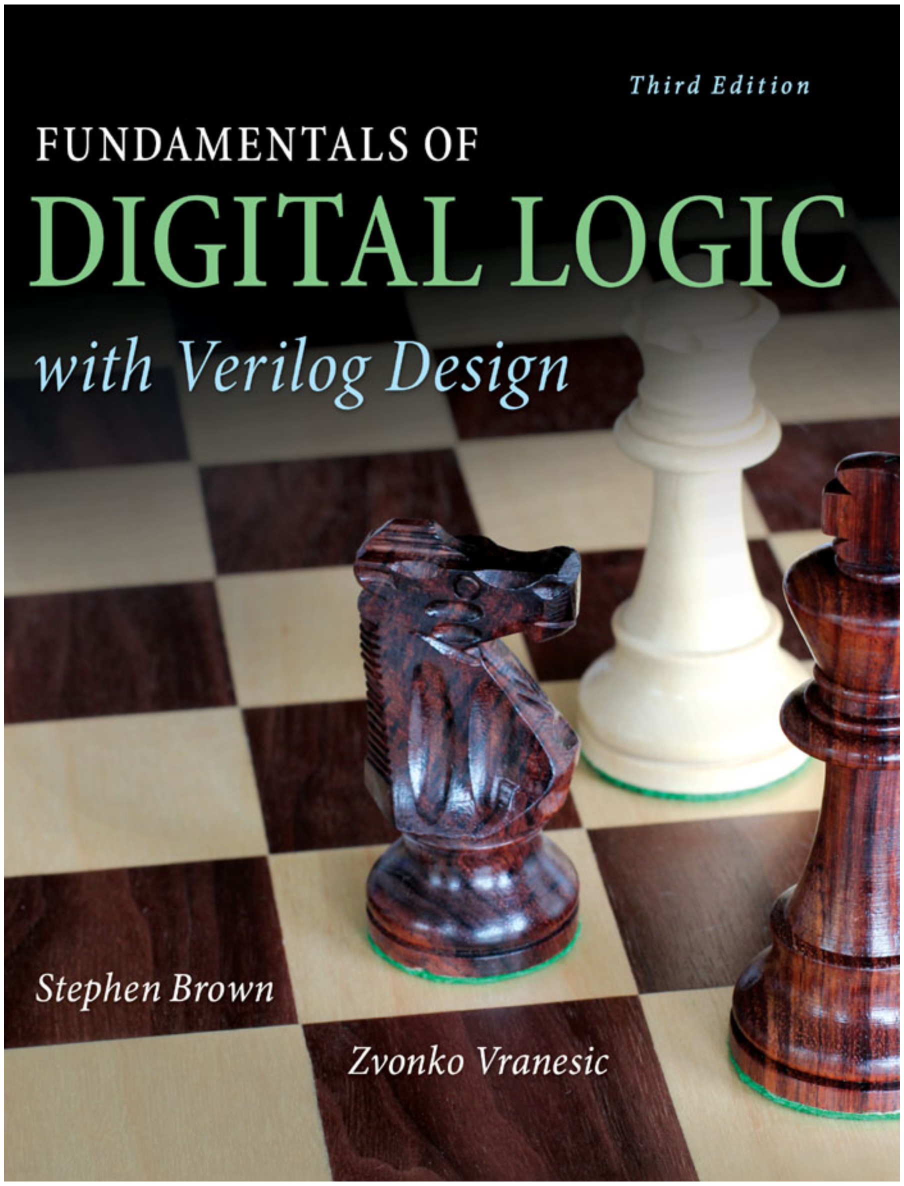


Third Edition

FUNDAMENTALS OF DIGITAL LOGIC with Verilog Design

Stephen Brown

Zvonko Vranesic

Fundamentals of Digital Logic with Verilog Design

THIRD EDITION

Stephen Brown and Zvonko Vranesic
Department of Electrical and Computer Engineering
University of Toronto

The McGraw·Hill Companies

Connect Learn Succeed™

FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN, THIRD EDITION

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 1 0 9 8 7 6 5 4 3

ISBN 978–0–07–338054–4
MHID 0–07–338054–7

Managing Director: *Thomas Timp*
Director: *Michael Lange*
Global Publisher: *Raghothaman Srinivasan*
Developmental Editor: *Vincent Bradshaw*
Marketing Manager: *Curt Reynolds*
Director, Content Production: *Terri Schiesl*
Senior Project Manager: *Melissa M. Leick*
Buyer: *Susan K. Culbertson*
Media Project Manager: *Prashanthi Nadipalli*
Cover Design: *Studio Montage, St. Louis, Missouri*
(USE) Cover Image: *Steven Brown and Zvonko Vranesic*
Compositor: *Techsetters, Inc.*
Typeface: *10/12 Times Roman*
Printer: *R. R. Donnelley, Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Brown, Stephen.
Fundamentals of digital logic with Verilog design / Stephen Brown and Zvonko Vranesic. — Third edition.
pages cm
ISBN 978–0–07–338054–4 (alk. paper)
1. Logic circuits—Design and construction—Data processing. 2. Verilog (Computer hardware description language). 3. Computer-aided design. I. Vranesic, Zvonko G. II. Title.

TK7868.L6B76 2014
621.39′2–dc23 2012042163

www.mhhe.com

To Susan and Anne

This page intentionally left blank

About the Authors

Stephen Brown received his B.A.Sc. degree in Electrical Engineering from the University of New Brunswick, Canada, and the M.A.Sc. and Ph.D. degrees in Electrical Engineering from the University of Toronto. He joined the University of Toronto faculty in 1992, where he is now a Professor in the Department of Electrical & Computer Engineering. He is also the Director of the worldwide University Program at Altera Corporation.

His research interests include field-programmable VLSI technology and computer architecture. He won the Canadian Natural Sciences and Engineering Research Council's 1992 Doctoral Prize for the best Ph.D. thesis in Canada and has published more than 100 scientific research papers.

He has won five awards for excellence in teaching electrical engineering, computer engineering, and computer science courses. He is a coauthor of two other books: *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. and *Field-Programmable Gate Arrays*.

Zvonko Vranesic received his B.A.Sc., M.A.Sc., and Ph.D. degrees, all in Electrical Engineering, from the University of Toronto. From 1963–1965 he worked as a design engineer with the Northern Electric Co. Ltd. in Bramalea, Ontario. In 1968 he joined the University of Toronto, where he is now a Professor Emeritus in the Departments of Electrical & Computer Engineering and Computer Science. During the 1978–1979 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984–1985 he was at the University of Paris, 6. From 1995 to 2000 he served as Chair of the Division of Engineering Science at the University of Toronto. He is also involved in research and development at the Altera Toronto Technology Center.

His current research interests include computer architecture and field-programmable VLSI technology.

He is a coauthor of four other books: *Computer Organization and Embedded Systems*, 6th ed.; *Fundamentals of Digital Logic with VHDL Design*, 3rd ed.; *Microcomputer Structures*; and *Field-Programmable Gate Arrays*. In 1990, he received the Wighton Fellowship for "innovative and distinctive contributions to undergraduate laboratory instruction." In 2004, he received the Faculty Teaching Award from the Faculty of Applied Science and Engineering at the University of Toronto.

He has represented Canada in numerous chess competitions. He holds the title of International Master.

Preface

This book is intended for an introductory course in digital logic design, which is a basic course in most electrical and computer engineering programs. A successful designer of digital logic circuits needs a good understanding of basic concepts and a firm grasp of the modern design approach that relies on computer-aided design (CAD) tools.

The main goals of the book are (1) to teach students the fundamental concepts in classical manual digital design and (2) illustrate clearly the way in which digital circuits are designed today, using CAD tools. Even though modern designers no longer use manual techniques, except in rare circumstances, our motivation for teaching such techniques is to give students an intuitive feeling for how digital circuits operate. Also, the manual techniques provide an illustration of the types of manipulations performed by CAD tools, giving students an appreciation of the benefits provided by design automation. Throughout the book, basic concepts are introduced by way of examples that involve simple circuit designs, which we perform using both manual techniques and modern CAD-tool-based methods. Having established the basic concepts, more complex examples are then provided, using the CAD tools. Thus our emphasis is on modern design methodology to illustrate how digital design is carried out in practice today.

Technology

The book discusses modern digital circuit implementation technologies. The emphasis is on programmable logic devices (PLDs), which is the most appropriate technology for use in a textbook for two reasons. First, PLDs are widely used in practice and are suitable for almost all types of digital circuit designs. In fact, students are more likely to be involved in PLD-based designs at some point in their careers than in any other technology. Second, circuits are implemented in PLDs by end-user programming. Therefore, students can be provided with an opportunity, in a laboratory setting, to implement the book's design examples in actual chips. Students can also simulate the behavior of their designed circuits on their own computers. We use the two most popular types of PLDs for targeting of designs: complex programmable logic devices (CPLDs) and field-programmable gate arrays (FPGAs).

