphic symbol

R	*Clk*	*D*	*Q*	*Q'*
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

FIGURE 5.14
***D* flip-flop with asynchronous reset**

after the reset input goes to logic 1. The clock at *Clk* is shown with an upward arrow to indicate that the flip-flop triggers on the positive edge of the clock. The value in *D* is transferred to *Q* with every positive-edge clock signal, provided that $R = 1$.

5.5 ANALYSIS OF CLOCKED SEQUENTIAL CIRCUITS

Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is determined from the inputs, the outputs, and the state of its flip-flops. The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or a diagram for the time sequence of inputs, outputs, and internal states. It is also possible to write Boolean expressions that describe the behavior of the sequential circuit. These expressions must include the necessary time sequence, either directly or indirectly.

A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs. The flip-flops may be of any type, and the logic diagram may or may not include combinational circuit gates. In this section, we introduce an algebraic representation for specifying the next-state condition in terms of the present state and inputs. A state table and state diagram are then presented to describe the behavior of the sequential circuit. Another algebraic representation is introduced for specifying the logic diagram of sequential circuits. Examples are used to illustrate the various procedures.

State Equations

The behavior of a clocked sequential circuit can be described algebraically by means of state equations. A *state equation* (also called a *transition equation*) specifies the next state as a function of the present state and inputs. Consider the sequential circuit shown in Fig. 5.15. It consists

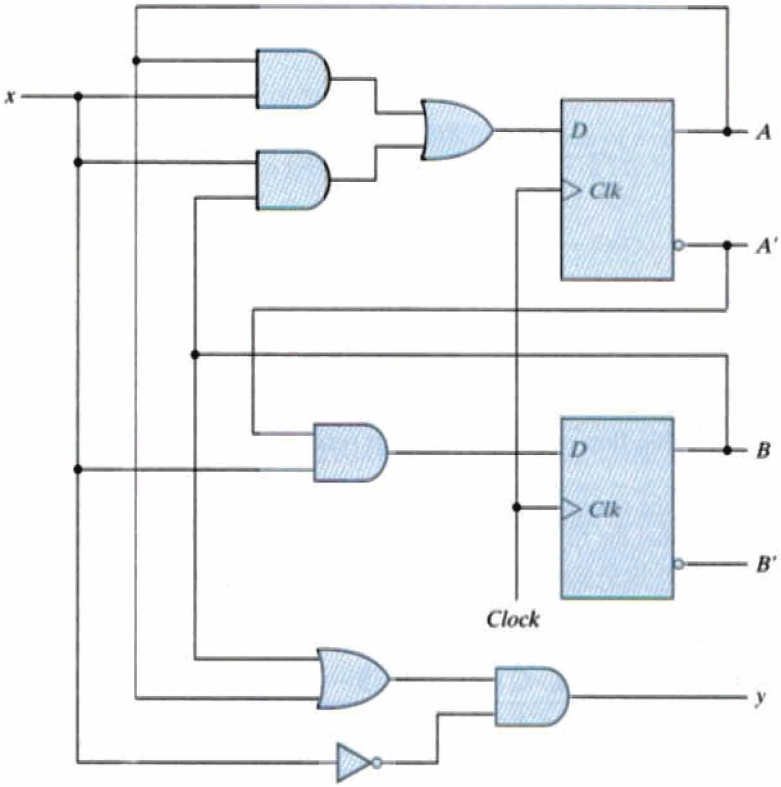


FIGURE 5.15
Example of sequential circuit

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.

Table 5.2
State Table for the Circuit of Fig. 5.15

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

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

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

The next-state section in the state table under column A has three 1's where the present state of A and input x are both equal to 1 or the present state of B and input x are both equal to 1. Similarly, the next state of flip-flop B is derived from the state equation

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

and is equal to 1 when the present state of A is 0 and input x is equal to 1. The output column is derived from the output equation

$$y = Ax' + Bx'$$

The state table of a sequential circuit with D-type flip-flops is obtained by the same procedure outlined in the previous example. In general, a sequential circuit with m flip-flops and n inputs needs 2^{m+n} rows in the state table. The binary numbers from 0 through $2^{m+n} - 1$ are listed under the present-state and input columns. The next-state section has m columns, one for each flip-flop. The binary values for the next state are derived directly from the state equations. The output section has as many columns as there are output variables. Its binary value is derived from the circuit or from the Boolean function in the same manner as in a truth table.

It is sometimes convenient to express the state table in a slightly different form having only three sections: present state, next state, and output. The input conditions are enumerated under the next-state and output sections. The state table of Table 5.2 is repeated in Table 5.3 in this second form. For each present state, there are two possible next states and outputs, depending on the value of the input. One form may be preferable to the other, depending on the application.

Table 5.3
Second Form of the State Table

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

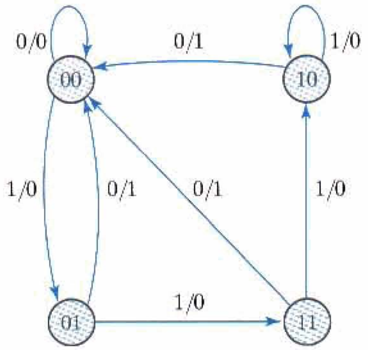


FIGURE 5.16
State diagram of the circuit of Fig. 5.15

State Diagram

The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle, and the (clock-triggered) transitions between states are indicated by directed lines connecting the circles. The state diagram of the sequential circuit of Fig. 5.15 is shown in Fig. 5.16. The state diagram provides the same information as the state table and is obtained directly from Table 5.2 or Table 5.3. The binary number inside each circle identifies the state of the flip-flops. The directed lines are labeled with two binary numbers separated by a slash. The input value during the present state is labeled first, and the number after the slash gives the output during the present state with the given input. (It is important to remember that the bit value listed for the output along the directed line occurs during the present state and with the indicated input, and has nothing to do with the transition to the next state.) For example, the directed line from state 00 to 01 is labeled 1/0, meaning that when the sequential circuit is in the present state 00 and the input is 1, the output is 0. After the next clock cycle, the circuit goes to the next state, 01. If the input changes to 0, then the output becomes 1, but if the input remains at 1, the output stays at 0. This information is obtained from the state diagram along the two directed lines emanating from the circle with state 01. A directed line connecting a circle with itself indicates that no change of state occurs.

There is no difference between a state table and a state diagram, except in the manner of representation. The state table is easier to derive from a given logic diagram and the state equation. The state diagram follows directly from the state table. The state diagram gives a pictorial view of state transitions and is the form more suitable for human interpretation of the circuit's operation. For example, the state diagram of Fig. 5.16 clearly shows that, starting from state 00, the output is 0 as long as the input stays at 1. The first 0 input after a string of 1's gives an output of 1 and transfers the circuit back to the initial state, 00. The machine represented by the state diagram acts to detect a zero in the bit stream of data.

Flip-Flop Input Equations

The logic diagram of a sequential circuit consists of flip-flops and gates. The interconnections among the gates form a combinational circuit and may be specified algebraically with Boolean expressions. The knowledge of the type of flip-flops and a list of the Boolean expressions of the combinational circuit provide the information needed to draw the logic diagram of the sequential circuit. The part of the combinational circuit that generates external outputs is described algebraically by a set of Boolean functions called *output equations*. The part of the circuit that generates the inputs to flip-flops is described algebraically by a set of Boolean functions called flip-flop *input equations* (or, sometimes, *excitation equations*). We will adopt the convention of using the flip-flop input symbol to denote the input equation variable and a subscript to designate the name of the flip-flop output. For example, the following input equation specifies an OR gate with inputs x and y connected to the D input of a flip-flop whose output is labeled with the symbol Q:

$$D_Q = x + y$$

The sequential circuit of Fig. 5.15 consists of two D flip-flops A and B, an input x, and an output y. The logic diagram of the circuit can be expressed algebraically with two flip-flop input equations and an output equation:

$$\begin{aligned} D_A &= Ax + Bx \\ D_B &= A'x \\ y &= (A + B)x' \end{aligned}$$

The three equations provide the necessary information for drawing the logic diagram of the sequential circuit. The symbol D_A specifies a D flip-flop labeled A. D_B specifies a second D flip-flop labeled B. The Boolean expressions associated with these two variables and the expression for output y specify the combinational circuit part of the sequential circuit.

The flip-flop input equations constitute a convenient algebraic form for specifying the logic diagram of a sequential circuit. They imply the type of flip-flop from the letter symbol, and they fully specify the combinational circuit that drives the flip-flops. Note that the expression for the input equation for a D flip-flop is identical to the expression for the corresponding state equation. This is because of the characteristic equation that equates the next state to the value of the D input: $Q(t + 1) = D_Q$.

Analysis with *D* Flip-Flops

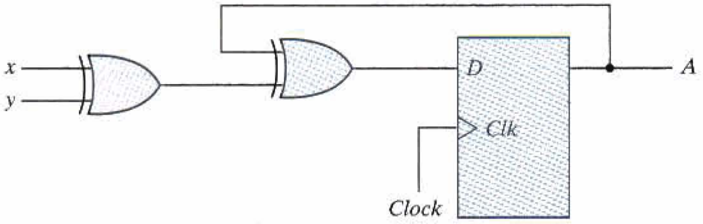
We will summarize the procedure for analyzing a clocked sequential circuit with D flip-flops by means of a simple example. The circuit we want to analyze is described by the input equation

$$D_A = A \oplus x \oplus y$$

The D_A symbol implies a D flip-flop with output A. The x and y variables are the inputs to the circuit. No output equations are given, which implies that the output comes from the output of the flip-flop. The logic diagram is obtained from the input equation and is drawn in Fig. 5.17(a).

The state table has one column for the present state of flip-flop A, two columns for the two inputs, and one column for the next state of A. The binary numbers under Axy are listed from 000 through 111 as shown in Fig. 5.17(b). The next-state values are obtained from the state equation

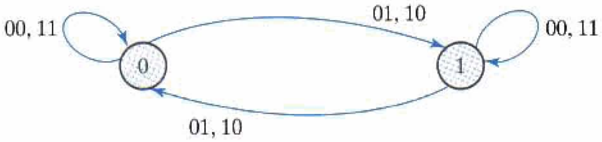
$$A(t + 1) = A \oplus x \oplus y$$



(a) Circuit diagram

Present state	Inputs	Next state
A	x y	A
0	0 0	0
0	0 1	1
0	1 0	1
0	1 1	0
1	0 0	1
1	0 1	0
1	1 0	0
1	1 1	1

(b) State table



(c) State diagram

FIGURE 5.17
Sequential circuit with D flip-flop

The expression specifies an odd function and is equal to 1 when only one variable is 1 or when all three variables are 1. This is indicated in the column for the next state of A.

The circuit has one flip-flop and two states. The state diagram consists of two circles, one for each state as shown in Fig. 5.17(c). The present state and the output can be either 0 or 1, as indicated by the number inside the circles. A slash on the directed lines is not needed, because there is no output from a combinational circuit. The two inputs can have four possible combinations for each state. Two input combinations during each state transition are separated by a comma to simplify the notation.

Analysis with *JK* Flip-Flops

A state table consists of four sections: present state, inputs, next state, and outputs. The first two are obtained by listing all binary combinations. The output section is determined from the output equations. The next-state values are evaluated from the state equations. For a D-type flip-flop, the state equation is the same as the input equation. When a flip-flop other than the D type is used, such as JK or T, it is necessary to refer to the corresponding characteristic table or characteristic equation to obtain the next-state values. We will illustrate the procedure first by using the characteristic table and again by using the characteristic equation.

The next-state values of a sequential circuit that uses *JK*- or *T*-type flip-flops can be derived as follows:

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

As an example, consider the sequential circuit with two *JK* flip-flops *A* and *B* and one input *x*, as shown in Fig. 5.18. The circuit has no outputs; therefore, the state table does not need an output column. (The outputs of the flip-flops may be considered as the outputs in this case.) The circuit can be specified by the flip-flop input equations

$$J_A = B \quad K_A = Bx'$$
$$J_B = x' \quad K_B = A'x + Ax' = A \oplus x$$

The state table of the sequential circuit is shown in Table 5.4. The present-state and input columns list the eight binary combinations. The binary values listed under the columns labeled *flip-flop inputs* are not part of the state table, but they are needed for the purpose of evaluating the next state as specified in step 2 of the procedure. These binary values are obtained directly from the four input equations in a manner similar to that for obtaining a truth table from a Boolean expression. The next state of each flip-flop is evaluated from the corresponding *J* and *K* inputs and the characteristic table of the *JK* flip-flop listed in Table 5.1. There are four cases to consider. When $J = 1$ and $K = 0$, the next state is 1. When $J = 0$ and

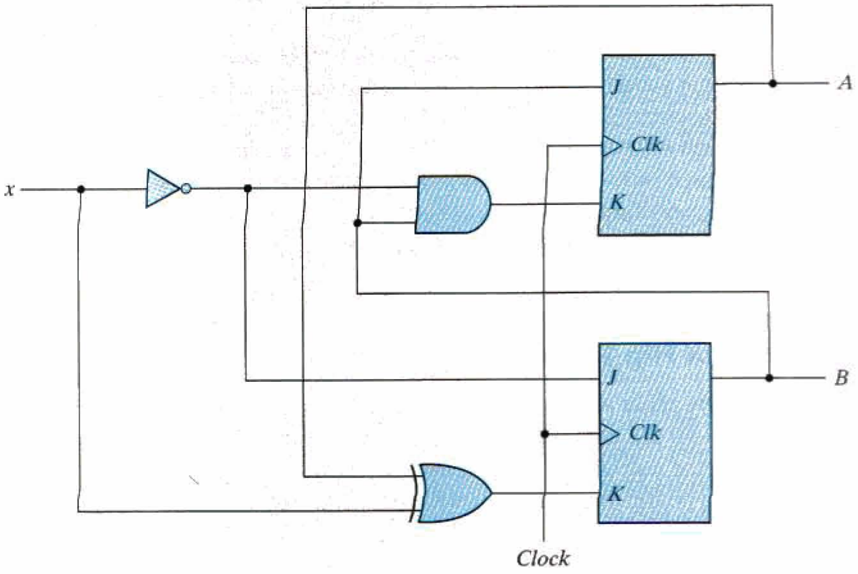


FIGURE 5.18
Sequential circuit with *JK* flip-flop

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

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

$K = 1$, 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:

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

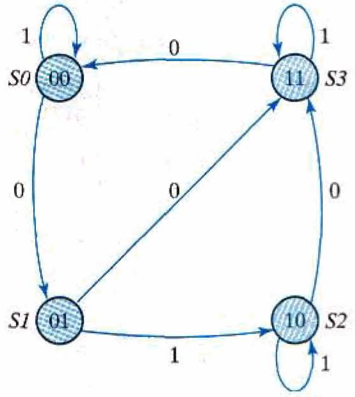


FIGURE 5.19
State diagram of the circuit of Fig. 5.18

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 the characteristic table listed in Table 5.1 or the characteristic equation

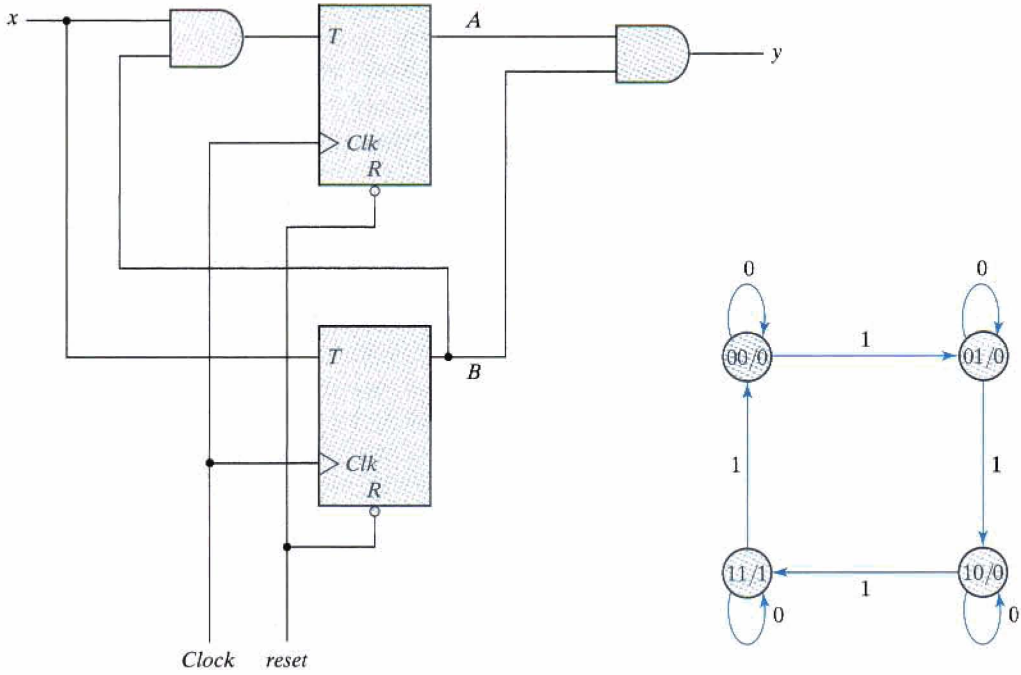
$$Q(t + 1) = T \oplus Q = T'Q + TQ'$$

Now consider the sequential circuit shown in Fig. 5.20. It has two flip-flops A and B, one input x, and one output y and can be described algebraically by two input equations and an output equation:

$$T_A = Bx$$
$$T_B = x$$
$$y = AB$$

The state table for the circuit is listed in Table 5.5. The values for y are obtained from the output equation. The values for the next state can be derived from the state equations by substituting T_A and T_B in the characteristic equations, yielding

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



(a) Circuit diagram

(b) State diagram

FIGURE 5.20
Sequential circuit with *T* flip-flops

The next-state values for *A* and *B* in the state table are obtained from the expressions of the two state equations.

The state diagram of the circuit is shown in Fig. 5.20(b). As long as input *x* is equal to 1, the circuit behaves as a binary counter with a sequence of states 00, 01, 10, 11, and back to 00.

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

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

When $x = 0$, the circuit remains in the same state. Output y is equal to 1 when the present state is 11. Here, the output depends on the present state only and is independent of the input. The two values inside each circle and separated by a slash are for the present state and output.

Mealy and Moore Models of Finite State Machines

The most general model of a sequential circuit has inputs, outputs, and internal states. It is customary to distinguish between two models of sequential circuits: the Mealy model and the Moore model. Both are shown in Figure 5.21. They differ only in the way the output 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.

An example of a Mealy model is given in Fig. 5.15. 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

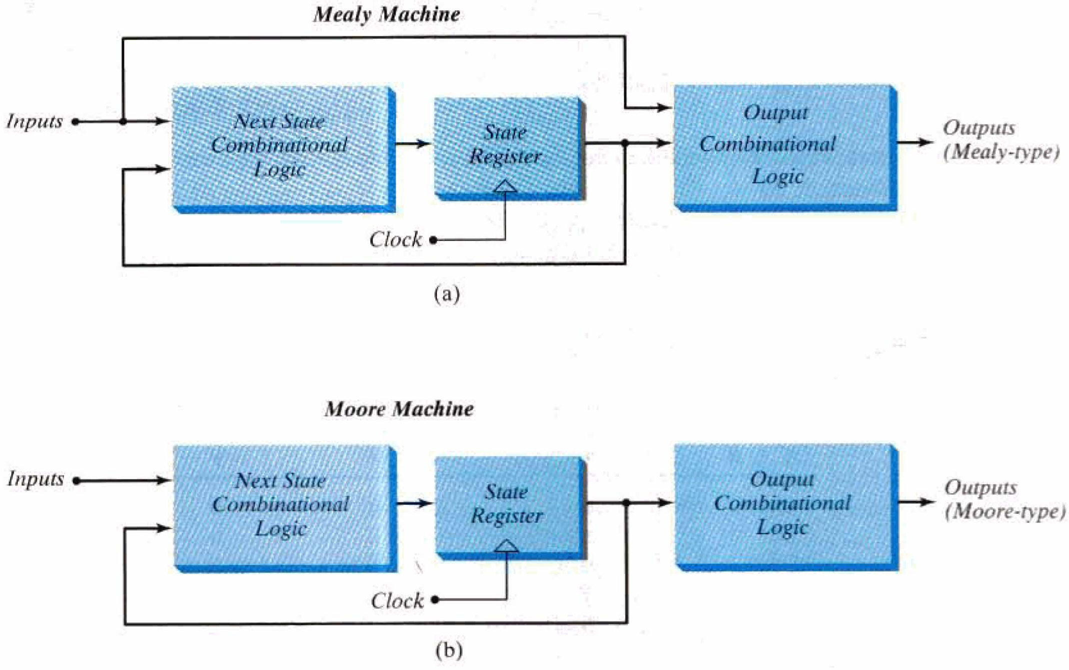


FIGURE 5.21
Block diagrams of Mealy and Moore state machines

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 cycle. Moreover, the outputs may have momentary false values because of the delay encountered from the time that the inputs change and the time that the flip-flop outputs change. In order to synchronize a Mealy-type circuit, the inputs of the sequential circuit must be synchronized with the clock and the outputs must be sampled immediately before the clock edge. The inputs are changed at the inactive edge of the clock to ensure that the inputs to the flip-flops stabilize before the active edge of the clock occurs. Thus, the output of the Mealy machine is the value that is present immediately before the active edge of the clock.

5.6 SYNTHESIZABLE HDL MODELS OF SEQUENTIAL CIRCUITS

The Verilog hardware description language (HDL) was introduced in Section 3.10. Combinational circuits were described in Section 4.12, and behavioral modeling with Verilog was introduced in that section as well. Behavioral models are abstract representations of the functionality of digital hardware. Designers write behavioral models to quickly describe how a circuit is to operate, without having to first specify its hardware. In this section, we continue the discussion of behavioral modeling and present description and examples of flip-flops and sequential circuits in preparation for modeling more complex circuits.

Behavioral Modeling

There are two kinds of abstract behaviors in the Verilog HDL. Behavior declared by the keyword **initial** is called *single-pass behavior* and specifies a single statement or a block statement (i.e., a list of statements enclosed by either a **begin** ... **end** or a **fork** ... **join** keyword pair). A single-pass behavior expires after the associated statement executes. In practice, designers use single-pass behavior primarily to prescribe stimulus signals in a test bench—never to model the behavior of a circuit—because synthesis tools do not accept descriptions that use the **initial** statement. The **always** keyword declares a *cyclic behavior.* Both types of behaviors begin executing when the simulator launches at time $t = 0$. The **initial** behavior expires after its statement executes; the **always** behavior executes and reexecutes indefinitely, until the simulation is stopped. A module may contain an arbitrary number of **initial** or **always** behavioral statements. They execute concurrently with respect to each other starting at time 0 and may interact through common variables. Here's a word description of how an **always** statement works for a simple model of a D flip-flop: Whenever the rising edge of the clock occurs, if the reset input is asserted, the output q gets 0; otherwise the output Q gets the value of the input D. The execution of statements triggered by the clock is repeated until the simulation ends. We'll see shortly how to write this description in Verilog.

An **initial** behavioral statement executes only once. It begins its execution at the start of simulation and expires after all of its statements have completed execution. As mentioned at the end of Section 4.12, the **initial** statement is useful for generating input signals to simulate a design. In simulating a sequential circuit, it is necessary to generate a clock source for triggering the flip-flops. The following are two possible ways to provide a free-running clock that operates for a specified number of cycles:

```
initial
 begin
  clock = 1'b0;
  repeat (30)
   #10 clock = ~clock;
 end
```

```
initial
 begin
  clock = 1'b0;
 end

initial 300 $finish;
always #10 clock = ~clock;
```

In the first version, the **initial** block contains two statements enclosed within the **begin** and **end** keywords. The first statement sets *clock* to 0 at time = 0. The second statement specifies a loop that reexecutes 30 times to wait 10 time units and then complement the value of *clock*. This produces 15 clock cycles, each with a cycle time of 20 time units. In the second version, the first **initial** behavior has a single statement that sets *clock* to 0 at time = 0, and it then expires (causes no further simulation activity). The second single-pass behavior declares a stopwatch for the simulation. The system task **finish** causes the simulation to terminate unconditionally after 300 time units have elapsed. Because this behavior has only one statement associated with it, there is no need to write the **begin** ... **end** keyword pair. After 10 time units, the **always** statement repeatedly complements *clock*, providing a clock generator having a cycle time of 20 time units. The three behavioral statements in the second example can be written in any order.

Here is another way to describe a free-running clock:

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

This version, with two statements on one line, initializes the clock and then executes an indefinite loop (**forever**) in which the clock is complemented after a delay of 10 time steps. Note that the single-pass behavior never finishes executing and so does not expire. Another behavior would have to terminate the simulation.

The activity associated with either type of behavioral statement can be controlled by a delay operator that waits for a certain time or by an event control operator that waits for certain conditions to become true or for specified events (changes in signals) to occur. Time delays specified with the # *delay control operator* are commonly used in single-pass behaviors. The delay control operator suspends execution of statements until a specified time has elapsed. We've already seen examples of its use to specify signals in a test bench. Another operator, @, is called the *event control operator* and is used to *suspend* activity until an event occurs. An event can be an unconditional change in a signal value (e.g., @ A) or a specified transition of a signal value (e.g., @ (**posedge** clock)). The general form of this type of statement is

```
always @ (event control expression) begin
  // Procedural assignment statements that execute when the condition is met
end
```

The event control expression specifies the condition that must occur to launch execution of the procedural assignment statements. The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such. The right-hand side can be any expression that produces a value using Verilog-defined operators.

The event control expression (also called the sensitivity list) specifies the events that must occur to initiate execution of the procedural statements associated with the **always** block. Statements within the block execute sequentially from top to bottom. After the last statement executes, the behavior waits for the event control expression to be satisfied. Then the statements are executed again. The sensitivity list can specify level-sensitive events, edge-sensitive events, or a combination of the two. In practice, designers do not make use of the third option, because this third form is not one that synthesis tools are able to translate into physical hardware. Level-sensitive events occur in combinational circuits and in latches. For example, the statement

```
always @ (A or B or C)
```

will initiate execution of the procedural statements in the associated **always** block if a change occurs in *A*, *B*, or *C*. In synchronous sequential circuits, changes in flip-flops occur only in response to a transition of a clock pulse. The transition may be either a positive edge or a negative edge of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the expression

```
always @(posedge clock or negedge reset)       // Verilog 1995
```

will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if *reset* goes through a negative transition. The 2001 and 2005 revisions to the Verilog language allow a comma-separated list for the event control expression (or sensitivity list):

```
always @(posedge clock, negedge reset)         // Verilog 2001, 2005
```

A procedural assignment is an assignment of a logic value to a variable within an **initial** or **always** statement. This is in contrast to a continuous assignment discussed in Section 4.12 with dataflow modeling. A continuous assignment has an implicit level-sensitive sensitivity list consisting of all of the variables on the right-hand side of its assignment statement. The updating of a continuous assignment is triggered whenever an event occurs in a variable listed on the right-hand side of its expression. In contrast, a procedural assignment is made only when an assignment statement is executed within a behavioral statement. For example, the clock signal in the preceding example was complemented only when the statement *clock* = ~*clock* executed; the statement did not execute until 10 time units after the simulation began. It is important to remember that a variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

There are two kinds of procedural assignments: *blocking* and *nonblocking*. The two are distinguished by the symbols that they use. Blocking assignments use the symbol (=) as the assignment operator, and nonblocking assignments use (<=) as the operator. Blocking assignment statements are executed sequentially in the order they are listed in a block of statements. Nonblocking assignments are executed concurrently by evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. The two types of assignments may be better understood by means of an illustration. Consider these two procedural blocking assignments:

```
B = A
C = B + 1
```

The first statement transfers A into B. The second statement increments the value of B and transfers the new value to C. At the completion of the assignments, C contains the value of $A + 1$.

Now consider the two statements as nonblocking assignments:

```
B <= A
C <= B + 1
```

When the statements are executed, the expressions on the right-hand side are evaluated and stored in a temporary location. The value of A is kept in one storage location and the value of $B + 1$ in another. After all the expressions in the block are evaluated and stored, the assignment to the targets on the left-hand side is made. In this case, C will contain the original value of B, plus 1. A general rule is to use blocking assignments when sequential ordering is imperative and in cyclic behavior that is level sensitive (i.e., in combinational logic). Use nonblocking assignments when modeling concurrent execution (e.g., edge-sensitive behavior such as synchronous, concurrent register transfers) and when modeling latched behavior. Nonblocking assignments are imperative in dealing with register transfer level design, as shown in Chapter 8. They model the concurrent operations of physical hardware synchronized by a common clock. Today's designers are expected to know what features of an HDL are useful in a practical way and how to avoid features that are not. Following these rules will prevent conditions that lead synthesis tools astray and create mismatches between the behavior of a model and the behavior of physical hardware that is produced by a synthesis tool.

Flip-Flops and Latches

HDL Examples 5.1 through 5.4 show descriptions of various flip-flops and a D latch. The D latch is transparent and responds to a change in data input with a change in output, as long as the enable input is asserted. The module description of a D latch is shown in HDL Example 5.1. It has two inputs, D and *enable*, and one output Q. Since Q is evaluated in a procedural statement, it must be declared as **reg** type. Latches respond to input signal levels, so the two inputs are listed without edge qualifiers in the event enable expression following the @ symbol in the **always** statement. There is only one blocking procedural assignment statement, and it specifies the transfer of input D to output Q if *enable* is true (logic 1). Note that this statement is executed every time there is a change in D if *enable* is 1.

A D-type flip-flop is the simplest example of a sequential machine. HDL Example 5.2 describes two positive-edge D flip-flops in two modules. The first responds only to the clock; the second includes an asynchronous reset input. Output Q must be declared as a **reg** data type in addition to being listed as an output. This is because it is a target output in a procedural assignment statement. The keyword **posedge** ensures that the transfer of input D into Q is synchronized by the positive-edge transition of *Clk*. A change in D at any other time does not change Q.

HDL Example 5.1

```
// Description of D latch (See Fig. 5.6)
module D_latch (Q, D, enable);
 output Q;
 input  D, enable;
 reg    Q;
 always @ (enable or D)
  if (enable) Q <= D;             // Same as: if (enable == 1)
 endmodule

// Alternative syntax (Verilog 2001, 2005)
module D_latch (output reg Q, input enable, D);
 always @ (enable, D)
  if (enable) Q <= D;             // No action if enable not asserted
endmodule
```

HDL Example 5.2

```
// D flip-flop without reset
module D_FF (Q, D, Clk);
 output Q;
 input  D, Clk;
 reg    Q;
 always @ (posedge Clk)
   Q <= D;
endmodule

// D flip-flop with asynchronous reset (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
 if (~rst) Q <= 1'b0;   // Same as: if (rst == 0)
 else Q <= D;
endmodule
```

The second module includes an asynchronous reset input in addition to the synchronous clock. A specific form of an **if** statement is used to describe such a flip-flop so that the model can be synthesized by a software tool. The event expression after the @ symbol in the **always** statement may have any number of edge events, either **posedge** or **negedge**. For modeling hardware, one of the events must be a clock event. The remaining events specify conditions under which asynchronous logic is to be executed. The designer knows which signal is the clock, but *clock* is not an identifier that software tools automatically recognize as the synchronizing signal of a circuit. The tool must be able to infer which signal is the clock, so you need to write the description in a way that enables the tool to infer the clock correctly. The rules are simple to follow: (1) Each **if** or **else if** statement in the procedural assignment statements is to correspond to an asynchronous event, (2) the last **else** statement corresponds to the clock event, and (3) the asynchronous events are tested first. There are two edge events in the second module of HDL Example 5.2. The **negedge** rst (reset) event is asynchronous, since it matches the **if** (~rst) statement. As long as *rst* is 0, *Q* is cleared to 0. If *Clk* has a positive transition, its effect is blocked. Only if $rst = 1$ can the **posedge** clock event synchronously transfer *D* into *Q*.

Hardware always has a reset signal. It is strongly recommended that all models of edge-sensitive behavior include a reset (or preset) input signal; otherwise, the initial state of the flip-flops of the sequential circuit cannot be determined. A sequential circuit cannot be tested with HDL simulation unless an initial state can be assigned with an input signal.

HDL Example 5.3 describes the construction of a *T* or *JK* flip-flop from a *D* flip-flop and gates. The circuit is described with the characteristic equations of the flip-flops:

$$Q(t+1) = Q \oplus T \qquad \text{for a } T \text{ flip-flop}$$
$$Q(t+1) = JQ' + K'Q \qquad \text{for a } JK \text{ flip-flop}$$

The first module, *TFF*, describes a *T* flip-flop by instantiating *DFF*. (Instantiation is explained in Section 4.12.) The declared **wire**, *DT*, is assigned the exclusive-OR of *Q* and *T*, as is required for building a *T* flip-flop with a *D* flip-flop. The instantiation with the value of *DT* replacing *D* in module *DFF* produces the required *T* flip-flop. The *JK* flip-flop is specified in a similar manner by using its characteristic equation to define a replacement for *D* in the instantiated *DFF*.

HDL Example 5.3

```
// T flip-flop from D flip-flop and gates
module TFF (Q, T, Clk, rst);
 output Q;
 input  T, Clk, rst;
 wire   DT;
 assign DT = Q ^ T ;                    // Continuous assignment
// Instantiate the D flip-flop
 DFF TF1 (Q, DT, Clk, rst);
endmodule

// JK flip-flop from D flip-flop and gates (V2001, 2005)
module JKFF (output reg Q, input J, K, Clk, rst);
 wire JK;
 assign JK = (J & ~Q) | (~K & Q);
// Instantiate D flip-flop
 DFF JK1 (Q, J, K, Clk, rst);
endmodule

// D flip-flop (V2001, V2005)
module DFF (output reg Q, input D, Clk, rst);
 always @ (posedge Clk, negedge rst)
  if (~rst) Q <= 1'b0;
  else Q <= D;
endmodule
```

HDL Example 5.4 shows another way to describe a *JK* flip-flop. Here, we choose to describe the flip-flop by using the characteristic table rather than the characteristic equation. The **case** multiway branch condition checks the two-bit number obtained by concatenating the bits of *J* and *K*. The **case** expression $(\{J, K\})$ is evaluated and compared with the values in the list of statements that follows. The first value that matches the true condition is executed. Since the concatenation of *J* and *K* produces a two-bit number, it can be equal to 00, 01, 10, or 11. The first bit gives the value of *J* and the second the value of *K*. The four possible conditions specify the value of the next state of *Q* after the application of a positive-edge clock.

HDL Example 5.4

```
// Functional description of JK flip-flop (V2001, 2005)
module  JK_FF (input J, K, Clk, output reg Q, output Q_b);
 assign Q_b = ~ 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
```

State Diagram

An HDL model of the operation of a sequential circuit can be based on the format of the circuit's state diagram. A Mealy HDL model is presented in HDL Example 5.5 for the state machine described by the state diagram shown in Figure 5.16. The input, output, clock, and reset are declared in the usual manner. The state of the flip-flops is declared with identifiers *state* and *next_state*. These variables hold the values of the present state and the next value of the sequential circuit. The state's binary assignment is done with a **parameter** statement. (Verilog allows constants to be defined in a module by the keyword **parameter**.) The four states *S0* through *S3* are assigned binary 00 through 11. The notation $S2 = 2'\text{b}10$ is preferable to the alternative $S2 = 2$. The former uses only two bits to store the constant, whereas the latter results in a binary number with 32 (or 64) bits.

HDL Example 5.5

```
// Mealy FSM zero detector (See Fig. 5.16)                    Verilog 2001, 2005 syntax
module Mealy_Zero_Detector (
 output reg y_out,
 input      x_in, clock, reset
);
 reg [1: 0]          state, next_state;
 parameter           S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
```

```
always @ (posedge clock, negedge reset)     Verilog 2001, 2005 syntax
 if (reset == 0) state <= S0;
 else state <= next_state;

always @ (state, x_in)                      // Form the next state
 case (state)
  S0:      if (x_in)   next_state = S1; else next_state = S0;
  S1:      if (x_in)   next_state = S3; else next_state = S0;
  S2:      if (~x_in)   next_state = S0; else next_state = S2;
  S3:      if (x_in)   next_state = S2; else next_state = S0;
 endcase

always @ (state, x_in)                      // Form the output
 case (state)
  S0:          y_out = 0;
  S1, S2, S3: y_out = ~x_in;
 endcase
endmodule

module t_Mealy_Zero_Detector;
 wire    t_y_out;
 reg     t_x_in, t_clock, t_reset;

Mealy_Zero_Detector M0 (t_y_out, t_x_in, t_clock, t_reset);
initial #200 $finish;
initial begin t_clock = 0; forever #5 t_clock = ~t_clock; end

initial fork
    t_reset = 0;
 #2 t_reset = 1;
 #87 t_reset = 0;
 #89 t_reset = 1;
 #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 Verilog model in HDL Example 5.5 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 executed only in response 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*

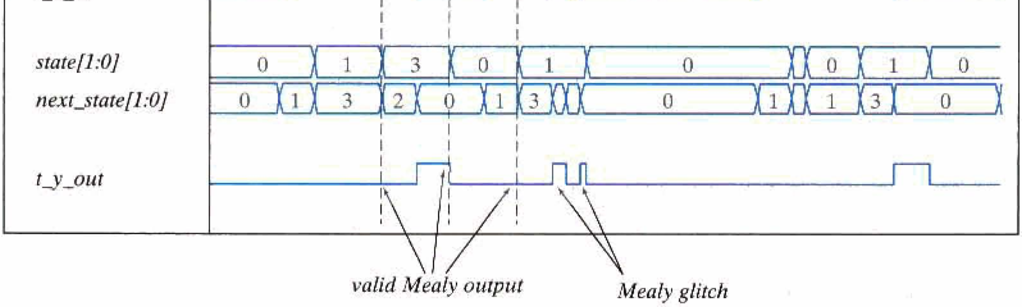


FIGURE 5.22
Simulation output of *Mealy_Zero_Detector*

responds to changes in both the state and the input and has a glitch (a transient logic value). The waveform description uses the **fork** ... **join** construct. Statements within the **fork** ... **join** block execute in parallel, so the time delays are relative to a common reference of $t = 0$. It is usually more convenient to use the **fork** ... **join** block instead of the **begin** ... **end** block in describing waveforms. The waveform of *reset* is triggered "on the fly" to demonstrate that the machine recovers from an unexpected 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. Their sensitivity lists include both the state and the input because their logic must respond to a change in either or both of them.

Note: the modeling style illustrated by *Mealy_Zero_Detector* is commonly used by designers. Notice that the reset signal is associated with the first **always** block. It is modeled here as an active-low reset. By including the reset in the model of the state transition, there is no need to include it in the combinational logic that specifies the next state and the output, producing a simpler and more readable description.

The behavior of the Moore FSM having the state diagram shown in Fig. 5.19 can be modeled by the Verilog description in HDL Example 5.6. This example shows that it is possible to describe the state transitions of a clocked sequential machine with only one **always** block. The present state of the circuit is identified by the variable *state*. The state transitions are triggered by the rising edge of the clock according to the conditions listed in the **case** statements. The combinational logic that implicitly determines the next state is included in the nonblocking assignment to *state*. In this example, the output of the circuit is independent of the input and is taken directly from the outputs of the flip-flops. The two-bit output *y_out* is specified with a continuous (**assign**) statement and is equal to the value of the present state vector. Figure 5.23 shows some simulation results for *Moore_Model_Fig_5_19*. Notice that the output of the Moore machine does not have glitches.

HDL Example 5.6

```
// Moore model FSM (see Fig. 5.19)                Verilog 2001, 2005 syntax
module Moore_Model_Fig_5_19 (
 output [1: 0]           y_out,
 input                   x_in, clock, reset
);
 reg [1: 0]              state;
 parameter               S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

 always @ (posedge clock, negedge reset)
  if (reset == 0) state <= S0;                        // Initialize to state S0
  else case (state)
```

```
      S0:   if (~x_in)  state <= S1; else state <= S0;
      S1:   if (x_in)   state <= S2; else state <= S3;
      S2:   if (~x_in)  state <= S3; else state <= S2;
      S3:   if (~x_in)  state <= S0; else state <= S3;
    endcase

   assign y_out = state;     // Output of flip-flops

endmodule
```

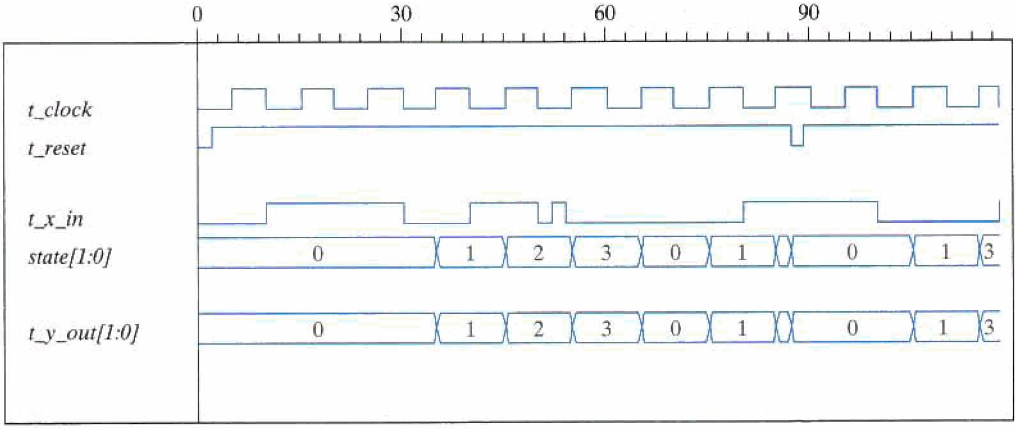


FIGURE 5.23
Simulation output of HDL Example 5.6

Structural Description of Clocked Sequential Circuits

Combinational logic circuits can be described in Verilog by a connection of gates (primitives and UDPs), by dataflow statements (continuous assignments), or by level-sensitive cyclic behaviors (**always** blocks). Sequential circuits are composed of combinational logic and flip-flops, and their HDL models use sequential UDPs and behavioral statements (edge-sensitive cyclic behaviors) to describe the operation of flip-flops. One way to describe a sequential circuit uses a combination of dataflow and behavioral statements. The flip-flops are described with an **always** statement. The combinational part can be described with **assign** statements and Boolean equations. The separate modules can be combined to form a structural model by instantiation within a **module**.

The structural description of a sequential circuit is shown in HDL Example 5.7. We want to encourage the reader to consider alternative ways to model a circuit, so as a point of comparison, we first present *Moore_Model_Fig_5_20*, a Verilog behavioral description of the machine having the state diagram shown in Fig. 5.20. This style of modeling is direct.

An alternative style, used in *Moore_Model_STR_Fig_5_20*, is to represent the structure shown in Fig. 5.20(a). This style uses two modules. The first describes the circuit of Fig. 5.20(a). The second describes the *T* flip-flop that will be used by the circuit. We also show two ways to model the T flip-flop. The first asserts that, at every clock tick, the value of the output of the flip-flop toggles if the toggle input is asserted. The second model describes the behavior of the toggle flip-flop in terms of its characteristic equation. The first style is attractive because it does not require the reader to remember the characteristic equation. Nonetheless, the models are interchangeable and will synthesize to the same hardware circuit. A test bench module provides a stimulus for verifying the functionality of the circuit. The sequential circuit is a two-bit binary counter controlled by input *x_in*. The output, *y_out*, is enabled when the count reaches binary 11. Flip-flops *A* and *B* are included as outputs in order to check their operation. The flip-flop input equations and the output equation are evaluated with continuous assignment (**assign**) statements having the corresponding Boolean expressions. The instantiated *T* flip-flops use *TA* and *TB* as defined by the input equations.

The second module describes the *T* flip-flop. The *reset* input resets the flip-flop to 0 with an active-low signal. The operation of the flip-flop is specified by its characteristic equation, $Q(t + 1) = Q \oplus T$.

The test bench includes both models of the machine. The stimulus module provides common inputs to the circuits to simultaneously display their output responses. The first **initial** block provides eight clock cycles with a period of 10 ns. The second **initial** block specifies a toggling of input *x_in* that occurs at the negative edge transition of the clock. The result of the simulation is shown in Fig. 5.24. The pair (*A*, *B*) goes through the binary sequence 00, 01, 10, 11, and back to 00. The change in the count is triggered by a positive edge of the clock, provided that $x\_in = 1$. If $x\_in = 0$, the count does not change. *y_out* is equal to 1 when both *A* and *B* are equal to 1. This verifies the main functionality of the circuit, but not a recovery from an unexpected reset event.

HDL Example 5.7