We emphasize the use of a hardware description language in specifying the logic circuits, because the HDL-based approach is the most efficient design method to use in practice. We describe in detail the IEEE Standard Verilog HDL language and use it extensively in examples.

Scope of the Book

This edition of the book has been extensively restructured. All of the material that should be covered in a one-semester course is now included in Chapters 1 to 6. More advanced material is presented in Chapters 7 to 11.

Chapter 1 provides a general introduction to the process of designing digital systems. It discusses the key steps in the design process and explains how CAD tools can be used to automate many of the required tasks. It also introduces the representation of digital information.

Chapter 2 introduces the logic circuits. It shows how Boolean algebra is used to represent such circuits. It introduces the concepts of logic circuit synthesis and optimization, and shows how logic gates are used to implement simple circuits. It also gives the reader a first glimpse at Verilog, as an example of a hardware description language that may be used to specify the logic circuits.

Chapter 3 concentrates on circuits that perform arithmetic operations. It discusses numbers and shows how they can be manipulated using logic circuits. This chapter illustrates how Verilog can be used to specify the desired functionality and how CAD tools provide a mechanism for developing the required circuits.

Chapter 4 presents combinational circuits that are used as building blocks. It includes the encoder, decoder, and multiplexer circuits. These circuits are very convenient for illustrating the application of many Verilog constructs, giving the reader an opportunity to discover more advanced features of Verilog.

Storage elements are introduced in Chapter 5. The use of flip-flops to realize regular structures, such as shift registers and counters, is discussed. Verilog-specified designs of these structures are included.

Chapter 6 gives a detailed presentation of synchronous sequential circuits (finite state machines). It explains the behavior of these circuits and develops practical design techniques for both manual and automated design.

Chapter 7 is a discussion of a number of practical issues that arise in the design of real systems. It highlights problems often encountered in practice and indicates how they can be overcome. Examples of larger circuits illustrate a hierarchical approach in designing digital systems. Complete Verilog code for these circuits is presented.

Chapter 8 deals with more advanced techniques for optimized implementation of logic functions. It presents algorithmic techniques for optimization. It also explains how logic functions can be specified using a cubical representation as well as using binary decision diagrams.

Asynchronous sequential circuits are discussed in Chapter 9. While this treatment is not exhaustive, it provides a good indication of the main characteristics of such circuits. Even though the asynchronous circuits are not used extensively in practice, they provide an excellent vehicle for gaining a deeper understanding of the operation of digital circuits in general. They illustrate the consequences of propagation delays and race conditions that may be inherent in the structure of a circuit.

Chapter 10 presents a complete CAD flow that the designer experiences when designing, implementing, and testing a digital circuit.

Chapter 11 introduces the topic of testing. A designer of logic circuits has to be aware of the need to test circuits and should be conversant with at least the most basic aspects of testing.

Appendix A provides a complete summary of Verilog features. Although use of Verilog is integrated throughout the book, this appendix provides a convenient reference that the reader can consult from time to time when writing Verilog code.

The electronic aspects of digital circuits are presented in Appendix B. This appendix shows how the basic gates are built using transistors and presents various factors that affect circuit performance. The emphasis is on the latest technologies, with particular focus on CMOS technology and programmable logic devices.

What Can Be Covered in a Course

Much of the material in the book can be covered in 2 one-quarter courses. A good coverage of the most important material can be achieved in a single one-semester, or even a one-quarter course. This is possible only if the instructor does not spend too much time teaching the intricacies of Verilog and CAD tools. To make this approach possible, we organized the Verilog material in a modular style that is conducive to self-study. Our experience in teaching different classes of students at the University of Toronto shows that the instructor may spend only three to four lecture hours on Verilog, describing how the code should be structured, including the use of design hierarchy, using scalar and vector variables, and on the style of code needed to specify sequential circuits. The Verilog examples given in the book are largely self-explanatory, and students can understand them easily.

The book is also suitable for a course in logic design that does not include exposure to Verilog. However, some knowledge of Verilog, even at a rudimentary level, is beneficial to the students, and it is a great preparation for a job as a design engineer.

One-Semester Course

The following material should be covered in lectures:

- Chapter 1—all sections.
- Chapter 2—all sections.
- Chapter 3—Sections 3.1 to 3.5.
- Chapter 4—all sections.
- Chapter 5—all sections.
- Chapter 6—all sections.

One-Quarter Course

In a one-quarter course the following material can be covered:

- Chapter 1—all sections.
- Chapter 2—all sections.

- Chapter 3—Sections 3.1 to 3.3 and Section 3.5.
- Chapter 4—all sections.
- Chapter 5—all sections.
- Chapter 6—Sections 6.1 to 6.4.

VERILOG

Verilog is a complex language, which some instructors feel is too hard for beginning students to grasp. We fully appreciate this issue and have attempted to solve it. It is not necessary to introduce the entire Verilog language. In the book we present the important Verilog constructs that are useful for the design and synthesis of logic circuits. Many other language constructs, such as those that have meaning only when using the language for simulation purposes, are omitted. The Verilog material is introduced gradually, with more advanced features being presented only at points where their use can be demonstrated in the design of relevant circuits.

The book includes more than 120 examples of Verilog code. These examples illustrate how Verilog is used to describe a wide range of logic circuits, from those that contain only a few gates to those that represent digital systems such as a simple processor.

All of the examples of Verilog code presented in the book are provided on the Authors' website at

www.eecg.toronto.edu/~brown/Verilog_3e

SOLVED PROBLEMS

The chapters include examples of solved problems. They show how typical homework problems may be solved.

HOMEWORK PROBLEMS

More than 400 homework problems are provided in the book. Answers to selected problems are given at the back of the book. Solutions to all problems are available to instructors in the *Solutions Manual* that accompanies the book.

POWERPOINT SLIDES AND SOLUTIONS MANUAL

PowerPoint slides that contain all of the figures in the book are available on the Authors' website. Instructors can request access to these slides, as well as access to the Solutions Manual for the book, at:

www.mhhe.com/brownvranesic

CAD TOOLS

Modern digital systems are quite large. They contain complex logic circuits that would be difficult to design without using good CAD tools. Our treatment of Verilog should enable the reader to develop Verilog code that specifies logic circuits of varying degrees of complexity. To gain proper appreciation of the design process, it is highly beneficial to implement the designs using commercially-available CAD tools. Some excellent CAD tools are available free of charge. For example, the Altera Corporation has its Quartus II CAD software, which is widely used for implementing designs in programmable logic devices such as FPGAs. The Web Edition of the Quartus II software can be downloaded from Altera's website and used free of charge, without the need to obtain a license. In previous editions of this book a set of tutorials for using the Quartus II software was provided in the appendices. Those tutorials can now be found on the Authors' website. Another set of useful tutorials about Quartus II can be found on Altera's University Program website, which is located at www.altera.com/education/univ.

ACKNOWLEDGMENTS

We wish to express our thanks to the people who have helped during the preparation of the book. Dan Vranesic produced a substantial amount of artwork. He and Deshanand Singh also helped with the preparation of the solutions manual. Tom Czajkowski helped in checking the answers to some problems. The reviewers, William Barnes, New Jersey Institute of Technology; Thomas Bradicich, North Carolina State University; James Clark, McGill University; Stephen DeWeerth, Georgia Institute of Technology; Sander Eller, Cal Poly Pomona; Clay Gloster, Jr., North Carolina State University (Raleigh); Carl Hamacher, Queen's University; Vincent Heuring, University of Colorado; Yu Hen Hu, University of Wisconsin; Wei-Ming Lin, University of Texas (San Antonio); Wayne Loucks, University of Waterloo; Kartik Mohanram, Rice University; Jane Morehead, Mississippi State University; Chris Myers, University of Utah; Vojin Oklobdzija, University of California (Davis); James Palmer, Rochester Institute of Technology; Gandhi Puvvada, University of Southern California; Teodoro Robles, Milwaukee School of Engineering; Tatyana Roziner, Boston University; Rob Rutenbar, Carnegie Mellon University; Eric Schwartz, University of Florida; Wen-Tsong Shiue, Oregon State University; Peter Simko, Miami University; Scott Smith, University of Missouri (Rolla); Arun Somani, Iowa State University; Bernard Svihel, University of Texas (Arlington); and Zeljko Zilic, McGill University provided constructive criticism and made numerous suggestions for improvements.