```
// State-diagram-based model (V2001, 2005)
module Moore_Model_Fig_5_20 (
 output y_out,
 input   x_in, clock, reset
);
 reg [1: 0]             state;
 parameter              S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

 always @ (posedge clock, negedge reset)
  if (reset == 0) state <= S0;                        // Initialize to state S0
  else case (state)
   S0:   if (x_in)  state <= S1; else state <= S0;
   S1:   if (x_in)  state <= S2; else state <= S1;
```

```
     S2:   if (x_in)  state <= S3; else state <= S2;
     S3:   if (x_in)  state <= S0; else state <= S3;
   endcase

  assign y_out = (state == S3);                    // Output of flip-flops
endmodule

// structural model
module Moore_Model_STR_Fig_5_20 (
  output    y_out, A, B,
  input     x_in, clock, reset
);
  wire      TA, TB;

// Flip-flop input equations
  assign TA = x_in & B;
  assign TB = x_in;
// Output equation
  assign y_out = A & B;

// Instantiate Toggle flip-flops
  Toggle_flip_flop_3 M_A (A, TA, clock, reset);
  Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule

module Toggle_flip_flop (Q, T, CLK, RST_b);
  output    Q;
  input     T, CLK, RST_b;
  reg       Q;

  always @ (posedge CLK, negedge RST_b)
    if (RST_b == 0) Q <= 1'b0;
    else if (T) Q <= ~Q;
endmodule

// Alternative model using characteristic equation
// module Toggle_flip_flop (Q, T, CLK, RST_b);
// output  Q;
// input   T, CLK, RST_b;
// reg     Q;

// always @ (posedge CLK, negedge RST)
//   if (RST_b == 0) Q <= 1'b0;
//   else  Q <= Q ^ T;
// endmodule
```

```
module t_Moore_Fig_5_20;
 wire              t_y_out_2, t_y_out_1;
 reg               t_x_in, t_clock, t_reset;

 Moore_Model_Fig_5_20          M1(t_y_out_1, t_x_in, t_clock, t_reset);
 Moore_Model_STR_Fig_5_20      M2 (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

 initial #200 $finish;
 initial begin
    t_reset = 0;
    t_clock = 0;
    #5 t_reset = 1;
  repeat (16)
   #5 t_clock = ~t_clock;
 end
 initial begin
      t_x_in = 0;
  #15 t_x_in = 1;
  repeat (8)
   #10 t_x_in = ~t_x_in;
 end
endmodule
```

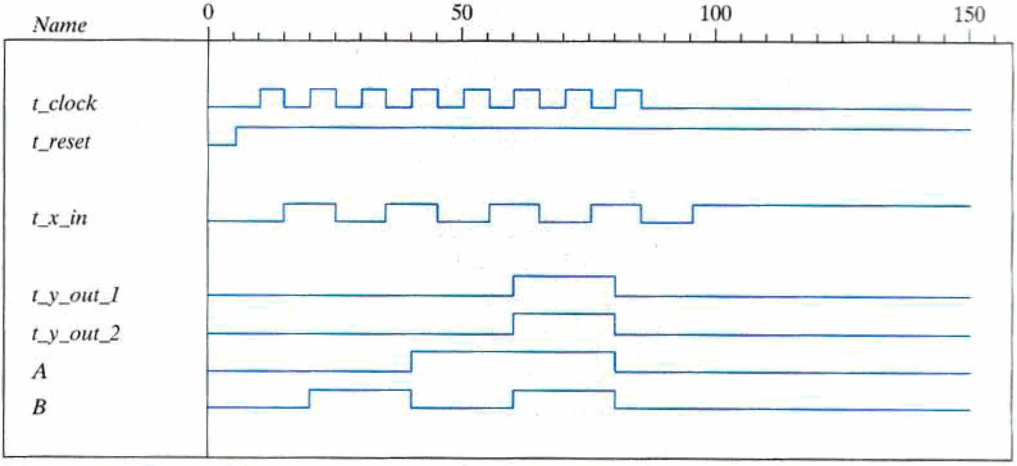


FIGURE 5.24
Simulation output of HDL Example 5.7

5.7 STATE REDUCTION AND ASSIGNMENT

The *analysis* of sequential circuits starts from a circuit diagram and culminates in a state table or diagram. The *design* (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Design procedures are presented in Section 5.8. Two sequential circuits may exhibit the same input–output behavior, but have a different number of internal states in their state diagram. The current section discusses certain properties of sequential circuits that may simplify a design by reducing the number of gates and flip-flops it uses. In general, reducing the number of flip-flops reduces the cost of a circuit.

State Reduction

The reduction in the number of flip-flops in a sequential circuit is referred to as the *state-reduction* problem. State-reduction algorithms are concerned with procedures for reducing the number of states in a state table, while keeping the external input–output requirements unchanged. Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates.

We will illustrate the state-reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram of Fig. 5.25. In our example, only the input–output sequences are important; the internal states are used merely to provide the required sequences. For that reason, the states marked inside the circles are denoted by letter symbols instead of their binary values. This is in contrast to a binary counter, wherein the binary value sequence of the states themselves is taken as the outputs.

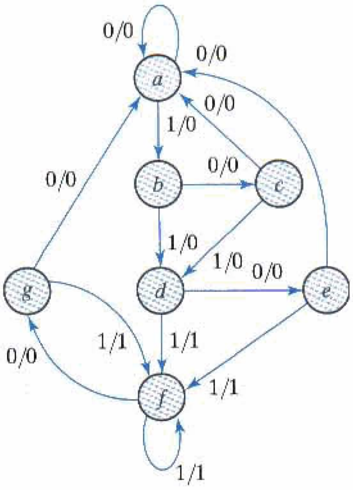


FIGURE 5.25
State diagram

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.

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

The following algorithm for the state reduction of a completely specified state table is given here without proof: "Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an

Table 5.6
State Table

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

Table 5.7
Reducing the State Table

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

equivalent state." When two states are equivalent, one of them can be removed without altering the input–output relationships.

Now apply this algorithm to Table 5.6. Going through the state table, we look for two present states that go to the same next state and have the same output for both input combinations. States g and e are two such states: They both go to states a and f and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states g and e are equivalent, and one of these states can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 5.7. The row with present state g is removed, and state g is replaced by state e each time it occurs in the columns headed "Next State."

Present state f now has next states e and f and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively. The same next states and outputs appear in the row with present state d. Therefore, states f and d are equivalent, and state f can be removed and replaced by d. The final reduced table is shown in Table 5.8. The state diagram for the reduced table consists of only five states and is shown in Fig. 5.26. This state diagram satisfies the original input–output specifications and will produce the required output sequence for any given input sequence. The following list derived from the state diagram of Fig. 5.26 is for the input sequence used previously (note that the same output sequence results, although the state sequence is different):

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

Table 5.8
Reduced State Table

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

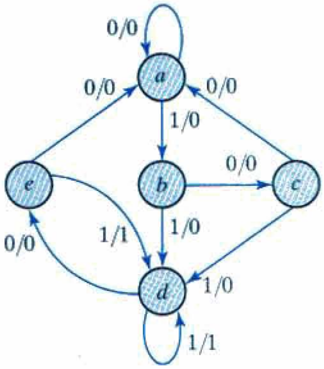


FIGURE 5.26
Reduced state diagram

In fact, this sequence is exactly the same as that obtained for Fig. 5.25 if we replace *g* by *e* and *f* by *d*.

Checking each pair of states for equivalency can be done systematically by means of a procedure that employs an implication table, which consists of squares, one for every suspected pair of possible equivalent states. By judicious use of the table, it is possible to determine all pairs of equivalent states in a state table. The use of the implication table for reducing the number of states in a state table is demonstrated in Section 9.5.

The sequential circuit of this example was reduced from seven to five states. In general, reducing the number of states in a state table may result in a circuit with less equipment. However, the fact that a state table has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates.

State Assignment

In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m states, the codes must contain n bits, where $2^n \geq m$. For example, with three bits, it is possible to assign codes to eight states, denoted by binary numbers 000 through 111. If the state table of Table 5.6 is used, we must assign binary values to seven states; the remaining state is unused. If the state table of Table 5.8 is used, only five states need binary assignment, and we are left with three unused states. Unused states are treated as don't-care conditions during the design. Since don't-care conditions usually help in obtaining a simpler circuit, it is more likely that the circuit with five states will require fewer combinational gates than the one with seven states.

The simplest way to code five states is to use the first five integers in binary counting order, as shown in the first assignment of Table 5.9. Another similar assignment is the Gray code shown in assignment 2. Here, only one bit in the code group changes when going from one number to the next. This code makes it easier for the Boolean functions to be placed in the map for simplification. Another possible assignment often used in the design of state machines to control data-path units is the one-hot assignment. This configuration uses as many bits as there are

Table 5.9
Three Possible Binary State Assignments

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

Table 5.10
Reduced State Table with Binary Assignment 1

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

states in the circuit. At any given time, only one bit is equal to 1 while all others are kept at 0. This type of assignment uses one flip-flop per state, which is not an issue for register-rich field-programmable gate arrays. (See Chapter 7.) One-hot encoding usually leads to simpler decoding logic for the next state and output. One-hot machines can be faster than machines with sequential binary encoding, and the silicon area required by the extra flip-flops can be offset by the area 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 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.

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 gate structure that, together with the flip-flops, produces a circuit which fulfills the stated specifications. The number of flip-flops is determined from the number of states needed in the circuit. The combinational circuit is derived from the state table by evaluating the flip-flop input equations and output equations. In fact, once the type and number of flip-flops are determined, the design process involves a transformation from a sequential circuit problem into a combinational circuit problem. In this way, the techniques of combinational circuit design can be applied.

The procedure for designing synchronous sequential circuits can be summarized by a list of recommended steps:

1. From the word description and specifications of the desired operation, derive a state diagram for the circuit.
2. Reduce the number of states if necessary.
3. Assign binary values to the states.
4. Obtain the binary-coded state table.
5. Choose the type of flip-flops to be used.
6. Derive the simplified flip-flop input equations and output equations.
7. Draw the logic diagram.

The word specification of the circuit behavior usually assumes that the reader is familiar with digital logic terminology. It is necessary that the designer use intuition and experience to arrive at the correct interpretation of the circuit specifications, because word descriptions may be incomplete and inexact. Once such a specification has been set down and the state diagram obtained, it is possible to use known synthesis procedures to complete the design. Although there are formal procedures for state reduction and assignment (steps 2 and 3), they are seldom used by experienced designers. Steps 4 through 7 in the design can be implemented by exact algorithms and therefore can be automated. The part of the design that follows a well-defined procedure is referred to as *synthesis*. Designers using logic synthesis tools (software) can follow a simplified process that develops an HDL description directly from a state diagram, letting the synthesis tool determine the circuit elements and structure that implement the description.

The first step is a critical part of the process, because succeeding steps depend on it. We will give one simple example to demonstrate how a state diagram is obtained from a word specification.

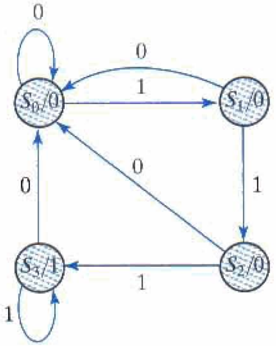


FIGURE 5.27
State diagram for sequence detector

Suppose we wish to design a circuit that detects a sequence of three or more consecutive 1's in a string of bits coming through an input line (i.e., the input is a *serial bit stream*). The state diagram for this type of circuit is shown in Fig. 5.27. It is derived by starting with state S_0, the reset state. If the input is 0, the circuit stays in S_0, but if the input is 1, it goes to state S_1 to indicate that a 1 was detected. If the next input is 1, the change is to state S_2 to indicate the arrival of two consecutive 1's, but if the input is 0, the state goes back to S_0. The third consecutive 1 sends the circuit to state S_3. If more 1's are detected, the circuit stays in S_3. Any 0 input sends the circuit back to S_0. In this way, the circuit stays in S_3 as long as there are three or more consecutive 1's received. This is a Moore model sequential circuit, since the output is 1 when the circuit is in state S_3 and is 0 otherwise.

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 can be obtained directly from the next-state columns of A and B and expressed in sum-of-minterms form as

$$A(t + 1) = D_A(A, B, x) = \Sigma(3, 5, 7)$$
$$B(t + 1) = D_B(A, B, x) = \Sigma(1, 5, 7)$$
$$y(A, B, x) = \Sigma(6, 7)$$

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

where A and B are the present-state values of flip-flops A and B, x is the input, and D_A and D_B are the input equations. The minterms for output y are obtained from the output column in the state table.

The Boolean equations are simplified by means of the maps plotted in Fig. 5.28. The simplified equations are

$$D_A = Ax + Bx$$
$$D_B = Ax + B'x$$
$$y = AB$$

The advantage of designing with D flip-flops is that the Boolean equations describing the inputs to the flip-flops can be obtained directly from the state table. Software tools automatically infer and select the D-type flip-flop from a properly written HDL model. The schematic of the sequential circuit is drawn in Fig. 5.29.

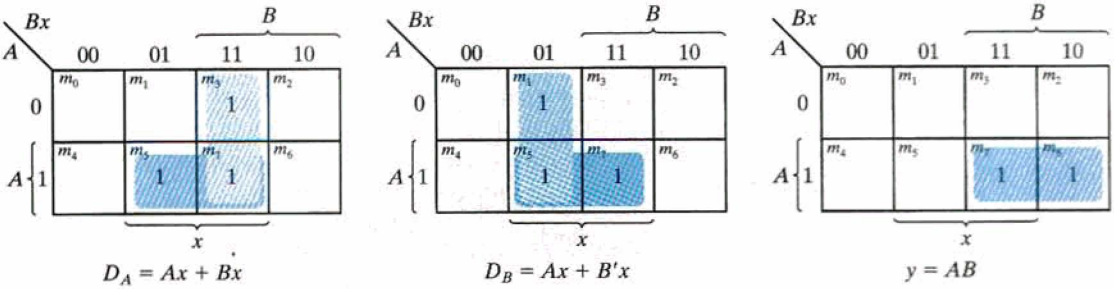


FIGURE 5.28
Maps for sequence detector

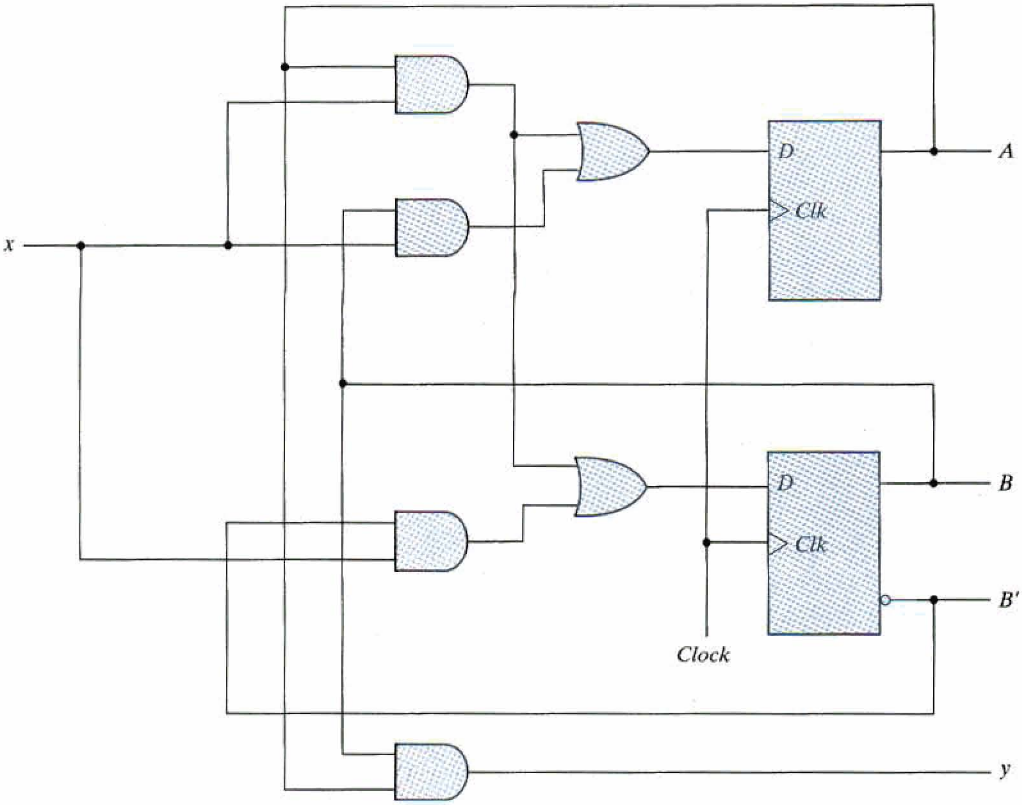


FIGURE 5.29
Logic diagram of sequence detector

Excitation Tables

The design of a sequential circuit with flip-flops other than the *D* type is complicated by the fact that the input equations for the circuit must be derived indirectly from the state table. When *D*-type flip-flops are employed, the input equations are obtained directly from the next state. This is not the case for the *JK* and *T* types of flip-flops. In order to determine the input equations for these flip-flops, it is necessary to derive a functional relationship between the state table and the input equations.

The flip-flop characteristic tables presented in Table 5.1 provide the value of the next state when the inputs and the present state are known. These tables are useful for analyzing sequential circuits and for defining the operation of the flip-flops. During the design process, we usually know the transition from the present state to the next state and wish to find the flip-flop input conditions that will cause the required transition. For this reason, we need a table that lists the required inputs for a given change of state. Such a table is called an *excitation table*.

Table 5.12 shows the excitation tables for the two flip-flops. Each table has a column for the present state $Q(t)$, a column for the next state $Q(t + 1)$, and a column for each input to show

Table 5.12
Flip-Flop Excitation Tables

$Q(t)$	$Q(t = 1)$	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(a) *JK*

$Q(t)$	$Q(t = 1)$	T
0	0	0
0	1	1
1	0	1
1	1	0

(b) *T*

how the required transition is achieved. There are four possible transitions from the present state to the next state. The required input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol X in the tables represents a don't-care condition, which means that it does not matter whether the input is 1 or 0.

The excitation table for the *JK* flip-flop is shown in part (a). When both present state and next state are 0, the J input must remain at 0 and the K input can be either 0 or 1. Similarly, when both present state and next state are 1, the K input must remain at 0, while the J input can be 0 or 1. If the flip-flop is to have a transition from the 0-state to the 1-state, J must be equal to 1, since the J input sets the flip-flop. However, input K may be either 0 or 1. If $K = 0$, the $J = 1$ condition sets the flip-flop as required; if $K = 1$ and $J = 1$, the flip-flop is complemented and goes from the 0-state to the 1-state as required. Therefore, the K input is marked with a don't-care condition for the 0-to-1 transition. For a transition from the 1-state to the 0-state, we must have $K = 1$, since the K input clears the flip-flop. However, the J input may be either 0 or 1, since $J = 0$ has no effect and $J = 1$ together with $K = 1$ complements the flip-flop with a resultant transition from the 1-state to the 0-state.

The excitation table for the T flip-flop is shown in part (b). From the characteristic table, we find that when input $T = 1$, the state of the flip-flop is complemented, and when $T = 0$, the state of the flip-flop remains unchanged. Therefore, when the state of the flip-flop must remain the same, the requirement is that $T = 0$. When the state of the flip-flop has to be complemented, T must equal 1.

Synthesis Using *JK* Flip-Flops

The manual synthesis procedure for sequential circuits with *JK* flip-flops is the same as with D flip-flops, except that the input equations must be evaluated from the present-state to the next-state transition derived from the excitation table. To illustrate the procedure, we will synthesize the sequential circuit specified by Table 5.13. In addition to having columns for the present state, input, and next state, as in a conventional state table, the table shows the flip-flop input conditions from which the input equations are derived. The flip-flop inputs are derived from the state table in conjunction with the excitation table for the *JK* flip-flop. For example, in the first row of Table 5.13, we have a transition for flip-flop A from 0 in the present state to 0 in the next state. In Table 5.12, for the *JK* flip-flop, we find that a transition of states from present state 0 to next state 0 requires that input J be 0 and input K be a don't-care. So 0 and X are

Table 5.13
State Table and JK Flip-Flop Inputs

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

entered in the first row under J_A and K_A, respectively. Since the first row also shows a transition for flip-flop B from 0 in the present state to 0 in the next state, 0 and X are inserted into the first row under J_B and K_B, respectively. The second row of the table shows a transition for flip-flop B from 0 in the present state to 1 in the next state. From the excitation table, we find that a transition from 0 to 1 requires that J be 1 and K be a don't-care, so 1 and X are copied into the second row under J_B and K_B, respectively. The process is continued for each row in the table and for each flip-flop, with the input conditions from the excitation table copied into the proper row of the particular flip-flop being considered.

The flip-flop inputs in Table 5.13 specify the truth table for the input equations as a function of present state A, present state B, and input x. The input equations are simplified in the maps of Fig. 5.30. The next-state values are not used during the simplification, since the input equations are a function of the present state and the input only. Note the advantage of using JK-type flip-flops when sequential circuits are designed manually. The fact that there are so many don't-care entries indicates that the combinational circuit for the input equations is likely to be simpler, because don't-care minterms usually help in obtaining simpler expressions. If there are unused states in the state table, there will be additional don't-care conditions in the map.

The four input equations for the pair of JK flip-flops are listed under the maps of Fig. 5.30. The logic diagram (schematic) of the sequential circuit is drawn in Fig. 5.31.

Synthesis Using *T* Flip-Flops

The procedure for synthesizing circuits using T flip-flops will be demonstrated by designing a binary counter. An n-bit binary counter consists of n flip-flops that can count in binary from 0 to $2^n - 1$. The state diagram of a three-bit counter is shown in Fig. 5.32. As seen from the binary states indicated inside the circles, the flip-flop outputs repeat the binary count sequence with a return to 000 after 111. The directed lines between circles are not marked with input and output values as in other state diagrams. Remember that state transitions in clocked sequential circuits occur during a clock edge; the flip-flops remain in their present states if no clock is applied. For that reason, the clock does not appear explicitly as an input variable in

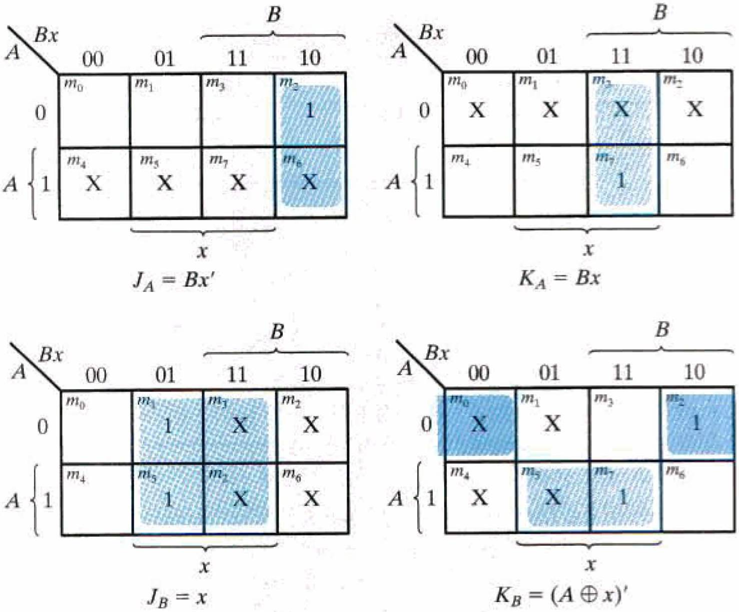


FIGURE 5.30
Maps for *J* and *K* input equations

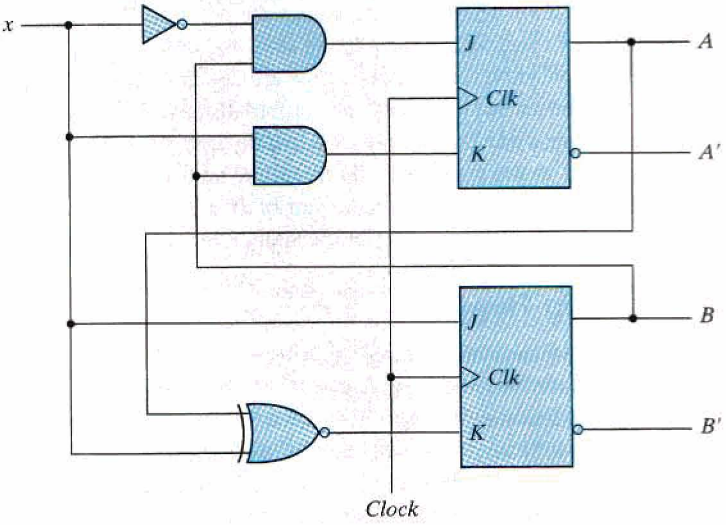


FIGURE 5.31
Logic diagram for sequential circuit with *JK* flip-flops

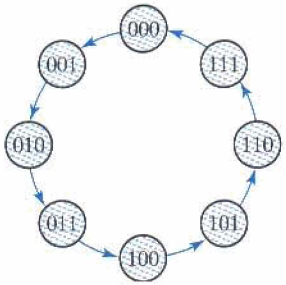


FIGURE 5.32
State diagram of three-bit binary counter

a state diagram or state table. From this point of view, the state diagram of a counter does not have to show input and output values along the directed lines. The only input to the circuit is the clock, and the outputs are specified by the present state of the flip-flops. The next state of a counter depends entirely on its present state, and the state transition occurs every time the clock goes through a transition.

Table 5.14 is the state table for the three-bit binary counter. The three flip-flops are symbolized by A_2, A_1, and A_0. Binary counters are constructed most efficiently with T flip-flops because of their complement property. The flip-flop excitation for the T inputs is derived from the excitation table of the T flip-flop and by inspection of the state transition of the present state to the next state. As an illustration, consider the flip-flop input entries for row 001. The present state here is 001 and the next state is 010, which is the next count in the sequence. Comparing these two counts, we note that A_2 goes from 0 to 0, so T_{A2} is marked with 0 because flip-flop A_2 must not change when a clock occurs. Also, A_1 goes from 0 to 1, so T_{A1} is marked with a 1 because this flip-flop must be complemented in the next clock edge. Similarly, A_0 goes from 1 to 0, indicating that it must be complemented, so T_{A0} is marked with a 1. The last row, with present state 111, is compared with the first count 000, which is its next state. Going from all 1's to all 0's requires that all three flip-flops be complemented.

Table 5.14
State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

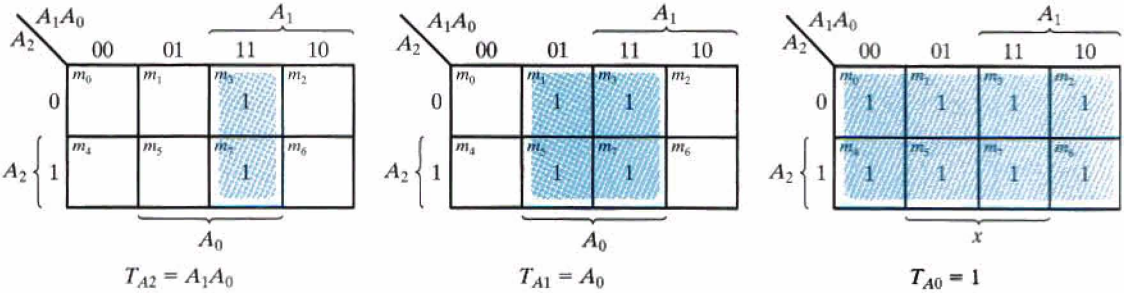


FIGURE 5.33
Maps for three-bit binary counter

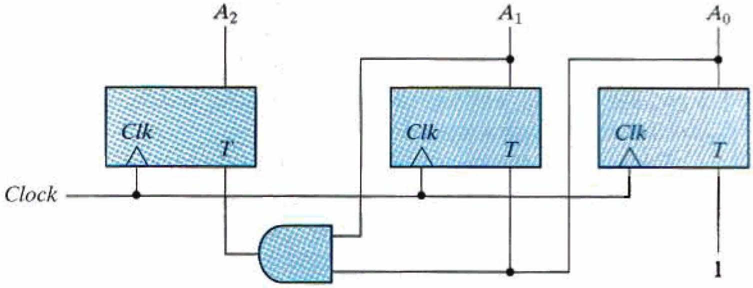


FIGURE 5.34
Logic diagram of three-bit binary counter

The flip-flop input equations are simplified in the maps of Fig. 5.33. Note that T_{A0} has 1's in all eight minterms because the least significant bit of the counter is complemented with each count. A Boolean function that includes all minterms defines a constant value of 1. The input equations listed under each map specify the combinational part of the counter. Including these functions with the three flip-flops, we obtain the logic diagram of the counter, as shown in Fig. 5.34. For simplicity, the reset signal is not shown, but be aware that every design should include a reset signal.

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 HDL model, a test plan should 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 are available, the Verilog descriptions developed for Problems 5.34–5.46 can be assigned as synthesis exercises. The gate-level circuit produced by the synthesis tools should be simulated and compared with the simulation results for the presynthesis model.

5.1 The *D* latch of Fig. 5.6 is constructed with four NAND gates and an inverter. Consider the following three other ways for obtaining a *D* latch, and in each case draw the logic diagram and verify the circuit operation:

(a) Use NOR gates for the *SR* latch part and AND gates for the other two. An inverter may be needed.

(b) Use NOR gates for all four gates. Inverters may be needed.

(c) Use four NAND gates only (without an inverter). This can be done by connecting the output of the upper gate in Fig. 5.6 (the gate that goes to the *SR* latch) to the input of the lower gate (instead of the inverter output).

5.2 Construct a *JK* flip-flop, using a *D* flip-flop, a two-to-one-line multiplexer, and an inverter. (HDL—see Problem 5.34.)

5.3 Show that the characteristic equation for the complement output of a *JK* flip-flop is

$$Q'(t + 1) = J'Q' + KQ$$

5.4 A *PN* flip-flop has four operations, clear to 0, no change, complement, and set to 1, when inputs *P* and *N* are 00, 01, 10, and 11, respectively.

(a) Tabulate the characteristic table.

(b)* Derive the characteristic equation.

(c) Tabulate the excitation table.

(d) Show how the *PN* flip-flop can be converted to a *D* flip-flop.

5.5 Explain the differences among a truth table, a state table, a characteristic table, and an excitation table. Also, explain the difference among a Boolean equation, a state equation, a characteristic equation, and a flip-flop input equation.

5.6 A sequential circuit with two *D* flip-flops *A* and *B*, two inputs *x* and *y*, and one output *z* is specified by the following next-state and output equations (HDL—see Problem 5.35):

$$A(t + 1) = x'y + xB$$

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

$$z = A$$

(a) Draw the logic diagram of the circuit.

(b) List the state table for the sequential circuit.

(c) Draw the corresponding state diagram.

5.7* A sequential circuit has one flip-flop *Q*, two inputs *x* and *y*, and one output *S*. It consists of a full-adder circuit connected to a *D* flip-flop, as shown in Fig. P5.7. Derive the state table and state diagram of the sequential circuit.

5.8* Derive the state table and the state diagram of the sequential circuit shown in Fig. P5.8. Explain the function that the circuit performs. (HDL—see Problem 5.36.)

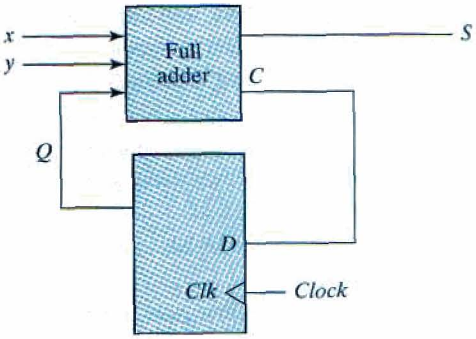


FIGURE P5.7

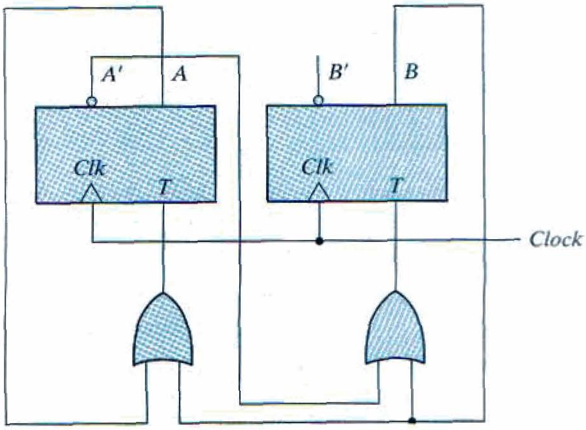


FIGURE P5.8

5.9 A sequential circuit has two *JK* flip-flops *A* and *B* and one input *x*. The circuit is described by the following flip-flop input equations:

$$J_A = x \quad K_A = B'$$
$$J_B = x \quad K_B = A$$

(a)* Derive the state equations $A(t + 1)$ and $B(t + 1)$ by substituting the input equations for the *J* and *K* variables.

(b) Draw the state diagram of the circuit.

5.10 A sequential circuit has two *JK* flip-flops *A* and *B*, two inputs *x* and *y*, and one output *z*. The flip-flop input equations and circuit output equation are

$$J_A = Bx + B'y' \qquad K_A = B'xy'$$
$$J_B = A'x \qquad K_B = A + xy'$$
$$z = Ax'y' + Bx'y'$$

(a) Draw the logic diagram of the circuit.

(b) Tabulate the state table.

(c)* Derive the state equations for A and B.

5.11* Starting from state 00 in the state diagram of Fig. 5.16, determine the state transitions and output sequence that will be generated when an input sequence of 010110111011110 is applied.

5.12* Reduce the number of states in the following state table, and tabulate the reduced state table:

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

5.13* Starting from state a and the input sequence 01110010011, determine the output sequence for

(a) the state table of the previous problem and

(b) the reduced state table from the previous problem. Show that the same output sequence is obtained for both.

5.14 Substitute binary assignment 2 from Table 5.9 to the states in Table 5.8, and obtain the binary state table.

5.15* List a state table for the JK flip-flop, using Q as the present and next state and J and K as inputs. Design the sequential circuit specified by the state table, and show that it is equivalent to Fig. 5.12(a).

5.16* Design a sequential circuit with two D flip-flops A and B and one input x_in.

(a) When $x\_in = 0$, the state of the circuit remains the same. When $x\_in = 1$, the circuit goes through the state transitions from 00 to 01, to 11, to 10, back to 00, and repeats.

(b) When $x\_in = 0$, the state of the circuit remains the same. When $x\_in = 1$, the circuit goes through the state transitions from 00 to 11, to 01, to 10, back to 00, and repeats. (HDL—see Problems 5.38.)

5.17 Design a one-input, one-output serial 2's complementer. The circuit accepts a string of bits from the input and generates the 2's complement at the output. The circuit can be reset asynchronously to start and end the operation. (HDL—see Problem 5.39.)

5.18* Design a sequential circuit with two *JK* flip-flops A and B and two inputs E and F. If $E = 0$, the circuit remains in the same state regardless of the value of F. When $E = 1$ and $F = 1$, the circuit goes through the state transitions from 00 to 01, to 10, to 11, back to 00, and repeats. When $E = 1$ and $F = 0$, the circuit goes through the state transitions from 00 to 11, to 10, to 01, back to 00, and repeats. (HDL—see Problem 5.40.)

5.19 A sequential circuit has three flip-flops A, B, and C; one input *x_in*; and one output *y_out*. The state diagram is shown in Fig. P5.19. The circuit is to be designed by treating the unused states as don't-care conditions. Analyze the circuit obtained from the design to determine the effect of the unused states. (HDL—see Problem 5.41.)

(a)* Use D flip-flops in the design.

(b) Use *JK* flip-flops in the design.

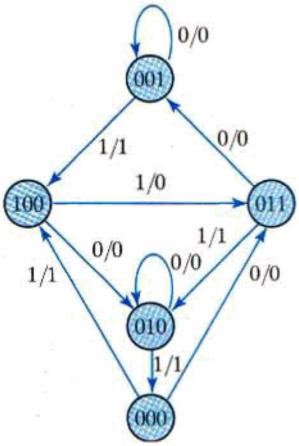


FIGURE P5.19

5.20 Design the sequential circuit specified by the state diagram of Fig. 5.19, using T flip-flops.

5.21 What is the main difference between an **initial** statement and an **always** statement in Verilog HDL?

5.22 Draw the waveform generated by the following statements:

(a)
```
initial begin
   w = 0;  #15 w = 1;  #60 w = 0;  #25 w = 1;  #40 w = 0;
 end
```

(b)
```
initial fork
   w = 0;  #15 w = 1;  #60 w = 0;  #25 w = 1;  #40 w = 0;
 join
```

5.23* Consider the following statements, assuming that *RegA* contains the value of 30 initially:

(a)
```
RegA = 75;
RegB = RegA;
```

(b)
```
RegA <= 75;
RegB <= RegA;
```

What are the values of *RegA* and *RegB* after execution?

5.24 Write and verify an HDL behavioral description of a positive-edge-sensitive *D* flip-flop with

(a) active-low asynchronous preset and clear. (This type of flip-flop is shown in Fig. 11.13.)

(b) active-low synchronous preset and clear.

5.25 A special positive-edge-triggered flip-flop has two inputs *D1* and *D2* and a control input that chooses between the two. Write and verify an HDL behavioral description of this flip-flop.

5.26 Write and verify an HDL behavioral description of the *JK* flip-flop, using an if-else statement based on the value of the present state.

(a)* Consider the characteristic equation when $Q = 0$ or $Q = 1$

(b) Consider how the *J* and *K* inputs affect the output of the flip-flop at each clock tick.

5.27 Rewrite and verify the description of HDL Example 5.5 by combining the state transitions and output into one **always** block.

5.28 Simulate the sequential circuit shown in Fig. 5.17.

(a) Write the HDL description of the state diagram (i.e., a behavioral model).

(b) Write the HDL description of the circuit diagram (i.e., a structural model).

(c) Write an HDL stimulus with the sequence 00, 01, 11, 10 of inputs. Verify that the response is the same for both descriptions.

5.29 Write a behavioral description of the state machine described by the state diagram shown in Fig. P5.19. Write a test bench and verify the functionality of the description.

5.30* Draw the logic diagram for the sequential circuit described by the following HDL module:

```
module Seq_Ckt (input A, B, C, CLK, output reg Q);
 reg E;

 always @ (posedge CLK);
 begin
  E <= A & B;
  Q <= E | C;
 end
endmodule
```

What changes, if any, must be included in the circuit if the last two statements use blocking instead of nonblocking assignment?

5.31* How should the description in Problem 5.30 be written so that the circuit has the same behavior when the assignments are made with = instead of with <= ?

5.32 Using an **initial** statement with a **begin** ... **end** block, write a Verilog description of the waveforms shown in Fig. P5.32. Repeat using a **fork** ... **join** block.

5.33 Explain why it is important that the stimulus signals in a test bench be synchronized to the inactive edge of the clock of the sequential circuit that is to be tested.

5.34 Using behavioral models for the *D* flip-flop and the inverter, write and verify an HDL model of the *J-K* flip-flop described in Problem 5.2.

5.35 Write and verify an HDL model of the sequential circuit described in Problem 5.6.

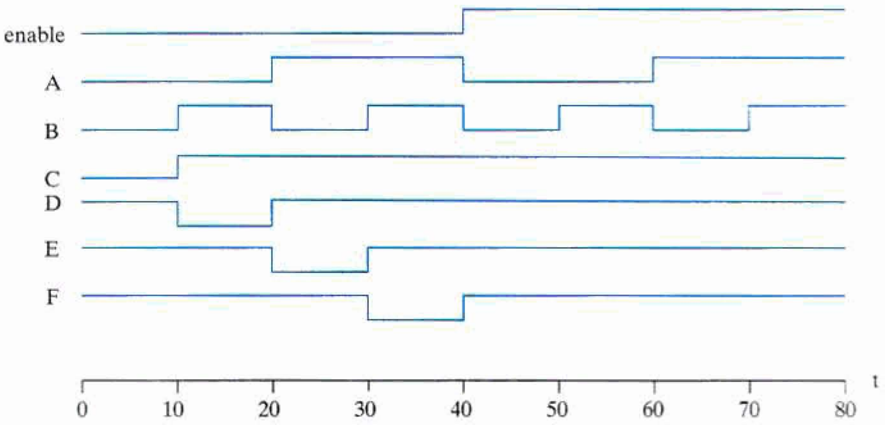


FIGURE P5.32
Waveforms for Problem 5.32

5.36 Write and verify an HDL structural description of the machine having the circuit diagram (schematic) shown in Fig. P5.8.

5.37 Write and verify HDL behavioral descriptions of the state machines shown in Fig. 5.25 and Fig. 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 shown in Figure 5.34.

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

5.45 Write and verify an HDL behavioral description of the sequence detector described in Figure 5.27

REFERENCES

1. Bhasker, J. 1997. *A Verilog HDL Primer.* Allentown, PA: Star Galaxy Press.
2. Bhasker, J. 1998. *Verilog HDL Synthesis.* Allentown, PA: Star Galaxy Press.

3. CILETTI, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL.* Upper Saddle River, NJ: Prentice Hall.
4. DIETMEYER, D. L. 1988. *Logic Design of Digital Systems*, 3d ed. Boston: Allyn Bacon.
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: Prentice Hall.
8. MANO, M. M., and C. R. KIME. 2005. *Logic and Computer Design Fundamentals & Xilinx 6.3 Student Edition*, 3rd 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. 2004. *Fundamentals of Logic Design*, 5th ed. St. Paul, MN: Brooks/Cole.
12. THOMAS, D. E., and P. R. MOORBY, 2002. *The Verilog Hardware Description Language*, 6th ed. Boston: Kluwer Academic Publishers.
13. WAKERLY, J. F. 2006. *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice Hall.

Chapter 6

Registers and Counters

6.1 REGISTERS

A clocked sequential circuit consists of a group of flip-flops and combinational gates connected to form a feedback path. The flip-flops are essential because, in their absence, the circuit reduces to a purely combinational circuit (provided that there is no feedback among the gates). A circuit with flip-flops is considered a sequential circuit even in the absence of combinational gates. Circuits that include flip-flops are usually classified by the function they perform rather than by the name of the sequential circuit. Two such circuits are registers and counters.

A *register* is a group of flip-flops, each one of which is capable of storing one bit of information. An n-bit register consists of a group of n flip-flops capable of storing n bits of binary information. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation. The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

A *counter* is essentially a register that goes through a predetermined sequence of binary states. The gates in the counter are connected in such a way as to produce the prescribed sequence of states. Although counters are a special type of register, it is common to differentiate them by giving them a different name.

Various types of registers are available commercially. The simplest register is one that consists of only flip-flops, without any gates. Figure 6.1 shows such a register constructed with four D-type flip-flops to form a four-bit data storage register. The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are

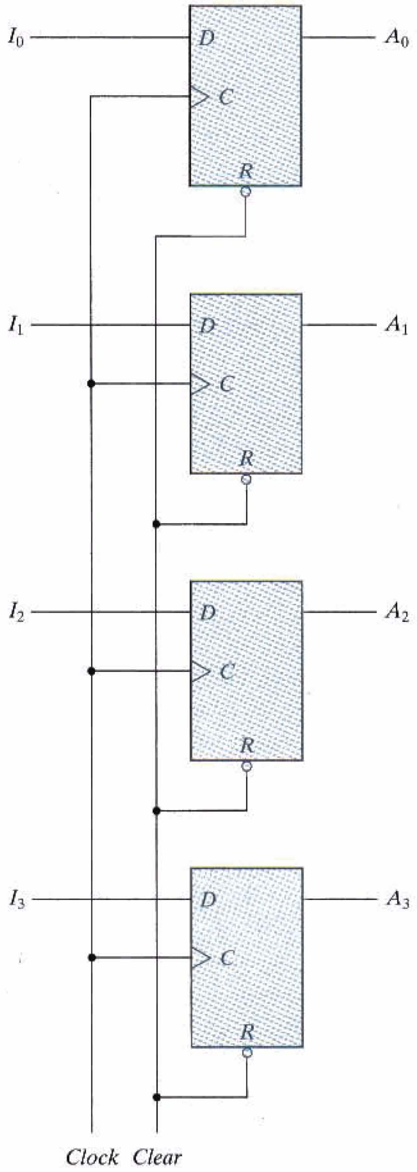


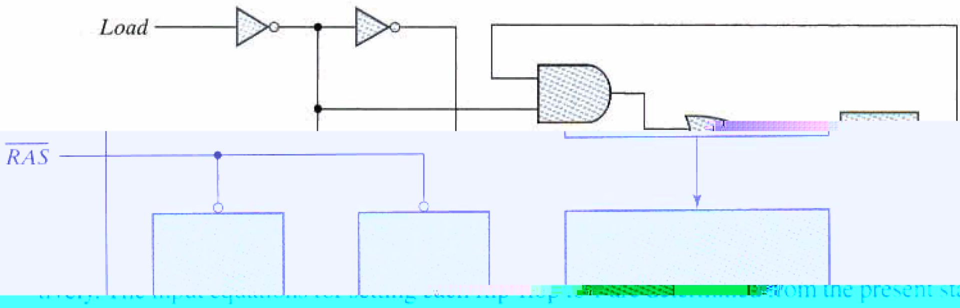
FIGURE 6.1
Four-bit register

transferred into the register. The four 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 at logic 1 during normal clocked operation. Note that, depending on the flip-flop, either *Clear*, *Clear_b*, *reset*, or *reset_b* can be used to indicate the transfer of the register to an all 0's state.

Register with Parallel Load

Synchronous digital systems have a master clock generator that supplies a continuous train of clock pulses. The pulses are applied to all flip-flops and registers in the system. The master clock acts like a drum that supplies a constant beat to all parts of the system. A separate control signal must be used to decide which register operation will execute at each clock pulse. The transfer of new information into a register is referred to as *loading* or *updating* the register. If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done *in parallel*. A clock edge applied to the *C* inputs of the register of Fig. 6.1 will load all four inputs in parallel. In this configuration, if the contents of the register must be left unchanged, the inputs must be held constant or the clock must be inhibited from the circuit. In the first case, the data bus driving the register would be unavailable for other traffic. In the second case, the clock can be inhibited from reaching the register by controlling the clock input signal with an enabling gate. However, inserting gates into the clock path is ill advised because it means that logic is performed with clock pulses. The insertion of logic gates produces uneven propagation delays between the master clock and the inputs of flip-flops. To fully synchronize the system, we must ensure that all clock pulses arrive at the same time anywhere in the system, so that all flip-flops trigger simultaneously. Performing logic with clock pulses inserts variable delays and may cause the system to go out of synchronism. For this reason, it is advisable to control the operation of the register with the *D* inputs, rather than controlling the clock in the *C* inputs of the flip-flops. This creates the effect of a gated clock, but without affecting the clock path of the circuit.

A four-bit data-storage register with a load control input that is directed through gates and into the *D* inputs of the flip-flops is shown in Fig. 6.2. The additional gates implement a two-channel mux whose output drives the input to the register with either the data bus or the output of the register. The load input to the register determines the action to be taken with each clock pulse. When the load input is 1, the data at the four external inputs are transferred into the register with the next positive edge of the clock. When the load input is 0, the outputs of the flip-flops are connected to their respective inputs. The feedback connection from output to input is necessary because a *D* flip-flop does not have a "no change" condition. With each clock edge, the *D* input determines the next state of the register. To leave the output unchanged, it is necessary to make the *D* input equal to the present value of the output (i.e., the output circulates to the input at each clock pulse). The clock pulses are applied to the *C* inputs without interruption. The load input determines whether the next pulse will accept new information or leave the information in the register intact. The transfer of information from the data inputs or the outputs of the register is done simultaneously with all four bits in response to a clock edge.



tively. The input equations for setting each flip-flop to [illegible] from the present state and

going into saturation. This series increases the speed of operation of the circuit without an excessive increase in power dissipation. The low-power Schottky TTL sacrifices some speed for reduced power dissipation. It is equal to the standard TTL in propagation delay, but has only one-fifth the power dissipation. Further innovations led to the development of the advanced Schottky series, which provides an improvement in propagation delay over the Schottky series and also lowers the power dissipation. The advanced low-power Schottky has the lowest speed–power product and is the most efficient series. The fast TTL family is the best choice for high-speed designs.

All TTL series are available in SSI components and in more complex forms, such as MSI and LSI components. The differences in the TTL series are not in the digital logic that they perform, but rather in the internal construction of the basic NAND gate. In any case, TTL gates in all the available series come in three different types of output configuration:

1. Open-collector output
2. Totem-pole output
3. Three-state output

These three types of outputs are considered next, in conjunction with the circuit description of the basic TTL gate.

Open-Collector Output Gate

The basic TTL gate shown in Fig. 10.11 is a modified circuit of the DTL gate. The multiple emitters in transistor *Q1* are connected to the inputs. Most of the time, these emitters behave like the input diodes in the DTL gate, since they form a *pn* junction with their common base. The base–collector junction of *Q1* acts as another *pn* junction diode corresponding to *D1* in the

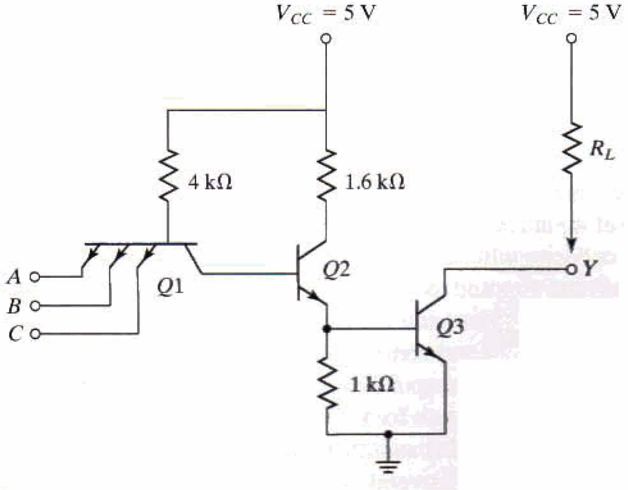


FIGURE 10.11
Open-collector TTL NAND gate

DTL gate. (See Fig. 10.9.) Transistor *Q2* replaces the second diode, *D2*, in the DTL gate. The output of the TTL gate is taken from the open collector of *Q3*. A resistor connected to V_{CC} must be inserted externally to the IC package for the output to "pull up" to the high voltage level when *Q3* is off; otherwise, the output acts as an open circuit. The reason for not providing the resistor internally will be discussed later.

The two voltage levels of the TTL gate are 0.2 V for the low level and from 2.4 to 5 V for the high level. The basic circuit is a NAND gate. If any input is low, the corresponding base–emitter junction in *Q1* is forward biased. The voltage at the base of *Q1* is equal to the input voltage of 0.2 V plus a V_{BE} drop of 0.7, or 0.9 V. In order for *Q3* to start conducting, the path from *Q1* to *Q3* must overcome a potential of one diode drop in the base–collector *pn* junction of *Q1* and two V_{BE} drops in *Q2* and *Q3*, or $3 \times 0.6 = 1.8$ V. Since the base of *Q*1 is maintained at 0.9 V by the input signal, the output transistor cannot conduct and is cut off. The output level will be high if an external resistor is connected between the output and V_{CC} (or an open circuit if a resistor is not used).

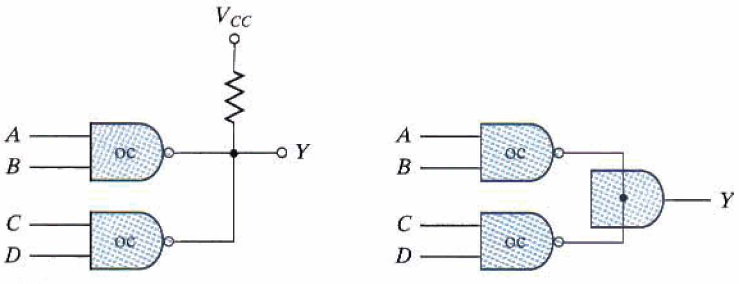
If all inputs are high, both *Q2* and *Q3* conduct and saturate. The base voltage of *Q1* is equal to the voltage across its base–collector *pn* junction plus two V_{BE} drops in *Q2* and *Q3*, or about $0.7 \times 3 = 2.1$ V. Since all inputs are high and greater than 2.4 V, the base–emitter junctions of *Q1* are all reverse biased. When output transistor *Q3* saturates (provided that it has a current path), the output voltage goes low to 0.2 V. This confirms the conditions of a NAND operation.

In the analysis presented thus far, we said that the base–collector junction of *Q1* acts like a *pn* diode junction. This is true in the steady-state condition. However, during the turnoff transition, *Q1* does exhibit transistor action, resulting in a reduction in propagation delay. When all inputs are high and then one of the inputs is brought to a low level, both *Q2* and *Q3* start turning off. At this time, the collector junction of *Q1* is reverse biased and the emitter is forward biased, so transistor *Q1* goes momentarily into the active region. The collector current of *Q1* comes from the base of *Q2* and quickly removes the excess charge stored in *Q2* during its previous saturation state. This causes a reduction in the storage time of the circuit compared with that of the DTL type of input. The result is a reduction in the turnoff time of the gate.

The open-collector TTL gate will operate without the external resistor when connected to inputs of other TTL gates, although this kind of operation is not recommended because of the low noise immunity encountered. Without an external resistor, the output of the gate will be an open circuit when *Q3* is off. An open circuit to an input of a TTL gate behaves as if it has a high-level input (but a small amount of noise can change this to a low level). When *Q3* conducts, its collector will have a current path supplied by the input of the loading gate through V_{CC}, the 4-kΩ resistor, and the forward-biased base–emitter junction.

Open-collector gates are used in three major applications: driving a lamp or relay, performing wired logic, and constructing a common-bus system. An open-collector output can drive a lamp placed in its output through a limiting resistor. When the output is low, the saturated transistor *Q3* forms a path for the current that turns the lamp on. When the output transistor is off, the lamp turns off because there is no path for the current.

If the outputs of several open-collector TTL gates are tied together with a single external resistor, a wired-AND logic is performed. Remember that a positive-logic AND function gives



(a) Physical connection (b) Wired-logic graphic symbol

FIGURE 10.12
Wired-AND of two open-collector (oc) gates, $Y = (AB + CD)'$

a high level only if all variables are high; otherwise, the function is low. With the outputs of open-collector gates connected together, the common output is high only when all output transistors are off (or high). If an output transistor conducts, it forces the output into the low state.

The wired logic performed with open-collector TTL gates is depicted in Fig. 10.12. The physical wiring in (a) shows how the outputs must be connected to a common resistor. The graphic symbol for such a connection is demonstrated in (b). The AND function formed by connecting the two outputs together is called a wired-AND function. The AND gate is drawn with the lines going through the center of the gate, to distinguish it from a conventional gate. The wired-AND gate is not a physical gate, but only a symbol to designate the function obtained from the indicated connection. The Boolean function obtained from the circuit of Fig. 10.12 is the AND operation between the outputs of the two NAND gates:

$$Y = (AB)' \cdot (CD)' = (AB + CD)'$$

The second expression is preferred, since it shows an operation commonly referred to as an AND–OR–INVERT function. (See Section 3.8.)

Open-collector gates can be tied together to form a common bus. At any time, all gate outputs tied to the bus, except one, must be maintained in their high state. The selected gate may be in either the high or low state, depending on whether we want to transmit a 1 or a 0 on the bus. Control circuits must be used to select the particular gate that drives the bus at any given time.

Figure 10.13 demonstrates the connection of four sources tied to a common bus line. Each of the four inputs drives an open-collector inverter, and the outputs of the inverters are tied together to form a single bus line. The figure shows that three of the inputs are 0, producing a 1, or high level, on the bus. The fourth input, I_4, can now transmit information through the common-bus line into inverter 5. Remember that an AND operation is performed in the wired logic. If $I_4 = 1$, the output of gate 4 is 0 and the wired-AND operation produces a 0. If $I_4 = 0$, the output of gate 4 is 1 and the wired-AND operation produces a 1. Thus, if all other outputs are maintained at 1, the selected gate can transmit its value through the bus. The value transmitted is the complement of I_4, but inverter 5 at the receiving end can easily invert this signal again to make $Y = I_4$.

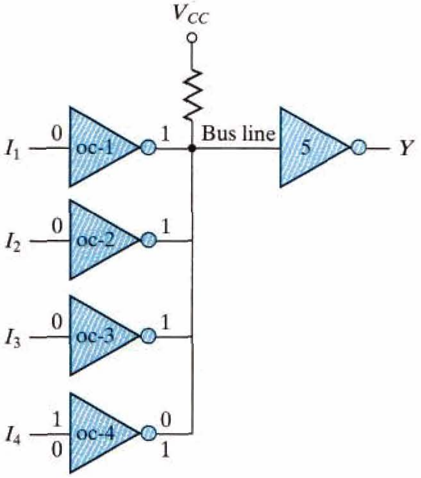


FIGURE 10.13
Open-collector gates forming a common bus line

Totem-Pole Output

The output impedance of a gate is normally a resistive plus a capacitive load. The capacitive load consists of the capacitance of the output transistor, the capacitance of the fan-out gates, and any stray wiring capacitance. When the output changes from the low to the high state, the output transistor of the gate goes from saturation to cutoff and the total load capacitance C charges exponentially from the low to the high voltage level with a time constant equal to RC. For the open-collector gate, R is the external resistor marked R_L. For a typical operating value of $C = 15$ pF and $R_L = 4\ \text{k}\Omega$, the propagation delay of a TTL open-collector gate during the turnoff time is 35 ns. With an *active pull-up* circuit replacing the passive pull-up resistor R_L, the propagation delay is reduced to 10 ns. This configuration, shown in Fig. 10.14, is called a *totem-pole* output because transistor *Q4* "sits" upon *Q3*.

The TTL gate with the totem-pole output is the same as the open-collector gate, except for the output transistor *Q4* and the diode *D1*. When the output *Y* is in the low state, *Q2* and *Q3* are driven into saturation as in the open-collector gate. The voltage in the collector of *Q2* is $V_{BE}(Q3) + V_{CE}(Q2)$, or $0.7 + 0.2 = 0.9$ V. The output $Y = V_{CE}(Q3) = 0.2$ V. Transistor *Q4* is cut off because its base must be one V_{BE} drop plus one diode drop, or $2 \times 0.6 = 1.2$ V, to start conducting. Since the collector of *Q2* is connected to the base of *Q4*, the latter's voltage is only 0.9 V instead of the required 1.2 V, so *Q4* is cut off. The reason for placing the diode in the circuit is to provide a diode drop in the output path and thus ensure that *Q4* is cut off when *Q3* is saturated.

When the output changes to the high state because one of the inputs drops to the low state, transistors *Q2* and *Q3* go into cutoff. However, the output remains momentarily low because the voltages across the load capacitance cannot change instantaneously. As soon as *Q2* turns off, *Q4* conducts, because its base is connected to V_{CC} through the 1.6-kΩ resistor. The current needed to charge the load capacitance causes *Q4* to saturate momentarily, and the output

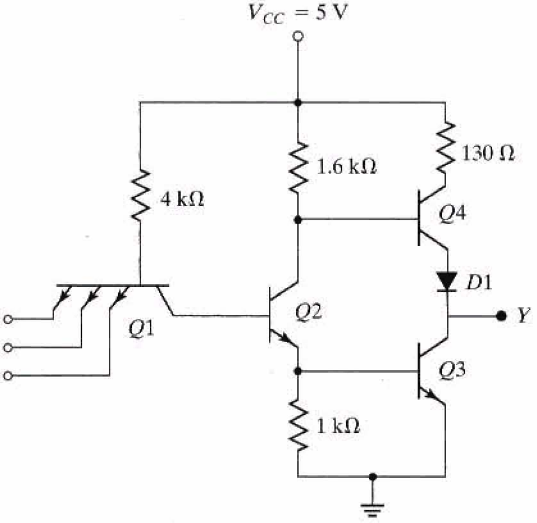


FIGURE 10.14
TTL gate with totem-pole output

voltage rises with a time constant RC. But R in this case is equal to 130 Ω, plus the saturation resistance of *Q4*, plus the resistance of the diode, for a total of approximately 150 Ω. This value of R is much smaller than the passive pull-up resistance used in the open-collector circuit. As a consequence, the transition from the low to high level is much faster.

As the capacitive load charges, the output voltage rises and the current in *Q4* decreases, bringing the transistor into the active region. Thus, in contrast to the other transistors, *Q4* is in the *active* region when Q4 is in a steady-state condition. The final value of the output voltage is then 5 V, minus a V_{BE} drop in *Q4*, minus a diode drop in *D1* to about 3.6 V. Transistor *Q3* goes into cutoff very fast, but during the initial transition time, both *Q3* and *Q4* are on and a peak current is drawn from the power supply. This current spike generates noise in the power-supply distribution system. When the change of state is frequent, the transient-current spikes increase the power-supply current requirement and the average power dissipation of the circuit increases.

The wired-logic connection is not allowed with totem-pole output circuits. When two totem poles are wired together, with the output of one gate high and the output of the second gate low, the excessive amount of current drawn can produce enough heat to damage the transistors in the circuit. (See Problem 10.7.) Some TTL gates are constructed to withstand the amount of current that flows under this condition. In any case, the collector current in the low gate may be high enough to move the transistor into the active region and produce an output voltage greater than 0.8 V in the wired connection. This voltage is not a valid binary signal for TTL gates.

Schottky TTL Gate

As mentioned before, a reduction in storage time results in a reduction in propagation delay. This is because the time needed for a transistor to come out of saturation delays the switching of the transistor from the on condition to the off condition. Saturation can be eliminated by placing a Schottky diode between the base and collector of each saturated transistor in the circuit. The Schottky diode is formed by the junction of a metal and semiconductor, in contrast to a conventional diode, which is formed by the junction of *p*-type and *n*-type semiconductor material. The voltage across a conducting Schottky diode is only 0.4 V, compared with 0.7 V in a conventional diode. The presence of a Schottky diode between the base and collector prevents the transistor from going into saturation. The resulting transistor is called a *Schottky transistor.* The use of Schottky transistors in a TTL decreases the propagation delay without sacrificing power dissipation.

The Schottky TTL gate is shown in Fig. 10.15. Note the special symbol used for the Schottky transistors and diodes. The diagram shows all transistors except *Q4* to be of the Schottky type. An exception is made for *Q4*, since it does not saturate, but stays in the active region. Note also that resistor values have been reduced in order to decrease the propagation delay further.

In addition to using Schottky transistors and lower resistor values, the circuit of Fig. 10.15 includes other modifications not available in the standard gate of Fig. 10.14. Two new transistors, *Q5* and *Q6*, have been added, and Schottky diodes are inserted between each input terminal and ground. There is no diode in the totem-pole circuit. However, the new combination of *Q5* and *Q4* still gives the two V_{BE} drops necessary to prevent *Q4* from conducting when the output is low. This combination constitutes a double emitter–follower called a *Darlington pair.* The Darlington

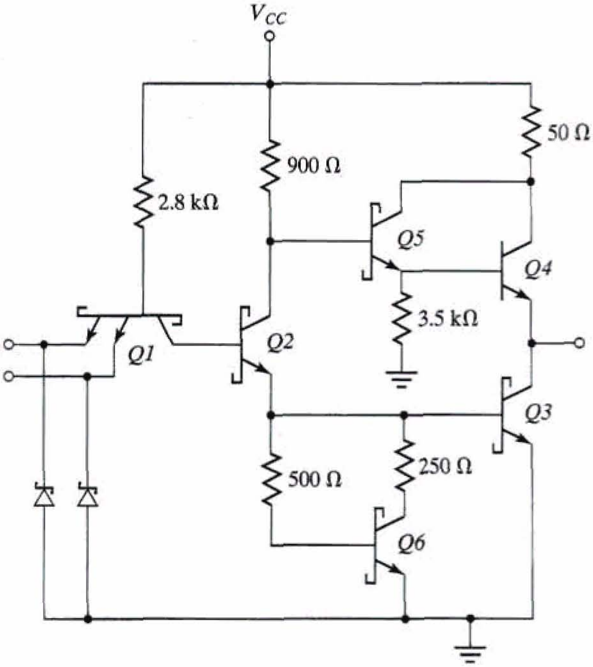


FIGURE 10.15
Schottky TTL gate

pair provides a very high current gain and extremely low resistance, exactly what is needed during the low-to-high swing of the output, resulting in a decrease in propagation delay.

The diodes in each input shown in the circuit help clamp any ringing that may occur in the input lines. Under transient switching conditions, signal lines appear inductive; this, along with stray capacitance, causes signals to oscillate, or "ring." When the output of a gate switches from the high to the low state, the ringing waveform at the input may have excursions as great as 2–3 V below ground, depending on the line length. The diodes connected to ground help clamp this ringing, since they conduct as soon as the negative voltage exceeds 0.4 V. When the negative excursion is limited, the positive swing is also reduced. Clamp diodes have been so successful in limiting line effects that all versions of TTL gates use them.

The emitter resistor *Q2* in Fig. 10.14 has been replaced in Fig. 10.15 by a circuit consisting of transistor *Q6* and two resistors. The effect of this circuit is to reduce the turnoff current spikes discussed previously. The analysis of such a circuit, whose operation helps to reduce the propagation time of the gate, is too involved to present in this brief discussion.

Three-State Gate

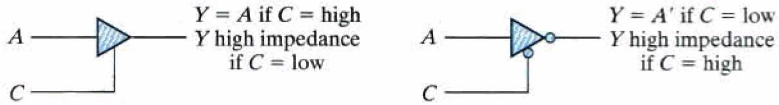
As mentioned earlier, the outputs of two TTL gates with totem-pole structures cannot be connected together as in open-collector outputs. There is, however, a special type of totem-pole gate that allows the wired connection of outputs for the purpose of forming a common-bus system. When a totem-pole output TTL gate has this property, it is called a *three-state* gate.

A three-state gate exhibits three output states: (1) a low-level state when the lower transistor in the totem pole is on and the upper transistor is off, (2) a high-level state when the upper transistor in the totem pole is on and the lower transistor is off, and (3) a third state when both transistors in the totem pole are off. The third state is an open-circuit, or high-impedance, state that allows a direct wire connection of many outputs to a common line. Three-state gates eliminate the need for open-collector gates in bus configurations.

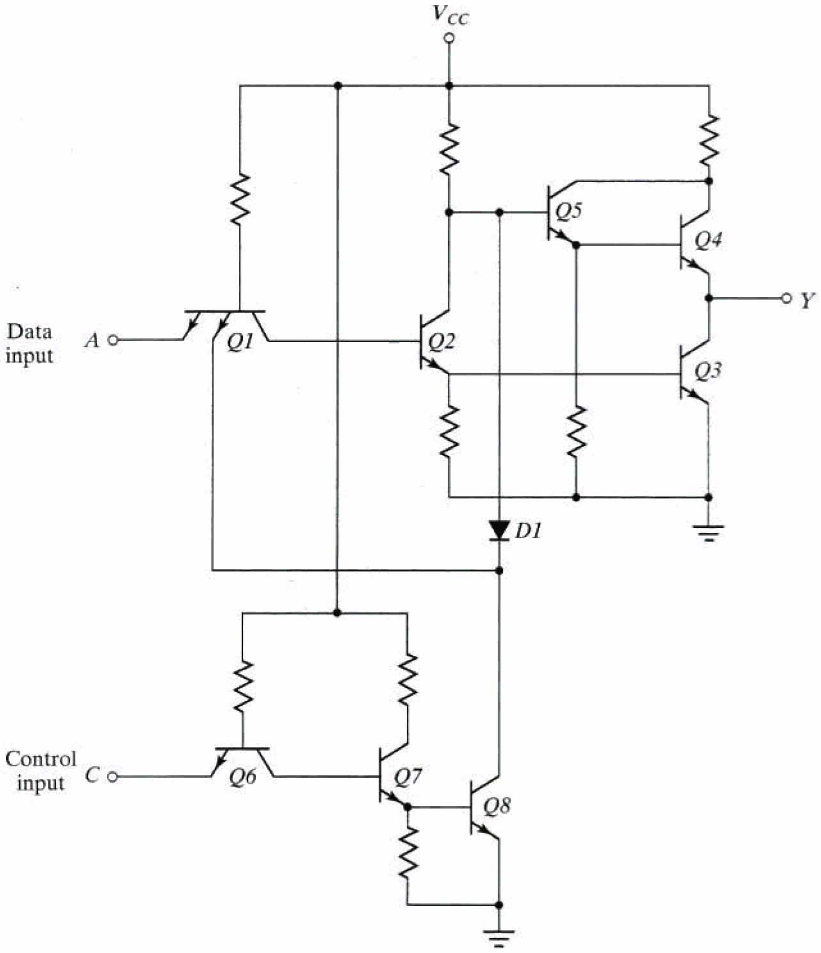
Figure 10.16(a) shows the graphic symbol of a three-state buffer gate. When the control input C is high, the gate is enabled and behaves like a normal buffer, with the output equal to the input binary value. When the control input is low, the output is an open circuit, which gives a high impedance (the third state) regardless of the value of input A. Some three-state gates produce a high-impedance state when the control input is high. This is shown symbolically in Fig. 10.16(b), where we have two small circles, one for the inverter output and the other to indicate that the gate is enabled when C is low.

The circuit diagram of the three-state inverter is shown in Fig. 10.16(c). Transistors *Q6*, *Q7*, and *Q8* associated with the control input form a circuit similar to the open-collector gate. Transistors *Q1*–*Q5*, associated with the data input, form a totem-pole TTL circuit. The two circuits are connected together through diode *D1*. As in an open-collector circuit, transistor *Q8* turns off when the control input at C is in the low-level state. This prevents diode *D1* from conducting. In addition, the emitter in *Q1* connected to *Q8* has no conduction path. Under this condition, transistor *Q8* has no effect on the operation of the gate and the output in Y depends only on the data input at A.

When the control input is high, transistor *Q8* turns on and the current flowing from V_{CC} through diode *D1* causes transistor *Q8* to saturate. The voltage at the base of *Q5* is now equal



(a) Three-state buffer gate (b) Three-state inverter gate



(c) Circuit diagram for the three-state inverter of (b)

FIGURE 10.16
Three-state TTL gate

to the voltage across the saturated transistor, *Q8*, plus one diode drop, or 0.9 V. This voltage turns off *Q5* and *Q4*, since it is less than two V_{BE} drops. At the same time, the low input to one of the emitters of *Q1* forces transistor *Q3* (and *Q2*) to turn off. Thus, both *Q3* and *Q4* in the totem pole are turned off, and the output of the circuit behaves like an open circuit with a very high output impedance.

A three-state bus is created by wiring several three-state outputs together. At any given time, only one control input is enabled while all other outputs are in the high-impedance state. The single gate not in a high-impedance state can transmit binary information through the common bus. Extreme care must be taken that all except one of the outputs be in the third state; otherwise, we have the undesirable condition of having two active totem-pole outputs connected together.

An important feature of most three-state gates is that the output enable delay is longer than the output disable delay. If a control circuit enables one gate and disables another at the same time, the disabled gate enters the high-impedance state before the other gate is enabled. This eliminates the situation of both gates being active at the same time.

There is a very small leakage current associated with the high-impedance condition in a three-state gate. Nevertheless, this current is so small that as many as 100 three-state outputs can be connected together to form a common-bus line.

10.6 EMITTER-COUPLED LOGIC

Emitter-coupled logic (ECL) is a nonsaturated digital logic family. Since transistors do not saturate, it is possible to achieve propagation delays as low as 1–2 ns. This logic family has the lowest propagation delay of any family and is used mostly in systems requiring very high speed operation. Its noise immunity and power dissipation, however, are the worst of all the logic families available.

A typical basic circuit of the ECL family is shown in Fig. 10.17. The outputs provide both the OR and NOR functions. Each input is connected to the base of a transistor. The two voltage levels are about -0.8 V for the high state and about -1.8 V for the low state. The circuit consists of a differential amplifier, a temperature- and voltage-compensated bias network, and an emitter–follower output. The emitter outputs require a pull-down resistor for current to flow. This is obtained from the input resistor R_P of another similar gate or from an external resistor connected to a negative voltage supply.

The internal temperature- and voltage-compensated bias circuit supplies a reference voltage to the differential amplifier. Bias voltage V_{BB} is set at -1.3 V, which is the midpoint of the signal's logic swing. The diodes in the voltage divider, together with *Q6*, provide a circuit that maintains a constant V_{BB} value despite changes in temperature or supply voltage. Any one of the power supply inputs could be used as ground. However, the use of the V_{CC} node as ground and V_{EE} at -5.2 V results in the best noise immunity.

If any input in the ECL gate is high, the corresponding transistor is turned on and *Q5* is turned off. An input of -0.8 V causes the transistor to conduct and places -1.6 V on the emitters of all of the transistors. (The V_{BE} drop in ECL transistors is 0.8 V.) Since $V_{BB} = -1.3$ V, the base voltage of *Q5* is only 0.3 V more positive than its emitter. *Q5* is cut off because its V_{BE}

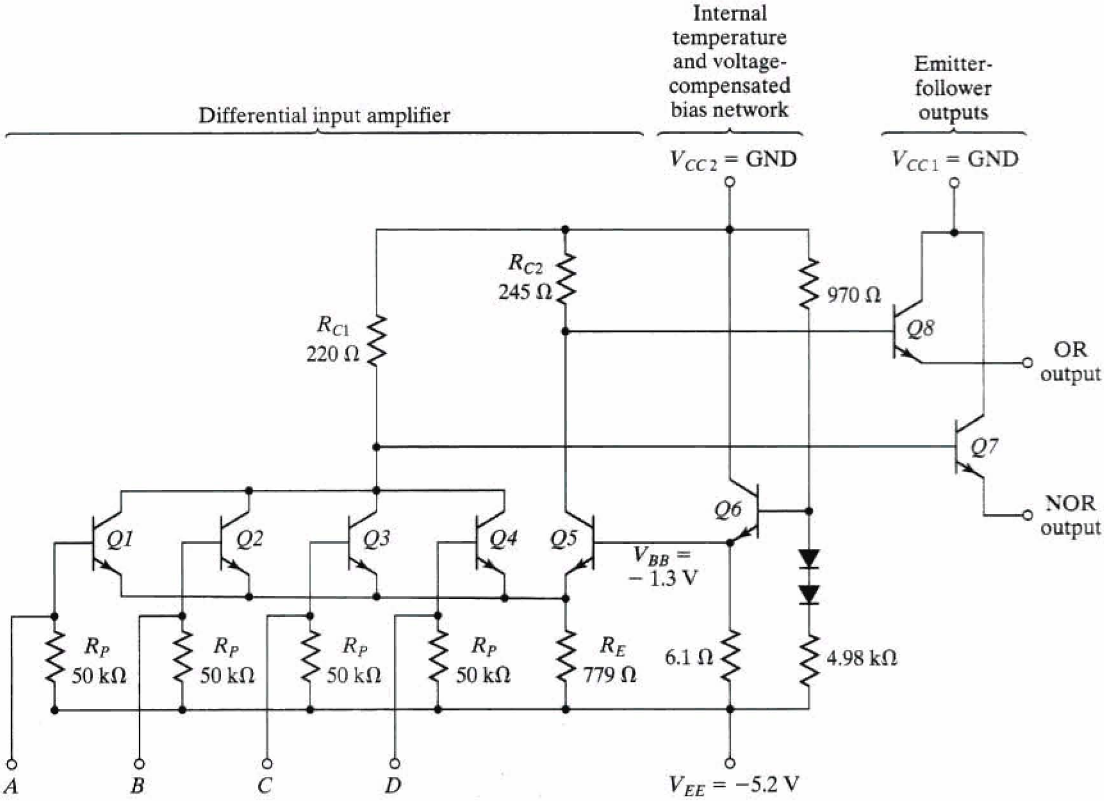
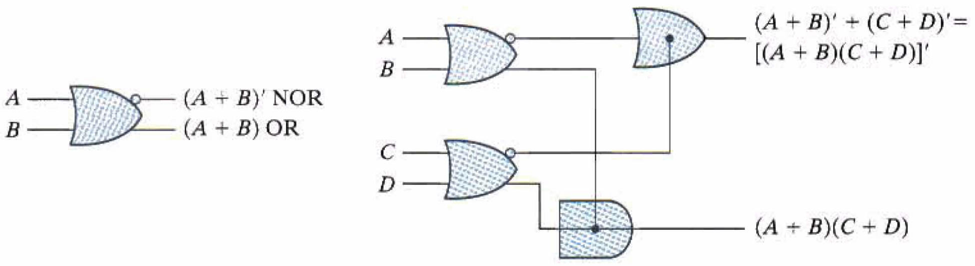


FIGURE 10.17
Basic ECL gate

voltage needs at least 0.6 V to start conducting. The current in resistor R_{C2} flows into the base of *Q8* (provided that there is a load resistor). This current is so small that only a negligible voltage drop occurs across R_{C2}. The OR output of the gate is one V_{BE} drop below ground, or −0.8 V, which is the high state. The current flowing through R_{C1} and the conducting transistor causes a drop of about 1 V below ground. (See Problem 10.9.) The NOR output is one V_{BE} drop below this level, or −1.8 V, which is the low state.

If all inputs are at the low level, all input transistors turn off and *Q5* conducts. The voltage in the common-emitter node is one V_{BE} drop below V_{BB}, or −2.1 V. Since the base of each input is at a low level of −1.8 V, each base–emitter junction has only 0.3 V and all input transistors are cut off. R_{C2} draws current through *Q5* that results in a voltage drop of about 1 V, making the OR output one V_{BE} drop below this, at −1.8 V, or the low level. The current in R_{C1} is negligible, and the NOR output is one V_{BE} drop below ground, at −0.8 V, or the high level. This analysis verifies the OR and NOR operations of the circuit.

The propagation delay of the ECL gate is 2 ns and the power dissipation is 25 mW, giving a speed–power product of 50, which is about the same as that for the Schottky TTL. The noise



(a) Single gate

(b) Wired combination of two gates

FIGURE 10.18
Graphic symbol and wired combination of ECL gates

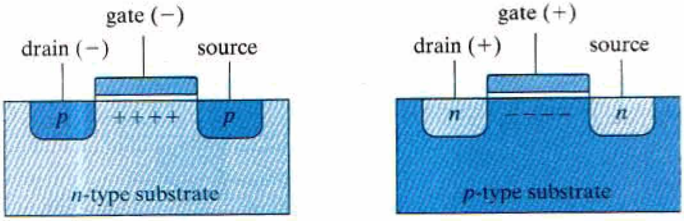
margin is about 0.3 V and is not as good as that in the TTL gate. High fan-out is possible in the ECL gate because of the high input impedance of the differential amplifier and the low output impedance of the emitter–follower. Because of the extreme high speed of the signals, external wires act like transmission lines. Except for very short wires of a few centimeters, ECL outputs must use coaxial cables with a resistor termination to reduce line reflections.

The graphic symbol for the ECL gate shown in Fig. 10.18(a). Two outputs are available: one for the NOR function and the other for the OR function. The outputs of two or more ECL gates can be connected together to form wired logic. As shown in Fig. 10.18(b), an *external* wired connection of two NOR outputs produces a wired-OR function. An *internal* wired connection of two OR outputs is employed in some ECL ICs to produce a wired-AND (sometimes called dot-AND) logic. This property may be utilized when ECL gates are used to form the OR–AND–INVERT and the OR–AND functions.

10.7 METAL-OXIDE SEMICONDUCTOR

The field-effect transistor (FET) is a unipolar transistor, since its operation depends on the flow of only one type of carrier. There are two types of FETs: the junction field-effect transistor (JFET) and the metal-oxide semiconductor (MOS). The former is used in linear circuits and the latter in digital circuits. MOS transistors can be fabricated in less area than bipolar transistors.

The basic structure of the MOS transistor is shown in Fig. 10.19. The *p*-channel MOS consists of a lightly doped substrate of *n*-type silicon material. Two regions are heavily doped by diffusion with *p*-type impurities to form the *source* and *drain*. The region between the two *p*-type sections serves as the *channel*. The *gate* is a metal plate separated from the channel by an insulated dielectric of silicon dioxide. A negative voltage (with respect to the substrate) at the gate terminal causes an induced electric field in the channel that attracts *p*-type carriers (holes) from the substrate. As the magnitude of the negative voltage on the gate increases, the region below the gate accumulates more positive carriers, the conductivity increases, and current can flow from source to drain, provided that a voltage difference is maintained between these two terminals.



(a) p-channel (b) n-channel

FIGURE 10.19
Basic structure of MOS transistor

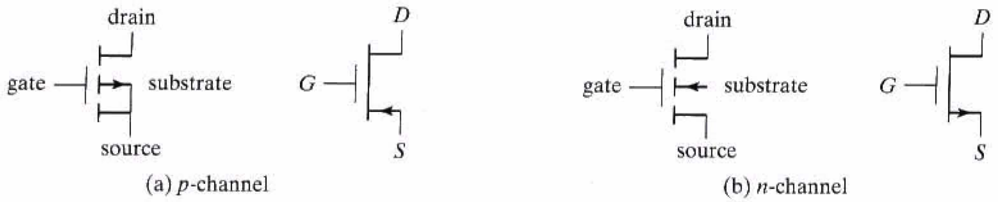
There are four basic types of MOS structures. The channel can be p or n type, depending on whether the majority carriers are holes or electrons. The mode of operation can be enhancement or depletion, depending on the state of the channel region at zero gate voltage. If the channel is initially doped lightly with p-type impurity (in which case it is called a *diffused channel*), a conducting channel exists at zero gate voltage and the device is said to operate in the *depletion* mode. In this mode, current flows unless the channel is depleted by an applied gate field. If the region beneath the gate is left initially uncharged, a channel must be induced by the gate field before current can flow. Thus, the channel current is enhanced by the gate voltage, and such a device is said to operate in the *enhancement* mode.

The source is the terminal through which the majority carriers enter the device. The drain is the terminal through which the majority carriers leave the device. In a p-channel MOS, the source terminal is connected to the substrate and a negative voltage is applied to the drain terminal. When the gate voltage is above a threshold voltage V_T (about -2 V), no current flows in the channel and the drain-to-source path is like an open circuit. When the gate voltage is sufficiently negative below V_T, a channel is formed and p-type carriers flow from source to drain. p-type carriers are positive and correspond to a positive current flow from source to drain.

In the n-channel MOS, the source terminal is connected to the substrate and a positive voltage is applied to the drain terminal. When the gate voltage is below the threshold voltage V_T (about 2 V), no current flows in the channel. When the gate voltage is sufficiently positive above V_T to form the channel, n-type carriers flow from source to drain. n-type carriers are negative and correspond to a positive current flow from drain to source. The threshold voltage may vary from 1 to 4 V, depending on the particular process used.

The graphic symbols for the MOS transistors are shown in Fig. 10.20. The symbol for the enhancement type is the one with the broken-line connection between source and drain. In this symbol, the substrate can be identified and is shown connected to the source. An alternative symbol omits the substrate, and instead an arrow is placed in the source terminal to show the direction of *positive* current flow (from source to drain in the p-channel MOS and from drain to source in the n-channel MOS).

Because of the symmetrical construction of source and drain, the MOS transistor can be operated as a bilateral device. Although normally operated so that carriers flow from source to drain, there are circumstances when it is convenient to allow carriers to flow from drain to source. (See Problem 10.12.)



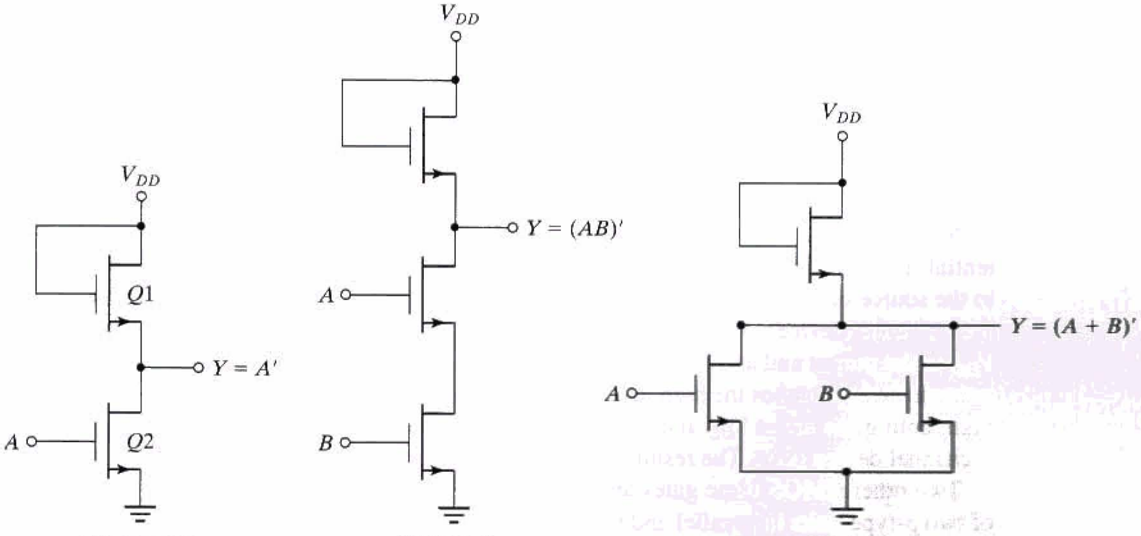
(a) p-channel

(b) n-channel

FIGURE 10.20
Symbols for MOS transistors

One advantage of the MOS device is that it can be used not only as a transistor, but as a resistor as well. A resistor is obtained from the MOS by permanently biasing the gate terminal for conduction. The ratio of the source–drain voltage to the channel current then determines the value of the resistance. Different resistor values may be constructed during manufacturing by fixing the channel length and width of the MOS device.

Three logic circuits using MOS devices are shown in Fig. 10.21. For an n-channel MOS, the supply voltage V_{DD} is positive (about 5 V), to allow positive current flow from drain to source. The two voltage levels are a function of the threshold voltage V_T. The low level is anywhere from zero to V_T, and the high level ranges from V_T to V_{DD}. The n-channel gates usually employ positive logic. The p-channel MOS circuits use a negative voltage for V_{DD}, to allow positive current flow from source to drain. The two voltage levels are both negative above and below the negative threshold voltage V_T. p-channel gates usually employ negative logic.



(a) Inverter

(b) NAND gate

(c) NOR gate

FIGURE 10.21
n-channel MOS logic circuits

The inverter circuit shown in Fig. 10.21(a) uses two MOS devices. *Q1* acts as the load resistor and *Q2* as the active device. The load-resistor MOS has its gate connected to V_{DD}, thus maintaining it in the conduction state. When the input voltage is low (below V_T), *Q*2 turns off. Since *Q1* is always on, the output voltage is about V_{DD}. When the input voltage is high (above V_T), *Q2* turns on. Current flows from V_{DD} through the load resistor *Q1* and into *Q2*. The geometry of the two MOS devices must be such that the resistance of *Q2*, when conducting, is much less than the resistance of *Q1* to maintain the output *Y* at a voltage below V_T.

The NAND gate shown in Fig. 10.21(b) uses transistors in series. Inputs *A* and *B* must both be high for all transistors to conduct and cause the output to go low. If either input is low, the corresponding transistor is turned off and the output is high. Again, the series resistance formed by the two active MOS devices must be much less than the resistance of the load-resistor MOS. The NOR gate shown in Fig. 10.21(c) uses transistors in parallel. If either input is high, the corresponding transistor conducts and the output is low. If all inputs are low, all active transistors are off and the output is high.

10.8 COMPLEMENTARY MOS

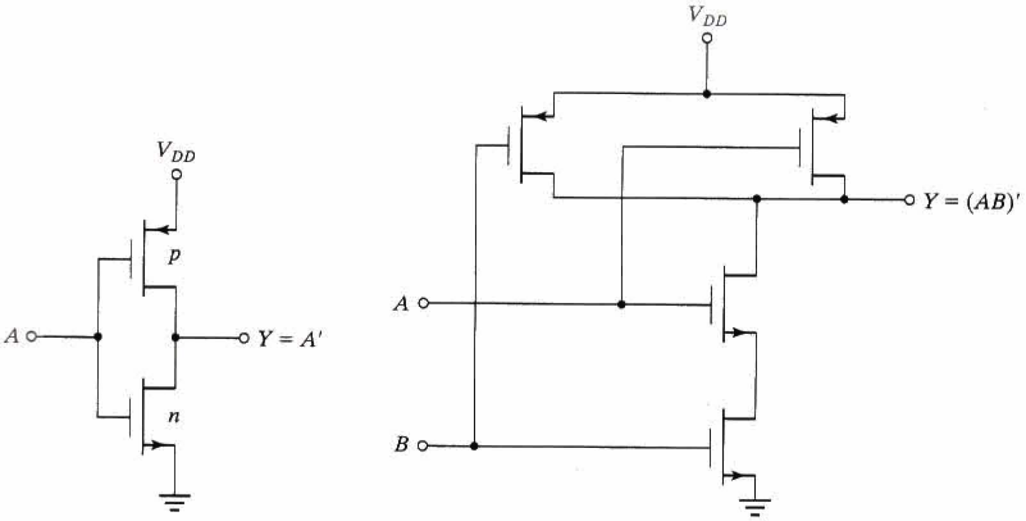
Complementary MOS (CMOS) circuits take advantage of the fact that both *n*-channel and *p*-channel devices can be fabricated on the same substrate. CMOS circuits consist of both types of MOS devices, interconnected to form logic functions. The basic circuit is the inverter, which consists of one *p*-channel transistor and one *n*-channel transistor, as shown in Fig. 10.22(a). The source terminal of the *p*-channel device is at V_{DD}, and the source terminal of the *n*-channel device is at ground. The value of V_{DD} may be anywhere from $+3$ to $+18$ V. The two voltage levels are 0 V for the low level and V_{DD} for the high level (typically, 5 V).

To understand the operation of the inverter, we must review the behavior of the MOS transistor from the previous section:

1. The *n*-channel MOS conducts when its gate-to-source voltage is positive.
2. The *p*-channel MOS conducts when its gate-to-source voltage is negative.
3. Either type of device is turned off if its gate-to-source voltage is zero.

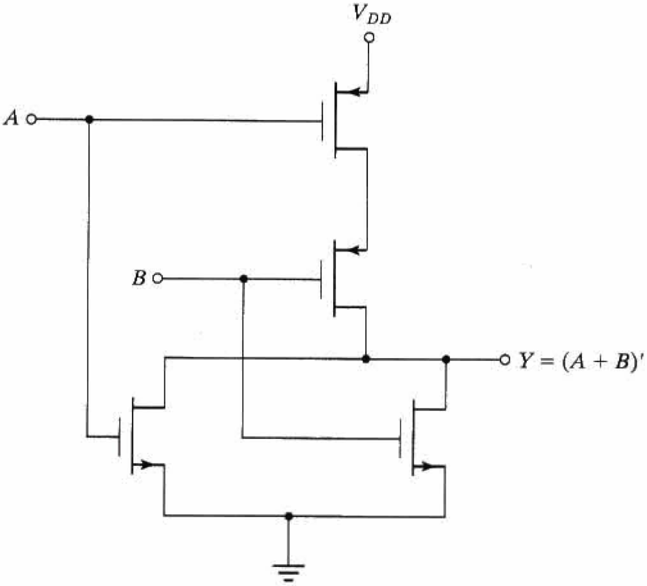
Now consider the operation of the inverter. When the input is low, both gates are at zero potential. The input is at $-V_{DD}$ relative to the source of the *p*-channel device and at 0 V relative to the source of the *n*-channel device. The result is that the *p*-channel device is turned on and the *n*-channel device is turned off. Under these conditions, there is a low-impedance path from V_{DD} to the output and a very high impedance path from output to ground. Therefore, the output voltage approaches the high level V_{DD} under normal loading conditions. When the input is high, both gates are at V_{DD} and the situation is reversed: The *p*-channel device is off and the *n*-channel device is on. The result is that the output approaches the low level of 0 V.

Two other CMOS basic gates are shown in Fig. 10.22. A two-input NAND gate consists of two *p*-type units in parallel and two *n*-type units in series, as shown in Fig. 10.22(b). If all inputs are high, both *p*-channel transistors turn off and both *n*-channel transistors turn on. The output has a low impedance to ground and produces a low state. If any input is low, the associated *n*-channel transistor is turned off and the associated *p*-channel transistor is turned



(a) Inverter

(b) NAND gate



(c) NOR gate

FIGURE 10.22
CMOS logic circuits

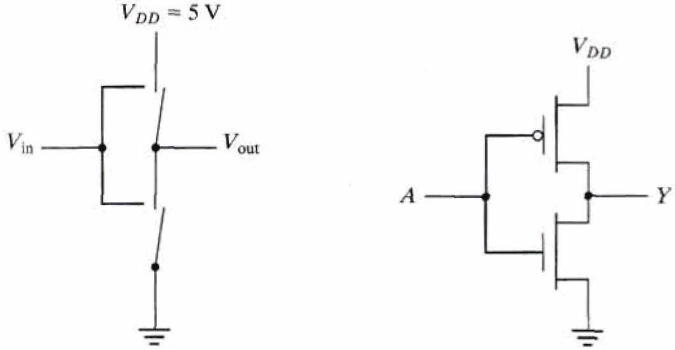
on. The output is coupled to V_{DD} and goes to the high state. Multiple-input NAND gates may be formed by placing equal numbers of *p*-type and *n*-type transistors in parallel and series, respectively, in an arrangement similar to that shown in Fig. 10.22(b).

A two-input NOR gate consists of two *n*-type units in parallel and two *p*-type units in series, as shown in Fig. 10.22(c). When all inputs are low, both *p*-channel units are on and both *n*-channel units are off. The output is coupled to V_{DD} and goes to the high state. If any input is high, the associated *p*-channel transistor is turned off and the associated *n*-channel transistor turns on, connecting the output to ground and causing a low-level output.

MOS transistors can be considered to be electronic switches that either conduct or are open. As an example, the CMOS inverter can be visualized as consisting of two switches as shown in Fig. 10.23(a). Applying a low voltage to the input causes the upper switch (p) to close, supplying a high voltage to the output. Applying a high voltage to the input causes the lower switch (n) to close, connecting the output to ground. Thus, the output V_{out} is the complement of the input V_{in}. Commercial applications often use other graphic symbols for MOS transistors to emphasize the logical behavior of the switches. The arrows showing the direction of current flow are omitted. Instead, the gate input of the *p*-channel transistor is drawn with an inversion bubble on the gate terminal to show that it is enabled with a low voltage. The inverter circuit is redrawn with these symbols in Fig. 10.23(b). A logic 0 in the input causes the upper transistor to conduct, making the output logic 1. A logic 1 in the input enables the lower transistor, making the output logic 0.

CMOS Characteristics

When a CMOS logic circuit is in a static state, its power dissipation is very low. This is because at least one transistor is always off in the path between the power supply and ground when the state of the circuit is not changing. As a result, a typical CMOS gate has static power dissipation on the order of 0.01 mW. However, when the circuit is changing state at the rate of 1 MHz, the power dissipation increases to about 1 mW, and at 10 MHz it is about 5 mW.



(a) Switch model

(b) Logical model

FIGURE 10.23
CMOS inverter

CMOS logic is usually specified for a single power-supply operation over a voltage range from 3 to 18 V with a typical V_{DD} value of 5 V. Operating CMOS at a larger power-supply voltage reduces the propagation delay time and improves the noise margin, but the power dissipation is increased. The propagation delay time with $V_{DD} = 5$ V ranges from 5 to 20 ns, depending on the type of CMOS used. The noise margin is usually about 40 percent of the power supply voltage. The fan-out of CMOS gates is about 30 when they are operated at a frequency of 1 MHz. The fan-out decreases with an increase in the frequency of operation of the gates.

There are several series of the CMOS digital logic family. The 74C series are pin and function compatible with TTL devices having the same number. For example, CMOS IC type 74C04 has six inverters with the same pin configuration as TTL type 7404. The high-speed CMOS 74HC series is an improvement over the 74C series, with a tenfold increase in switching speed. The 74HCT series is electrically compatible with TTL ICs. This means that circuits in this series can be connected to inputs and outputs of TTL ICs without the need of additional interfacing circuits. Newer versions of CMOS are the high-speed series 74VHC and its TTL-compatible version 74VHCT.

The CMOS fabrication process is simpler than that of TTL and provides a greater packing density. Thus, more circuits can be placed on a given area of silicon at a reduced cost per function. This property, together with the low power dissipation of CMOS circuits, good noise immunity, and reasonable propagation delay, makes CMOS the most popular standard as a digital logic family.

10.9 CMOS TRANSMISSION GATE CIRCUITS

A special CMOS circuit that is not available in the other digital logic families is the *transmission gate*. The transmission gate is essentially an electronic switch that is controlled by an input logic level. It is used to simplify the construction of various digital components when fabricated with CMOS technology.

Figure 10.24(a) shows the basic circuit of the transmission gate. Whereas a CMOS inverter consists of a p-channel transistor connected in series with an n-channel transistor, a transmission gate is formed by one n-channel and one p-channel MOS transistor connected in parallel.

The n-channel substrate is connected to ground and the p-channel substrate is connected to V_{DD}. When the N gate is at V_{DD} and the P gate is at ground, both transistors conduct and there is a closed path between input X and output Y. When the N gate is at ground and the P gate is at V_{DD}, both transistors are off and there is an open circuit between X and Y. Figure 10.24(b) shows the block diagram of the transmission gate. Note that the terminal of the p-channel gate is marked with the negation symbol. Figure 10.24(c) demonstrates the behavior of the switch in terms of positive-logic assignment with V_{DD} equivalent to logic 1 and ground equivalent to logic 0.

The transmission gate is usually connected to an inverter, as shown in Fig. 10.25. This type of arrangement is referred to as a *bilateral switch*. The control input C is connected directly to the n-channel gate and its inverse to the p-channel gate. When $C = 1$, the switch is

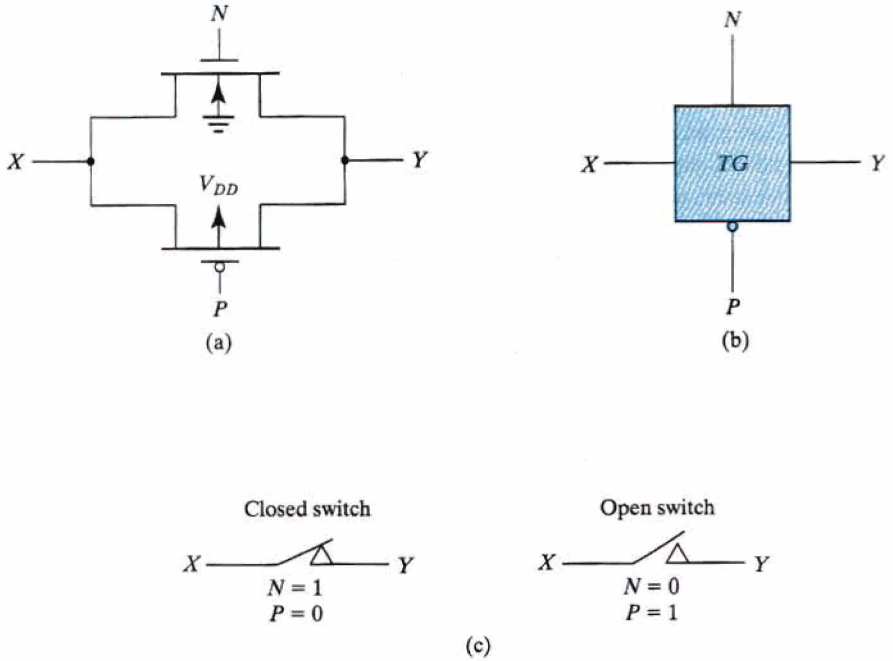


FIGURE 10.24
Transmission gate (TG)

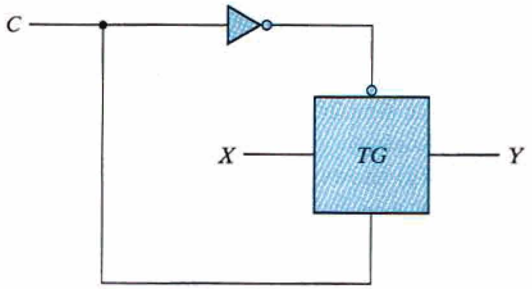
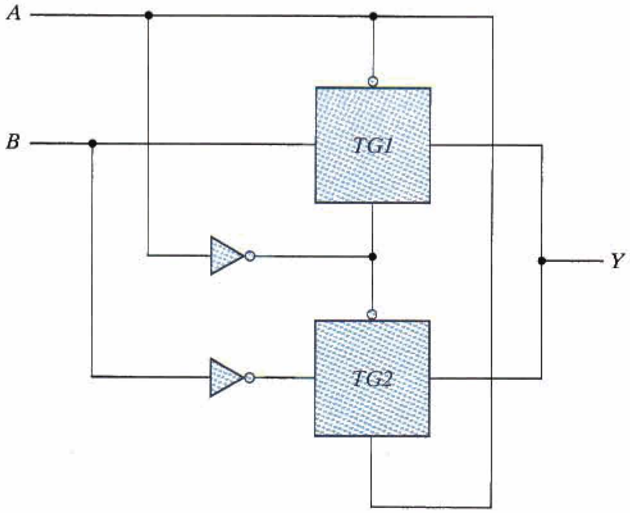


FIGURE 10.25
Bilateral switch

closed, producing a path between X and Y. When $C = 0$, the switch is open, disconnecting the path between X and Y.

Various circuits can be constructed that use the transmission gate. To demonstrate its usefulness as a component in the CMOS family, we will show three examples.

The exclusive-OR gate can be constructed with two transmission gates and two inverters, as shown in Fig. 10.26. Input A controls the paths in the transmission gates and input B is



A	*B*	*TG1*	*TG2*	*Y*
0	0	close	open	0
0	1	close	open	1
1	1	open	close	1
1	0	open	close	0

FIGURE 10.26
Exclusive-OR constructed with transmission gates

connected to output Y through the gates. When input A is equal to 0, transmission gate *TG1* is closed and output Y is equal to input B. When input A is equal to 1, *TG2* is closed and output Y is equal to the complement of input B. This results in the exclusive-OR truth table, as indicated in Fig. 10.26.

Another circuit that can be constructed with transmission gates is the multiplexer. A four-to-one-line multiplexer implemented with transmission gates is shown in Fig. 10.27. The *TG* circuit provides a transmission path between its horizontal input and 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.

The level-sensitive D flip-flop commonly referred to as the gated D latch can be constructed with transmission gates, as shown in Fig. 10.28. The C input controls two transmission gates *TG*. When $C = 1$, the *TG* connected to input D has a closed path and the one connected to output Q has an open path. This configuration produces an equivalent circuit from input D through two inverters to output Q. Thus, the output follows the data input as long as C remains active. When C switches to 0, the first *TG* disconnects input D from the circuit and the second *TG* produces a closed path between the two inverters at the output.

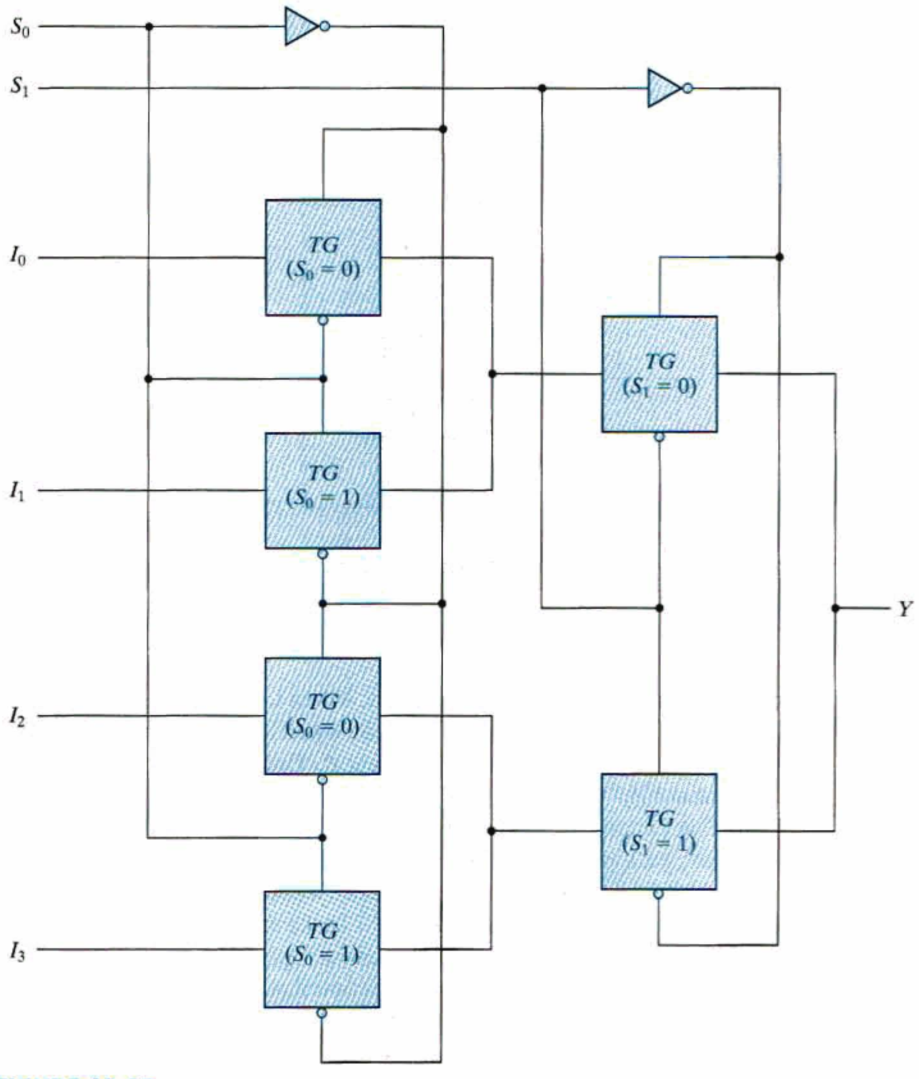


FIGURE 10.27
Multiplexer with transmission gates

Thus, the value that was present at input D at the time that C went from 1 to 0 is retained at the Q output.

A master–slave D flip-flop can be constructed with two circuits of the type shown in Fig. 10.28. The first circuit is the master and the second is the slave. Thus, a master–slave D flip-flop can be constructed with four transmission gates and six inverters.

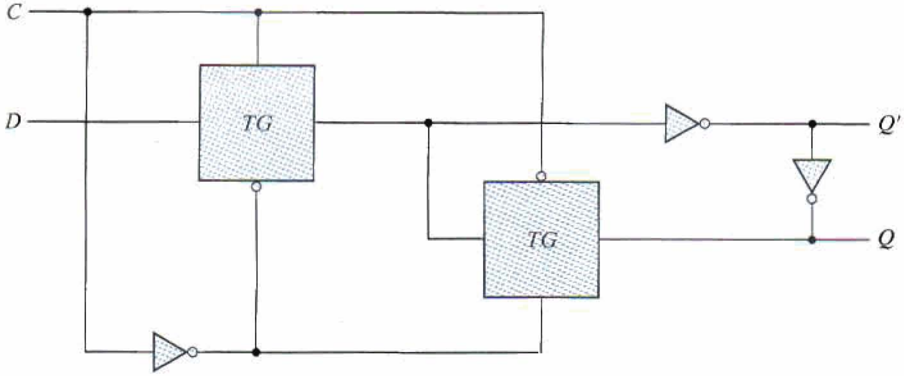


FIGURE 10.28
Gated *D* latch with transmission gates

10.10 SWITCH-LEVEL MODELING WITH HDL

CMOS is the dominant digital logic family used with integrated circuits. By definition, CMOS is a complementary connection of an NMOS and a PMOS transistor. MOS transistors can be considered to be electronic switches that either conduct or are open. By specifying the connections among MOS switches, the designer can describe a digital circuit constructed with CMOS. This type of description is called *switch-level modeling* in Verilog HDL.

The two types of MOS switches are specified in Verilog HDL with the keywords **nmos** and **pmos**. They are instantiated by specifying the three terminals of the transistor, as shown in Fig. 10.20:

```
nmos (drain, source, gate);
pmos (drain, source, gate);
```

Switches are considered to be primitives, so the use of an instance name is optional.

The connections to a power source (V_{DD}) and to ground must be specified when MOS circuits are designed. Power and ground are defined with the keywords **supply1** and **supply0**. They are specified, for example, with the following statements:

```
supply1 PWR;
supply0 GRD;
```

Sources of type **supply1** are equivalent to V_{DD} and have a value of logic 1. Sources of type **supply0** are equivalent to ground connection and have a value of logic 0.

The description of the CMOS inverter of Fig. 10.22(a) is shown in HDL Example 10.1. The input, the output, and the two supply sources are declared first. The module instantiates a PMOS and an NMOS transistor. The output *Y* is common to both transistors at their drain terminals. The input is also common to both transistors at their gate terminals. The source terminal of the PMOS transistor is connected to PWR and the source terminal of the NMOS transistor is connected to GRD.

HDL Example 10.1

```
// CMOS inverter of Fig. 10.22(a)
module inverter (Y, A);
 input A;
 output Y;
 supply1  PWR;
 supply0  GRD;
 pmos (Y, PWR, A);              // (Drain, source, gate)
 nmos (Y, GRD, A);              // (Drain, source, gate)
endmodule
```

The second module, set forth in HDL Example 10.2, describes the two-input CMOS NAND circuit of Fig. 10.22(b). There are two PMOS transistors connected in parallel, with their source terminals connected to PWR. There are also two NMOS transistors connected in series and with a common terminal *W1*. The drain of the first NMOS is connected to the output, and the source of the second NMOS is connected to GRD.

HDL Example 10.2

```
// CMOS two-input NAND of Fig. 10.22(b)
module NAND2 (Y, A, B);
 input A, B;
 output Y;
 supply1  PWR;
 supply0  GRD;
 wire W1;                       // terminal between two nmos
 pmos (Y, PWR, A);              // source connected to Vdd
 pmos (Y, PWR, B);              // parallel connection
 nmos (Y, W1, A);               // serial connection
 nmos (W1, GRD, B);             // source connected to ground
endmodule
```

Transmission Gate

The transmission gate is instantiated in Verilog HDL with the keyword **cmos**. It has an output, an input, and two control signals, as shown in Fig. 10.24. It is referred to as a **cmos** switch. The relevant code is as follows:

```
cmos (output, input, ncontrol, pcontrol); // general description

cmos (Y, X, N, P);  // transmission gate of Fig. 10.24(b)
```

Normally, ncontrol and pcontrol are the complement of each other. The **cmos** switch does not need power sources, since V_{DD} and ground are connected to the substrates of the MOS transistors. Transmission gates are useful for building multiplexers and flip-flops with CMOS circuits.

HDL Example 10.3 describes a circuit with **cmos** switches. The exclusive-OR circuit of Fig. 10.26 has two transmission gates and two inverters. The two inverters are instantiated within the module describing a CMOS inverter. The two **cmos** switches are instantiated without an instance name, since they are primitives in the language. A test module is included to test the circuit's operation. Applying all possible combinations of the two inputs, the result of the simulator verifies the operation of the exclusive-OR circuit. The output of the simulation is as follows:

```
A = 0   B = 0   Y = 0
A = 0   B = 1   Y = 1
A = 1   B = 0   Y = 1
A = 1   B = 1   Y = 0
```

HDL Example 10.3

//CMOS_XOR with CMOS switches, Fig. 10.26

```
module CMOS_XOR (A, B, Y);
 input A, B;
 output Y;
 wire A_b, B_b;
 // instantiate inverter
 inverter v1 (A_b, A);
 inverter v2 (B_b, B);
 // instantiate cmos switch
 cmos (Y, B, A_b, A);                    //(output, input, ncontrol, pcontrol)
 cmos (Y, B_b, A, A_b);
endmodule
// CMOS inverter Fig. 10-22(a)
module inverter (Y, A);
 input A;
 output Y;
 supply1  PWR;
 supply0  GND;
 pmos (Y, PWR, A);                       //(Drain, source, gate)
 nmos (Y, GND, A);                       //(Drain, source, gate)
endmodule
// Stimulus to test CMOS_XOR
module test_CMOS_XOR;
 reg A,B;
 wire Y;
 //Instantiate CMOS_XOR
 CMOS_XOR X1 (A, B, Y);
 // Apply truth table
 initial
  begin
```