The support of McGraw-Hill people has been exemplary. We truly appreciate the help of Raghu Srinivasan, Vincent Bradshaw, Darlene Schueller, Curt Reynolds, and Michael Lange. We are also grateful for the excellent support in the typesetting of the book that has been provided by Techsetters, Inc.

Stephen Brown and Zvonko Vranesic

CONTENTS

chapter

1

INTRODUCTION

CHAPTER OBJECTIVES

In this chapter you will be introduced to:

- Digital hardware components
- An overview of the design process
- Binary numbers
- Digital representation of information

This book is about logic circuits—the circuits from which computers are built. Proper understanding of logic circuits is vital for today's electrical and computer engineers. These circuits are the key ingredient of computers and are also used in many other applications. They are found in commonly-used products like music and video players, electronic games, digital watches, cameras, televisions, printers, and many household appliances, as well as in large systems, such as telephone networks, Internet equipment, television broadcast equipment, industrial control units, and medical instruments. In short, logic circuits are an important part of almost all modern products.

The material in this book will introduce the reader to the many issues involved in the design of logic circuits. It explains the key ideas with simple examples and shows how complex circuits can be derived from elementary ones. We cover the classical theory used in the design of logic circuits because it provides the reader with an intuitive understanding of the nature of such circuits. But, throughout the book, we also illustrate the modern way of designing logic circuits using sophisticated *computer aided design (CAD)* software tools. The CAD methodology adopted in the book is based on the industry-standard design language called the *Verilog hardware description language*. Design with Verilog is first introduced in Chapter 2, and usage of Verilog and CAD tools is an integral part of each chapter in the book.

Logic circuits are implemented electronically, using transistors on an integrated circuit chip. Commonly available chips that use modern technology may contain more than a billion transistors, as in the case of some computer processors. The basic building blocks for such circuits are easy to understand, but there is nothing simple about a circuit that contains billions of transistors. The complexity that comes with large circuits can be handled successfully only by using highly-organized design techniques. We introduce these techniques in this chapter, but first we briefly describe the hardware technology used to build logic circuits.

1.1 Digital Hardware

Logic circuits are used to build computer hardware, as well as many other types of products. All such products are broadly classified as [illegible] The reason that the name [illegible] is used will be explained in Section 1.5—it derives from the way in which information is represented in computers, as electronic signals that correspond to digits of information.

The technology used to build digital hardware has evolved dramatically over the past few decades. Until the 1960s logic circuits were constructed with bulky components, such as transistors and resistors that came as individual parts. The advent of integrated circuits made it possible to place a number of transistors, and thus an entire circuit, on a single chip. In the beginning these circuits had only a few transistors, but as the technology improved they became more complex. Integrated circuit chips are manufactured on a silicon wafer, such as the one shown in Figure 1.1. The wafer is cut to produce the individual chips, which are then placed inside a special type of chip package. By 1970 it was possible to implement all circuitry needed to realize a microprocessor on a single chip. Although early microprocessors had modest computing capability by today's standards, they opened the door for the information processing revolution by providing the means for implementation of affordable personal computers.

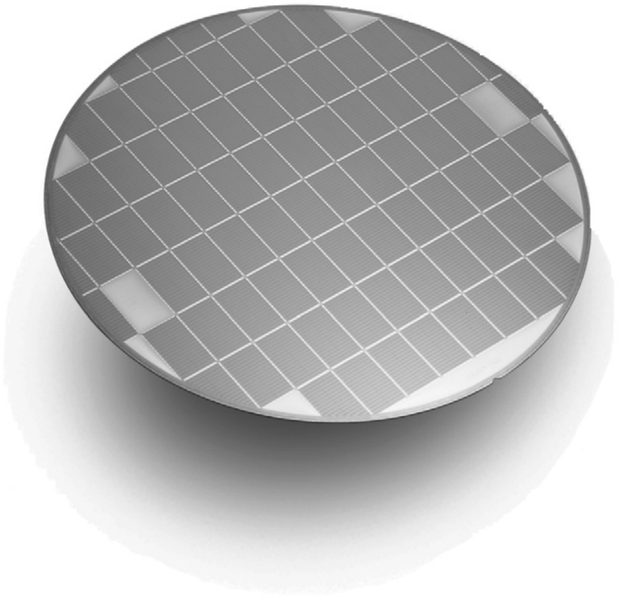


Figure 1.1 A silicon wafer (courtesy of Altera Corp.).

About 30 years ago Gordon Moore, chairman of Intel Corporation, observed that integrated circuit technology was progressing at an astounding rate, approximately doubling the number of transistors that could be placed on a chip every two years. This phenomenon, informally known as *Moore's law*, continues to the present day. Thus in the early 1990s microprocessors could be manufactured with a few million transistors, and by the late 1990s it became possible to fabricate chips that had tens of millions of transistors. Presently, chips can be manufactured containing billions of transistors.

Moore's law is expected to continue to hold true for a number of years. A consortium of integrated circuit associations produces a forecast of how the technology is expected to evolve. Known as the *International Technology Roadmap for Semiconductors (ITRS)* [1], this forecast discusses many aspects of technology, including the maximum number of transistors that can be manufactured on a single chip. A sample of data from the ITRS is given in Figure 1.2. It shows that chips with about 10 million transistors could be successfully manufactured in 1995, and this number has steadily increased, leading to today's chips with over a billion transistors. The roadmap predicts that chips with as many as 100 billion transistors will be possible by the year 2022. There is no doubt that this technology will have a huge impact on all aspects of people's lives.

The designer of digital hardware may be faced with designing logic circuits that can be implemented on a single chip or designing circuits that involve a number of chips placed on a *printed circuit board (PCB)*. Frequently, some of the logic circuits can be realized

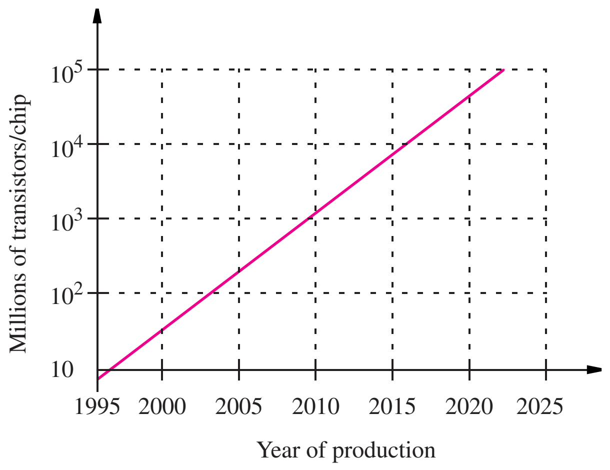


Figure 1.2 An estimate of the maximum number of transistors per chip over time.

in existing chips that are readily available. This situation simplifies the design task and shortens the time needed to develop the final product. Before we discuss the design process in detail, we should introduce the different types of integrated circuit chips that may be used.

There exists a large variety of chips that implement various functions that are useful in the design of digital hardware. The chips range from simple ones with low functionality to extremely complex chips. For example, a digital hardware product may require a microprocessor to perform some arithmetic operations, memory chips to provide storage capability, and interface chips that allow easy connection to input and output devices. Such chips are available from various vendors.

For many digital hardware products, it is also necessary to design and build some logic circuits from scratch. For implementing these circuits, three main types of chips may be used: standard chips, programmable logic devices, and custom chips. These are discussed next.

1.1.1 Standard Chips

Numerous chips are available that realize some commonly-used logic circuits. We will refer to these as *standard chips*, because they usually conform to an agreed-upon standard in terms of functionality and physical configuration. Each standard chip contains a small amount of circuitry (usually involving fewer than 100 transistors) and performs a simple function. To build a logic circuit, the designer chooses the chips that perform whatever functions are needed and then defines how these chips should be interconnected to realize a larger logic circuit.

Standard chips were popular for building logic circuits until the early 1980s. However, as integrated circuit technology improved, it became inefficient to use valuable space on PCBs for chips with low functionality. Another drawback of standard chips is that the functionality of each chip is fixed and cannot be changed.

1.1.2 Programmable Logic Devices

In contrast to standard chips that have fixed functionality, it is possible to construct chips that contain circuitry which can be configured by the user to implement a wide range of different logic circuits. These chips have a very general structure and include a collection of *programmable switches* that allow the internal circuitry in the chip to be configured in many different ways. The designer can implement whatever functions are required for a particular application by setting the programmable switches as needed. The switches are programmed by the end user, rather than when the chip is manufactured. Such chips are known as *programmable logic devices (PLDs)*.