```
    A = 1'b0; B = 1'b0;
  #5 A = 1'b0; B = 1'b1;
  #5 A = 1'b1; B = 1'b0;
  #5 A = 1'b1; B = 1'b1;
 end
 // Display results
 initial
  $monitor ("A =%b  B= %b  Y =%b", A, B, Y);
endmodule
```

PROBLEMS

Answers to problems marked with * appear at the end of the book.

10.1* Following are the specifications for the Schottky TTL 74S00 quadruple two-input NAND gates:

Parameter	**Name**	**Value**
V_{CC}	Supply voltage	5 V
I_{CCH}	High-level supply current (four gates)	10 mA
I_{CCL}	Low-level supply current (four gates)	20 mA
V_{OH}	High-level output voltage (min)	2.7 V
V_{OL}	Low-level output voltage (max)	0.5 V
V_{IH}	High-level input voltage (min)	2 V
V_{IL}	Low-level input voltage (max)	0.8 V
I_{OH}	High-level output current (max)	1 mA
I_{OL}	Low-level output current (max)	20 mA
I_{IH}	High-level input current (max)	0.05 mA
I_{IL}	Low-level input current (max)	2 mA
t_{PLH}	Low-to-high delay	3 ns
t_{PHL}	High-to-low delay	3 ns

Calculate the fan-out, power dissipation, propagation delay, and noise margin of the Schottky NAND gate.

10.2* (a) Determine the high-level output voltage of the RTL gate for a fan-out of 5.

(b) Determine the minimum input voltage required to drive an RTL transistor to saturation when $h_{FE} = 20$.

(c) From the results in parts (a) and (b), determine the noise margin of the RTL gate when the input is high and the fan-out is 5.

10.3* Show that the output transistor of the DTL gate of Fig. 10.9 goes into saturation when all inputs are high. Assume that $h_{FE} = 20$.

10.4* Connect the output Y of the DTL gate shown in Fig. 10.9 to N inputs of other, similar gates. Assume that the output transistor is saturated and its base current is 0.44 mA. Let $h_{FE} = 20$.

(a) Calculate the current in the 2-kΩ resistor.

(b) Calculate the current coming from each input connected to the gate.

(c) Calculate the total collector current in the output transistor as a function of N.

(d) Find the value of N that will keep the transistor in saturation.

(e) What is the fan-out of the gate?

10.5* Let all inputs in the open-collector TTL gate of Fig. 10.11 be in the high state of 3 V.

(a) Determine the voltages in the base, collector, and emitter of all transistors in the circuit.

(b) Determine the minimum h_{FE} of $Q2$ which ensures that this transistor saturates.

(c) Calculate the base current of $Q3$.

(d) Assume that the minimum h_{FE} of $Q3$ is 6.18. What is the maximum current that can be tolerated in the collector to ensure saturation of $Q3$?

(e) What is the minimum value of R_L that can be tolerated to ensure saturation of $Q3$?

10.6 (a) Using the actual output transistors of two open-collector TTL gates, show (by means of a truth table) that, when connected together to an external resistor and V_{CC}, the wired connection produces an AND function.

(b) Prove that two open-collector TTL inverters, when connected together, produce the NOR function.

10.7 It was stated in Section 10.5 that totem-pole outputs should not be tied together to form wired logic. To see why this is prohibitive, connect two such circuits together and let the output of one gate be in the high state and the output of the other gate be in the low state. Show that the load current (which is the sum of the base and collector currents of the saturated transistor $Q4$ in Fig. 10.14) is about 32 mA. Compare this value with the recommended load current in the high state of 0.4 mA.

10.8 For the following conditions, list the transistors that are off and the transistors that are conducting in the three-state TTL gate of Fig. 10.16(c) (for $Q1$ and $Q6$, it is necessary to list the states in the base–emitter and base–collector junctions separately):

(a) when C is low and A is low.

(b) when C is low and A is high.

(c) when C is high.

What is the state of the output in each case?

10.9* (a) Calculate the emitter current I_E across R_E in the ECL gate of Fig. 10.17 when at least one input is high at -0.8 V.

(b) Calculate the same current when all inputs are low at -1.8 V.

(c) Now assume that $I_C = I_E$. Calculate the voltage drop across the collector resistor in each case and show that it is about 1 V, as required.

10.10* Calculate the noise margin of the ECL gate.

10.11 Using the NOR outputs of two ECL gates, show that, when connected together to an external resistor and a negative supply voltage, the wired connection produces an OR function.

10.12 The MOS transistor is bilateral (i.e., current may flow from source to drain or from drain to source). On the basis of this property, derive a circuit that implements the Boolean function

$$Y = (AB + CD + AED + CEB)'$$

using six MOS transistors.

10.13 (a) Show the circuit of a four-input NAND gate using CMOS transistors.
(b) Repeat for a four-input NOR gate.

10.14 Construct an exclusive-NOR circuit with two inverters and two transmission gates.

10.15 Construct an eight-to-one-line multiplexer using transmission gates and inverters.

10.16 Draw the logic diagram of a master–slave *D* flip-flop using transmission gates and inverters.

10.17 Write a test bench that will test the NAND circuit of HDL Example 10.2. The simulation should verify the truth table of the gate.

REFERENCES

1. Ciletti, M. D. 1999. *Modeling, Synthesis, and Rapid Prototyping with Verilog HDL*. Upper Saddle River, NJ: Prentice Hall.
2. *CMOS Logic Data Book*. 1994. Dallas: Texas Instruments.
3. Hodges, D. A. 2003. *Analysis and Design of Digital Integrated Circuits*, New York: McGraw-Hill.
4. Tocci, R. J., and N. S. Widmer. 2004. *Digital Systems: Principles and Applications*, 9th ed. Upper Saddle River, NJ: Prentice Hall.
5. *The TTL Logic Data Book*. 1988. Dallas: Texas Instruments.
6. Wakerly, J. F. 2006. *Digital Design: Principles and Practices*, 4th ed. Upper Saddle River, NJ: Prentice Hall.
7. Weste, N. E., and K. Eshraghian. 2005. *Principles of CMOS VLSI Design: A System Perspective*, 2d ed. Reading, MA: Addison-Wesley.

Chapter 11

Laboratory Experiments with Standard ICs and FPGAs

11.1 INTRODUCTION TO EXPERIMENTS

This chapter presents 18 laboratory experiments in digital circuits and logic design. The experiments give the student using this book hands-on experience. The digital circuits can be constructed by using standard integrated circuits (ICs) mounted on breadboards that are easily assembled in the laboratory. The experiments are ordered according to the material presented in the book. The last section consists of a number of supplements with suggestions for using the Verilog HDL to simulate and verify the functionality of the digital circuits presented in the experiments. If an FPGA prototyping board is available, the experiments can be implemented in an FPGA as an alternative to standard ICs.

A logic breadboard suitable for performing the experiments must have the following equipment:

1. Light-emitting diode (LED) indicator lamps.
2. Toggle switches to provide logic-1 and logic-0 signals.
3. Pulsers with push buttons and debounce circuits to generate single pulses.
4. A clock-pulse generator with at least two frequencies: a low frequency of about 1 pulse per second to observe slow changes in digital signals and a higher frequency for observing waveforms in an oscilloscope.
5. A power supply of 5 V.
6. Socket strips for mounting the ICs.
7. Solid hookup wires and a pair of wire strippers for cutting the wires.

Digital logic trainers that include the required equipment are available from several manufacturers. A digital logic trainer contains LED lamps, toggle switches, pulsers, 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.

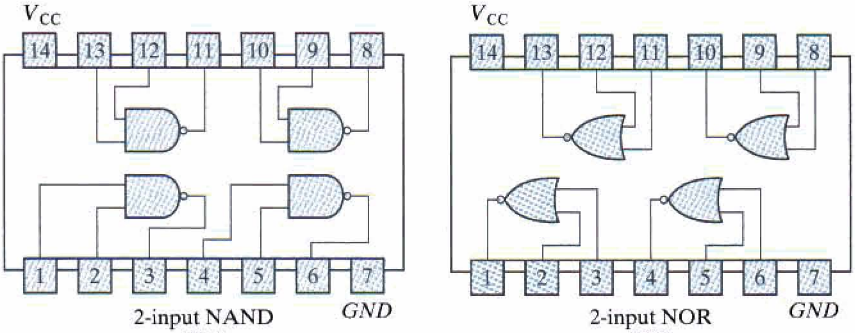
The integrated circuits to be used in the experiments can be classified as small-scale integration (SSI) or medium-scale integration (MSI) circuits. SSI circuits contain individual gates or flip-flops, and MSI circuits perform specific digital functions. The eight SSI gate ICs needed for the experiments—two-input NAND, NOR, AND, OR, and XOR gates, inverters, and three-input and four-input NAND gates—are shown in Fig. 11.1. 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.

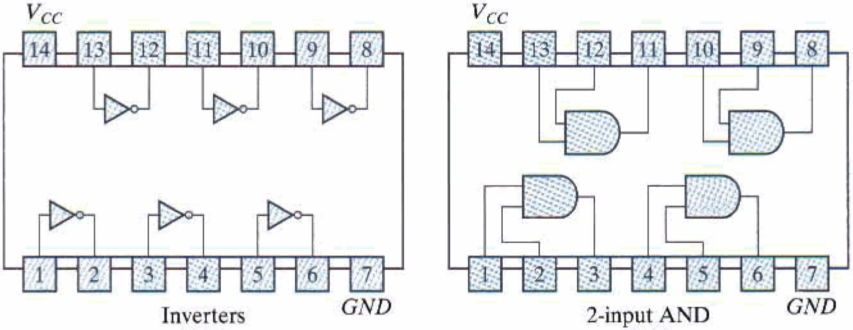
The information about the 7493 IC that is found in a data book is shown in Figs. 11.2(a) and (b). Part (a) shows a diagram of the internal logic circuit and its connection to external pins. All inputs and outputs are given symbolic letters and assigned to pin numbers. Part (b) shows the physical layout 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. 11.2(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 and output terminals. Inputs or outputs may sometimes be placed on the top or the bottom of the block for convenience.

The operation of the circuit is similar to the ripple counter shown in Fig. 6.8(a) with an asynchronous clear to each flip-flop. When input *R1* or *R2* or both are equal to logic 0 (ground), all asynchronous clears are equal to 1 and are disabled. To clear all four flip-flops to 0, the output



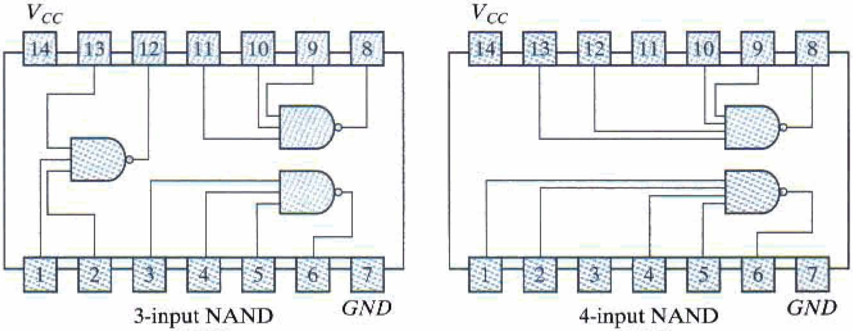
2-input NAND
7400

2-input NOR
7402



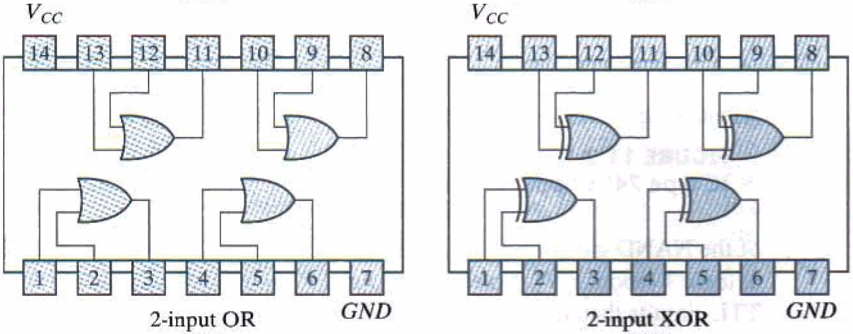
Inverters
7404

2-input AND
7408



3-input NAND
7410

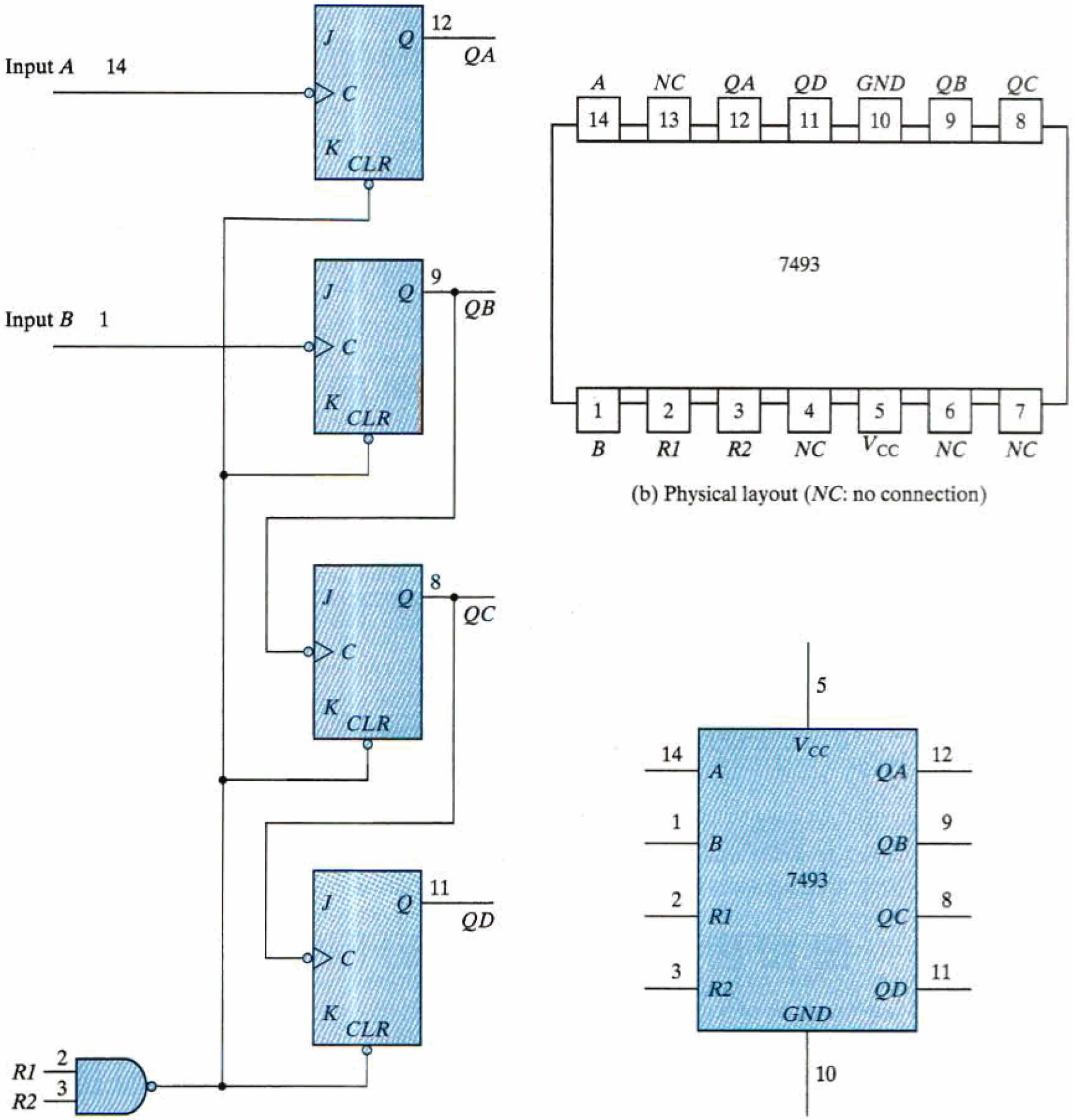
4-input NAND
7420



2-input OR
7432

2-input XOR
7486

FIGURE 11.1
Digital gates in IC packages with identification numbers and pin assignments



(b) Physical layout (*NC*: no connection)

(a) Internal circuit diagram

(c) Schematic diagram

FIGURE 11.2
IC type 7493 ripple counter

of the NAND gate must be equal to 0. This is accomplished by having both inputs *R1* and *R2* at logic 1 (about 5 V). Note that the *J* and *K* inputs show no connections. It is characteristic of TTL circuits that an input terminal with no external connections has the effect of producing a signal equivalent to logic 1. Note also that output *QA* is not connected to input *B* internally.

The 7493 IC can operate as a three-bit counter using input B and flip-flops QB, QC, and QD. It can operate as a four-bit counter using input A if output QA is connected to input B. Therefore, to operate the circuit as a four-bit counter, it is necessary to have an external connection between pin 12 and pin 1. The reset inputs, $R1$ and $R2$, at pins 2 and 3, respectively, must be grounded. Pins 5 and 10 must be connected to a 5-V power supply. The input pulses must be applied to input A at pin 14, and the four flip-flop outputs of the counter are taken from QA, QB, QC, and QD at pins 12, 9, 8, and 11, respectively, with QA being the least significant bit.

Figure 11.2(c) demonstrates the way that all MSI circuits will be symbolized graphically in this chapter. Only a block diagram similar to the one shown in this figure will be given for each IC. The letter symbols for the inputs and outputs in the IC block diagram will be according to the symbols used in the data book. The operation of the circuit will be explained with reference to logic diagrams from previous chapters. The operation of the circuit will be specified by means of a truth table or a function table.

Other possible graphic symbols for the ICs are presented in Chapter 12. These are standard graphic symbols approved by the Institute of Electrical and Electronics Engineers and are given in IEEE Standard 91-1984. The standard graphic symbols for SSI gates have rectangular shapes, as shown in Fig. 12.1. The standard graphic symbol for the 7493 IC is shown in Fig. 12.13. This symbol can be substituted in place of the one shown in Fig. 11.2(c). The standard graphic symbols of the other ICs that are needed to run the experiments are presented in Chapter 12. They can be used to draw schematic diagrams of the logic circuits if the standard symbols are preferred.

Table 11.1 lists the ICs that are needed for the experiments, together with the numbers of the figures in which they are presented in this chapter. In addition, the table lists the numbers of the figures in Chapter 12 in which the equivalent standard graphic symbols are drawn.

Table 11.1
Integrated Circuits Required for the Experiments

		Graphic Symbol	
IC Number	**Description**	**In Chapter 11**	**In Chapter 12**
	Various gates	Fig. 11.1	Fig. 12.1
7447	BCD-to-seven-segment decoder	Fig. 11.8	—
7474	Dual D-type flip-flops	Fig. 11.13	Fig. 12.9(b)
7476	Dual JK-type flip-flops	Fig. 11.12	Fig. 12.9(a)
7483	Four-bit binary adder	Fig. 11.10	Fig. 12.2
7493	Four-bit ripple counter	Fig. 11.2	Fig. 12.13
74151	8 × 1 multiplexer	Fig. 11.9	Fig. 12.7(a)
74155	3 × 8 decoder	Fig. 11.7	Fig. 12.6
74157	Quadruple 2 × 1 multiplexers	Fig. 11.17	Fig. 12.7(b)
74161	Four-bit synchronous counter	Fig. 11.15	Fig. 12.14
74189	16 × 4 random-access memory	Fig. 11.18	Fig. 12.15
74194	Bidirectional shift register	Fig. 11.19	Fig. 12.12
74195	Four-bit shift register	Fig. 11.16	Fig. 12.11
7730	Seven-segment LED display	Fig. 11.8	—
72555	Timer (same as 555)	Fig. 11.21	—

The next 18 sections present 18 hardware experiments requiring the use of digital integrated circuits. Section 11.20 outlines HDL simulation experiments requiring a Verilog HDL compiler and simulator.

11.2 EXPERIMENT 1: BINARY AND DECIMAL NUMBERS

This experiment demonstrates the count sequence of binary numbers and the binary-coded decimal (BCD) representation. It serves as an introduction to the breadboard used in the laboratory and acquaints the student with the cathode-ray oscilloscope. Reference material from the text that may be useful to know while performing the experiment can be found in Section 1.2, on binary numbers, and Section 1.7, on BCD numbers.

Binary Count

IC type 7493 consists of four flip-flops, as shown in Fig. 11.2. They can be connected to count in binary or in BCD. Connect the IC to operate as a four-bit binary counter by wiring the external terminals, as shown in Fig. 11.3. This is done by connecting a wire from pin 12 (output *QA*) to pin 1 (input *B*). Input *A* at pin 14 is connected to a pulser that provides single pulses. The two reset inputs, *R1* and *R2*, are connected to ground. The four outputs go to four indicator lamps, with the low-order bit of the counter from *QA* connected to the rightmost indicator lamp. Do not forget to supply 5 V and ground to the IC. All connections should be made with the power supply in the off position.

Turn the power on and observe the four indicator lamps. The four-bit number in the output is incremented by 1 for every pulse generated in the push-button pulser. The count goes to binary

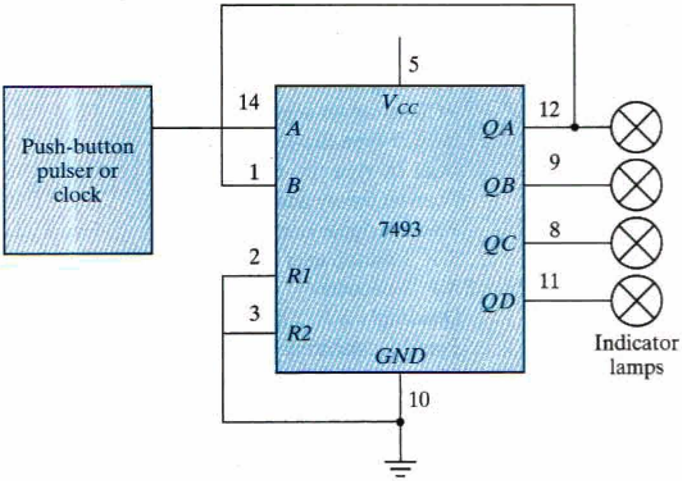


FIGURE 11.3
Binary counter

15 and then back to 0. Disconnect the input of the counter at pin 14 from the pulser, and connect it to a clock generator that produces a train of pulses at a low frequency of about 1 pulse per second. This will provide an automatic binary count. Note that the binary counter will be used in subsequent experiments to provide the input binary signals for testing combinational circuits.

Oscilloscope Display

Increase the frequency of the clock to 10 kHz or higher and connect its output to an oscilloscope. Observe the clock output on the oscilloscope and sketch its waveform. Using a dual-trace oscilloscope, connect the output of *QA* to one channel and the output of the clock to the second channel. Note that the output of *QA* is complemented every time the clock pulse goes through a negative transition from 1 to 0. Note also that the clock frequency at the output of the first flip-flop is one-half that of the input clock frequency. Each flip-flop in turn divides its incoming frequency by 2. The four-bit counter divides the incoming frequency by 16 at output *QD*. Obtain a timing diagram showing the relationship of the clock to the four outputs of the counter. Make sure that you include at least 16 clock cycles. The way to proceed with a dual-trace oscilloscope is as follows: First, observe the clock pulses and *QA*, and record their timing waveforms. Then repeat by observing and recording the waveforms of *QA* together with *QB*, followed by the waveforms of *QB* with *QC* and then *QC* with *QD*. Your final result should be a diagram showing the relationship of the clock to the four outputs in one composite diagram having at least 16 clock cycles.

BCD Count

The BCD representation uses the binary numbers from 0000 to 1001 to represent the coded decimal digits from 0 to 9. IC type 7493 can be operated as a BCD counter by making the external connections shown in Fig. 11.4. Outputs *QB* and *QD* are connected to the two reset inputs,

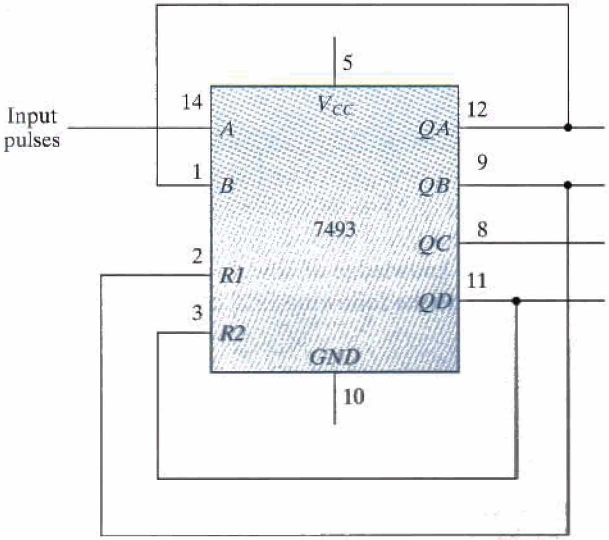


FIGURE 11.4
BCD counter

R1 and *R2*. When both *R1* and *R2* are equal to 1, all four cells in the counter clear to 0 irrespective of the input pulse. The counter starts from 0, and every input pulse increments it by 1 until it reaches the count of 1001. The next pulse changes the ouput to 1010, making *QB* and *QD* equal to 1. This momentary output cannot be sustained, because the four cells immediately clear to 0, with the result that the output goes to 0000. Thus, the pulse after the count of 1001 changes the output to 0000, producing a BCD count.

Connect the IC to operate as a BCD counter. Connect the input to a pulser and the four outputs to indicator lamps. Verify that the count goes from 0000 to 1001.

Disconnect the input from the pulser and connect it to a clock generator. Observe the clock waveform and the four outputs on the oscilloscope. Obtain an accurate timing diagram showing the relationship between the clock and the four outputs. Make sure to include at least 10 clock cycles in the oscilloscope display and in the composite timing diagram.

Output Pattern

When the count pulses into the BCD counter are continuous, the counter keeps repeating the sequence from 0000 to 1001 and back to 0000. This means that each bit in the four outputs produces a fixed pattern of 1's and 0's that is repeated every 10 pulses. These patterns can be predicted from a list of the binary numbers from 0000 to 1001. The list will show that output *QA*, being the least significant bit, produces a pattern of alternate 1's and 0's. Output *QD*, being the most significant bit, produces a pattern of eight 0's followed by two 1's. Obtain the pattern for the other two outputs and then check all four patterns on the oscilloscope. This is done with a dual-trace oscilloscope by displaying the clock pulses in one channel and one of the output waveforms in the other channel. The pattern of 1's and 0's for the corresponding output is obtained by observing the output levels at the vertical positions where the pulses change from 1 to 0.

Other Counts

IC type 7493 can be connected to count from 0 to a variety of final counts. This is done by connecting one or two outputs to the reset inputs, *R1* and *R2*. Thus, if *R1* is connected to *QA* instead of to *QB* in Fig. 11.4, the resulting count will be from 0000 to 1000, which is 1 less than 1001 ($QD = 1$ and $QA = 1$).

Utilizing your knowledge of how *R1* and *R2* affect the final count, connect the 7493 IC to count from 0000 to the following final counts:

(a) 0101

(b) 0111

(c) 1011

Connect each circuit and verify its count sequence by applying pulses from the pulser and observing the output count in the indicator lamps. If the initial count starts with a value greater than the final count, keep applying input pulses until the output clears to 0.

11.3 EXPERIMENT 2: DIGITAL LOGIC GATES

In this experiment, you will investigate the logic behavior of various IC gates:

7400 quadruple two-input NAND gates

7402 quadruple two-input NOR gates

7404 hex inverters

7408 quadruple two-input AND gates

7432 quadruple two-input OR gates

7486 quadruple two-input XOR gates

The pin assignments to the various gates are shown in Fig. 11.1. "Quadruple" means that there are four gates within the package. The digital logic gates and their characteristics are discussed in Section 2.8. A NAND implementation is discussed in Section 3.7.

Truth Tables

Use one gate from each IC listed and obtain the truth table of the gate. The truth table is obtained by connecting the inputs of the gate to switches and the output to an indicator lamp. Compare your results with the truth tables listed in Fig. 2.5.

Waveforms

For each gate listed, obtain the input–output waveform of the gate. The waveforms are to be observed in the oscilloscope. Use the two low-order outputs of a binary counter (Fig. 11.3) to provide the inputs to the gate. As an example, the circuit and waveforms for the NAND gate are illustrated in Fig. 11.5. The oscilloscope display will repeat this waveform, but you should record only the nonrepetitive portion.

Propagation Delay

Connect the six inverters inside the 7404 IC in cascade. The output will be the same as the input, except that it will be delayed by the time it takes the signal to propagate through all six inverters. Apply clock pulses to the input of the first inverter. Using the oscilloscope, determine

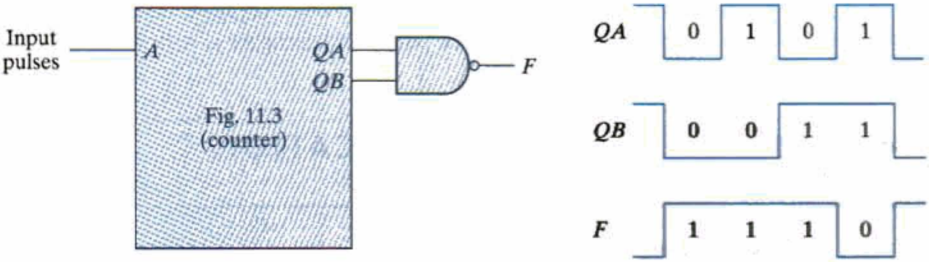


FIGURE 11.5
Waveforms for NAND gate

the delay from the input to the output of the sixth inverter during the upswing of the pulse and again during the downswing. This is done with a dual-trace oscilloscope by applying the input clock pulses to one of the channels and the output of the sixth inverter to the second channel. Set the time-base knob to the lowest time-per-division setting. The rise or fall time of the two pulses should appear on the screen. Divide the total delay by 6 to obtain an average propagation delay per inverter.

Universal NAND Gate

Using a single 7400 IC, connect a circuit that produces

(a) an inverter.

(b) a two-input AND.

(c) a two-input OR.

(d) a two-input NOR.

(e) a two-input XOR. (See Fig. 3.32.)

In each case, verify your circuit by checking its truth table.

NAND Circuit

Using a single 7400 IC, construct a circuit with NAND gates that implements the Boolean function

$$F = AB + CD$$

1. Draw the circuit diagram.
2. Obtain the truth table for F as a function of the four inputs.
3. Connect the circuit and verify the truth table.
4. Record the patterns of 1's and 0's for F as inputs A, B, C, and D go from binary 0 to binary 15.
5. Connect the four outputs of the binary counter shown in Fig. 11.3 to the four inputs of the NAND circuit. Connect the input clock pulses from the counter to one channel of a dual-trace oscilloscope and output F to the other channel. Observe and record the 1's and 0's pattern of F after each clock pulse, and compare it with the pattern recorded in step 4.

11.4 EXPERIMENT 3: SIMPLIFICATION OF BOOLEAN FUNCTIONS

This experiment demonstrates the relationship between a Boolean function and the corresponding logic diagram. The Boolean functions are simplified by using the map method, as discussed in Chapter 3. The logic diagrams are to be drawn with NAND gates, as explained in Section 3.7.

The gate ICs to be used for the logic diagrams must be those from Fig. 11.1 which contain the following NAND gates:

7400 two-input NAND
7404 inverter (one-input NAND)
7410 three-input NAND
7420 four-input NAND

If an input to a NAND gate is not used, it should not be left open, but instead should be connected to another input that is used. For example, if the circuit needs an inverter and there is an extra two-input gate available in a 7400 IC, then both inputs of the gate are to be connected together to form a single input for an inverter.

Logic Diagram

This part of the experiment starts with a given logic diagram from which we proceed to apply simplification procedures to reduce the number of gates and, possibly, the number of ICs. The logic diagram shown in Fig. 11.6 requires two ICs—a 7400 and a 7410. Note that the inverters for inputs x, y, and z are obtained from the remaining three gates in the 7400 IC. If the inverters were taken from a 7404 IC, the circuit would have required three ICs. Note

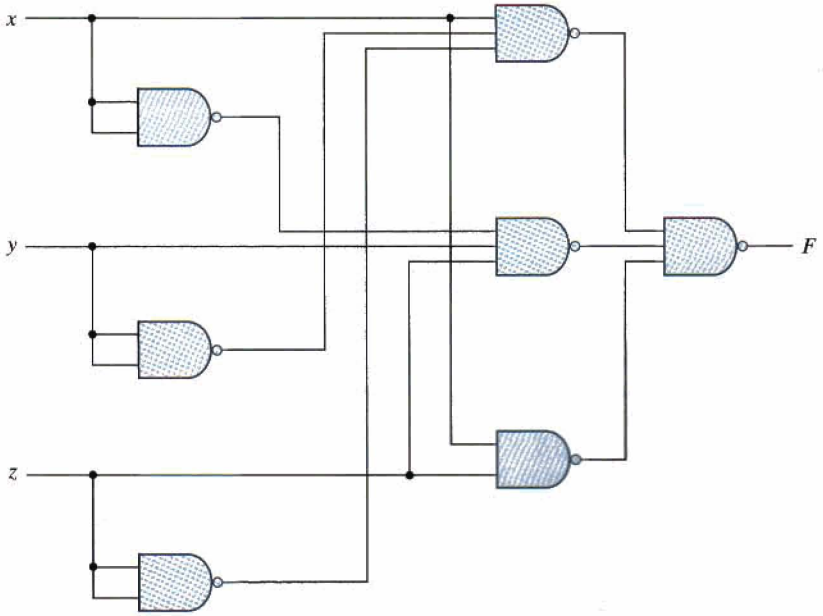


FIGURE 11.6
Logic diagram for Experiment 3

also that, in drawing SSI circuits, the gates are not enclosed in blocks as is done with MSI circuits.

Assign pin numbers to all inputs and outputs of the gates, and connect the circuit with the x, y, and z inputs going to three switches and the output F to an indicator lamp. Test the circuit by obtaining its truth table.

Obtain the Boolean function of the circuit and simplify it, using the map method. Construct the simplified circuit without disconnecting the original circuit. Test both circuits by applying identical inputs to each and observing the separate outputs. Show that, for each of the eight possible input combinations, the two circuits have identical outputs. This will prove that the simplified circuit behaves exactly like the original circuit.

Boolean Functions

Consider two Boolean functions in sum-of-minterms form:

$$F_1(A, B, C, D) = (0, 1, 4, 5, 8, 9, 10, 12, 13)$$
$$F_2(A, B, C, D) = (3, 5, 7, 8, 10, 11, 13, 15)$$

Simplify these functions by means of maps. Obtain a composite logic diagram with four inputs, A, B, C, and D, and two outputs, F_1 and F_2. Implement the two functions together, using a minimum number of NAND ICs. Do not duplicate the same gate if the corresponding term is needed for both functions. Use any extra gates in existing ICs for inverters when possible. Connect the circuit and check its operation. The truth table for F_1 and F_2 obtained from the circuit should conform with the minterms listed.

Complement

Plot the following Boolean function in a map:

$$F = A'D + BD + B'C + AB'D$$

Combine the 1's in the map to obtain the simplified function for F in sum-of-products form. Then combine the 0's in the map to obtain the simplified function for F', also in sum-of-products form. Implement both F and F' with NAND gates, and connect the two circuits to the same input switches, but to separate output indicator lamps. Obtain the truth table of each circuit in the laboratory and show that they are the complements of each other.

11.5 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

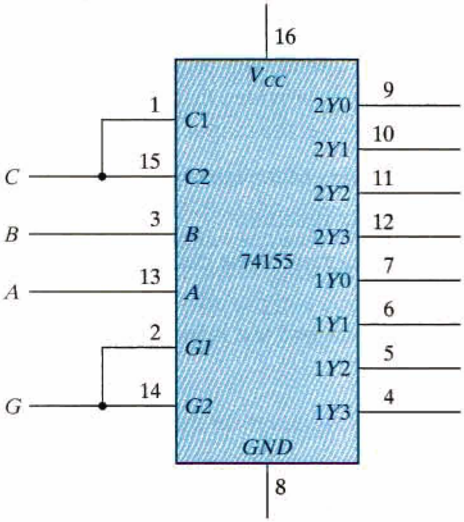
$$F_1 = xz + x'y'z'$$

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

$$F_3 = xy + x'y'z$$

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

The block diagram of the decoder and its truth table are shown in Fig. 11.7. The 74155 can be connected as a dual 2 × 4 decoder or as a single 3 × 8 decoder. When a 3 × 8 decoder is desired, inputs *C1* and *C2*, as well as inputs *G1* and *G2*, must be connected together, as shown in the block diagram. The function of the circuit is similar to that illustrated in Fig. 4.18. *G* is the enable input and must be equal to 0 for proper operation. The eight outputs are labeled with symbols given in the data book. The 74155 uses NAND gates, with the result that the selected output goes to 0 while all other outputs remain at 1. The implementation with the decoder is as shown in Fig. 4.21, except that the OR gates must be replaced with external NAND gates when the 74155 is used.



Truth table

Inputs				Outputs							
G	*C*	*B*	*A*	2*Y*0	2*Y*1	2*Y*2	2*Y*3	1*Y*0	1*Y*1	1*Y*2	1*Y*3
1	X	X	X	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	0

FIGURE 11.7
IC type 74155 connected as a 3 × 8 decoder

11.6 EXPERIMENT 5: CODE CONVERTERS

The conversion from one binary code to another is common in digital systems. In this experiment, you will design and construct three combinational-circuit converters. Code conversion is discussed in Section 4.4.

Gray Code to Binary

Design a combinational circuit with four inputs and four outputs that converts a four-bit Gray code number (Table 1.6) into the equivalent four-bit binary number. Implement the circuit with exclusive-OR gates. (This can be done with one 7486 IC.) Connect the circuit to four switches and four indicator lamps, and check for proper operation.

9's Complementer

Design a combinational circuit with four input lines that represent a decimal digit in BCD and four output lines that generate the 9's complement of the input digit. Provide a fifth output that detects an error in the input BCD number. This output should be equal to logic 1 when the four inputs have one of the unused combinations of the BCD code. Use any of the gates listed in Fig. 11.1, but minimize the total number of ICs used.

Seven-Segment Display

A seven-segment indicator is used to display any one of the decimal digits 0 through 9. Usually, the decimal digit is available in BCD. A BCD-to-seven-segment decoder accepts a decimal digit in BCD and generates the corresponding seven-segment code, as is shown pictorially in Problem 4.9.

Figure 11.8 shows the connections necessary between the decoder and the display. The 7447 IC is a BCD-to-seven-segment decoder/driver that has four inputs for the BCD digit. Input *D* is the most significant and input *A* the least significant. The four-bit BCD digit is converted to a seven-segment code with outputs *a* through *g*. The outputs of the 7447 are applied to the inputs of the 7730 (or equivalent) seven-segment display. This IC contains the seven light-emitting diode (LED) segments on top of the package. The input at pin 14 is the common anode (*CA*) for all the LEDs. A 47-Ω resistor to V_{CC} is needed in order to supply the proper current to the selected LED segments. Other equivalent seven-segment display ICs may have additional anode terminals and may require different resistor values.

Construct the circuit shown in Fig. 11.8. Apply the four-bit BCD digits through four switches, and observe the decimal display from 0 to 9. Inputs 1010 through 1111 have no meaning in BCD.

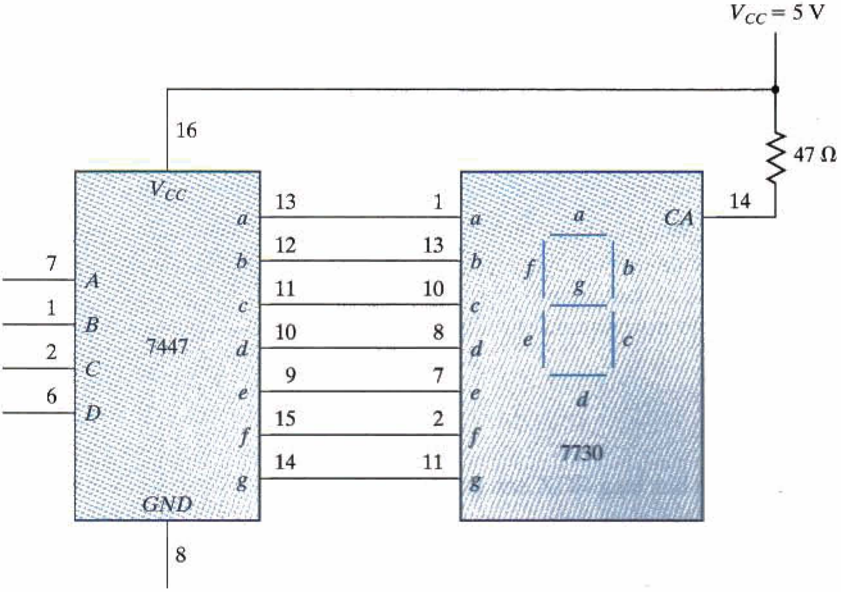
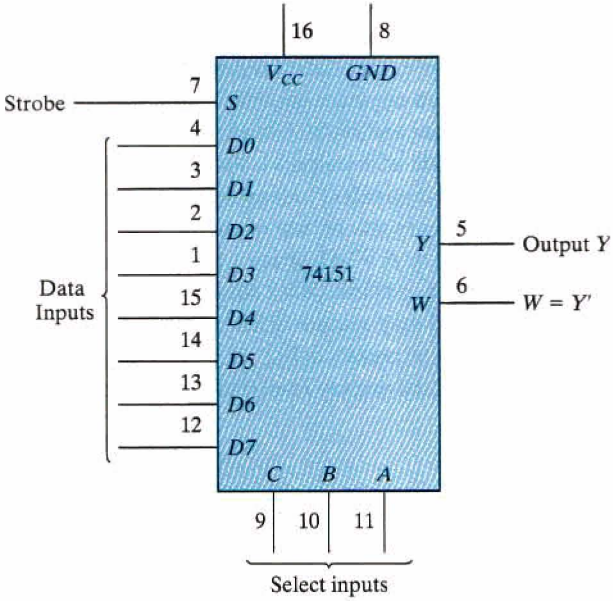


FIGURE 11.8
BCD-to-seven-segment decoder (7447) and seven-segment display (7730)

Depending on the decoder, these values may cause either a blank or a meaningless pattern to be displayed. Observe and record the output patterns of the six unused input combinations.

11.7 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. 11.9. The internal construction of the 74151 is similar to the diagram shown in Fig. 4.25, except that



Function table

Strobe	Select			Output
S	*C*	*B*	*A*	*Y*
1	X	X	X	0
0	0	0	0	*D0*
0	0	0	1	*D1*
0	0	1	0	*D2*
0	0	1	1	*D3*
0	1	0	0	*D4*
0	1	0	1	*D5*
0	1	1	0	*D6*
0	1	1	1	*D7*

FIGURE 11.9
IC type 74151 38 × 1 multiplexer

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

Each of these persons has a switch to close when voting yes and to open when voting no for his or her shares.

It is necessary to design a circuit that displays the total number of shares that vote yes for each measure. Use a seven-segment display and a decoder, as shown in Fig. 11.8, to display the required number. If all shares vote no for a measure, the display should be blank. (Note that binary input 15 into the 7447 blanks out all seven segments.) If 10 shares vote yes for a measure, the display should show 0. Otherwise, the display shows a decimal number equal to the number of shares that vote yes. Use four 74151 multiplexers to design the combinational circuit that converts the inputs from the stock owners' switches into the BCD digit for the 7447. Do not use 5 V for logic 1. Use the output of an inverter whose input is grounded.

11.8 EXPERIMENT 7: ADDERS AND SUBTRACTORS

In this experiment, you will construct and test various adder and subtractor circuits. The subtractor circuit is then used to compare the relative magnitudes of two numbers. Adders are discussed in Section 4.3. Subtraction with 2's complement is explained in Section 1.6. A four-bit parallel adder–subtractor is shown in Fig. 4.13, and the comparison of two numbers is explained in Section 4.8.

Half Adder

Design, construct, and test a half-adder circuit using one XOR gate and two NAND gates.

Full Adder

Design, construct, and test a full-adder circuit using two ICs, 7486 and 7400.

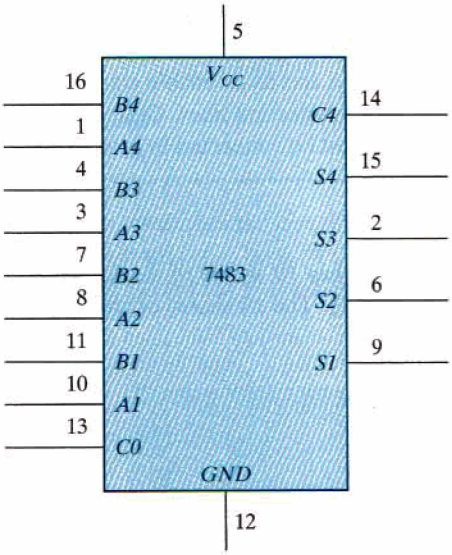


FIGURE 11.10
IC type 7483 four-bit binary adder

Parallel Adder

IC type 7483 is a four-bit binary parallel adder. The pin assignment is shown in Fig. 11.10. The 2 four-bit input binary numbers are *A1* through *A4* and *B1* through *B4*. The four-bit sum is obtained from *S1* through *S4*. *C0* is the input carry and *C4* the output carry.

Test the four-bit binary adder 7483 by connecting the power supply and ground terminals. Then connect the four *A* inputs to a fixed binary number, such as 1001, and the *B* inputs and the input carry to five toggle switches. The five outputs are applied to indicator lamps. Perform the addition of a few binary numbers and check that the output sum and output carry give the proper values. Show that when the input carry is equal to 1, it adds 1 to the output sum.

Adder–Subtractor

Two binary numbers can be subtracted by taking the 2's complement of the subtrahend and adding it to the minuend. The 2's complement can be obtained by taking the 1's complement and adding 1. To perform $A - B$, we complement the four bits of *B*, add them to the four bits of *A*, and add 1 through the input carry. This is done as shown in Fig. 11.11. The four XOR gates complement the bits of *B* when the mode select $M = 1$ (because $x \oplus 1 = x'$) and leave the bits of *B* unchanged when $M = 0$ (because $x \oplus 0 = x$). Thus, when the mode select *M* is equal to 1, the input carry *C0* is equal to 1 and the sum output is *A* plus the 2's complement of *B*. When *M* is equal to 0, the input carry is equal to 0 and the sum generates $A + B$.

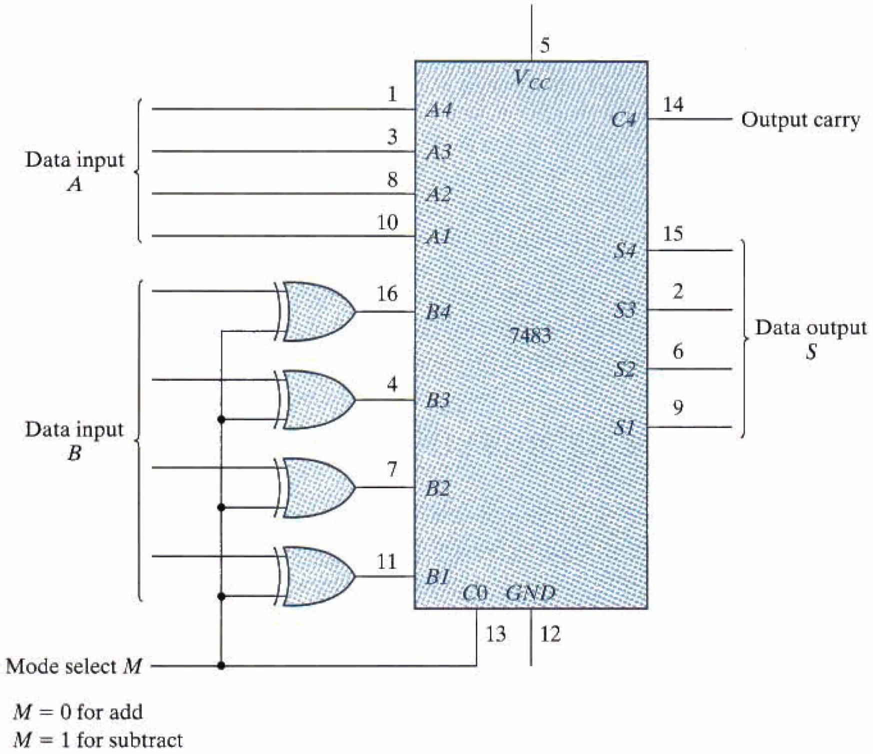


FIGURE 11.11
Four-bit adder–subtractor

Connect the adder–subtractor circuit and test it for proper operation. Connect the four A inputs to a fixed binary number 1001 and the B inputs to switches. Perform the following operations and record the values of the output sum and the output carry $C4$:

$$
\begin{array}{ll}
9+5 & 9-5 \\
9+9 & 9-9 \\
9+15 & 9-15
\end{array}
$$

Show that during addition, the output carry is equal to 1 when the sum exceeds 15. Also, show that when $A \geq B$, the subtraction operation gives the correct answer, $A - B$, and the output carry $C4$ is equal to 1, but when $A < B$, the subtraction gives the 2's complement of $B - A$ and the output carry is equal to 0.

Magnitude Comparator

The comparison of two numbers is an operation that determines whether one number is greater than, equal to, or less than the other number. Two numbers, A and B, can be compared by first subtracting $A - B$ as is done in Fig. 11.11. If the output in S is equal to zero, then $A = B$. The output carry from $C4$ determines the relative magnitudes of the numbers: When $C4 = 1$, $A \geq B$; when $C4 = 0$, $A < B$; and when $C4 = 1$ and $S \neq 0$, $A > B$.

It is necessary to supplement the subtractor circuit of Fig. 11.11 to provide the comparison logic. This is done with a combinational circuit that has five inputs—*S1* through *S4* and *C4*—and three outputs, designated by x, y, and z, so that

$$\begin{array}{lll} x = 1 & \text{if } A = B & (S = 0000) \\ y = 1 & \text{if } A < B & (C4 = 0) \\ z = 1 & \text{if } A > B & (C4 = 1 \text{ and } S \neq 0000) \end{array}$$

The combinational circuit can be implemented with the 7404 and 7408 ICs.

Construct the comparator circuit and test its operation. Use at least two sets of numbers for A and B to check each of the outputs x, y, and z.

11.9 EXPERIMENT 8: FLIP-FLOPS

In this experiment, you will construct, test, and investigate the operation of various latches and flip-flops. The internal construction of latches and flip-flops can be found in Sections 5.3 and 5.4.

SR Latch