PLDs are available in a wide range of sizes, and can be used to implement very large logic circuits. The most commonly-used type of PLD is known as a *field-programmable gate array (FPGA)*. The largest FPGAs contain billions of transistors [2, 3], and support the implementation of complex digital systems. An FPGA consists of a large number of small logic circuit elements, which can be connected together by using programmable switches in the FPGA. Because of their high capacity, and their capability to be tailored to meet the requirements of a specific application, FPGAs are widely used today.

1.1.3 Custom-Designed Chips

FPGAs are available as off-the-shelf components that can be purchased from different suppliers. Because they are programmable, they can be used to implement most logic circuits found in digital hardware. However, they also have a drawback in that the programmable switches consume valuable chip area and limit the speed of operation of implemented circuits. Thus in some cases FPGAs may not meet the desired performance or cost objectives. In such situations it is possible to design a chip from scratch; namely, the logic circuitry that must be included on the chip is designed first and then the chip is manufactured by a company that has the fabrication facilities. This approach is known as *custom* or *semi-custom design*, and such chips are often called *application-specific integrated circuits (ASICs)*.

The main advantage of a custom chip is that its design can be optimized for a specific task; hence it usually leads to better performance. It is possible to include a larger amount of logic circuitry in a custom chip than would be possible in other types of chips. The cost of producing such chips is high, but if they are used in a product that is sold in large quantities, then the cost per chip, amortized over the total number of chips fabricated, may be lower than the total cost of off-the-shelf chips that would be needed to implement the same function(s). Moreover, if a single chip can be used instead of multiple chips to achieve the same goal, then a smaller area is needed on a PCB that houses the chips in the final product. This results in a further reduction in cost.

A disadvantage of the custom-design approach is that manufacturing a custom chip often takes a considerable amount of time, on the order of months. In contrast, if an FPGA can be used instead, then the chips are programmed by the end user and no manufacturing delays are involved.

1.2 The Design Process

The availability of computer-based tools has greatly influenced the design process in a wide variety of environments. For example, designing an automobile is similar in the general approach to designing a furnace or a computer. Certain steps in the development cycle must be performed if the final product is to meet the specified objectives.

The flowchart in Figure 1.3 depicts a typical development process. We assume that the process is to develop a product that meets certain expectations. The most obvious requirements are that the product must function properly, that it must meet an expected level of performance, and that its cost should not exceed a given target.

The process begins with the definition of product specifications. The essential features of the product are identified, and an acceptable method of evaluating the implemented features in the final product is established. The specifications must be tight enough to ensure that the developed product will meet the general expectations, but should not be unnecessarily constraining (that is, the specifications should not prevent design choices that may lead to unforeseen advantages).

From a complete set of specifications, it is necessary to define the general structure of an initial design of the product. This step is difficult to automate. It is usually performed by a human designer because there is no clear-cut strategy for developing a product's overall structure—it requires considerable design experience and intuition.

After the general structure is established, CAD tools are used to work out the details. Many types of CAD tools are available, ranging from those that help with the design of individual parts of the system to those that allow the entire system's structure to be represented in a computer. When the initial design is finished, the results must be verified against the original specifications. Traditionally, before the advent of CAD tools, this step involved constructing a physical model of the designed product, usually including just the key parts. Today it is seldom necessary to build a physical model. CAD tools enable designers to simulate the behavior of incredibly complex products, and such simulations are used to determine whether the obtained design meets the required specifications. If errors are found, then appropriate changes are made and the verification of the new design is repeated through simulation. Although some design flaws may escape detection via simulation, usually all but the most subtle problems are discovered in this way.

When the simulation indicates that the design is correct, a complete physical prototype of the product is constructed. The prototype is thoroughly tested for conformance with the specifications. Any errors revealed in the testing must be fixed. The errors may be minor, and often they can be eliminated by making small corrections directly on the prototype of the product. In case of large errors, it is necessary to redesign the product and repeat the steps explained above. When the prototype passes all the tests, then the product is deemed to be successfully designed and it can go into production.

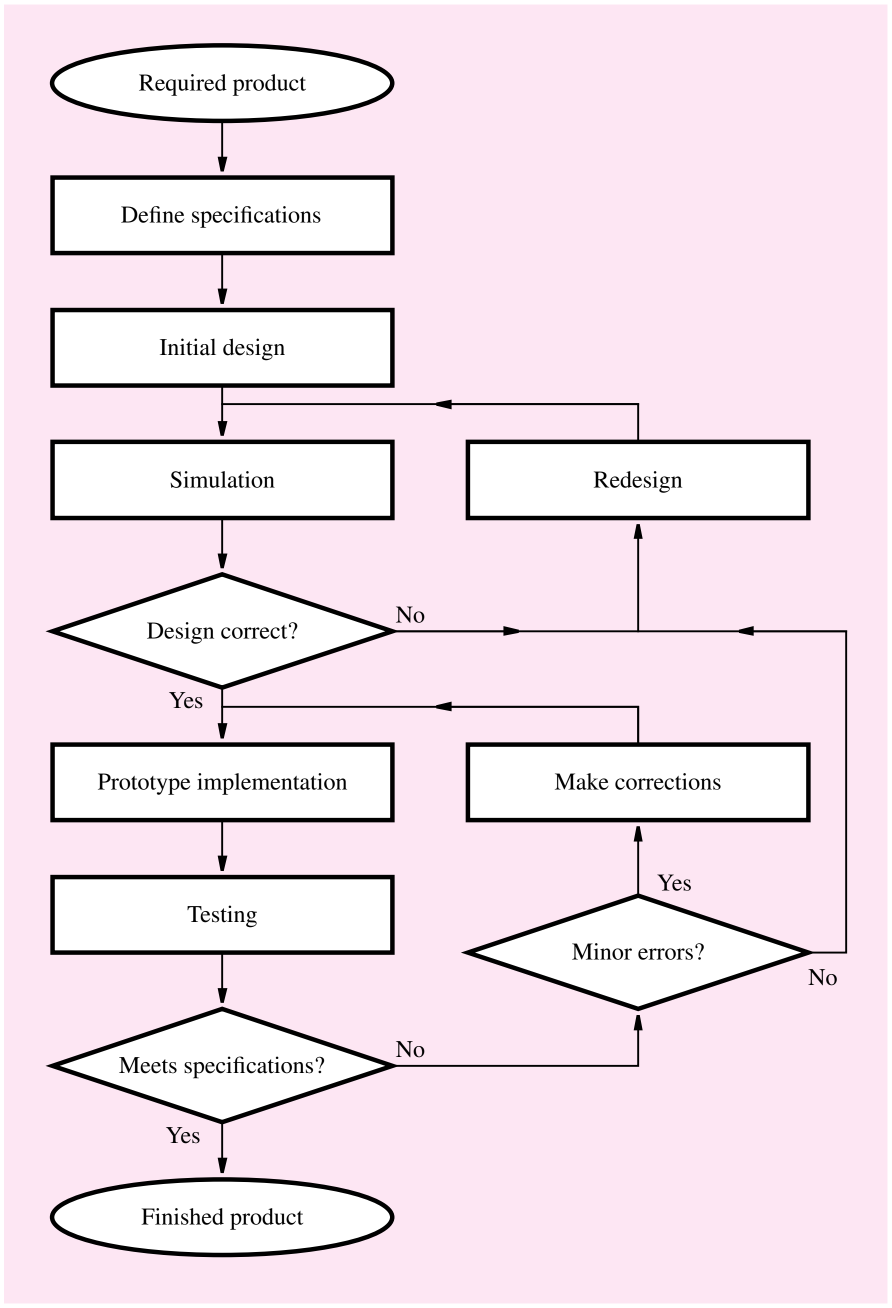


Figure 1.3 The development process.

1.3 Structure of a Computer

To understand the role that logic circuits play in digital systems, consider the structure of a typical computer, as illustrated in Figure 1.4*a*. The computer case houses a number of printed circuit boards (PCBs), a power supply, and (not shown in the figure) storage units, like a hard disk and DVD or CD-ROM drives. Each unit is plugged into a main PCB, called the *motherboard*. As indicated on the bottom of the figure, the motherboard holds several integrated circuit chips, and it provides slots for connecting other PCBs, such as audio, video, and network boards.

Figure 1.4*b* illustrates the structure of an integrated circuit chip. The chip comprises a number of subcircuits, which are interconnected to build the complete circuit. Examples of subcircuits are those that perform arithmetic operations, store data, or control the flow of data. Each of these subcircuits is a logic circuit. As shown in the middle of the figure, a logic circuit comprises a network of connected *logic gates*. Each logic gate performs a very simple function, and more complex operations are realized by connecting gates together. Logic gates are built with transistors, which in turn are implemented by fabricating various layers of material on a silicon chip.