Construct an *SR* latch with two cross-coupled NAND gates. Connect the two inputs to switches and the two outputs to indicator lamps. Set the two switches to logic 1, and then momentarily turn each switch separately to the logic-0 position and back to 1. Obtain the function table of the circuit.

D Latch

Construct a *D* latch with four NAND gates (only one 7400 IC) and verify its function table.

Master–Slave Flip-Flop

Connect a master–slave *D* flip-flop using two *D* latches and an inverter. Connect the *D* input to a switch and the clock input to a pulser. Connect the output of the master latch to one indicator lamp and the output of the slave latch to another indicator lamp. Set the value of the input to the complement value of the output. Press the push button in the pulser and then release it to produce a single pulse. Observe that the master changes when the pulse goes positive and the slave follows the change when the pulse goes negative. Press the push button again a few times while observing the two indicator lamps. Explain the transfer sequence from input to master and from master to slave.

Disconnect the clock input from the pulser and connect it to a clock generator. Connect the complement output of the flip-flop to the *D* input. This causes the flip-flop to be complemented with each clock pulse. Using a dual-trace oscilloscope, observe the waveforms of the clock and the master and slave outputs. Verify that the delay between the master and the slave outputs is equal to the positive half of the clock cycle. Obtain a timing diagram showing the relationship between the clock waveform and the master and slave outputs.

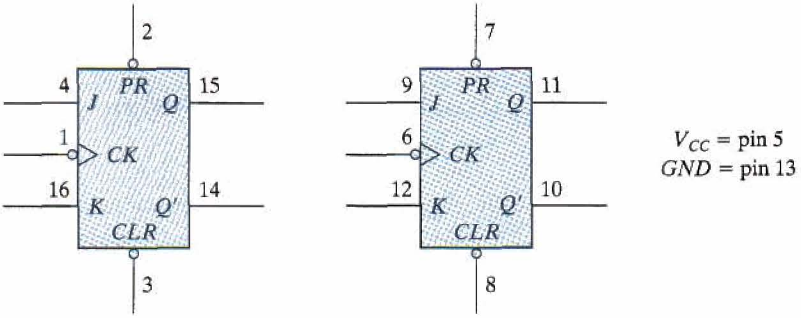
Edge-Triggered Flip-Flop

Construct a *D*-type positive-edge-triggered flip-flop using six NAND gates. Connect the clock input to a pulser, the *D* input to a toggle switch, and the output *Q* to an indicator lamp. Set the value of *D* to the complement of *Q*. Show that the flip-flop output changes only in response to a positive transition of the clock pulse. Verify that the output does not change when the clock input is logic 1, when the clock goes through a negative transition, or when the clock input is logic 0. Continue changing the *D* input to correspond to the complement of the *Q* output at all times.

Disconnect the input from the pulser and connect it to the clock generator. Connect the complement output Q' to the *D* input. This causes the output to be complemented with each positive transition of the clock pulse. Using a dual-trace oscilloscope, observe and record the timing relationship between the input clock and the output *Q*. Show that the output changes in response to a positive edge transition.

IC Flip-Flops

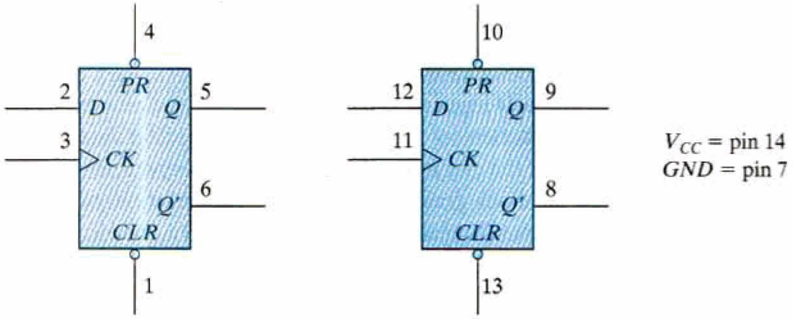
IC type 7476 consists of two *JK* master–slave flip-flops with preset and clear. The pin assignment for each flip-flop is shown in Fig. 11.12. The function table specifies the circuit's operation. The first three entries in the table specify the operation of the asynchronous preset and



Function table

Inputs					Outputs	
Preset	Clear	Clock	*J*	*K*	*Q*	*Q'*
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	1	1
1	1	⎍	0	0	**No change**	
1	1	⎍	0	1	**0**	**1**
1	1	⎍	1	0	**1**	**0**
1	1	⎍	1	1	**Toggle**	

FIGURE 11.12
IC type 7476 dual *JK* master–slave flip-flops



Function table

Inputs				Outputs	
Preset	Clear	Clock	D	Q	Q'
0	1	X	X	1	0
1	0	X	X	0	1
0	0	X	X	1	1
1	1	↑	0	0	1
1	1	↑	1	1	0
1	1	0	X	No change	

FIGURE 11.13
IC type 7474 dual *D* positive-edge-triggered flip-flops

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. 11.13. 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.

11.10 EXPERIMENT 9: SEQUENTIAL CIRCUITS

In this experiment, you will design, construct, and test three synchronous sequential circuits. Use IC type 7476 (Fig. 11.12) or 7474 (Fig. 11.13). 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.

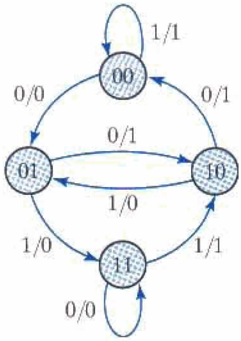


FIGURE 11.14
State diagram for Experiment 9

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. 11.14. 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.

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.

11.11 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. 11.12). 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.

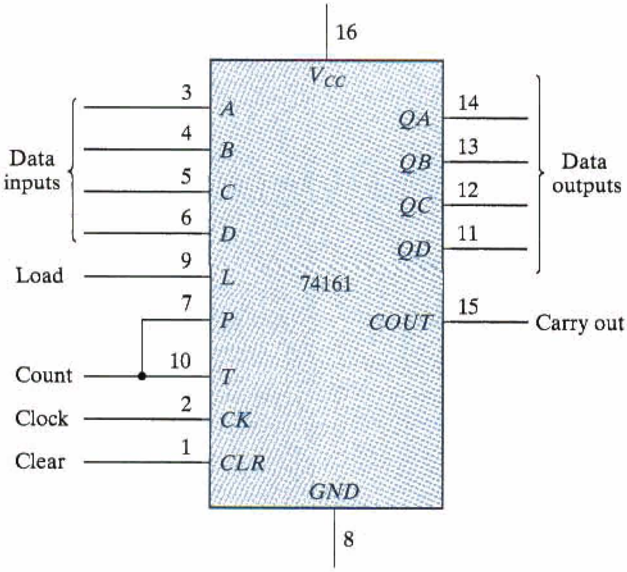
Decimal Counter

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.

Binary Counter with Parallel Load

IC type 74161 is a four-bit synchronous binary counter with parallel load and asynchronous clear. The internal logic is similar to that of the circuit shown in Fig. 6.14. The pin assignments to the inputs and outputs are shown in Fig. 11.15. When the load signal is enabled, the four data inputs are transferred into four internal flip-flops, QA through QD, with QD being the most significant bit. There are two count-enable inputs called P and T. Both must be equal to 1 for the counter to operate. The function table is similar to Table 6.6, with one exception: The load input in the 74161 is enabled when equal to 0. To load the input data, the clear input must be equal to 1 and the load input must be equal to 0. The two count inputs have don't-care conditions and may be equal to either 1 or 0. The internal flip-flops trigger on the positive transition of the clock pulse. The circuit functions as a counter when the load input is equal to 1 and both count inputs P and T are equal to 1. If either P or T goes to 0, the output does not change. The carry-out output is equal to 1 when all four data outputs are equal to 1. Perform an experiment to verify the operation of the 74161 IC according to the function table.

Show how the 74161 IC, together with a two-input NAND gate, can be made to operate as a synchronous BCD counter that counts from 0000 to 1001. Do not use the clear input. Use the NAND gate to detect the count of 1001, which then causes all 0's to be loaded into the counter.



Function table

Clear	Clock	Load	Count	Function
0	X	X	X	Clear outputs to 0
1	↑	0	X	Load input data
1	↑	1	1	Count to next binary value
1	↑	1	0	No change in output

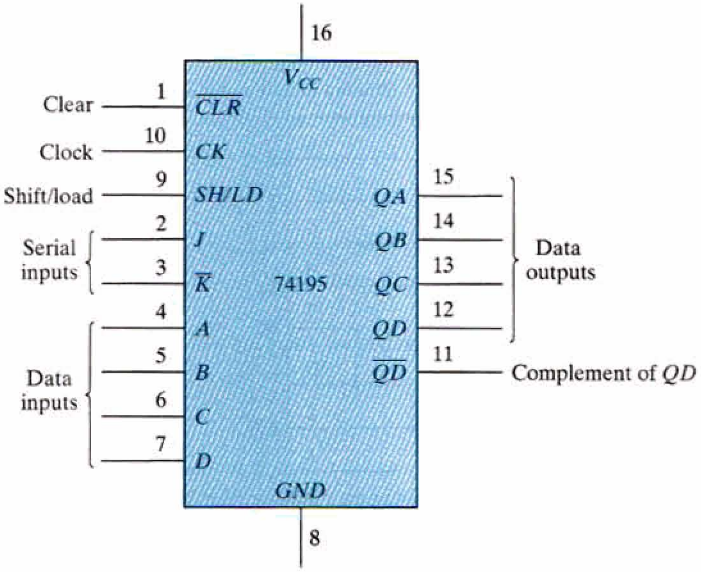
FIGURE 11.15
IC type 74161 binary counter with parallel load

11.12 EXPERIMENT 11: SHIFT REGISTERS

In this experiment, you will investigate the operation of shift registers. The IC to be used is the 74195 shift register with parallel load. Shift registers are explained in Section 6.2.

IC Shift Register

IC type 74195 is a four-bit shift register with parallel load and asynchronous clear. The pin assignments to the inputs and outputs are shown in Fig. 11.16. The single control line labeled *SH/LD* (shift/load) determines the synchronous operation of the register. When *SH/LD* = 0, the control input is in the load mode and the four data inputs are transferred into the four internal flip-flops, *QA* through *QD*. When *SH/LD* = 1, the control input is in the shift mode and the information in the register is shifted right from *QA* toward *QD*. The serial input into *QA* during the shift is determined from the *J* and $\overline{K}$ inputs. The two inputs behave like the *J* and the complement of *K* of a *JK* flip-flop. When both *J* and $\overline{K}$ are equal to 0, flip-flop *QA* is



Function table

Clear	Shift/ load	Clock	J	$\overline{K}$	Serial input	Function
0	X	X	X	X	X	Asynchronous clear
1	X	0	X	X	X	No change in output
1	0	↑	X	X	X	Load input data
1	1	↑	0	0	0	Shift from QA toward QD. $QA = 0$
1	1	↑	1	1	1	Shift from QA toward QD, $QA = 1$

FIGURE 11.16
IC type 74195 shift register with parallel load

cleared to 0 after the shift. If both inputs are equal to 1, QA is set to 1 after the shift. The other two conditions for the J and $\overline{K}$ inputs provide a complement or no change in the output of flip-flop QA after the shift.

The function table for the 74195 shows the mode of operation of the register. When the clear input goes to 0, the four flip-flops clear to 0 asynchronously—that is, without the need of a clock. Synchronous operations are affected by a positive transition of the clock. To load the input data, *SH/LD* must be equal to 0 and a positive clock-pulse transition must occur. To shift right, *SH/LD* must be equal to 1. The J and $\overline{K}$ inputs must be connected together to form the serial input.

Perform an experiment that will verify the operation of the 74195 IC. Show that it performs all the operations listed in the function table. Include in your function table the two conditions for $J\overline{K} = 01$ and 10.

Ring Counter

A ring counter is a circular shift register with the signal from the serial output *QD* going into the serial input. Connect the *J* and $\overline{K}$ input together to form the serial input. Use the load condition to preset the ring counter to an initial value of 1000. Rotate the single bit with the shift condition and check the state of the register after each clock pulse.

A switch-tail ring counter uses the complement output of *QD* for the serial input. Preset the switch-tail ring counter to 0000 and predict the sequence of states that will result from shifting. Verify your prediction by observing the state sequence after each shift.

Feedback Shift Register

A feedback shift register is a shift register whose serial input is connected to some function of selected register outputs. Connect a feedback shift register whose serial input is the exclusive-OR of outputs *QC* and *QD*. Predict the sequence of states of the register, starting from state 1000. Verify your prediction by observing the state sequence after each clock pulse.

Bidirectional Shift Register

The 74195 IC can shift only right from *QA* toward *QD*. It is possible to convert the register to a bidirectional shift register by using the load mode to obtain a shift-left operation (from *QD* toward *QA*). This is accomplished by connecting the output of each flip-flop to the input of the flip-flop on its left and using the load mode of the *SH/LD* input as a shift-left control. Input *D* becomes the serial input for the shift-left operation.

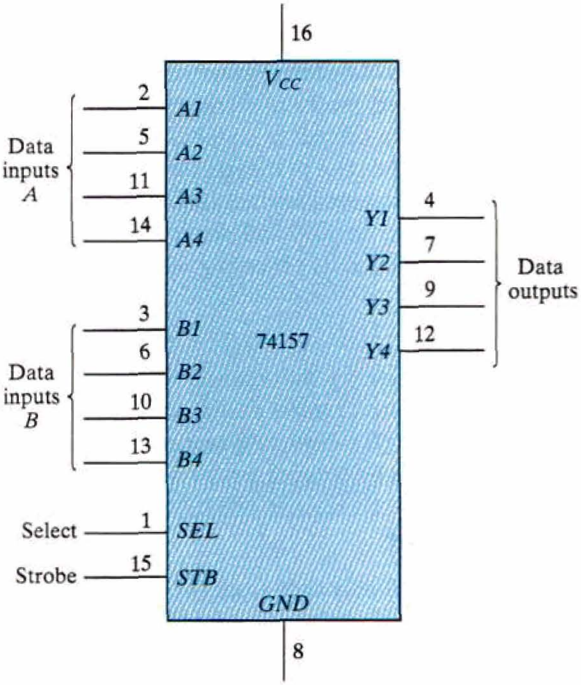
Connect the 74195 as a bidirectional shift register (without parallel load). Connect the serial input for shift right to a toggle switch. Construct the shift left as a ring counter by connecting the serial output *QA* to the serial input *D*. Clear the register and then check its operation by shifting a single 1 from the serial input switch. Shift right three more times and insert 0's from the serial input switch. Then rotate left with the shift-left (load) control. The single 1 should remain visible while shifting.

Bidirectional Shift Register with Parallel Load

The 74195 IC can be converted to a bidirectional shift register with parallel load in conjunction with a multiplexer circuit. We will use IC type 74157 for this purpose. The 74157 is a quadruple two-to-one-line multiplexer whose internal logic is shown in Fig. 4.26. The pin assignments to the inputs and outputs of the 74157 are shown in Fig. 11.17. Note that the enable input is called a strobe in the 74157.

Construct a bidirectional shift register with parallel load using the 74195 register and the 74157 multiplexer. The circuit should be able to perform the following operations:

1. Asynchronous clear
2. Shift right
3. Shift left
4. Parallel load
5. Synchronous clear



Function table

Strobe	Select	Data outputs Y
1	X	All 0's
0	0	Select data inputs A
0	1	Select data inputs B

FIGURE 11.17
IC type 74157 quadruple 2 × 1 multiplexers

Derive a table for the five operations as a function of the clear, clock, and *SH/LD* inputs of the 74195 and the strobe and select inputs of the 74157. Connect the circuit and verify your function table. Use the parallel-load condition to provide an initial value to the register, and connect the serial outputs to the serial inputs of both shifts in order not to lose the binary information while shifting.

11.13 EXPERIMENT 12: SERIAL ADDITION

In this experiment, you will construct and test a serial adder–subtractor circuit. Serial addition of two binary numbers can be done by means of shift registers and a full adder, as explained in Section 6.2.

Serial Adder

Starting from the diagram of Fig. 6.6, design and construct a four-bit serial adder using the following ICs: 74195 (two), 7408, 7486, and 7476. Provide a facility for register B to accept parallel data from four toggle switches, and connect its serial input to ground so that 0's are shifted into register B during the addition. Provide a toggle switch to clear 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. 11.12.) 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.

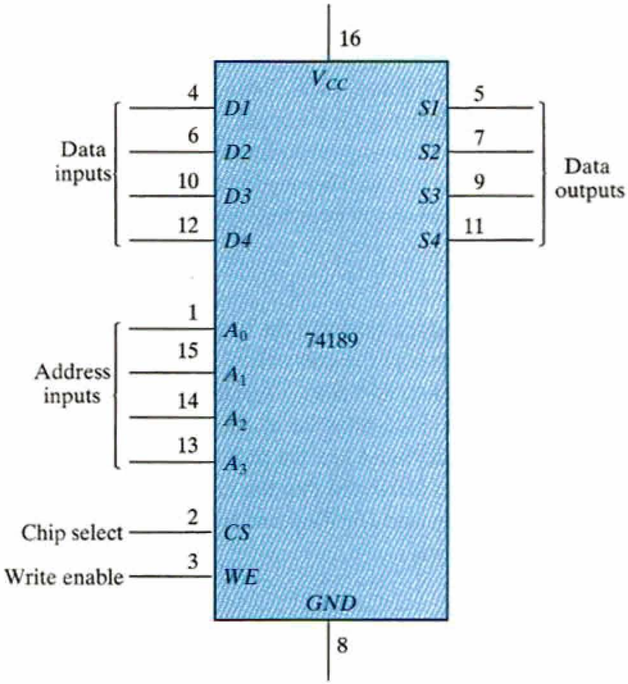
Test the adder part of the circuit by repeating the operations recommended to ensure that the modification did not change the operation. Test the serial subtractor part by performing the subtraction $15 - 4 - 5 - 13 = -7$. Binary 15 can be transferred to register A by first clearing it to 0 and adding 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.

11.14 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 random-access memory. 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. 11.18. The four address inputs select 1 of 16 words in the memory. The least significant bit



Function table

CS	*WE*	Operation	Data outputs
0	0	Write	High impedance
0	1	Read	Complement of selected word
1	X	Disable	High impedance

FIGURE 11.18
IC type 74189 16 × 4 RAM

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 operation is a transfer of the binary number from the data inputs into the selected word in memory. The read operation is performed when $WE = 1$. This operation transfers the complemented value stored in the selected word into the output data lines. The memory has three-state outputs to facilitate memory expansion.

Testing the RAM

Since the outputs of the 74189 produce the complemented values, we need to insert four inverters to change the outputs to their normal value. The RAM can be tested after making the following connections: Connect the address inputs to a binary counter using the 7493 IC (shown in Fig. 11.3). Connect the four data inputs to toggle switches and the 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 defined 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.

11.15 EXPERIMENT 14: LAMP HANDBALL

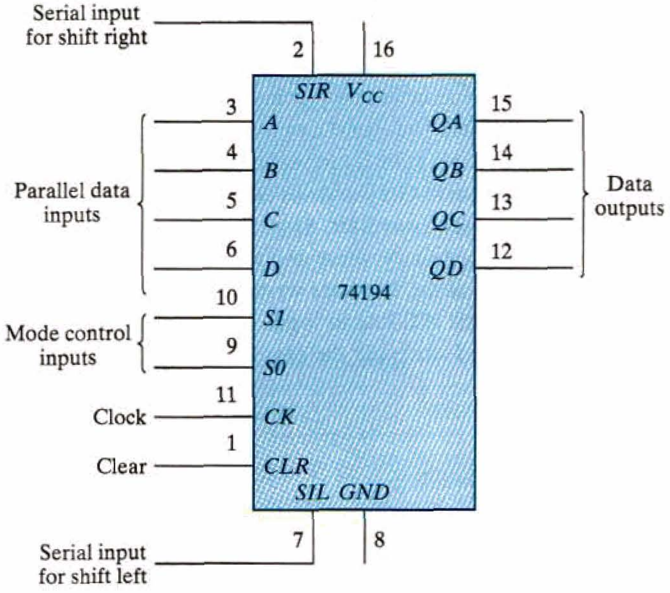
In this experiment, you will construct an electronic game of handball, using a single light to simulate the moving ball. The experiment demonstrates the application of a bidirectional shift register with parallel load. It also shows the operation of the asynchronous inputs of flip-flops. We will first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

IC Type 74194

This is a four-bit bidirectional shift register with parallel load. The internal logic is similar to that shown in Fig. 6.7. The pin assignments to the inputs and outputs are shown in Fig. 11.19. The two mode-control inputs determine the type of operation, as specified in the function table.

Logic Diagram

The logic diagram of the electronic lamp handball game is shown in Fig. 11.20. It consists of two 74194 ICs, a dual *D* flip-flop 7474 IC, and three gate ICs: the 7400, 7404, and 7408. The ball is simulated by a moving light that is shifted left or right through the bidirectional shift register. The rate at which the light moves is determined by the frequency of the clock. The



Function table

Clear	Clock	Mode		Function
		*S*1	*S*0	
0	X	X	X	Clear outputs to 0
1	↑	0	0	No change in output
1	↑	0	1	Shift right in the direction from *QA* to *QD*. *SIR* to *QA*
1	↑	1	0	Shift left in the direction from *QD* to *QA*. *SIL* to *QD*
1	↑	1	1	Parallel-load input data

FIGURE 11.19
IC type 74194 bidirectional shift register with parallel load

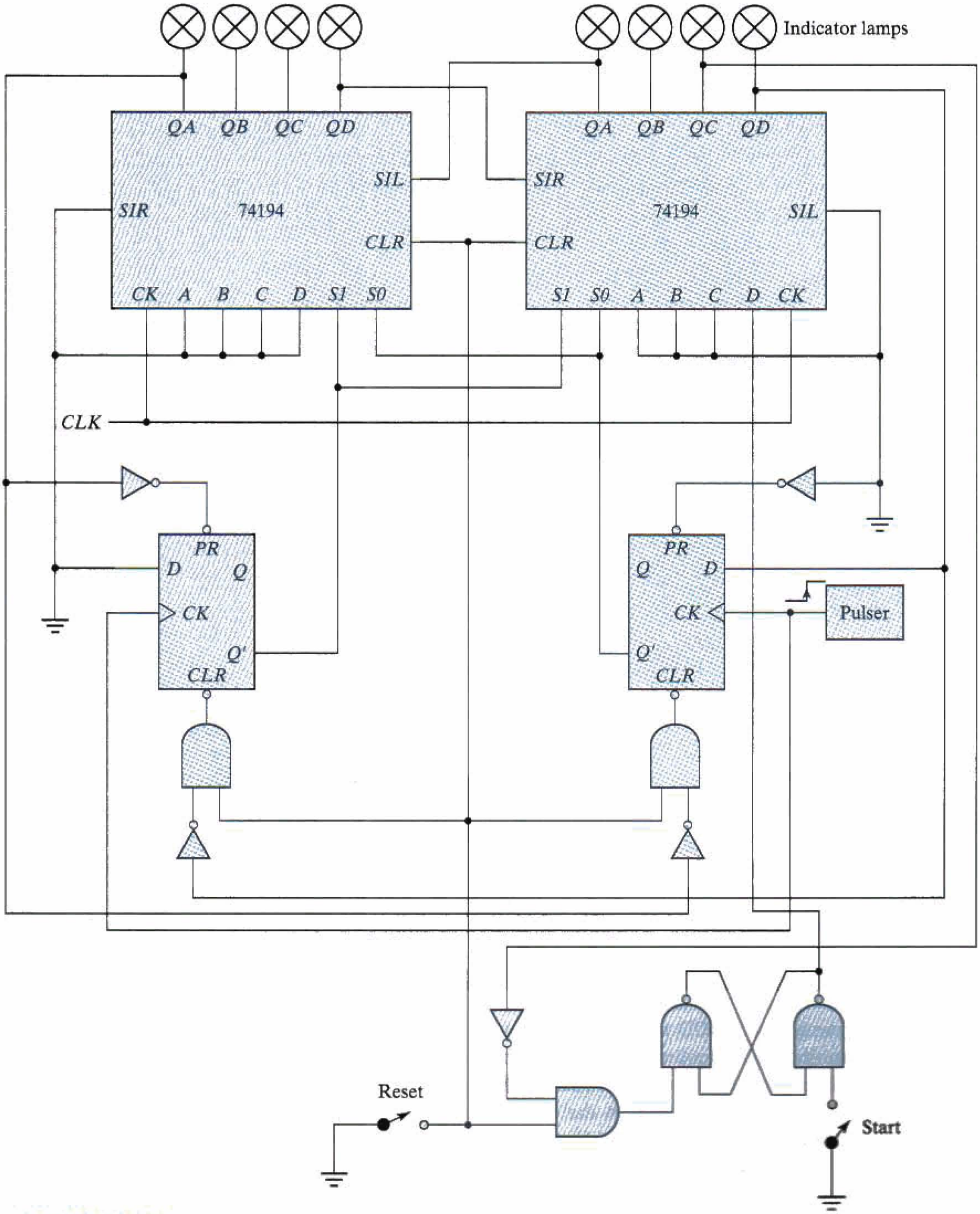


FIGURE 11.20
Lamp handball logic diagram

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. 11.20. 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. 11.8, 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. 11.20 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.

11.16 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 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. 11.21. (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 11.21 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 between two voltage levels at the output of the flip-flop. The output remains high for a duration equal to the charge time. This duration is determined from the equation

$$t_H = 0.693(R_A + R_B)C$$

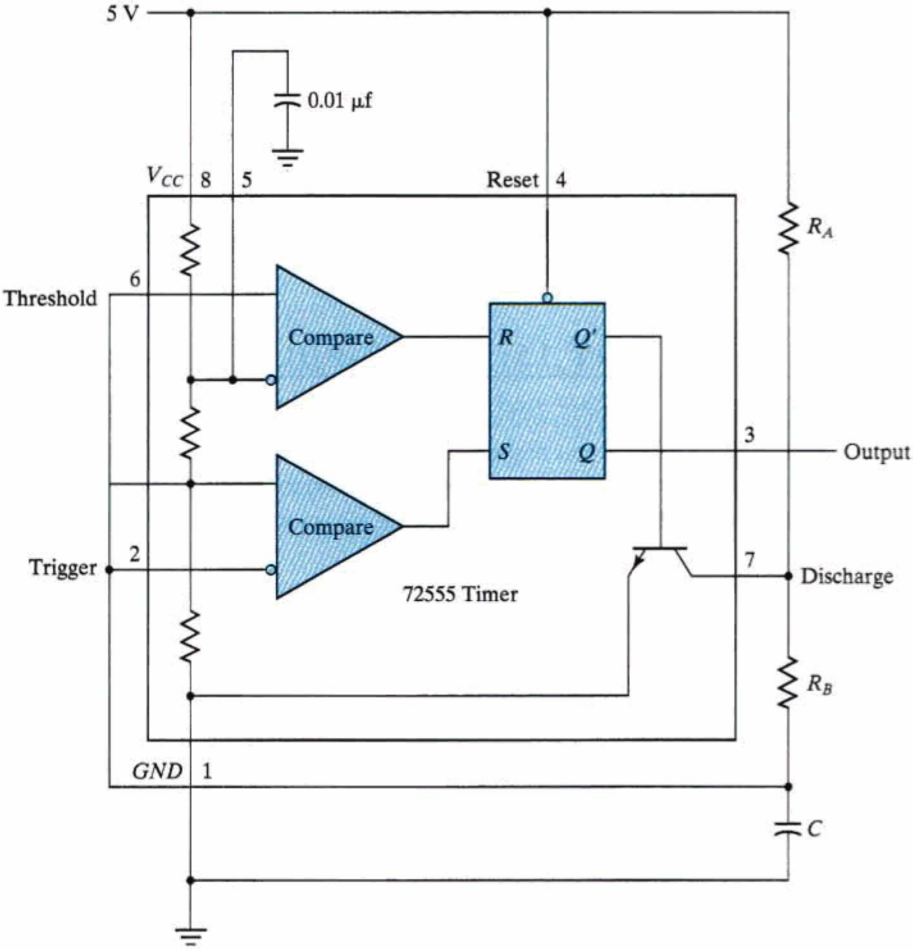


FIGURE 11.21
IC type 72555 timer connected as a clock-pulse generator

The output remains low for a duration equal to the discharge time. This duration is determined from the equation

$$t_L = 0.693 R_B C$$

Clock-Pulse Generator

Starting with a capacitor C of 0.001 μF, calculate values for R_A and R_B to produce clock pulses, as shown in Fig. 11.22. The pulse width is 1 μs in the low level and repeats at a frequency rate of 100 kHz (every 10 μs). Connect the circuit and check the output in the oscilloscope.

Observe the output across the capacitor C, and record its two levels to verify that they are between the trigger and threshold values.

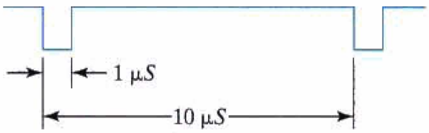


FIGURE 11.22
Output waveform for clock generator

Observe the waveform in the collector of the transistor at pin 7 and record all pertinent information. Explain the waveform by analyzing the circuit's action.

Connect a variable resistor (potentiometer) in series with R_A to produce a variable-frequency pulse generator. The low-level duration remains at 1 μs. The frequency should range from 20 to 100 kHz.

Change the low-level pulses to high-level pulses with a 7404 inverter. This will produce positive pulses of 1 μs with a variable-frequency range.

11.17 EXPERIMENT 16: PARALLEL ADDER AND ACCUMULATOR

In this experiment, you will construct a four-bit parallel adder whose sum can be loaded into a register. The numbers to be added will be stored in a random-access memory. A set of binary numbers will be selected from memory and their sum will be accumulated in the register.

Block Diagram

Use the RAM circuit from the memory experiment of Section 11.14, a four-bit parallel adder, a four-bit shift register with parallel load, a carry flip-flop, and a multiplexer to construct the circuit. The block diagram and the ICs to be used are shown in Fig. 11.23. Information can be written into RAM from data in four switches or from the four-bit data available in the outputs of the register. The selection is done by means of a multiplexer. The data in RAM can be added to the contents of the register and the sum transferred back to the register.

Control of Register

Provide toggle switches to control the 74194 register and the 7476 carry flip-flop as follows:

(a) A LOAD condition transfers the sum to the register and the output carry to the flip-flop upon the application of a clock pulse.

(b) A SHIFT condition shifts the register right with the carry from the carry flip-flop transferred into the leftmost position of the register upon the application of a clock pulse. The value in the carry flip-flop should not change during the shift.

(c) A NO-CHANGE condition leaves the contents of the register and flip-flop unchanged even when clock pulses are applied.

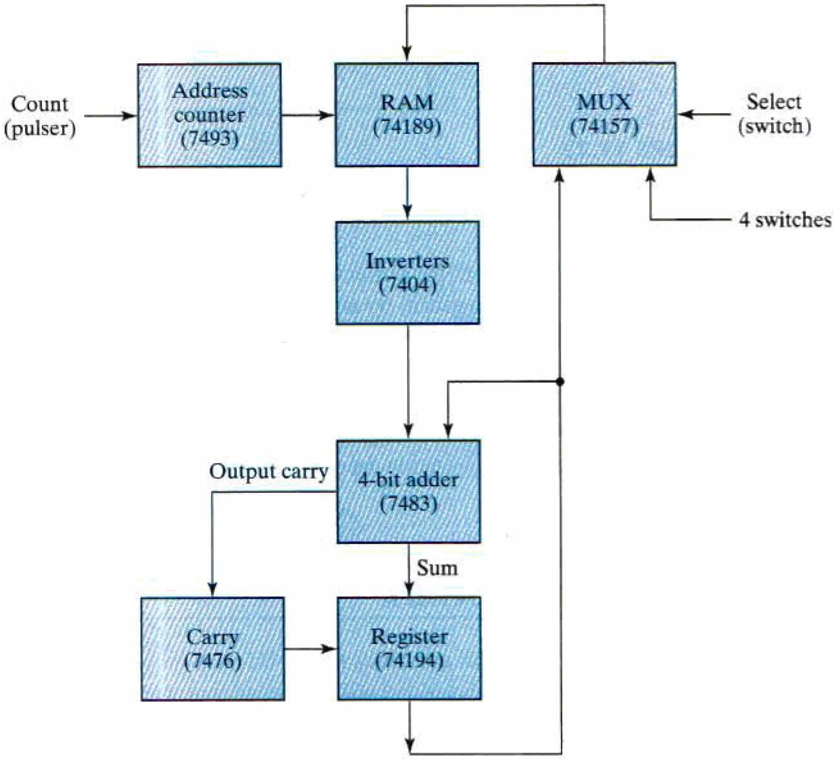


FIGURE 11.23
Block diagram of a parallel adder for Experiment 16

Carry Circuit

To conform with the preceding specifications, it is necessary to provide a circuit between the output carry from the adder and the *J* and *K* inputs of the 7476 flip-flop so that the output carry is transferred into the flip-flop (whether it is equal to 0 or 1) only when the LOAD condition is activated and a pulse is applied to the clock input of the flip-flop. The carry flip-flop should not change if the LOAD condition is disabled or the SHIFT condition is enabled.

Detailed Circuit

Draw a detailed diagram showing all the wiring between the ICs. Connect the circuit, and provide indicator lamps for the outputs of the register and carry flip-flop and for the address and output data of the RAM.

Checking the Circuit

Store the numbers 0110, 1110, 1101, 0101, and 0011 in RAM and then add them to the register one at a time. Start with a cleared register and flip-flop. Predict the values in the output of the register and carry after each addition in the following sum, and verify your results:

$$0110 + 1110 + 1101 + 0101 + 0011$$

Circuit Operation

Clear the register and the carry flip-flop to zero, and store the following four-bit numbers in RAM in the indicated addresses:

Address	Content
0	0110
3	1110
6	1101
9	0101
12	0011

Now perform the following four operations:

1. Add the contents of address 0 to the contents of the register, using the LOAD condition.
2. Store the sum from the register into RAM at address 1.
3. Shift right the contents of the register and carry with the SHIFT condition.
4. Store the shifted contents of the register at address 2 of RAM.

Check that the contents of the first three locations in RAM are as follows:

Address	Contents
0	0110
1	0110
2	0011

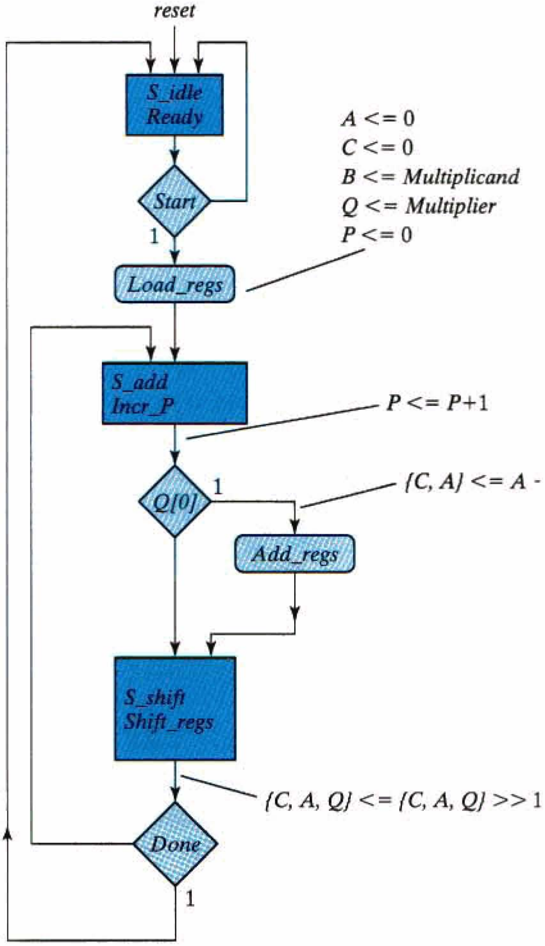
Repeat the foregoing four operations for each of the other four binary numbers stored in RAM. Use addresses 4, 7, 10, and 13 to store the sum from the register in step 2. Use addresses 5, 8, 11, and 14 to store the shifted value from the register in step 4. Predict what the contents of RAM at addresses 0 through 14 would be, and check to verify your results.

11.18 EXPERIMENT 17: BINARY MULTIPLIER

In this experiment, you will design and construct a circuit that multiplies 2 four-bit unsigned numbers to produce an eight-bit product. An algorithm for multiplying two binary numbers is presented in Section 8.7. The algorithm implemented in this experiment differs from the one described in Figures 8.14 and 8.15, by treating only a four-bit datapath and by incrementing, instead of decrementing, a bit counter.

Block Diagram

The ASMD chart and block diagram of the binary multiplier with those ICs recommended to be used are shown in Fig. 11.24(a) and (b). The multiplicand, B, is available from four switches instead of a register. The multiplier, Q, is obtained from another set of four switches. The product is displayed with eight indicator lamps. Counter P is initialized to 0 and then incremented after each partial product is formed. When the counter reaches the count of four, output *Done* becomes 1 and the multiplication operation terminates.

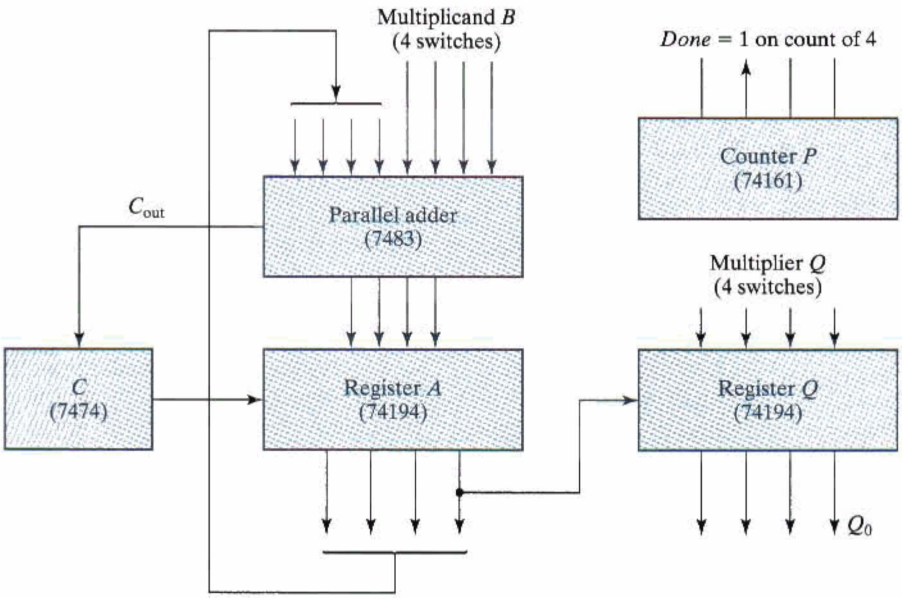


(a) ASMD chart

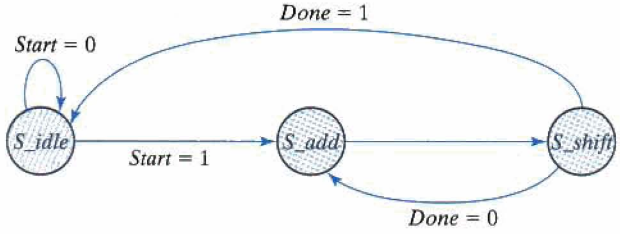
FIGURE 11.24
ASMD chart, block diagram of the datapath, control state diagram, and register operations of the binary multiplier circuit

Control of Registers

The ASMD chart for the binary multiplier in Figure 11.24(a) shows that the three registers and the carry flip-flop of the datapath unit are controlled with signals *Load_regs*, *Incr_P*, *Add_regs*, and *Shift_regs*. The external input signals of the control unit are *clock*, *reset_b* (active-low), and *Start*; another input to the control unit is the internal status signal, *Done*, which is formed by the datapath unit to indicate that the counter has reached a count of four, corresponding to the number of bits in the multiplier. *Load_regs* clears the product register (A) and the carry flip-flop (C), loads the multiplicand into register B, loads the multiplier into register Q, and clears the bit counter. *Incr_P* increments the bit counter concurrently with the accumulation of a partial product. *Add_regs* adds the multiplicand to A, if the least significant bit of the shifted multiplier



(b) Datapath block program



(c) Control state diagram

State Transition		Register Operations	Control signal
From	To		
S_idle		Initial state reached by reset action	
S_idle	*S_add*	$A <= 0, C <= 0, P <= 0$	***Load_regs***
S_add	*S_shift*	$P <= P + 1$ if (*Q[0]*) then ($A <= A + B, C <= C_{out}$)	***Incr_P*** ***Add_regs***
S_shift		shift right {*CAQ*}, $C <= 0$	***Shift_regs***

(d) Register operations

FIGURE 11.24
(Continued)

(Q*[0]*) is 1. Flip-flop *C* accommodates a carry that results from the addition. The concatenated register *CAQ* is updated by storing the result of shifting its contents one bit to the right. *Shift_regs* shifts *CAQ* one bit to the right, which also clears flip-flop *C*.

The state diagram for the control unit is shown in Fig. 11.24(c). Note that it does not show the register operations of the datapath unit or the output signals that control them. That information is apparent in Figure 11.24(d). Note that *Incr_P* and *Shift_regs* are generated unconditionally in states *S_add* and *S_shift*, respectively. *Load_regs* is generated under the condition that *Start* is asserted conditionally while the state is in *S_idle*; *Add_regs* is asserted conditionally in *S_add* if $Q[0] = 1$

Multiplication Example

Before connecting the circuit, make sure that you understand the operation of the multiplier. To do this, construct a table similar to Table 8.5, but with $B = 1111$ for the multiplicand and $Q = 1011$ for the multiplier. Along with each comment listed on the left side of the table, specify the state.

Datapath Design

Draw a detailed diagram of the datapath part of the multiplier, showing all IC pin connections. Generate the four control signals with switches, and use them to provide the required control operations for the various registers. Connect the circuit and check that each component is functioning properly. With the control signals at 0, set the multiplicand switches to 1111 and the multiplier switches to 1011. Assert the control signals manually by means of the control switches, as specified by the state diagram of Fig. 11.24(c). Apply a single pulse while in each control state, and observe the outputs of registers *A* and *Q* and the values in *C* and *P*. Compare these outputs with the numbers in your numerical example to verify that the circuit is functioning properly. Note that IC type 74161 has master–slave flip-flops. To operate it manually, it is necessary that the single clock pulse be a negative pulse.

Design of Control

Design the control circuit specified by the state diagram. You can use any method of control implementation discussed in Section 8.8.

Choose the method that minimizes the number of ICs. Verify the operation of the control circuit prior to its connection to the datapath unit.

Checking the Multiplier

Connect the outputs of the control circuit to the datapath unit, and verify the total circuit operation by repeating the steps of multiplying 1111 by 1011. The single clock pulses should now sequence the control states as well. (Remove the manual switches.) The start signal (*Start*) can be generated with a switch that is on while the control is in state *S_idle*.

Generate the start signal (*Start*) with a pulser or any other short pulse, and operate the multiplier with continuous clock pulses from a clock generator. Pressing the pulser for *Start* should initiate the multiplication operation, and upon its completion, the product should be displayed in the *A* and *Q* registers. Note that the multiplication will be repeated as long as signal *Start* is enabled. Make sure that *Start* goes back to 0. Then set the switches to two other four-bit numbers and press *Start* again. The new product should appear at the outputs. Repeat the multiplication of a few numbers to verify the operation of the circuit.

11.19 EXPERIMENT 18: ASYNCHRONOUS SEQUENTIAL CIRCUITS

In this experiment, you will analyze and design asynchronous sequential circuits. These types of circuits are presented in Chapter 9.

Analysis Example

The analysis of asynchronous sequential circuits with *SR* latches is outlined in Section 9.3. Analyze the circuit of Fig. P9.9 (shown with Problem 9.9) by deriving the transition table and output map of the circuit. From the transition table and output map, determine (a) what happens to output Q when input x_1 is a 1 irrespective of the value of input x_2, (b) what happens to output Q when input x_2 is a 1 and x_1 is equal to 0, and (c) what happens to output Q when both inputs go back to 0?

Connect the circuit and show that it operates according to the way you analyzed it.

Design Example

The circuit of a positive-edge-triggered D-type flip-flop is shown in Fig. 5.10. The circuit of a negative-edge T-type flip-flop is shown in Fig. 9.46. Using the six-step procedure recommended in Section 9.8, design, construct, and test a D-type flip-flop that triggers on both the positive and negative transitions of the clock. The circuit has two inputs—D and C—and a single output, Q. The value of D at the time C changes from 0 to 1 becomes the flip-flop output Q. The output remains unchanged irrespective of the value of D, as long as C remains at 1. On the next clock transition, the output is again updated to the value of D when C changes from 1 to 0. The output then remains unchanged as long as C remains at 0.

11.20 VERILOG HDL SIMULATION EXPERIMENTS AND RAPID PROTOTYPING WITH FPGAS

Field programmable gate arrays (FPGAs) are used by industry to implement logic when the system is complex, the time-to-market is short, the performance (e.g., speed) of an FPGA is acceptable, and the volume of potential sales does not warrant the investment in a standard cell-based ASIC. Circuits can be rapidly prototyped into an FPGA using an HDL. Once the HDL model is verified, the description is synthesized and mapped into the FPGA. FPGA vendors provide software tools for synthesizing the HDL description of a circuit into an optimized gate-level description and mapping (fitting) the resulting netlist into the resources of their FPGA. This process avoids the detailed assembly of ICs that is required by composing a circuit on a breadboard, and the process involves significantly less risk of failure, because it is easier and faster to edit an HDL description than to re-wire a breadboard.

Most of the hardware experiments outlined in this chapter can be supplemented by a corresponding software procedure using the Verilog hardware description language (HDL). A Verilog compiler and simulator are necessary for these supplements. The supplemental experiments have two levels of engagement. In the first, the circuits that are specified in the hands-on laboratory experiments can be described, simulated, and verified using Verilog and a simulator. In the second, if a suitable FPGA prototyping board is available (e.g., see www.digilentinc.com), the hardware experiments can be done by synthesizing the Verilog descriptions and implementing the circuits in an FPGA. Where appropriate, the identity of the individual (structural) hardware units (e.g., a 4-bit counter) can be preserved by encapsulating them in separate Verilog modules whose internal detail is described behaviorally or by a mixture of behavioral and structural models.

Prototyping a circuit with an FPGA requires synthesizing a Verilog description to produce a bit stream that can be downloaded to configure the internal resources (e.g., CLBS of a Xilinx FPGA) and connectivity of the FPGA. Three details require attention: (1) The pins of the prototyping board are connected to the pins of the FPGA, and the hardware implementation of the synthesized circuit requires that its input and output signals be associated with the pins of the prototyping board (this association is made using the synthesis tool provided by the vendor of the FPGA (such tools are available free)), (2) FPGA prototyping boards have a clock generator, but it will be necessary, in some cases, to implement a clock divider (in Verilog) to obtain an internal clock whose frequency is suitable for the experiment, and (3) inputs to an FPGA-based circuit can be made using switches and pushbuttons located on the prototyping board, but it might be necessary to implement a pulser circuit in software to control and observe the activity of a counter or a state machine (see the supplement to Experiment 1).

Supplement to Experiment 1 (Section 11.2)

The functionality of the counters specified in Experiment 1 can be described in Verilog and synthesized for implementation in an FPGA. Note that the circuit shown in Fig. 11.3 uses a push-button pulser or a clock to cause the count to increment in a circuit built with standard ICs. A software pulser circuit can be developed to work with a switch on the prototyping board of an FPGA so that the operation of the counters can be verified by visual inspection.

The software pulser has the ASM chart shown in Fig. 11.25, where the external input (*Pushed*) is obtained from a mechanical switch or pushbutton. This circuit asserts *Start* for one cycle of the clock and then waits for the switch to be opened (or the pushbutton to be released) to ensure that each action of the switch or pushbutton will produce only one pulse of *Start*. If the counter, or a state machine, is in the reset state (*S_idle*) when the switch is closed, the pulse will launch the activity of the counter or state machine. It will be necessary to open the switch (or release the pushbutton) before *Start* can be reasserted. Using the software pulser will allow each value of the count to be observed. If necessary, a simple synchronizer circuit can be used with *Pushed*.

Supplement to Experiment 2 (Section 11.3)

The various logic gates and their propagation delays were introduced in the hardware experiment. In Section 3.10, a simple circuit with gate delays was investigated. As an introduction

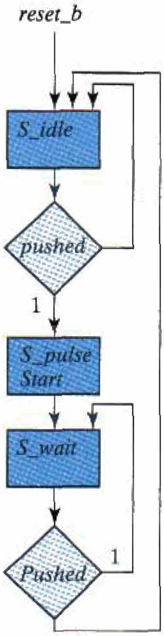


FIGURE 11.25
Pulser circuit for FPGA implementation of Experiment 1

to the laboratory Verilog program, compile the circuit described in HDL Example 3.3 and then run the simulator to verify the waveforms shown in Fig. 3.38.

Assign the following delays to the exclusive-OR circuit shown in Fig. 3.32(a): 10 ns for an inverter, 20 ns for an AND gate, and 30 ns for an OR gate. The input of the circuit goes from $xy = 00$ to $xy = 01$.

(a) Determine the signals at the output of each gate from $t = 0$ to $t = 50$ ns.
(b) Write the HDL description of the circuit including the delays.
(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 11.5)

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 an FPGA and test its operation.

Supplement to Experiment 5 (Section 11.6)

This experiment deals with code conversion. A BCD-to-excess-3 converter was designed in Section 4.4. Use the result 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 11.8)

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. 11.11.

(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.

Supplement to Experiment 8 (Section 11.9)

The edge-triggered *D* flip-flop 7474 is shown in Fig. 11.13. The flip-flop has asynchronous preset and clear inputs.

(a) Write an HDL behavioral description of the 7474 *D* flip-flop, using only the *Q* output. (Note that when $Preset = 0$, *Q* goes to 1, and when $Preset = 1$ and $Clear = 0$, *Q* goes to 0. Thus, *Preset* takes precedence over *Clear*.)

(b) Write an HDL behavioral description of the 7474 *D* flip-flop, using both outputs. Label the second output *Q_not*, and note that this is not always the complement of Q. (When $Preset = Clear = 0$, both *Q* and *Q_not* go to 1.)

Supplement to Experiment 9 (Section 11.10)

In this hardware experiment, you are asked to design and test a sequential circuit whose state diagram is given by Fig. 11.14. This is a Mealy model sequential circuit similar to the one described in HDL Example 5.5 (Section 5.6).