This book is primarily concerned with the center portion of Figure 1.4*b*—the design of logic circuits. We explain how to design circuits that perform important functions, such as adding, subtracting, or multiplying numbers, counting, storing data, and controlling the processing of information. We show how the behavior of such circuits is specified, how the circuits are designed for minimum cost or maximum speed of operation, and how the circuits can be tested to ensure correct operation. We also briefly explain how transistors operate, and how they are built on silicon chips.

1.4 Logic Circuit Design in This Book

In this book we use a modern design approach based on the Verilog hardware description language and CAD tools to illustrate many aspects of logic circuit design. We selected this technology because it is widely used in industry and because it enables the readers to implement their designs in FPGA chips, as discussed below. This technology is particularly well-suited for educational purposes because many readers have access to facilities for using CAD tools and programming FPGA devices.

To gain practical experience and a deeper understanding of logic circuits, we advise the reader to implement the examples in this book using CAD software. Most of the major vendors of CAD systems provide their software at no cost to university students for educational use. Some examples are Altera, Cadence, Mentor Graphics, Synopsys, and Xilinx. The CAD systems offered by any of these companies can be used equally well with this book. Two CAD systems that are particularly well-suited for use with this book are the Quartus II software from Altera and the ISE software from Xilinx. Both of these CAD systems support all phases of the design cycle for logic circuits and are powerful and easy to use. The reader is encouraged to visit the website for these companies, where

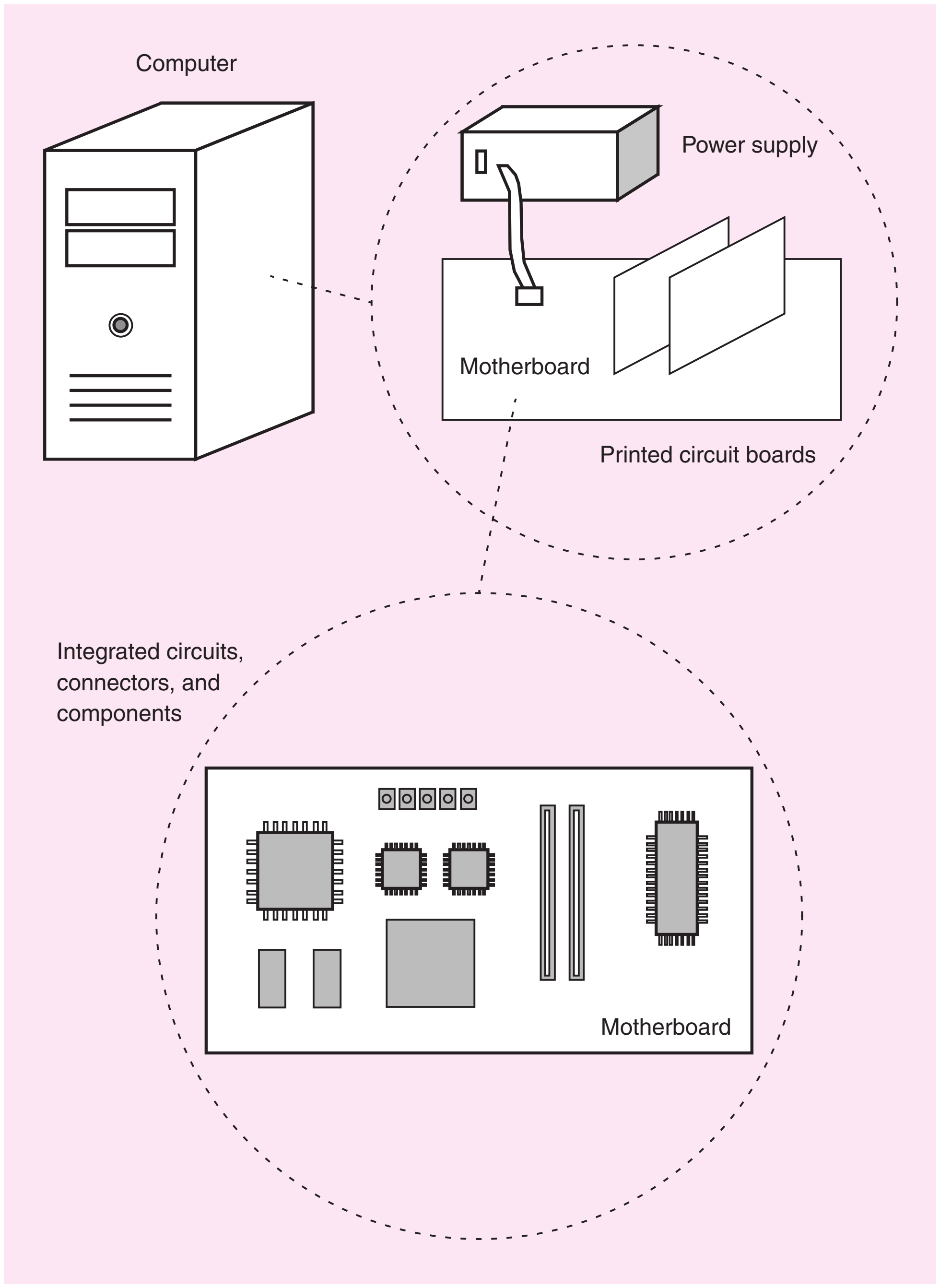


Figure 1.4 A digital hardware system (Part *a*).

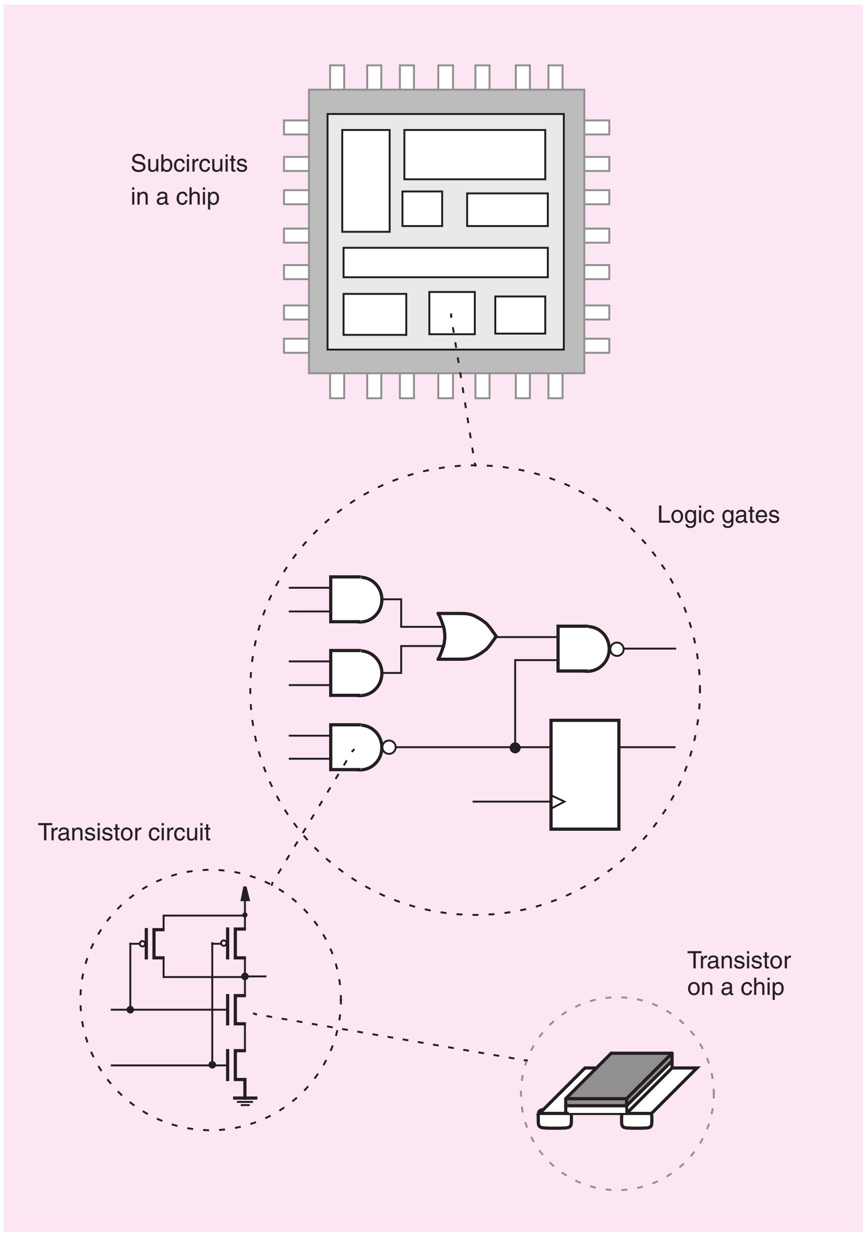


Figure 1.4 A digital hardware system (Part *b*).

the software tools and tutorials that explain their use can be downloaded and installed onto any personal computer.

To facilitate experimentation with logic circuits, some FPGA manufacturers provide special PCBs that include one or more FPGA chips and an interface to a personal computer.

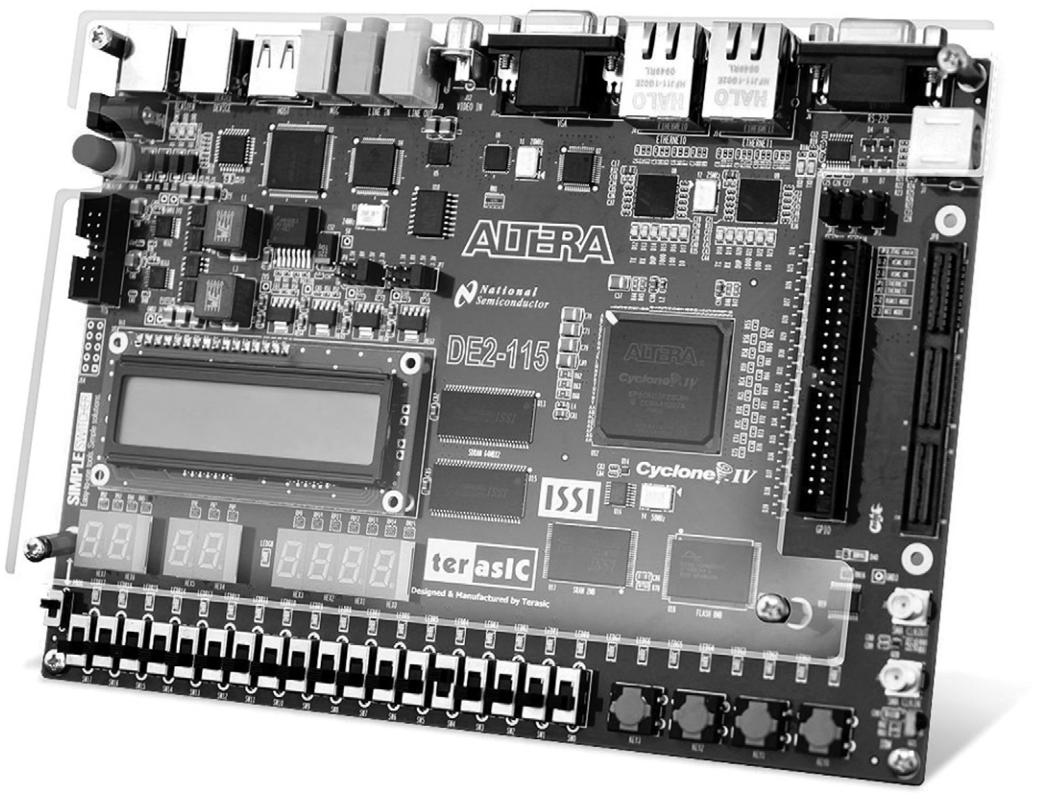


Figure 1.5 An FPGA board.

Once a logic circuit has been designed using the CAD tools, the circuit can be *programmed* into an FPGA on the board. Inputs can then be applied to the FPGA by way of switches and other devices, and the generated outputs can be examined. An example of such a board is depicted in Figure 1.5. This type of board is an excellent vehicle for learning about logic circuits, because it provides a collection of simple input and output devices. Many illustrative experiments can be carried out by designing and implementing logic circuits using the FPGA chip on the board.

1.5 Digital Representation of Information

In Section 1.1 we mentioned that information is represented in logic circuits as electronic signals. Each of these signals can be thought of as representing one *digit* of information. To make the design of logic circuits easier, each digit is allowed to take on only two possible values, usually denoted as 0 and 1. These logic values are implemented as voltage levels in a circuit; the value 0 is usually represented as 0 V (ground), and the value 1 is the voltage level of the circuit's power supply. As we discuss in Appendix B, typical power-supply voltages in logic circuits range from 1 V DC to 5 V DC.

In general, all information in logic circuits is represented as combinations of 0 and 1 digits. Before beginning our discussion of logic circuits in Chapter 2, it will be helpful to examine how numbers, alphanumeric data (text), and other information can be represented using the digits 0 and 1.

1.5.1 Binary Numbers

In the familiar decimal system, a number consists of digits that have 10 possible values, from 0 to 9, and each digit represents a multiple of a power of 10. For example, the number 8547 represents $8 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$. We do not normally write the powers of 10 with the number, because they are implied by the positions of the digits. In general, a decimal integer is expressed by an n-tuple comprising n decimal digits

$$D = d_{n-1}d_{n-2} \cdots d_1 d_0$$

which represents the value

$$V(D) = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10^1 + d_0 \times 10^0$$

This is referred to as the *positional number representation*.

Because the digits have 10 possible values and each digit is weighted as a power of 10, we say that decimal numbers are *base*-10 numbers. Decimal numbers are familiar, convenient, and easy to understand. However, since digital circuits represent information using only the values 0 and 1, it is not practical to have digits that can assume ten values. In these circuits it is more appropriate to use the binary, or *base*-2, system which has only the digits 0 and 1. Each binary digit is called a *bit*. In the binary number system, the same positional number representation is used so that

$$B = b_{n-1}b_{n-2} \cdots b_1 b_0$$

represents an integer that has the value

$$\begin{aligned} V(B) &= b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= \sum_{i=0}^{n-1} b_i \times 2^i \end{aligned} \qquad \textbf{[1.1]}$$

For example, the binary number 1101 represents the value

$$V = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Because a particular digit pattern has different meanings for different bases, we will indicate the base as a subscript when there is potential for confusion. Thus to specify that 1101 is a base-2 number, we will write $(1101)_2$. Evaluating the preceding expression for V gives $V = 8 + 4 + 1 = 13$. Hence

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

Table 1.1 Numbers in decimal and binary.

Decimal representation	Binary representation
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

The range of integers that can be represented by a binary number depends on the number of bits used. Table 1.1 lists the first 15 positive integers and shows their binary representations using four bits. An example of a larger number is $(10110111)_2 = (183)_{10}$. In general, using n bits allows representation of positive integers in the range 0 to $2^n - 1$.

In a binary number the right-most bit is usually referred to as the *least-significant bit (LSB)*. The left-most bit, which has the highest power of 2 associated with it, is called the *most-significant bit (MSB)*. In digital systems it is often convenient to consider several bits together as a group. A group of four bits is called a *nibble*, and a group of eight bits is called a *byte*.

1.5.2 Conversion between Decimal and Binary Systems

A binary number is converted into a decimal number simply by applying Equation 1.1 and evaluating it using decimal arithmetic. Converting a decimal number into a binary number is not quite as straightforward, because we need to construct the number by using powers of 2. For example, the number $(17)_{10}$ is $2^4 + 2^0 = (10001)_2$, and the number $(50)_{10}$ is $2^5 + 2^4 + 2^1 = (110010)_2$. In general, the conversion can be performed by successively dividing the decimal number by 2 as follows. Suppose that a decimal number $D = d_{k-1} \cdots d_1 d_0$, with a value V, is to be converted into a binary number $B = b_{n-1} \cdots b_2 b_1 b_0$. Then, we can write V in the form

$$V = b_{n-1} \times 2^{n-1} + \cdots + b_2 \times 2^2 + b_1 \times 2^1 + b_0$$

Convert $(857)_{10}$

				Remainder	
857	2	=	428	1	LSB
428	2	=	214	0	
214	2	=	107	0	
107	2	=	53	1	
53	2	=	26	1	
26	2	=	13	0	
13	2	=	6	1	
6	2	=	3	0	
3	2	=	1	1	
1	2	=	0	1	MSB

Result is $(1101011001)_2$

Figure 1.6 Conversion from decimal to binary.