(a) Write the HDL description of the state diagram of Fig. 11.14.

(b) Write the HDL structural description of the sequential circuit obtained from the design. (This is similar to HDL Example 5.7 in Section 5.6.)

(c) Figure 11.24(c) (Section 11.18) shows a control state diagram. Write the HDL description of the state diagram, using the one-hot binary assignment (see Table 5.9 in Section 5.7) and four outputs—T_0, T_1, T_2, and T_3, where T_0 asserts if the state is 4'b0001, T_1 asserts if the state is 4'b0010, etc.

(d) Write a behavioral model of the datapath unit, and verify that the interconnected control unit and datapath unit operate correctly.

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

Supplement to Experiment 10 (Section 11.11)

The synchronous counter with parallel load IC type 74161 is shown in Fig. 11.15. This circuit is similar to the one described in HDL Example 6.3 (Section 6.6), with two exceptions: The load input is enabled when equal to 0, and there are two inputs (*P* and *T*) that control the count. Write the HDL description of the 74161 IC. Implement the counter with an FPGA and test its operation.

Supplement to Experiment 11 (Section 11.12)

A bidirectional shift register with parallel load is designed in this experiment by using the 74195 and 74157 IC types.

(a) Write the HDL description of the 74195 shift register. Assume that inputs J and $\overline{K}$ are connected together to form the serial input.

(b) Write the HDL description of the 74157 multiplexer.

(c) Obtain the HDL description of the four-bit bidirectional shift register that has been designed in this experiment. (1) Write the structural description by instantiating the two ICs and specifying their interconnection, and (2) write the behavioral description of the circuit, using the function table that is derived in this design experiment.

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

Supplement to Experiment 13 (Section 11.14)

This experiment investigates the operation of a random-access memory (RAM). The way a memory is described in HDL is explained in Section 7.2 in conjunction with HDL Example 7.1.

(a) Write the HDL description of IC type 74189 RAM, shown in Fig. 11.18.

(b) Test the operation of the memory by writing a stimulus program that stores binary 3 in address 0 and binary 1 in address 14. Then read the stored numbers from the two addresses to check whether the numbers were stored correctly.

(c) Implement the RAM with an FPGA and test its operation.

Supplement to Experiment 14 (Section 11.15)

(a) Write the HDL behavioral description of the 74194 bidirectional shift register with parallel load shown in Fig. 11.19.

(b) Implement the shift register with an FPGA and test its operation.

Supplement to Experiment 16 (Section 11.17)

A parallel adder with an accumulator register and a memory unit is shown in the block diagram of Fig. 11.23. Write the structural description of the circuit specified by the block diagram. The HDL structural description of this circuit can be obtained by instantiating the various components. An example of a structural description of a design can be found in HDL Example 8.4 in Section 8.6. First, it is necessary to write the behavioral description of each component. Use counter 74161 instead of 7493, and substitute the *D* flip-flop 7474 instead of the *JK* flip-flop 7476. The block diagram of the various components can be found from the list in Table 11.1. Write a test bench for each model, and then write a test bench to verify the entire design. Implement the circuit with an FPGA and test its operation.

Supplement to Experiment 17 (Section 11.18)

The block diagram of a four-bit binary multiplier is shown in Fig. 11.24. The multiplier can be described in one of two ways: (1) by using the register transfer level statements listed in part (b) of the figure or (2) by using the block diagram shown in part (a) of the figure. The description of the multiplier in terms of the register transfer level (RTL) format is carried out in HDL Example 8.5 (Section 8.7).

(a) Use the integrated circuit components specified in the block diagram to write the HDL structural description of the binary multiplier. The structural description is obtained by using the module description of each component and then instantiating all the components to show how they are interconnected. (See Section 8.5 for an example.) The HDL descriptions of the components may be available from the solutions to previous experiments. The 7483 is described with a solution to Experiment 7(a), the 7474 with Experiment 8(a), the 74161 with Experiment 10, and the 74194 with Experiment 14. The description of the control is available from a solution to Experiment 9(c). Be sure to verify each structural unit before attempting to verify the multiplier.

(b) Implement the binary multiplier with an FPGA. Use the pulser described in the supplement to Experiment 1.

Chapter 12

Standard Graphic Symbols

12.1 RECTANGULAR-SHAPE SYMBOLS

Digital components such as gates, decoders, multiplexers, and registers are available commercially in integrated circuits and are classified as SSI or MSI circuits. Standard graphic symbols have been developed for these and other components so that the user can recognize each function from the unique graphic symbol assigned to it. This standard, known as ANSI/IEEE Std. 91-1984, has been approved by industry, government, and professional organizations and is consistent with international standards.

The standard uses a rectangular-shape outline to represent each particular logic function. Within the outline, there is a general qualifying symbol denoting the logical operation performed by the unit. For example, the general qualifying symbol for a multiplexer is MUX. The size of the outline is arbitrary and can be either a square or a rectangular shape with an arbitrary length–width ratio. Input lines are placed on the left and output lines are placed on the right. If the direction of signal flow is reversed, it must be indicated by arrows.

The rectangular-shape graphic symbols for SSI gates are shown in Fig. 12.1. The qualifying symbol for the AND gate is the ampersand (&). The OR gate has the qualifying symbol that designates greater than or equal to 1, indicating that at least one input must be active for the output to be active. The symbol for the buffer gate is 1, showing that only one input is present. The exclusive-OR symbol designates the fact that only one input must be active for the output to be active. The inclusion of the logic negation small circle in the output converts the gates to their complement values. Although the rectangular-shape symbols for the gates are recommended, the standard also recognizes the distinctive-shape symbols for the gates shown in Fig. 2.5.

An example of an MSI standard graphic symbol is the four-bit parallel adder shown in Fig. 12.2. The qualifying symbol for an adder is the Greek letter Σ. The preferred letters for the arithmetic operands are P and Q. The bit-grouping symbols in the two types of inputs and

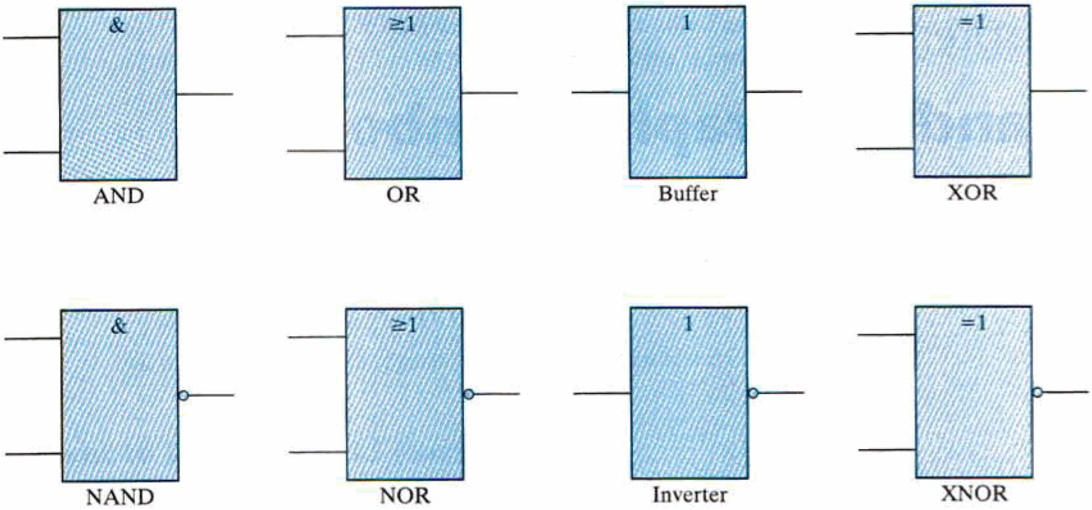


FIGURE 12.1
Rectangular-shape graphic symbols for gates

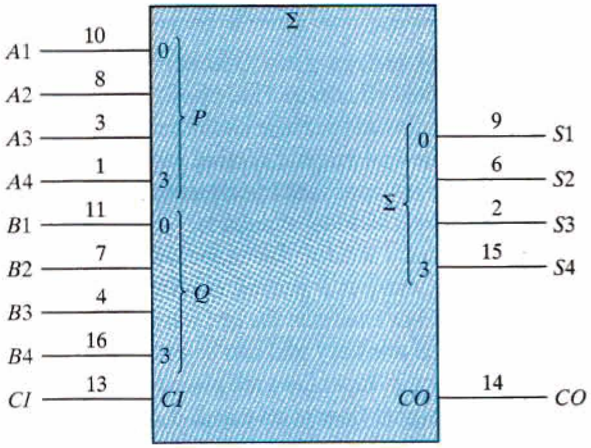


FIGURE 12.2
Standard graphic symbol for a four-bit parallel adder, IC type 7483

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 for the nine inputs and five outputs are shown in Fig. 12.2. The other two pins are for the power supply.

Before introducing the graphic symbols of other components, it is necessary to review some terminology. As mentioned in Section 2.8, a positive-logic system defines the more positive of two signal levels (designated by H) as logic 1 and the more negative signal level (designated by L) as logic 0. Negative logic assumes the opposite assignment. A third alternative is to employ a mixed-logic convention, where the signals are considered entirely in terms of their H and L values. At any point in the circuit, the user is allowed to define the logic polarity by assigning logic 1 to either the H or L signal. The mixed-logic notation uses a small right-angle-triangle graphic symbol to designate a negative-logic polarity at any input or output terminal. (See Fig. 2.10(f).)

Integrated-circuit manufacturers specify the operation of integrated circuits in terms of H and L signals. When an input or output is considered in terms of positive logic, it is defined as *active high*. When it is considered in terms of negative logic, it is defined as *active low*. Active-low inputs or outputs are recognized by the presence of the small-triangle polarity-indicator symbol. When positive logic is used exclusively throughout the entire system, the small-triangle polarity symbol is equivalent to the small circle that designates negation. In this book, we have assumed positive logic throughout and employed the small circle when drawing logic diagrams. When an input or output line does not include the small circle, we define it to be active if it is logic 1. An input or output that includes the small-circle symbol is considered active if it is in the logic-0 state. However, we will use the small-triangle polarity symbol to indicate active-low assignment in all drawings that represent standard diagrams. This will conform with integrated-circuit data books, where the polarity symbol is usually employed. Note that the bottom four gates in Fig. 12.1 could have been drawn with a small triangle in the output lines instead of a small circle.

Another example of a graphic symbol for an MSI circuit is shown in Fig. 12.3. This is a 2-to-4-line decoder representing one-half of IC type 74155. Inputs are on the left and outputs on the right. The identifying symbol X/Y indicates that the circuit converts from code X to code Y. Data inputs A and B are assigned binary weights 1 and 2 equivalent to 2^0 and 2^1, respectively. The outputs are assigned numbers from 0 to 3, corresponding to outputs D_0 through D_3, respectively. The decoder has one active-low input E_1 and one active-high input E_2. These two inputs go through an internal AND 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.

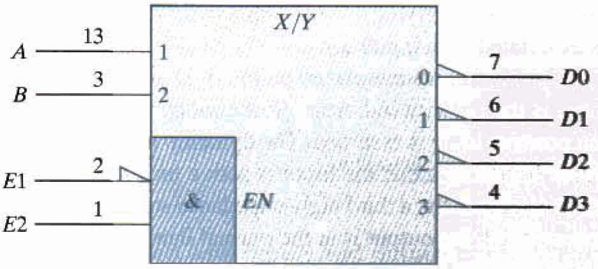


FIGURE 12.3
Standard graphic symbol for a 2-to-4-line decoder (one-half of IC type 74155)

12.2 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 12.1 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.

Table 12.1
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
Π	Multiplier
COMP	Magnitude comparator
ALU	Arithmetic logic unit
SRG	Shift register
CTR	Counter
RCTR	Ripple counter
ROM	Read-only memory
RAM	Random-access memory

Some of the qualifying symbols associated with inputs and outputs are shown in Fig. 12.4. 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 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.

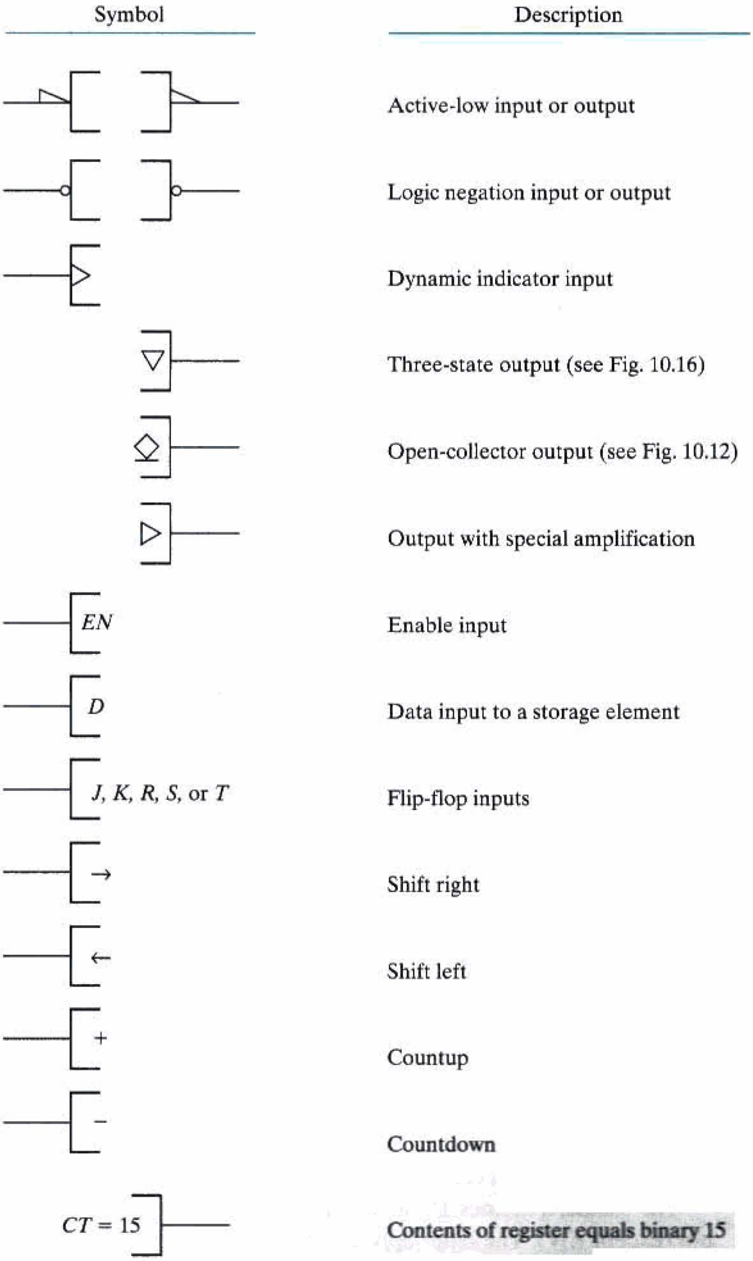

Symbol
Description
Active-low input or output
Logic negation input or output
Dynamic indicator input
Three-state output (see Fig. 10.16)
Open-collector output (see Fig. 10.12)
Output with special amplification
EN
Enable input
D
Data input to a storage element
J, K, R, S, or T
Flip-flop inputs
→
Shift right
←
Shift left
+
Countup
−
Countdown
CT = 15
Contents of register equals binary 15

FIGURE 12.4
Qualifying symbols associated with inputs and outputs

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].

12.3 DEPENDENCY NOTATION

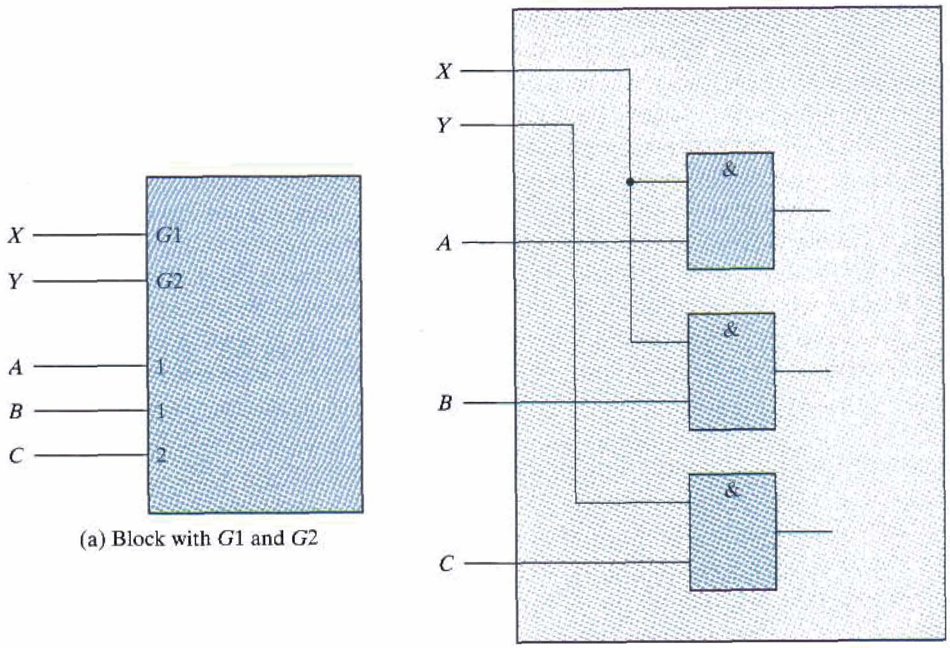
The most important aspect of the standard logic symbols is the dependency notation. Dependency notation is used to provide the means of denoting the relationship between different inputs or outputs without actually showing all the elements and interconnections between them. We will first demonstrate 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. 12.5. 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 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



(a) Block with $G1$ and $G2$

(b) Equivalent interpretation

FIGURE 12.5
Example of G (AND) dependency

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, $EN2$). 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 12.5

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 12.6 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 12.8 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.

12.4 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 11.

The graphic symbols for the adder and decoder were shown in Section 12.2. IC type 74155 can be connected as a 3 × 8 decoder, as shown in Fig. 12.6. (The truth table of this decoder is shown in Fig. 11.7.) 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

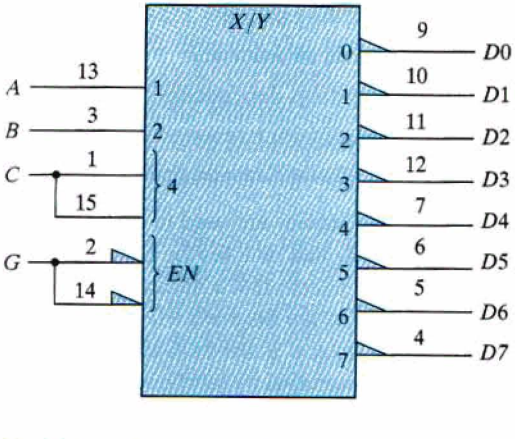


FIGURE 12.6
IC type 74155 connected as a 3 × 8 decoder

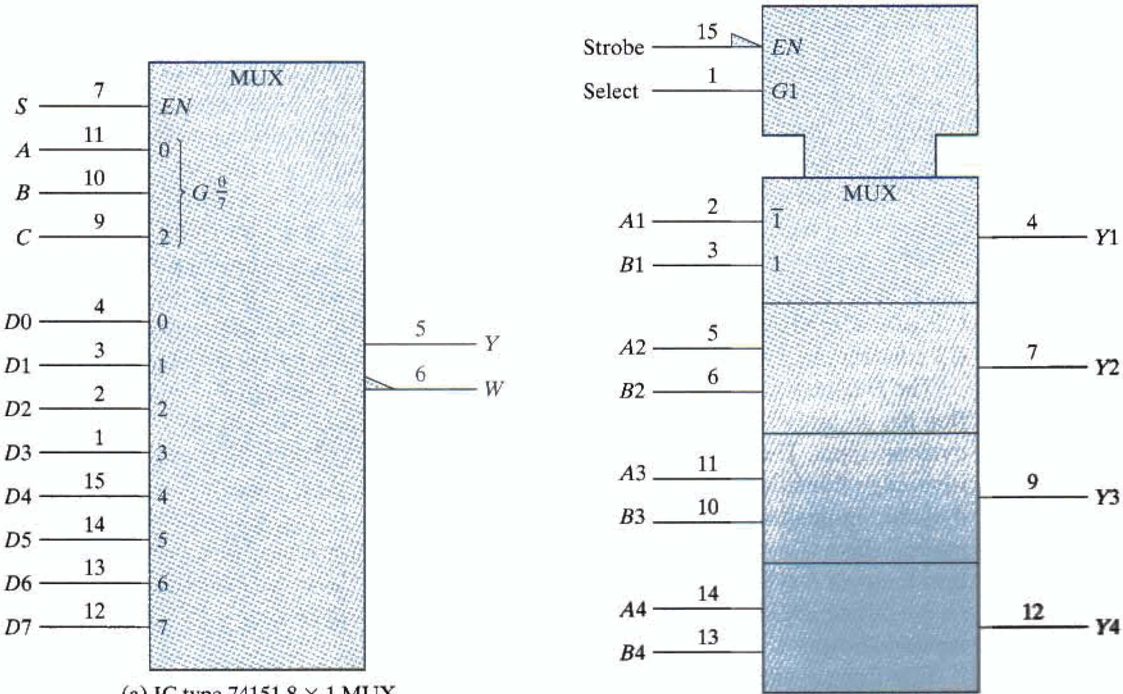
the code name. For example, the 3-to-8-line decoder of Fig. 12.6 can be symbolized with the name *BIN*/*OCT* since the circuit converts a 3-bit binary number into 8 octal values, 0 through 7.

Before showing the graphic symbol for the multiplexer, it is necessary to show a variation of the AND dependency. The AND dependency is sometimes represented by a shorthand notation like $G\frac{0}{7}$. This symbol stands for eight AND dependency symbols from 0 to 7 as follows:

$$G0, G1, G2, G3, G4, G5, G6, G7$$

At any given time, only one out of the eight AND gates can be active. The active AND gate is determined from the inputs associated with the G symbol. These inputs are marked with weights equal to the powers of 2. For the eight AND gates just listed, the weights are 0, 1, and 2, corresponding to the numbers 2^0, 2^1, and 2^2, respectively. The AND gate that is active at any given time is determined from the sum of the weights of the active inputs. Thus, if inputs 0 and 2 are active, then the AND gate that is active has the number $2^0 + 2^2 = 5$. This makes $G5$ active and the other seven AND gates inactive.

The standard graphic symbol for a 8 × 1 multiplexer is shown in Fig. 12.7(a). The qualifying symbol MUX identifies the device as a multiplexer. The symbols inside the block are part of the standard notation, but the symbols marked outside are user-defined symbols. The function table of the 741551 IC can be found in Fig. 11.9. The AND dependency is marked with $G\frac{0}{7}$ and is associated with the inputs enclosed in brackets. These inputs have weights of 0, 1,



(a) IC type 74151 8 × 1 MUX

(b) IC type 74157 quadruple 2 × 1 MUX

FIGURE 12.7
Graphic symbols for multiplexers

and 2. They are actually what we have called the selection inputs. The eight data inputs are marked with numbers from 0 to 7. The net weight of the active inputs associated with the G symbol specifies the number in the data input that is active. For example, if selection inputs $CBA = 110$, then inputs 1 and 2 associated with G are active. This gives a numerical value for the AND dependency of $2^2 + 2^1 = 6$, which makes $G6$ active. Since $G6$ is ANDed with data input number 6, it makes this input active. Thus, the output will be equal to data input D_6 provided that the enable input is active.

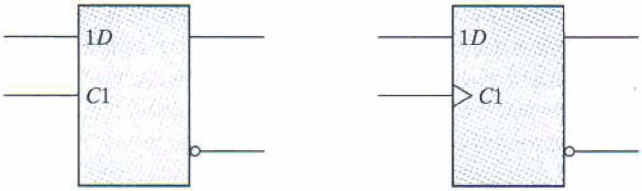
Fig.12.7(b) represents the quadruple 2 × 1 multiplexer IC type 74157 whose function table is listed in Fig. 11.17. The enable and selection inputs are common to all four multiplexers. This is indicated in the standard notation by the indented box at the top of the diagram, which represents a *common control block.* The inputs to a common control block control all lower sections of the diagram. The common enable input EN is active when in the low-level state. The AND dependency, $G1$, determines which input is active in each multiplexer section. When $G1 = 0$, the A inputs marked with $\overline{1}$ are active. When $G1 = 1$, the B inputs marked with 1 are active. The active inputs are applied to the corresponding outputs if EN is active. Note that the input symbols $\overline{1}$ and 1 are marked in the upper section only instead of repeating them in each section.

12.5 SYMBOLS FOR FLIP-FLOPS

The standard graphic symbols for different types of flip-flops are shown in Fig. 12.8. A flip-flop is represented by a rectangular-shaped block with inputs on the left and outputs on the right. One output designates the normal state of the flip-flop and the other output with a small-circle negation symbol (or polarity indicator) designates the complement output. The graphic symbols distinguish between three types of flip-flops: the D latch, whose internal construction is shown in Fig. 6.5; the master–slave flip-flop, shown in Fig. 6.9; and the edge-triggered flip-flop, introduced in Fig. 6.12. The graphic symbol for the D latch or D flip-flop has inputs D and C indicated inside the block. The graphic symbol for the JK flip-flop has inputs J, K, and C inside. The notation $C1$, $1D$, $1J$, and $1K$ are examples of control dependency. The input in $C1$ controls input $1D$ in a D flip-flop and inputs $1J$ and $1K$ in a JK flip-flop.

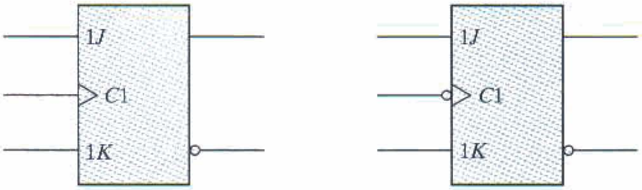
The D latch has no other symbols besides the $1D$ and $C1$ inputs. The edge-triggered flip-flop has an arrowhead-shaped symbol in front of the control dependency $C1$ to designate a dynamic input. The dynamic indicator symbol denotes that the flip-flop responds to the positive-edge transition of the input clock pulses. A small circle outside the block along the dynamic indicator designates a negative-edge transition for triggering the flip-flop. The master–slave is considered to be a pulse-triggered flip-flop and is indicated as such with an upside-down L symbol in front of the outputs. This is to show that the output signal changes on the falling edge of the pulse. Note that the master–slave flip-flop is drawn without the dynamic indicator.

Flip-flops available in integrated-circuit packages provide special inputs for setting and resetting the flip-flop asynchronously. These inputs are usually called direct set and direct reset. They affect the output on the negative level of the signal without the need of a clock. The graphic symbol of a master–slave JK flip-flop with direct set and reset is shown in Fig. 12.9(a).



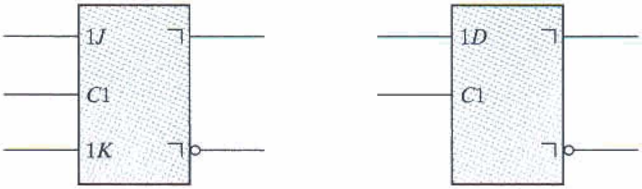
D latch

Positive-edge-triggered *D* flip-flop



Positive-edge-triggered *JK* flip-flop

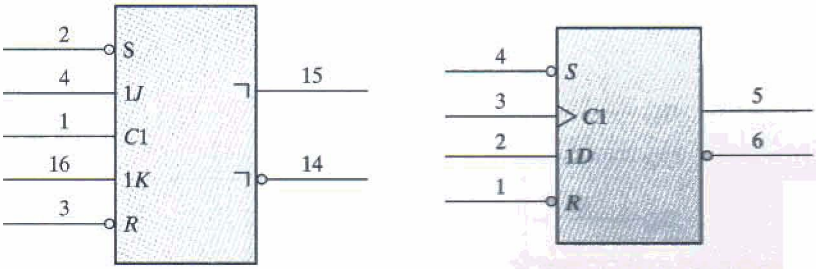
Negative-edge-triggered *JK* flip-flop



Master-slave *JK* flip-flop

Master-slave *D* flip-flop

FIGURE 12.8
Standard graphic symbols for flip-flops



(a) One-half 7476 *JK* flip-flop

(b) One-half 7474 *D* flip-flop

FIGURE 12.9
IC flip-flops with direct set and reset

The notations $C1$, $1J$, and $1K$ represent control dependency, showing that the clock input at $C1$ controls inputs $1J$ and $1K$. S and R have no 1 in front of the letters and, therefore, they are not controlled by the clock at $C1$. The S and R inputs have a small circle along the input lines to indicate that they are active when in the logic-0 level. The function table for the 7476 flip-flop is shown in Fig. 11.12.

The graphic symbol for a positive-edge-triggered D flip-flop with direct set and reset is shown in Fig. 12.9(b). The positive-edge transition of the clock at input $C1$ controls input $1D$. The S and R inputs are independent of the clock. This is IC type 7474, whose function table is listed in Fig. 11.13.

12.6 SYMBOLS FOR REGISTERS

The standard graphic symbol for a register is equivalent to the symbol used for a group of flip-flops with a common clock input. Fig. 12.10 shows the standard graphic symbol of IC type 74175, consisting of four D flip-flops with common clock and clear inputs. The clock input $C1$ and the clear input R appear in the common control block. The inputs to the common control block are connected to each of the elements in the lower sections of the diagram. The notation $C1$ is the control dependency that controls all the $1D$ inputs. Thus, each flip-flop is triggered

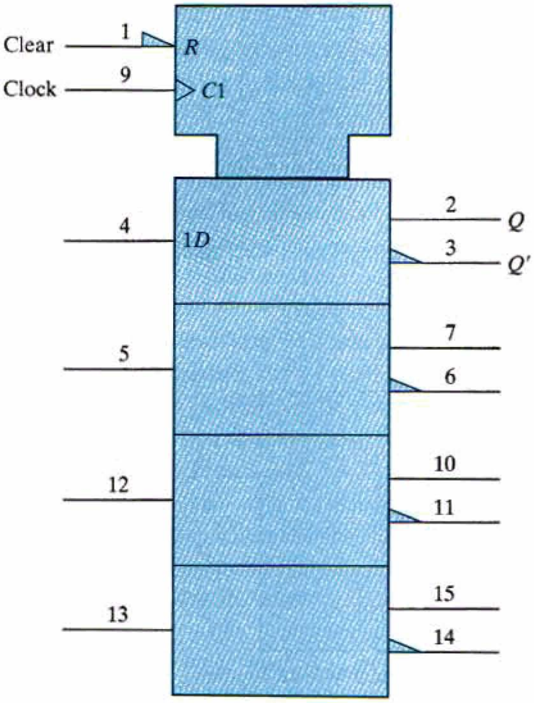


FIGURE 12.10
Graphic symbol for a four-bit register, IC type 74175

by the common clock input. The dynamic input symbol associated with $C1$ indicates that the flip-flops are triggered on the positive edge of the input clock. The common R input resets all flip-flops when its input is at a low-level state. The $1D$ symbol is placed only once in the upper section instead of repeating it in each section. The complement outputs of the flip-flops in this diagram are marked with the polarity symbol rather than the negation symbol.

The standard graphic symbol for a shift register with parallel load is shown in Fig. 12.11. This is IC type 74195, whose function table can be found in Fig. 11.16. The qualifying symbol for a shift register is SRG followed by a number that designates the number of stages. Thus, $SRG4$ denotes a four-bit shift register. The common control block has two mode dependencies, $M1$ and $M2$, for the shift and load operations, respectively. Note that the IC has a single input labeled SH/LD (shift/load), which is split into two lines to show the two modes. $M1$ is active when the SH/LD input is high and $M2$ is active when the SH/LD input is low. $M2$ is recognized as active low from the polarity indicator along its input line. Note the convention in this symbology: We must recognize that a single input actually exists in pin 9, but it is split into two parts in order to assign to it the two modes, $M1$ and $M2$. The control dependency $C3$ is for the clock input. The dynamic symbol along the $C3$ input indicates that the flip-flops trigger on the positive edge of the clock. The symbol $/1 \rightarrow$ following $C3$ indicates that the register shifts to the right or in the downward direction when mode $M1$ is active.

The four sections below the common control block represent the four flip-flops. Flip-flop QA has three inputs: Two are associated with the serial (shift) operation and one with the

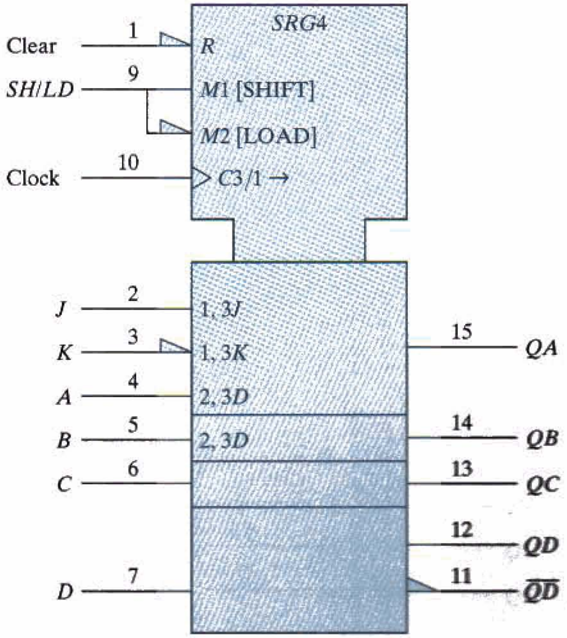


FIGURE 12.11
Graphic symbol for a shift register with parallel load, IC type 74195

parallel (load) operation. The serial input label 1, 3 J indicates that the J input of flip-flop QA is active when $M1$ (shift) is active and $C3$ goes through a positive clock transition. The other serial input with label 1, 3 K has a polarity symbol in its input line corresponding to the complement of input K in a JK flip-flop. The third input of QA and the inputs of the other flip-flops are for the parallel input data. Each input is denoted by the label 2, 3D. The 2 is for $M2$ (load), and 3 is for the clock $C3$. If the input in pin number 9 is in the low level, $M1$ is active, and a positive transition of the clock at $C3$ causes a parallel transfer from the four inputs, A through D, into the four flip-flops, QA through QD. Note that the parallel input is labeled only in the first and second sections. It is assumed to be in the other two sections below.

Figure 12.12 shows the graphic symbol for the bidirectional shift register with parallel load, IC type 74194. The function table for this IC is listed in Fig. 11.19. The common control block shows an R input for resetting all flip-flops to 0 asynchronously. The mode select has two inputs and the mode dependency M may take binary values from 0 to 3. This is indicated by the symbol $M\frac{0}{3}$, which stands for $M0$, $M1$, $M2$, $M3$, and is similar to the notation for the G dependency in multiplexers. The symbol associated with the clock is

$$C4/1 \rightarrow \quad /2 \leftarrow$$

$C4$ is the control dependency for the clock. The $/1 \rightarrow$ symbol indicates that the register shifts right (down in this case) when the mode is $M1$ $(S_1S_0 = 01)$. The $/2 \leftarrow$ symbol indicates that the register shifts left (up in this case) when the mode is M_2 $(S_1S_0 = 10)$. The right and left directions are obtained when the page is turned 90 degrees counterclockwise.

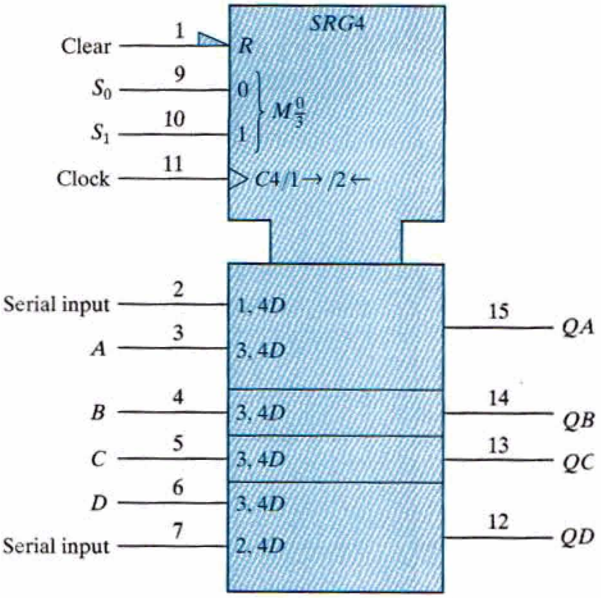


FIGURE 12.12

Graphic symbol for a bidirectional shift register with parallel load, IC type 74194

The sections below the common control block represent the four flip-flops. The first flip-flop has a serial input for shift right, denoted by $1, 4D$ (mode $M1$, clock $C4$, input D). The last flip-flop has a serial input for shift left, denoted by $2, 4D$ (mode $M2$, clock $C4$, input D). All four flip-flops have a parallel input denoted by the label $3, 4D$ (mode $M3$, clock $C4$, input D). Thus, $M3$ $(S_1S_0 = 11)$ is for parallel load. The remaining mode $M0$ $(S_1S_0 = 00)$ has no effect on the outputs because it is not included in the input labels.

12.7 SYMBOLS FOR COUNTERS

The standard graphic symbol of a binary ripple counter is shown in Fig. 12.13. The qualifying symbol for a ripple counter is *RCTR*. The designation *DIV* 2 stands for the divide-by-2 circuit that is obtained from the single flip-flop *QA*. The *DIV* 8 designation is for the divide-by-8 counter obtained from the other three flip-flops. The diagram represents IC type 7493, whose internal circuit diagram is shown in Fig. 11.2. The common control block has an internal AND gate, with inputs $R1$ and $R2$. When both of these inputs are equal to 1, the content of the counter goes to zero. This is indicated by the symbol $CT = 0$. Since the count input does not go to the clock inputs of all flip-flops, it has no $C1$ label and, instead, the symbol + is used to indicate a count-up operation. The dynamic symbol next to the + together with the polarity symbol along the input line signify that the count is affected with a negative-edge transition of the input signal. The bit grouping from 0 to 2 in the output represents values for the weights to the power of 2. Thus, 0 represents the value of $2^0 = 1$ and 2 represents the value $2^2 = 4$.

The standard graphic symbol for the four-bit counter with parallel load, IC type 74161, is shown in Fig. 12.14. The qualifying symbol for a synchronous counter is *CTR* followed by the symbol *DIV* 16 (divide by 16), which gives the cycle length of the counter. There is a single

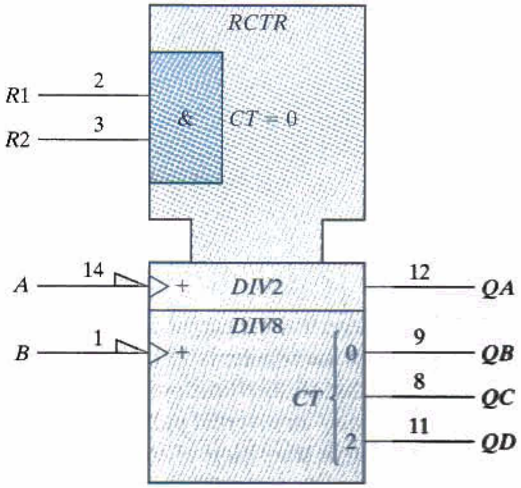


FIGURE 12.13
Graphic symbol for ripple counter, IC type 7493

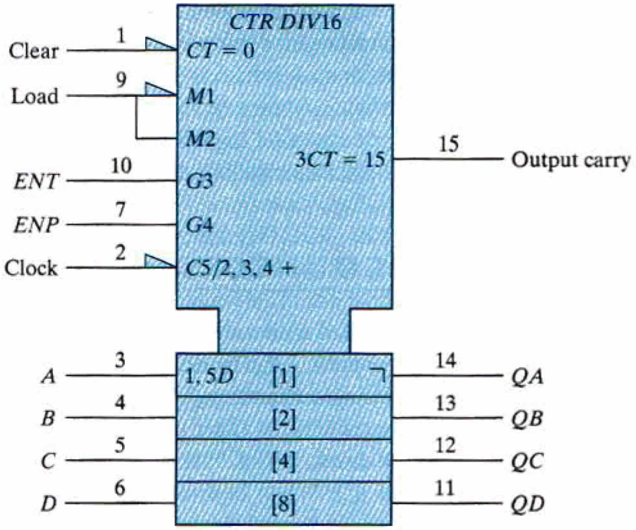


FIGURE 12.14
Graphic Symbol for 4-Bit Binary Counter with Parallel Load, IC Type 74161

load input at pin 9 that is split into the two modes, $M1$ and $M2$. $M1$ is active when the load input at pin 9 is low and $M2$ is active when the load input at pin 9 is high. $M1$ is recognized as active low from the polarity indicator along its input line. The count-enable inputs use the G dependencies. $G3$ is associated with the T input and $G4$ with the P input of the count enable. The label associated with the clock is

$$C5/2, 3, 4\ +$$

This means that the circuit counts up (the + symbol) when $M2$, $G3$, and $G4$ are active (load $= 1$, $ENT = 1$, and $ENP = 1$) and the clock in $C5$ goes through a positive transition. This condition is specified in the function table of the 74161 listed in Fig. 11.15. The parallel inputs have the label $1, 5D$, meaning that the D inputs are active when $M1$ is active (load $= 0$) and the clock goes through a positive transition. The output carry is designated by the label

$$3CT = 15$$

This is interpreted to mean that the output carry is active (equal to 1) if $G3$ is active $(ENT = 1)$ and the content (CT) of the counter is 15 (binary 1111). Note that the outputs have an inverted L symbol, indicating that all the flip-flops are of the master–slave type. The polarity symbol in the $C5$ input designates an inverted pulse for the input clock. This means that the master is triggered on the negative transition of the clock pulse and the slave changes state on the positive transition. Thus, the output changes on the positive transition of the clock pulse. It should be noted that IC type 74LS161 (low-power Schottky version) has positive-edge-triggered flip-flops.

12.8 SYMBOL FOR RAM

The standard graphic symbol for the random-access memory (RAM) 74189 is shown in Fig. 12.15. The numbers 16×4 that follow the qualifying symbol RAM designate the number of words and the number of bits per word. The common control block is shown with four address lines and two control inputs. Each bit of the word is shown in a separate section with an input and output data line. The address dependency A is used to identify the address inputs of the memory. Data inputs and outputs affected by the address are labeled with the letter A. The bit grouping from 0 through 3 provides the binary address that ranges from $A0$ through $A15$. The inverted triangle signifies three-state outputs. The polarity symbol specifies the inversion of the outputs.

The operation of the memory is specified by means of the dependency notation. The RAM graphic symbol uses four dependencies: A (address), G (AND), EN (enable), and C (control). Input $G1$ is to be considered ANDed with $1EN$ and $1C2$ because $G1$ has a 1 after the letter G and the other two have a 1 in their label. The EN dependency is used to identify an enable input that controls the data outputs. The dependency $C2$ controls the inputs as indicated by the $2D$ label. Thus, for a write operation, we have the $G1$ and $1C2$ dependency ($CS = 0$), the $C2$ and $2D$ dependency ($WE = 0$), and the A dependency, which specifies the binary address in the four address inputs. For a read operation, we have the $G1$ and $1EN$ dependencies ($CS = 0, WE = 1$) and the A dependency for the outputs. The interpretation of these dependencies results in the operation of the memory as listed in the function table of Fig. 11.18.

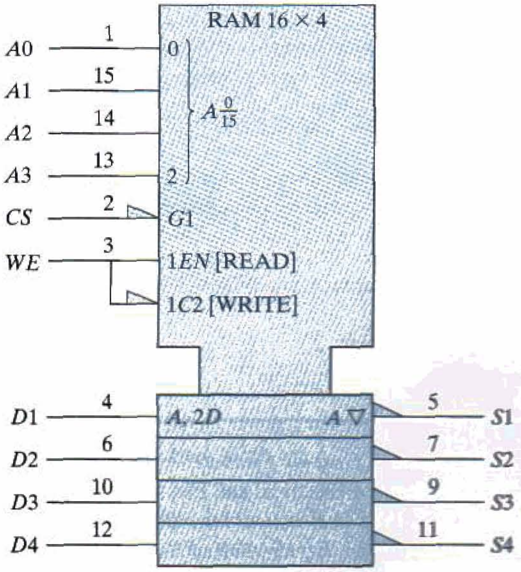


FIGURE 12.15
Graphic symbol for 16 × 4 RAM, IC type 74189

PROBLEMS

12.1 Figure 11.1 shows various small-scale integration circuits with pin assignment. Using this information, draw the rectangular-shaped graphic symbols for the 7400, 7404, and 7486 ICs.

12.2 Define the following in your own words:

(a) Positive and negative logic.
(b) Active high and active low.
(c) Polarity indicator.
(d) Dynamic indicator.
(e) Dependency notation.

12.3 Show an example of a graphic symbol that has the three Boolean dependencies—G, V, and N. Draw the equivalent interpretation.

12.4 Draw the graphic symbol of a BCD-to-decimal decoder. This is similar to a decoder with 4 inputs and 10 outputs.

12.5 Draw the graphic symbol for a binary-to-octal decoder with three enable inputs, $E1$, $E2$, and $E3$. The circuit is enabled if $E1 = 1$, $E2 = 0$, and $E3 = 0$ (assuming positive logic).

12.6 Draw the graphic symbol of dual 4-to-1-line multiplexers with common selection inputs and a separate enable input for each multiplexer.

12.7 Draw the graphic symbol for the following flip-flops:

(a) Negative-edge-triggered D flip-flop.
(b) Master–slave RS flip-flop.
(c) Positive-edge-triggered T flip-flop.

12.8 Explain the function of the common control block when used with the standard graphic symbols.

12.9 Draw the graphic symbol of a four-bit register with parallel load using the label $M1$ for the load input and $C2$ for the clock.

12.10 Explain all the symbols used in the standard graphic diagram of Fig. 12.12.

12.11 Draw the graphic symbol of an up–down synchronous binary counter with mode input (for up or down) and count-enable input with G dependency. Show the output carries for the up count and the down count.

12.12 Draw the graphic symbol of a 256 × 1 RAM. Include the symbol for three-state outputs.

REFERENCES

1. *IEEE Standard Graphic Symbols for Logic Functions* (ANSI/IEEE Std. 91-1984). 1984. New York: Institute of Electrical and Electronics Engineers.
2. KAMPEL, I. 1985. *A Practical Introduction to the New Logic Symbols.* Boston: Butterworth.
3. MANN, F. A. 1984. *Explanation of New Logic Symbols.* Dallas: Texas Instruments.
4. *The TTL Data Book,* Volume 1. 1985. Dallas: Texas Instruments.

Asynchronous sequential circuits *(cont.)*
  design of, 415–416
  flow table, 420–422
  fundamental mode, 417
  hazards, 452–457
    in combinational circuits, 452–454
    defined, 452
    detection of, 453
    essential, 456–457
    implementation with SR latches, 454–456
    remedy for eliminating, 454
    in sequential circuits, 454
  implementation example, 430–431
  laboratory experiment, 553
  latch excitation table, 430
  logic diagram, 436–437
  primitive flow table, 433–435
    reduction of, 435–436
  race conditions, 422–424
  race-free state assignment, 446–452
    four-row flow-table example, 449–450
    multiple-row method, 450–452
    three-row flow-table example, 447–449
  reduction of state and flow tables, 439–446
    closed-covering condition, 445–446
    compatible pairs, 443–444
    implication table and implied states, 440–442
    maximal compatibles, 444–445
    merging of the flow table, 442–443
  SR latch, 425–427, 430–431
    analysis example, 428–429
    transition table, 430
  stability, 424–425
  transition table, 417–420, 436–437
  unstable states:
    assigning outputs to, 437–439
Asynchronous sequential logic, 415–470

B

Base, 3–4
BCD 841, 21–22
BCD adder, 140–142
  block diagram, 141
BCD addition, 19–20
BCD (binary-coded decimal) code, 18–19
BCD ripple counters, 256–258
  logic diagram, 257
  state diagram, 256
Behavioral modeling, 168–170, 207–210
Bidirectional shift register, 251, 537, 537–538
  with parallel load, 537–538
Binary adder, 133–134
Binary adder–subtractor, 129–139
  binary adder, 133–134
  binary subtractor, 136–138
  carry propagation, 134–136
  full adder, 131–133
  half adder, 130–131
  overflow, 138–139
Binary and decimal numbers (laboratory experiment), 516
  BCD count, 517–518
  binary count, 516–517
  counts, 518
  oscilloscope display, 517
  output pattern, 517–518
Binary cell, 25
Binary codes, 1, 17–25
  American Standard Code for Information Interchange (ASCII), 23–25
  BCD 841, 21–22
  BCD addition, 19–20
  BCD (binary-coded decimal) code, 18–19
  decimal arithmetic, 20–21
  error-detecting code, 25
  Excess-3 code, 21–22
  Gray code, 22–23
  n-bit binary code, 17–18
  2421 code, 21–22
  weighted codes, 21
Binary countdown counters, 255
Binary counter with parallel load, 262–265
  Clear input, 262–264
  CLK input, 264
  Count input, 264
  Load input, 264
Binary counters:
  defined, 253
  with parallel load, 534–535
Binary decision box, algorithmic state machines (ASMs), 350
Binary digits, 1
Binary logic, 28–31
  defined, 29
  logic gates, 30–31
Binary multiplier, 142–143
  control state diagram for, 376
  HDL description of, 382–389
    datapath unit, 382
    next-state logic of the controller, 382
  laboratory experiment, 549–553
    block diagram, 549–550
    checking, 552–553
    control of registers, 550–552
    datapath design, 552
    design of control circuit, 552
    multiplication example, 552
  parallel multiplier, behavioral description of, 388–390
  testing, 384–388
Binary numbers, 3–5
  conversion to octal numbers, 9
  signed, 14–17
  sum of, 5
  unsigned, 14
Binary operator, 36
Binary ripple counters, 253–256
  binary count sequence, 255
  defined, 253
  four-bit, 254
Binary storage and registers, 25–28
Binary storage cell, 291–292
Binary subtractor, 136–138
Binary systems, 3–4
Bipolar IC transistors, 477
Bipolar junction transistor (BJT), 473
Bipolar transistors:
  base–emitter graphical characteristic, 479
  characteristics of, 477–481
  collector and base currents, 478
  dc current gain, 479
  graphical collector–emitter characteristics, 479
  pulled down output, 479
  pulled out output, 479
  saturation region, 479
Bits, 1, 4, 285
Bitwise operators, 338
Block statement, 112
Blocking assignments, 209–210, 337
Boole, George, 38
Boolean algebra, 28, 38
  associative law, 37
  axiomatic definition of, 38–39
  basic theorems, 41–43
  binary operator, 36
  canonical forms, 48–55
    conversion between, 52–53
  closure, 37
  commutative law, 37
  defined, 36, 44
  distributive law, 37
  duality, 41
  field, 37–38
  identity element, 37
  inverse, 37
  and logic gates, 30–31, 36–39
  manipulation of, 46–47
  maxterms, 48–50
    product of, 52

- Control unit, 345–346
- Controllers, 335
- Count operation registers, 334
- Counters:
  - binary countdown counters, 255
  - defined, 242, 253
  - divide-by-*N* counter, 265
  - HDL for, 269–276
  - Johnson, 268–269
  - laboratory experiment, 533–535
    - binary counter with parallel load, 534–535
    - decimal counter, 534
    - ripple counter, 534
    - synchronous counter, 534
  - ring, 267–268
  - ripple, 253–258
    - BCD, 256–258
    - binary, 253–256
  - symbols, 572–574
  - synchronous, 258–264
  - with unused states, 265–266
- Critical race, 422–423
  - avoiding, 447
  - examples of, 423
- Crosspoint, 300
- Cycle time, memory, 289
- Cycles, 423–424
- Cyclic behavior, 207, 343
  - edge-sensitive, 344

D

- *D* flip-flops:
  - advantage of designing with, 228
  - analysis of clocked sequential circuits with, 204–206
  - analysis with, 200–201
  - characteristic equation, 194
  - characteristic table, 193–194
  - as example of a sequential machine, 210–212
- *D* latch (transparent latch), 187–188
- Darlington pair, 490–491
- Data selector, 154
- Dataflow modeling, 165–167
- *DataIn*, 288
- *DataOut*, 288
- Datapath unit, 345–346
- Data-processing path, 345–346
- Dc current gain, 479
- Debounce circuit, 431
- Decade counter, 256
- Decimal adder, 139–142
  - BCD adder, 140–142
- Decimal arithmetic, 20–21
- Decimal counter, 534
- Declaration, 110
- Decoders, 146–149, 566
  - combinational logic implementation, 149
  - *n*-to-*m*-line decoders, 146
  - three-to-eight-line, 146
    - truth table for, 147
  - two-to-four-line, with enable input, 147–148
- Delay control operator, 208
- DeMorgan's theorem, 43, 47–48
- Demultiplexer:
  - defined, 148
  - with enable input, 147–148
- Dependency notation, 564–566
- Design:
  - asynchronous sequential circuits, 433–439, 457–463
    - logic diagram, 461–463
    - merging of the flow table, 459–460
    - primitive flow table, 457–458
    - specifications, 457
    - state assignment, 460–461
    - transition table, 460–461
  - combinational circuits, 523
  - with complex programmable logic device (CPLD), 315
  - with *D* flip-flops, 228
  - digital systems, 345
  - electronic design automation (EDA), 65
  - with field-programmable gate array (FPGA), 315
  - multiplexers, 390–401
    - testing the ones counter, 400–401
  - one-hot design (one flip-flop per state), 380–382
  - with programmable array logic (PAL), 309, 311
  - race-free, 401–403
    - software race conditions, 403
  - register transfer example, 352–361
    - algorithmic-based behavioral description, 362
    - behavioral descriptions, 361
    - control logic, 360–361
    - controller and datapath hardware design, 357–358
    - datapath unit, 352
    - HDL description of, 361–370
    - register transfer representation, 358
    - RTL description, 361–365
    - sequence of operations, 356
    - state table, 358–360
    - structural description, 361, 366–370
    - system chart, 353–355
    - testing the design description, 365–366
    - timing sequence, 355–356
  - at register transfer level (RTL), 161, 334–414
  - synchronous sequential logic, 161
  - top-down, 161
- Design entry, 107
- Design procedure, synchronous sequential logic, 225–234
- Digital age, 1
- Digital computers, 1–3
  - general-purpose, 2
- Digital integrated circuits, 471–510
  - bipolar transistors:
    - base–emitter graphical characteristic, 479
    - characteristics of, 477–481
    - collector and base currents, 478
    - dc current gain, 479
    - graphical collector–emitter characteristics, 479
    - pulled down output, 479
    - pulled out output, 479
    - saturation region, 479
    - types of, 477
  - CMOS transmission gate circuits, 501–504
    - basic circuit of, 501
    - bilateral switch, 501–502
    - connection to inverter, 501–502
    - exclusive-OR gate, construction of, 502–503
    - gated D latch, construction of, 503–504
    - master–slave D flip-flop, construction of, 504–505
    - multiplexer construction, 503
  - complementary MOS (CMOS) circuits, 498–501
    - characteristics of, 500–501
    - CMOS digital logic family, 501
    - CMOS transmission gate circuits, 501–504
    - examples of, 498–500
    - fabrication process, 501
    - graphic symbols, 500
    - inverter, 498
    - two-input NOR gate, 500
  - diodes, 479–480
  - DTL digital logic family:
    - analysis of, 482–483
    - fan-out, 483

    NAND gate, 482
    power dissipation of a DTL gate, 483
  emitter-coupled logic (ECL), 493–495
    basic circuit, 493
    defined, 493
    external wired connection of two OR outputs, 495
    graphic symbol, 495
    internal temperature- and voltage-compensated bias circuit, 493
    internal wired connection of two OR outputs, 495
    propagation delay, 494–495
  metal-oxide semiconductor (MOS), 495–498
    advantage of, 497
    basic structure of, 495
    channel, 495
    depletion mode, 496
    diffused channel, 496
    drain, 495
    enhancement mode, 496
    gate, 495
    graphic symbols, 496
    n-channel MOS, 496–497
    p-channel MOS, 496–497
    source, 495–496
    types of, 496
  RTL digital logic family:
    analysis of, 481
    fan-out, 481–482
    NOR gate, 481
  switch-level modeling, 505–508
    transmission gate, 506–507
  transistor–transistor logic, 484–493
    advanced low-power Schottky TTL gate, 485
    characteristics (table), 484
    fast TTL family, 485
    high-speed TTL gate, 484–485
    low-power Schottky TTL, 485
    low-power TTL gate, 484
    open-collector output gate, 485–488
    original, 484
    propagation delay, 484
    Schottky TTL gate, 484–485, 489–491
    standard, 484
    three-state gate, 491–493
    totem-pole output, 488–489
Digital logic circuits, 27–28
Digital logic gates, 57–62
  exclusive-OR gate, 59
  extension to multiple inputs, 59–61
  integrated circuits, 63–65
    computer-aided design, 64–65
    digital logic families, 63–64
    levels of integration, 63
  laboratory experiment, 519–520
    NAND circuit, 520
    propagation delay, 519–520
    truth tables, 519
    universal NAND gate, 520
    waveforms, 519
  NAND function, 59
  positive and negative logic, 61
Digital logic trainers, 511–512
Digital systems, 1–3
  defined, 2, 334
  logic design of, 345
  relationship between control logic and data-processing operations in, 345–346
Digital versatile disk (DVD), 2
Diminished radix, 10
Diminished radix complement, 10
Diodes, 479–480
  symbol and characteristic, 480
Direct (dedicated) interconnect lines, 317
Direct inputs, flip-flops, 194–195
Direct reset input, flip-flops, 194
Distributed RAM, 317
Distributive law, 37
Divide-by-*N* counter, 265
Don't-care conditions, 86–88
DRAM, *See* Dynamic RAM (DRAM)
DTL digital logic family:
  analysis of, 482–483
  fan-out, 483
  NAND gate, 482
  power dissipation of a DTL gate, 483
DTL (diode–transistor logic), 471
Duality, 41
Dual-trace oscilloscope, 512
Dynamic hazard, 453
Dynamic memory, refreshing, 291
Dynamic RAM (DRAM), 291
  address coding of, 295

E

ECL (emitter-coupled logic), 471
Edge-sensitive cyclic behavior, 344
Edge-triggered *D* flip-flop, 189–191, 531
  graphic symbol for, 191
  hold time, 191
  master, 189–190
  setup time, 191
  slave, 189–190
  with three *SR* latches, 190–191
  with two *D* latches and an inverter, 189–190
Electrically erasable PROM (EEPROM/E^{2}PROM), 304
Electronic design automation (EDA), 65
Emitter-coupled logic (ECL), 493–495
  basic circuit, 493
  defined, 493
  external wired connection of two OR outputs, 495
  graphic symbol, 495
  internal temperature- and voltage-compensated bias circuit, 493
  internal wired connection of two OR outputs, 495
  propagation delay, 494–495
Enable input, 288
Encoders, 150–152
  octal-to-binary, truth table for, 150
  priority, 151–152
endmodule (keyword), 109
Erasable PROM (EPROM), 304
Error detection scheme, parity bit as, 296
Error detection/correction, 296–299
  Hamming code, 296–298, 299
  single-error correction, double-error detection, 298–299
Error-correcting code, 296
Error-detecting code, 25
Essential hazards, 456–457
Event control operator, 208
Excess-3 code, 21–22
Excitation equations, flip-flops, 200
Excitation table, 229
Exclusive-NOR function (XNOR), 57
Exclusive-OR gate, 59
Exclusive-OR symbol (⊕), 55
Exclusive-OR (XOR) function, 57, 101–102, 104

F

Fan-in, 64
Fan-out, 64, 473–474
Fast TTL family, 485
Fault simulation, 108
Feedback shift register, 537
Feedback-free continuous assignment, 403
Field, 37–38
Field-effect transistor (FET), 473, 495
Field-programmable gate array (FPGA), 65, 284, 311, 315, 343
  design with, 315
  logic block, 315
  Xilinx FPGAs, 316
  Xilinx Spartan II FPGAs, 323–327
  Xilinx Spartan XL FPGAs, 322–323
  Xilinx Virtex FPGAs, 327–329

Field-programmable logic sequencer (FPLS), 313
Finite state machines, Mealy and Moore models of, 206–207
First-in, first-out register files (FIFOs), 320–321
Five-variable map, 81–83
Flash memory devices, 304
Flip-flops, 183–184, 188–195, 242
  characteristic equations, 194
  characteristic tables, 193–194
  construction of, 188–189
  defined, 183
  direct inputs, 194–195
  dynamic indicator, 191
  edge-triggered *D* flip-flop, 189–191
  input equations, 199–200
  JK flip-flops, 192
  laboratory experiment, 530–532
    D latch, 530
    edge-triggered flip-flop, 531
    IC flip-flops, 531–532
    master–slave flip-flop, 530
    SR latch, 530
  operation of, 189
  operations performed with, 192
  as registers, 335
  and signal transition, 189
  symbols, 568–570
  T flip-flops, 192
  timing of the response of, 191
Flow table, 420–422
  defined, 420
  examples of, 420
  obtaining the logic diagram from, 422
  primitive, 421
Flowcharts, 346
for loop, 340–341
forever loop, 340–341
Four-bit data-storage register, 244
Four-to-one-line multiplexers, 153
Four-variable map, 76–80
  prime implicants, 79–80
FPGA (field-programmable gate array), 284
Full adder, 130, 131–133, 527
Function blocks, 315

G
Gate array, 315
Gate instance, 110
Gate instantiation, 110
Gate-level minimization, 70–121
  AND–OR–INVERT function, 96
    implementation, 97–99
  defined, 70
  don't-care conditions, 86–88
  exclusive-OR (XOR) function, 101–102
  five-variable map, 81–83
  four-variable map, 76–80
    prime implicants, 79–80
  gate delays, 110–113
  hardware description language (HDL), 106–116
  Karnaugh map (K-map), 70–71
  map method, 70–71
  multilevel NAND circuits, 92–93
  NAND gate, 89–90
  nondegenerate forms, 97
  NOR gate, 93–96
  odd function, 102–104
  OR–AND–INVERT function, 96
    implementation, 98–100
  parity checker, 104–106
  parity generation, 104–106
  product-of-sums simplification, 83–86
  three-variable map, 72–76
  two-level implementation, 90–92
  two-variable map, 71–72
Gate-level (structural) modeling, 159
Giga (G), 4
Gray code, 22–23

H
Half adder, 130–131, 527
Hamming code, 296–298
  modified, 299
  using for data words, 298
Hardware algorithm, 346
Hardware description language (HDL), 65, 106–116, 159, 315
  Boolean expressions, 113–114
  defined, 106
  design entry, 107
  as documentation language, 107
  fault simulation, 108
  logic simulation, 107
  module declaration, 108–109
  register transfer level (RTL) in, 336–345
  for registers and counters, 269–276
  switch-level modeling, 505–508
    transmission gate, 506–507
  test bench, 107
  timing verification, 107–108
  user-defined primitives, 114–116
Hardware signal generators, 112
Hazards, 452–457
  in combinational circuits, 452–454
  defined, 452
  detection of, 453
  dynamic, 453
  essential, 456–457
  implementation with SR latches, 454–456
  remedy for eliminating, 454
  in sequential circuits, 454
  static 0-hazard, 453
  static 1-hazard, 453
Hexadecimal (base-16) number system, 3–4
Hexadecimal numbers, 8–9
High impedance, 159
High-impedance state, 156
High-speed TTL gate, 484–485
Huntington, E. V., 38

I
IC digital logic families, 471
  basic circuits in, 471
  bipolar junction transistor (BJT), 473
  data book, 471–472
  field-effect transistor (FET), 473
  NAND gates, 471–472
  NOR gates, 471–472
  special characteristics, 473–477
    fan-out, 473–474
    noise margin, 476–477
    power dissipation, 474–475
    propagation delay, 475–476
Identity element, 37
Implication table, 440–441
incompletely specified functions, 87
Information transfer, between registers, 335
initial block, 218
Input equations, flip-flops, 199–200
Instantiation, 110, 160
Institute of Electronics and Electrical Engineers (IEEE), 65
integer data type, 342
Integrated circuit RAM units, 291
Integrated circuits (ICs), 63–65, 471
  computer-aided design, 64–65
  digital logic families, 63–64
  levels of integration, 63
Interconnect resources, 317–318
  programmable, 319
Intra-assignment delay, 274
Inverse, 37

J
JK flip-flops, 192
  analysis of clocked sequential circuits with, 204–206

  analysis with, 201–204
  characteristic table, 193
Johnson counters, 268–269
Junction field-effect transistor (JFET), 495

K

Karnaugh map (K-map), 70–71, 343
Keywords, 108–109
Kilo (k), 4

L

Laboratory experiments, 511–558
  adders and subtractors (experiment 7), 527
    adder–subtractor (four-bit), 528–529
    full adder, 527
    half adder, 527
    magnitude comparator, 529–530
    parallel adder, 528
  asynchronous sequential circuits (experiment 18), 553
  binary and decimal numbers (experiment 1), 516
    BCD count, 517–518
    binary count, 516–517
    counts, 518
    oscilloscope display, 517
    output pattern, 517–518
  binary multiplier (experiment 17), 549–553
    block diagram, 549–550
    checking, 552–553
    control of registers, 550–552
    datapath design, 552
    design of control circuit, 552
    multiplication example, 552
  Boolean function simplification (experiment 3), 520–522
    Boolean functions in sum-of-minterms form, 522
    complement, 522
    gate ICs, 521
    logic diagram, 521–522
  clock-pulse generator (experiment 15), 545–547
    circuit operation, 545–546
    clock-pulse generator operation, 546–547
    IC timer, 545
  code converters (experiment 5), 524–526
    Gray code to equivalent binary, 524
    nine's complementer, 525
    seven-segment display, 525–526
  combinational circuits (experiment 4), 522–524
    decoder and truth table block diagram, 523–524
    decoder implementation, 523–524
    design example, 523
    majority logic, 523
    parity generator, 523
  counters (experiment 10), 533–535
    binary counter with parallel load, 534–535
    decimal counter, 534
    ripple counter, 534
    synchronous counter, 534
  digital logic gates (experiment 2), 519–520
    NAND circuit, 520
    propagation delay, 519–520
    truth tables, 519
    universal NAND gate, 520
    waveforms, 519
  digital logic trainers, 511–512
  dual-trace oscilloscope, 512
  flip-flops (experiment 8), 530–532
    D latch, 530
    edge-triggered flip-flop, 531
    IC flip-flops, 531–532
    master–slave flip-flop, 530
    SR latch, 530
  gate ICs needed for, 512
  graphic symbols, 515
  IC type 7493 ripple counter, 512
    operation of, 512–515
  integrated circuits required, 515
  lamp handball (experiment 15), 541–545
    circuit analysis, 544
    counting the number of losses, 544
    IC type 74194, 542
    Lamp Ping-Pong™, 545
    logic diagram, 542–544
    playing the game, 544
  logic breadboard suitable for performing, 511
  medium-scale integration (MSI) circuits, 512
  memory unit (experiment 14), 539–541
    IC RAM, 539–540
    memory expansion, 541
    ROM simulator, 541
    testing the RAM, 540–541
  multiplexer design (experiment 6), 526–527
    specifications, 527
  parallel adder and accumulator (experiment 16), 547–549
    block diagram, 547
    carry circuit, 548
    checking the circuit, 548
    circuit operation, 549
    control of register, 547
    detailed diagram of circuit, 548
  sequential circuits (experiment 9), 532–533
    counter design, 533
    state diagram, 533
    up–down counter with enable, 533
  serial addition (experiment 12), 538–539
    serial adder, 539
    serial adder–subtractor, 539
    testing the adder, 539
  shift registers (experiment 11), 535–538
    bidirectional shift register, 537
    bidirectional shift register with parallel load, 537–538
    feedback shift register, 537
    IC shift register, 535–536
    ring counter, 537
  small-scale integration (SSI) circuits, 512
  Verilog HDl simulation experiments and rapid prototyping with FPGAs, 553
    experiment 1, 554
    experiment 2, 554–555
    experiment 4, 555–556
    experiment 5, 556
    experiment 7, 556
    experiment 8, 556
    experiment 9, 557
    experiment 10, 557
    experiment 11, 557
    experiment 13, 557–558
    experiment 14, 558
    experiment 16, 558
    experiment 17, 558
Lamp handball (laboratory experiment), 541–545
  circuit analysis, 544
  counting the number of losses, 544
  IC type 74194, 542
  Lamp Ping-Pong™, 545
  logic diagram, 542–544
  playing the game, 544
Large-scale integration (LSI) devices, 63
Latches, 184–188
  D latch (transparent latch), 187–188
  reset state, 185
  sensitivity of, 184
  set state, 185

Latches *(cont.)*
  SR latch, 185–187
  trigger, 188
Latch-free design, 403–404
Literals, 46
Load operation, registers, 244, 334
Logic circuits, *See* Digital systems
Logic diagram:
  obtaining output Boolean functions from, 124
  of three-bit binary counter, 234
Logic gates, 30–31
Logic operations, digital systems, 336
Logic operators, Verilog 2001 HDL, 339
Logic operators, for binary words, 338
Logic simulation, 107
Logic simulators, 122
Logic synthesis, 343–345
  advantages to designer, 345
  assign statement, 343
  tools, 343
Logical operators, Verilog 2001 HDL, 339
Loop statements, 340–341
Low-power Schottky TTL, 485
Low-power TTL gate, 484

M
Magnetic disk, 290–291
Magnitude comparator, 143–145, 529–530
  four-bit, 145
Mask programming, 303
Master–slave flip-flop, 530
Maximal compatibles, 444–445
Mealy FSM (Mealy machine), 206
Mealy model, 206–207
Mealy_Zero_Detector, 215–216
Medium-scale integration (MSI) circuits, 512
Medium-scale integration (MSI) devices, 63
Mega (M), 4
Mem, 288
Memory:
  access time, 289
  address, 286
  architecture of, 285
  communication between the environment and, 285
  cycle time, 289
  integrated circuit RAM units, 291
  programmable logic device (PLD), 284
  random-access (RAM), 284
  sequential-access, 290
  types of, 290–291
Memory cell, 291–292
Memory chip, control inputs to, 288
Memory cycle timing waveforms, 289–290
Memory decoding, 291–295
  address multiplexing, 294–295
  coincident decoding, 293–294
  internal construction, 291–292
Memory depth, 288
Memory description in HDL, 288–289
Memory enable, 287, 289
Memory system, mode of access of, 290
Memory timing, 289
Memory units, 284–287
  block diagram, 285–286
  capacity of, 285
  defined, 284–285
  laboratory experiment, 539–541
    IC RAM, 539–540
    memory expansion, 541
    ROM simulator, 541
    testing the RAM, 540–541
  operation of, 288
  reliability of, 296
  volatile, 291
  words, 285, 287
memword, 288
Merger diagram, 444
Metal-oxide semiconductor (MOS):
  advantage of, 497
  basic structure of, 495
  channel, 495
  depletion mode, 496
  diffused channel, 496
  drain, 495
  enhancement mode, 496
  gate, 495
  graphic symbols, 496
  n-channel MOS, 496–497
  p-channel MOS, 496–497
  source, 495–496
  types of, 496
Metal-oxide-semiconductor field-effect transistor (MOSFET), 473
Module, 207
module ... endmodule (keyword pair), 109
Module declaration, 108–110
Modules, 108–110, 217
Modulo-*N* counter, 265
Moore FSM (Moore machine), 206
Moore model, 206–207
MOS, *See* Metal-oxide semiconductor (MOS)
Multilevel NAND circuits, 92–93
Multiple-row method, 450–452
Multiplexers, 152–158, 313
  Boolean function implementation, 154–156
  data selector, 154
  defined, 152
  design example, 393–400
    testing the ones counter, 400–401
  design with, 390–401
    laboratory experiment, 526–527
  four-to-one-line, 153
  graphics symbols for, 567
  implementing a Boolean function with, 156
  implementing a four-input function with, 157
  input conditions, 393
  quadruple two-to-one-line, 155
  three-state gates, 156–158
  two-to-one-line, 152–153, 153

N
NAND gate, 89–90
n-bit binary code, 17–18
Negation (~) operator, 338
negedge (keyword), 209
Nets, 164
Noise, defined, 476
Noise margin, 64, 476–477
Nonblocking assignments, 209–210, 337–338
Noncritical race, 422
Nondegenerate forms, 97
Nonvolatile memory, 291
NOR gate, 93–96
NOT operation, 29
npn type, bipolar transistor, 477–478
  silicon transistor parameters, 479
n-to-*m*-line decoders, 146
Number-base conversions, 5–7

O
Octal number system, 4
Octal numbers, conversion to hexadecimal, 9
Octal-to-binary encoder, truth table for, 150
Odd function, 102–104
One-hot assignment, 224–225
One-hot design, 380–382
Open Verilog International (OVI), 108
Open-collector output gate, 485–488
  AND–OR–INVERT function, 487
  applications, 486
  forming a common bus line, 487–488
  NAND gate, 485
  wired-AND, 487

Operator precedence, 43–44
OR gate, conventional and array logic diagrams for, 285
OR operation, 29
Output equations, flip-flops, 200
Overflow, 138–139

P

PAL, *See* Programmable array logic (PAL)
Parallel adder, 528
  laboratory experiment, 547–549
    block diagram, 547
    carry circuit, 548
    checking the circuit, 548
    circuit operation, 549
    control of register, 547
    detailed diagram of circuit, 548
Parallel load:
  bidirectional shift register with, 537–538
  binary counters with, 534–535
    Clear input, 262–264
    CLK input, 264
    Count input, 264
    Load input, 264
  registers with, 244–245
Parallel multiplier, behavioral description of, 388–390
Parallel-load control, shift registers, 251
parameter (keyword), 213
Parity bit, 25
  as error detection scheme, 296
Parity checker, 104–106
Parity generation, 104–106
Parity generator, combinational circuits, 523
PIP-based interconnection, architecture of, 319–320
PLA, *See* Programmable logic array (PLA)
PLD, *See* Programmable logic device (PLD)
pnp type, bipolar transistor, 477–478
posedge (keyword), 209
Positive-edge-triggered *D* flip-flop, 194–195
Power dissipation, 64, 474–475
Powers of two (table), 5
Predefined primitives, 110
Preset inputs, flip-flops, 194–195
Prime implicants, 79–80
Primitive flow table, 421, 457–458
Primitive gates, 110
Primitives, 110
Priority encoders, 151–152
  four-input, 152
  maps for, 151
  truth table for, 151
Procedural assignments, 337
Product of sums, 54
  expression, 95
  simplification, 83–86
Programmable array logic (PAL), 284–285, 309–311
  commercial, 309
  defined, 309
  designing with, 309, 311
  fuse map for, 311
  programming table, 309–311
Programmable logic array (PLA), 284, 305–308
  defined, 305
  designing a digital system with, 307
  fuse map of, 306, 308
  implementing a combinational circuit with, 307–308
  internal logic of, 305
  programming table:
    generation of, 308
    sections of, 306
  size of, 307
Programmable logic device (PLD), 65, 284–285, 343
  design with, 315
Programmable read-only memory (PROM), 303–304
Programming, 284–285
Propagation delay, 64, 475–476
  digital logic gates (laboratory experiment), 519–520
  emitter-coupled logic (ECL), 494–495
  IC digital logic families, 475–476
  transistor–transistor logic (TTL), 484

Q

Quadruple two-to-one-line multiplexers, 155
Qualifying symbols, 562–564
Qualitative analysis, 480
Quantitative analysis, 480

R

Race conditions, 422–424
  critical race, 422–423
    avoiding, 447
    examples of, 423
  cycles, 423–424
  noncritical race, 422
Race-free design, 401–403
  software race conditions, 403
Race-free state assignment, 446–452
  four-row flow-table example, 449–450
  multiple-row method, 450–452
  three-row flow-table example, 447–449
Radix, 3–4, 10–11
Radix complement, 10–11
RAM, *See* Random-access memory (RAM)
Random-access memory (RAM), 284–291
  commercial, word capacity of, 292
  memory description in HDL, 288–289
  memory, types of, 290–291
  symbol for, 574
  timing waveforms, 289–290
  write and read operations, 287–288
Read cycle, 289
Read input, 285
Read operation, 284
Read-only memory (ROM), 284, 299–305
  block diagram, 299–300
  combinational circuit implementation, 302
  combinational programmable logic device (PLD), 304–305
  defined, 299
  electrically erasable PROM (EEPROM/E^{2}PROM), 304
  erasable PROM (EPROM), 304
  flash memories, 304
  internal binary storage of, 300
  internal operation of, 302
  mask programming, 303
  number of words in, 299
  programmable read-only memory (PROM), 303–304
  programming, 300–301
  truth table, 301
  types of, 303–304
ReadWrite input, 288
Read/write signals, 289
Rectangular-shape symbols, 559–561
Reduction operators, 338
reg variable, 342
Register operations, 334
Register symbols, 570–572
Register transfer, 26–28
Register transfer level (RTL), 2
  algorithmic state machines (ASMs), 345–352
    design example, 352–361
  binary multiplier:
    control state diagram for, 376
    HDL description of, 382–389

- Register transfer level (RTL) *(cont.)*
  - continuous assignments, 337
  - control logic, 376–382
  - design at, 334–414
  - digital system represented at, 334
  - in HDL, 336–345
    - HDL operators, 338–340
    - logic synthesis, 343–345
    - loop statements, 340–341
  - latch-free design, 403–404
  - multiplexers, design with, 390–401
  - notation, 334–335
  - procedural assignments, 337–338
  - race-free design, 401–403
  - sequential binary multiplier, 370–376
- Register transfer operations, 334, 336
- Registers, 26, 242–253, 334, *See also* Register transfer level (RTL)
  - defined, 242, 335
  - four-bit data-storage register, 244
  - HDL for, 269–276
    - ripple counter, 274–276
    - shift register, 269–273
    - synchronous counter, 273–274
  - loading, 244
  - with parallel load, 244–245
  - shift registers, 245–253
    - defined, 245
    - serial addition, 248–250
    - serial input, 246
    - serial output, 246
    - serial transfer, 246–248
    - simple, 245–246
    - universal, 250–253
  - types of, 242–243
  - updating, 244
- Relational operators, Verilog 2001 HDL, 339
- **repeat** loop, 340
- Reset state, latches, 185
- Ring counters, 267–268, 537
- Ripple counters, 253–258, 534
  - BCD, 256–258
  - binary, 253–256
  - defined, 253, 256
  - HDL for, 274–276
- ROM (read-only memory), *See* Read-only memory (ROM)
- RTL digital logic family:
  - analysis of, 481
  - fan-out, 481–482
  - NOR gate, 481
- RTL (resistor–transistor logic), 471

S

- Schematic capture, 65
- Schematic entry, 65
- Schottky transistor, defined, 489
- Schottky TTL gate, 484–485, 489–491
  - symbol for Schottky transistors/diodes, 490
- Scratchpad memories, 321
- Sensitivity list, 208
- Sequence detector:
  - maps for, 228
  - state diagram for, 227
  - state table for, 228
- Sequential binary multiplier, 370–376
  - ASMD chart, 373–376
  - register configuration, 372–373
- Sequential circuits, 182–184, 415–470
  - asynchronous, 183
  - block diagram, 182
  - hazards in, 454
  - laboratory experiment, 532–533
    - counter design, 533
    - state diagram, 533
    - up–down counter with enable, 533
  - synchronous, 183
- Sequential (or simple) programmable logic device (SPLD), 311, 313
- Sequential programmable devices, 311–329
  - complex programmable logic device (CPLD), 311, 313–315
    - design with, 315
  - configurable logic block (CLB), 317
  - distributed RAM, 317
  - enhancements, 320–321
  - field-programmable gate array (FPGA), 65, 284, 311, 315
    - design with, 315
    - logic block, 315
    - Xilinx FPGAs, 316
    - Xilinx Spartan II FPGAs, 323–327
    - Xilinx Spartan XL FPGAs, 322–323
    - Xilinx Virtex FPGAs, 327–329
  - interconnect resources, 317–318
  - I/O block (IOB), 320
  - sequential (or simple) programmable logic device (SPLD), 311, 313
  - Xilinx:
    - basic architecture, 316–317
    - FPGAs, 316
- Sequential-access memory, 290
- Serial adder, 248, 539
  - second form of, 250
  - state table for, 250
- Serial addition, 248–250
  - laboratory experiment, 538–539
    - serial adder, 539
    - serial adder–subtractor, 539
    - testing the adder, 539
- Serial bit stream, 227
- Serial input, 246
- Serial output, 246
- Serial transfer, 246–248
- Set, 36
- Set state, latches, 185
- Shannon, C. E., 38
- Shared-row method, 451
- Shift operation, registers, 334
- Shift operations, digital systems, 336
- Shift operators, Verilog 2001 HDL, 339
- Shift register, HDL for, 269–273
- Shift registers, 245–253, 321
  - bidirectional, 251
  - clear control, 251
  - clock input, 251
  - defined, 245
  - laboratory experiment, 535–538
    - bidirectional shift register, 537–538
    - bidirectional shift register with parallel load, 537–538
    - feedback shift register, 537
    - IC shift register, 535–536
    - ring counter, 537
  - parallel-load control, 251
  - serial addition, 248–250
  - serial input, 246
  - serial output, 246
  - serial transfer, 246–248
  - shift-left control, 251
  - shift-right control, 251
  - simple, 245–246
  - unidirectional, 251
  - universal, 250–253
- Shift-left control, shift registers, 251
- Shift-right control, shift registers, 251
- Signed binary numbers, 14–17
  - arithmetic addition, 16
  - arithmetic subtraction, 17
- Signed-complement system, 14
- Signed-magnitude convention, 14
- Simple shift registers, 245–246
- *Simple_Circuit_prop_delay*, 111–112
- Single-error correction, double-error detection, 298–299
- Single-pass behavior, 207
- Small-scale integration (SSI) circuits, 512
- Small-scale integration (SSI) devices, 63
- Software race conditions, 403
- Spartan chips, 320–321
- Spartan device families, comparison chart, 324
- Spartan devices, 317
- Spartan II FPGAs, 323–327
  - device attributes (table), 324
- Spartan XL FPGAs, 322–323
  - architecture of, 323
  - device attributes (table), 323

SPLD, *See* Sequential (or simple) programmable logic device (SPLD)
SR latch, 185–187
SRAM, *See* Static RAM (SRAM)
Stable circuits, 424–425
Standard forms:
  Boolean algebra, 48–55
    defined, 54
    expression of a Boolean function in, 55
    product of sums, 54
    sum of products, 54–55
Standard graphic symbols, 559–576
  combinational element symbols, 566–568
  counter symbols, 572–574
  dependency notation, 564–566
  flip-flop symbols, 568–570
  qualifying symbols, 562–564
  RAM symbol, 574
  rectangular-shape symbols, 559–561
  register symbols, 570–572
State assignment, 224–225
State diagram, 199, 213–217
  compared to a state table, 198
  reducing, 223–224
  for sequence detector, 227
State equations, 196–197
  Boolean expressions for, 197
  defined, 196–197
State machine, defined, 346
State reduction, 220–223
State table, 197–198
  binary form of, 225
  compared to a state diagram, 198
  and *JK* flip-flop inputs, 231
  reducing, 222–223
  sections, 198
  for sequence detector, 228
  for three-bit binary counter, 231–233
State tables, reduction of, 439–441
Static 0-hazard, 453
Static 1-hazard, 453
Static RAM (SRAM), 291
Storage elements:
  defined, 184
  flip-flops, 188–195
  latches, 184–188
Sum of products, 54–55
Switch-level modeling, 505–508
  transmission gate, 506–507
Switch matrices, and CLB architecture, 317–319
Switching algebra, 38
Synchronous counters, 258–264, 534
  BCD counter, 260–262
    state table for, 260–262
  binary counter, 258–260
    four-bit, 259
  binary counter with parallel load, 262–265
    Clear input, 262–264
    CLK input, 264
    Count input, 264
    Load input, 264
  defined, 253
  HDL for, 273–274
  up-down binary counter, 260
    four-bit, 261
Synchronous RAM (SelectRAM), 320–321
Synchronous sequential circuits, 183, 225
Synchronous sequential logic, 182–241
  clocked sequential circuits, analysis of, 195–207
  design procedure, 225–234
  sequential circuits, 182–184
  state assignment, 224–225
  state reduction, 220–223
  storage elements:
    defined, 184
    flip-flops, 188–195
    latches, 184–188
  synthesizable HDL models of sequential circuits, 207–220
Syndrome, 296
Synthesis, 226
Synthesis tools, 315
Synthesizable HDL models of sequential circuits, 207–220
  behavioral modeling, 207–210
  clocked sequential circuits, structural description of, 217–220
  flip-flops and latches, 210–213
  state diagram, 199
System primitives, 114

T

T flip-flops, 192
  analysis of clocked sequential circuits with, 204–206
  analysis with, 204–206
  characteristic table, 193–194
    conditions, 194
t_Simple_Circuit_prop_delay, 112
Tape unit, 290
Tera (T), 4
Test access port (TAP) controller, 320
Test bench, 107, 111
Test bench module, 218
Three-bit binary counter:
  logic diagram of, 234
  maps for, 234
  state diagram of, 233
  state table for, 231–233
Three-state buffer gate, graphic symbol for, 157
Three-state gates, 156–158, 491–493
  buffer gate, graphic symbol of, 491–492
  bus, creation of, 493
  inverter, 491–492
  output enable delay compared to output disable delay, 493
  output states, 491
Three-to-eight-line decoder, 146
  truth table for, 147
Three-variable map, 72–76
Time units, 110
Time-delay devices, 183
Timing verification, 107–108
Timing waveforms, 289–290
Top-down design, 161
Total state, 419
Totem-pole output, 488–489
  defined, 488
  wired-logic connection, 489
Transfer function, 57
Transfer operations, digital systems, 336
Transistors, 1
Transistor–transistor logic (TTL):
  advanced low-power Schottky TTL gate, 485
  characteristics (table), 484
  defined, 471
  fast TTL family, 485
  high-speed TTL gate, 484–485
  low-power Schottky TTL, 485
  low-power TTL gate, 484
  open-collector output gate, 485–488
  original, 484
  propagation delay, 484
  Schottky TTL gate, 484–485, 489–491
  standard, 484
  three-state gate, 491–493
  totem-pole output, 488–489
  TTL gate, operation of, 471
Transition diagram, 447
Transition equation, *See* State equations
Transition table, 225, 417–420, *See* State table
  of asynchronous sequential circuits, 419–420
Transparent latch, 187–188
Trigger, latches, 188
Truth tables, 29
TTL, *See* Transistor–transistor logic (TTL)
2421 code, 21–22
Two-level implementation, 90–92
Two-to-four-line decoder, with enable input, 147–148
Two-to-one-line multiplexers, 152–153, 153

Two-valued Boolean algebra, 39–41
Two-variable map, 71–72

U
UDPs, *See* User-defined primitives (UDPs)
Unidirectional shift registers, 251
Universal NAND gate, 520
Universal shift registers, 250–253
  four-bit, 252
Unknown value, 159
Unsigned binary numbers, 14
Unstable circuits, 424–425
Unused states, 224
  counters with, 265–266
Updating a register, 244
User-defined primitives (UDPs), 114–116

V
Vectors, 160
Verilog HDL, 108, 159, 207, 209, 315
  memory description in, 288
  operators, 165, 338–340
  switch-level modeling, 505–508
    transmission gate, 506–507
Very large-scale integration (VLSI) devices, 63–65
VHDL, 108, 315
Virtex FPGAs, 327–329
Volatile memory units, 291

W
Waveforms, 519
Weighted codes, 21
while loop, 340–342
Wired logic, 96
Word length, 288
Word locations, 290
Words, 285, 287
Write input, 285
Write operation, 284

X
Xilinx:
  basic architecture, 316–317
  FPGAs, 316
  Spartan II FPGAs, 323–327
  Spartan XL FPGAs, 322–323
  Virtex FPGAs, 327–329
XNOR function, 57
XOR function, 57