If we now divide V by 2, the result is

$$\frac{V}{2} = b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2^1 + b_1 + \frac{b_0}{2}$$

The quotient of this integer division is $b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2 + b_1$, and the remainder is b_0. If the remainder is 0, then $b_0 = 0$; if it is 1, then $b_0 = 1$. Observe that the quotient is just another binary number, which comprises $n - 1$ bits, rather than n bits. Dividing this number by 2 yields the remainder b_1. The new quotient is

$$b_{n-1} \times 2^{n-3} + \cdots + b_2$$

Continuing the process of dividing the new quotient by 2, and determining one bit in each step, will produce all bits of the binary number. The process continues until the quotient becomes 0. Figure 1.6 illustrates the conversion process, using the example $(857)_{10} = (1101011001)_2$. Note that the least-significant bit (LSB) is generated first and the most-significant bit (MSB) is generated last.

So far, we have considered only the representation of positive integers. In Chapter 3 we will complete the discussion of number representation by explaining how negative numbers are handled and how fixed-point and floating-point numbers may be represented. We will also explain how arithmetic operations are performed in computers.

1.5.3 ASCII Character Code

Alphanumeric information, such as letters and numbers typed on a computer keyboard, is represented as codes consisting of 0 and 1 digits. The most common code used for this type of information is known as the *ASCII code*, which stands for the American Standard Code for Information Interchange. The code specified by this standard is presented in Table 1.2.

Table 1.2 The seven-bit ASCII code.

Bit positions	Bit positions 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	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	\	1	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	—	o	DEL

NUL	Null/Idle	SI	Shift in
SOH	Start of header	DLE	Data link escape
STX	Start of text	DC1-DC4	Device control
ETX	End of text	NAK	Negative acknowledgement
EOT	End of transmission	SYN	Synchronous idle
ENQ	Enquiry	ETB	End of transmitted block
ACQ	Acknowledgement	CAN	Cancel (error in data)
BEL	Audible signal	EM	End of medium
BS	Back space	SUB	Special sequence
HT	Horizontal tab	ESC	Escape
LF	Line feed	FS	File separator
VT	Vertical tab	GS	Group separator
FF	Form feed	RS	Record separator
CR	Carriage return	US	Unit separator
SO	Shift out	DEL	Delete/Idle

Bit positions of code format = $\boxed{6}\boxed{5}\boxed{4}\boxed{3}\boxed{2}\boxed{1}\boxed{0}$

The ASCII code uses seven-bit patterns to denote 128 different characters. Ten of the characters are decimal digits 0 to 9. As the table shows, the high-order bits have the same pattern, $b_6b_5b_4 = 011$, for all 10 digits. Each digit is identified by the low-order four bits, b_{3-0}, using the binary patterns for these digits. Capital and lowercase letters are encoded in a way that makes sorting of textual information easy. The codes for A to Z are in ascending numerical sequence, which means that the task of sorting letters (or words) can be accomplished by a simple arithmetic comparison of the codes that represent the letters.

In addition to codes that represent characters and letters, the ASCII code includes punctuation marks such as ! and ?, commonly used symbols such as & and %, and a collection of control characters. The control characters are those needed in computer systems to handle and transfer data among various devices. For example, the carriage return character, which is abbreviated as CR in the table, indicates that the carriage, or cursor position, of an output device, such as a printer or display, should return to the left-most column.

The ASCII code is used to encode information that is handled as text. It is not convenient for representation of numbers that are used as operands in arithmetic operations. For this purpose, it is best to convert ASCII-encoded numbers into a binary representation that we discussed before.

The ASCII standard uses seven bits to encode a character. In computer systems a more natural size is eight bits, or one byte. There are two common ways of fitting an ASCII-encoded character into a byte. One is to set the eighth bit, b_7, to 0. Another is to use this bit to indicate the *parity* of the other seven bits, which means showing whether the number of 1s in the seven-bit code is even or odd. We discuss parity in Chapter 4.

1.5.4 Digital and Analog Information

Binary numbers can be used to represent many types of information. For example, they can represent music that is stored in a personal music player. Figure 1.7 illustrates a music player, which contains an electronic memory for storing music files. A music file comprises a sequence of binary numbers that represent tones. To convert these binary numbers into sound, a *digital-to-analog (D/A) converter* circuit is used. It converts digital values into corresponding voltage levels, which create an analog voltage signal that drives the speakers inside the headphones. The binary values stored in the music player are referred to as *digital* information, whereas the voltage signal that drives the speakers is *analog* information.

1.6 Theory and Practice

Modern design of logic circuits depends heavily on CAD tools, but the discipline of logic design evolved long before CAD tools were invented. This chronology is quite obvious because the very first computers were built with logic circuits, and there certainly were no computers available on which to design them!

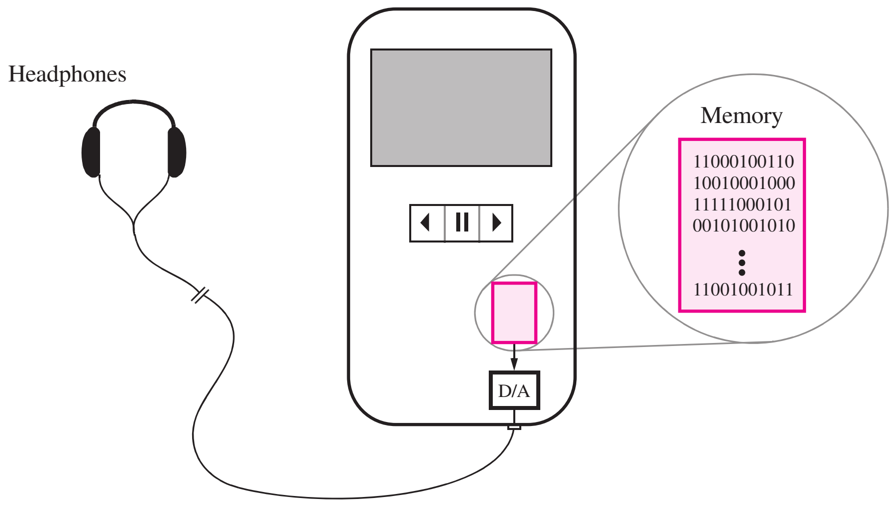


Figure 1.7 Using digital technology to represent music.

Numerous manual design techniques have been developed to deal with logic circuits. Boolean algebra, which we will introduce in Chapter 2, was adopted as a mathematical means for representing such circuits. An enormous amount of "theory" was developed showing how certain design issues may be treated. To be successful, a designer had to apply this knowledge in practice.

CAD tools not only made it possible to design incredibly complex circuits but also made the design work much simpler in general. They perform many tasks automatically, which may suggest that today's designer need not understand the theoretical concepts used in the tasks performed by CAD tools. An obvious question would then be, Why should one study the theory that is no longer needed for manual design? Why not simply learn how to use the CAD tools?

There are three big reasons for learning the relevant theory. First, although the CAD tools perform the automatic tasks of optimizing a logic circuit to meet particular design objectives, the designer has to give the original description of the logic circuit. If the designer specifies a circuit that has inherently bad properties, then the final circuit will also be of poor quality. Second, the algebraic rules and theorems for design and manipulation of logic circuits are directly implemented in today's CAD tools. It is not possible for a user of the tools to understand what the tools do without grasping the underlying theory. Third, CAD tools offer many optional processing steps that a user can invoke when working on a design. The designer chooses which options to use by examining the resulting circuit produced by the CAD tools and deciding whether it meets the required objectives. The only way that the designer can know whether or not to apply a particular option in a given situation is to know what the CAD tools will do if that option is invoked—again, this implies that the designer must be familiar with the underlying theory. We discuss the logic circuit theory extensively in this book, because it is not possible to become an effective logic circuit designer without understanding the fundamental concepts.

There is another good reason to learn some logic circuit theory even if it were not required for CAD tools. Simply put, it is interesting and intellectually challenging. In the modern world filled with sophisticated automatic machinery, it is tempting to rely on tools as a substitute for thinking. However, in logic circuit design, as in any type of design process, computer-based tools are not a substitute for human intuition and innovation. Computer-based tools can produce good digital hardware designs only when employed by a designer who thoroughly understands the nature of logic circuits.

PROBLEMS

Answers to problems marked by an asterisk are given at the back of the book.

***1.1** Convert the following decimal numbers into binary, using the method shown in Figure 1.6.
(a) $(20)_{10}$
(b) $(100)_{10}$
(c) $(129)_{10}$
(d) $(260)_{10}$
(e) $(10240)_{10}$

1.2 Convert the following decimal numbers into binary, using the method shown in Figure 1.6.
(a) $(30)_{10}$
(b) $(110)_{10}$
(c) $(259)_{10}$
(d) $(500)_{10}$
(e) $(20480)_{10}$

1.3 Convert the following decimal numbers into binary, using the method shown in Figure 1.6.
(a) $(1000)_{10}$
(b) $(10000)_{10}$
(c) $(100000)_{10}$
(c) $(1000000)_{10}$

***1.4** In Figure 1.6 we show how to convert a decimal number into binary by successively dividing by 2. Another way to derive the answer is to constuct the number by using powers of 2. For example, if we wish to convert the number $(23)_{10}$, then the largest power of 2 that is not larger than 23 is $2^4 = 16$. Hence, the binary number will have five bits and the most-significant bit is $b_4 = 1$. We then perform the subtraction $23 - 16 = 7$. Now, the largest power of 2 that is not larger than 7 is $2^2 = 4$. Hence, $b_3 = 0$ (because $2^3 = 8$ is larger than 7) and $b_2 = 1$. Continuing this process gives

$$\begin{aligned} 23 &= 16 + 4 + 2 + 1 \\ &= 2^4 + 2^2 + 2^1 + 2^0 \\ &= 10000 + 00100 + 00010 + 00001 \\ &= 10111 \end{aligned}$$

Using this method, convert the following decimal numbers into binary.
(a) $(17)_{10}$
(b) $(33)_{10}$
(c) $(67)_{10}$
(d) $(130)_{10}$
(e) $(2560)_{10}$
(f) $(51200)_{10}$

1.5 Repeat Problem 3 using the method described in Problem 4.

***1.6** Convert the following binary numbers into decimal.
(a) $(1001)_2$
(b) $(11100)_2$
(c) $(111111)_2$
(d) $(101010101010)_2$

1.7 Convert the following binary numbers into decimal.
(a) $(110010)_2$
(b) $(1100100)_2$
(c) $(11001000)_2$
(d) $(110010000)_2$

***1.8** What is the minimum number of bits needed to represent the following decimal numbers in binary?
(a) $(270)_{10}$
(b) $(520)_{10}$
(c) $(780)_{10}$
(d) $(1029)_{10}$

1.9 Repeat Problem 8 for the following decimal numbers:
(a) $(111)_{10}$
(b) $(333)_{10}$
(c) $(555)_{10}$
(d) $(1111)_{10}$

REFERENCES

1. "International Technology Roadmap for Semiconductors," http://www.itrs.net
2. Altera Corporation, "Altera Field Programmable Gate Arrays Product Literature," http://www.altera.com
3. Xilinx Corporation, "Xilinx Field Programmable Gate Arrays Product Literature," http://www.xilinx.com

chapter

2

Introduction to Logic Circuits

Chapter Objectives

In this chapter you will be introduced to:

- Logic functions and circuits
- Boolean algebra for dealing with logic functions
- Logic gates and synthesis of simple circuits
- CAD tools and the Verilog hardware description language
- Minimization of functions and Karnaugh maps

The study of logic circuits is motivated mostly by their use in digital computers. But such circuits also form the foundation of many other digital systems, such as those that perform control applications or are involved in digital communications. All such applications are based on some simple logical operations that are performed on input information.

In Chapter 1 we showed that information in computers is represented as electronic signals that can have two discrete values. Although these values are implemented as voltage levels in a circuit, we refer to them simply as logic values, 0 and 1. Any circuit in which the signals are constrained to have only some number of discrete values is called a [illegible]. Logic circuits can be designed with different numbers of logic values, such as three, four, or even more, but in this book we deal only with the *binary* logic circuits that have two logic values.

Binary logic circuits have the dominant role in digital technology. We hope to provide the reader with an understanding of how these circuits work, how are they represented in mathematical notation, and how are they designed using modern design automation techniques. We begin by introducing some basic concepts pertinent to the binary logic circuits.

2.1 VARIABLES AND FUNCTIONS

The dominance of binary circuits in digital systems is a consequence of their simplicity, which results from constraining the signals to assume only two possible values. The simplest binary element is a switch that has two states. If a given switch is controlled by an input variable x, then we will say that the switch is open if $x = 0$ and closed if $x = 1$, as illustrated in Figure 2.1*a*. We will use the graphical symbol in Figure 2.1*b* to represent such switches in the diagrams that follow. Note that the control input x is shown explicitly in the symbol. In Appendix B we explain how such switches are implemented with transistors.

Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure 2.2*a*. A battery provides the power source. The lightbulb glows when a sufficient amount of current passes through it.

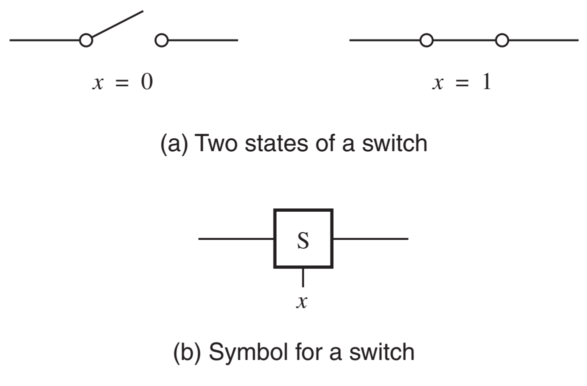
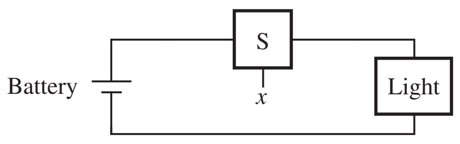
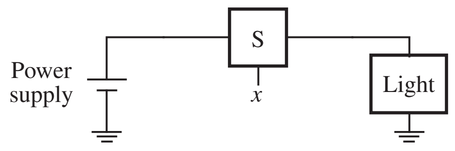


Figure 2.1 A binary switch.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

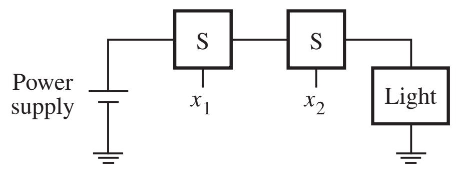
The current flows when the switch is closed, that is, when $x = 1$. In this example the input that causes changes in the behavior of the circuit is the switch control x. The output is defined as the state (or condition) of the light, which we will denote by the letter L. If the light is on, we will say that $L = 1$. If the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light as a function of the input variable x. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

$$L(x) = x$$

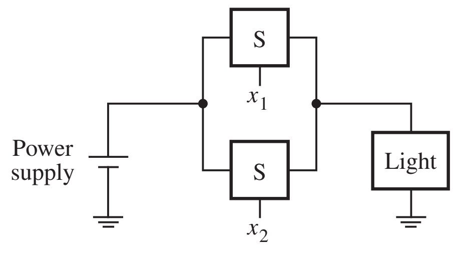
This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.

The circuit in Figure 2.2*a* can be found in an ordinary flashlight, where the switch is a simple mechanical device. In an electronic circuit the switch is implemented as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, usually in the range of 1 to 5 volts. One side of the power supply provides the *circuit ground*, as illustrated in Figure 2.2*b*. The circuit ground is a common reference point for voltages in the circuit. Rather than drawing wires in a circuit diagram for all nodes that return to the circuit ground, the diagram can be simplified by showing a connection to a ground symbol, as we have done for the bottom terminal of the light L in the figure. In the circuit diagrams that follow we will use this convention, because it makes the diagrams look simpler.

Consider now the possibility of using two switches to control the state of the light. Let x_1 and x_2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches are closed. If either switch is open, the light will be off.



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

This behavior can be described by the expression

$$
\begin{aligned}
& L(x_1, x_2) = x_1 \cdot x_2 \\
\text{where}\quad & L = 1 \text{ if } x_1 = 1 \text{ and } x_2 = 1, \\
& L = 0 \text{ otherwise.}
\end{aligned}
$$

The "·" symbol is called the *AND operator*, and the circuit in Figure 2.3*a* is said to implement a *logical AND function*.

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either the x_1 or x_2 switch is closed. The light will also be on if both switches are closed. The light will be off only if both switches are open. This behavior can be stated as

$$
\begin{aligned}
& L(x_1, x_2) = x_1 + x_2 \\
\text{where}\quad & L = 1 \text{ if } x_1 = 1 \text{ or } x_2 = 1 \text{ or if } x_1 = x_2 = 1, \\
& L = 0 \text{ if } x_1 = x_2 = 0.
\end{aligned}
$$

The + symbol is called the *OR operator*, and the circuit in Figure 2.3*b* is said to implement a *logical OR function*. It is important not to confuse the use of the + symbol with its more common meaning, which is for arithmetic addition. In this chapter the + symbol represents the logical OR operation unless otherwise stated.

In the above expressions for AND and OR, the output $L(x_1, x_2)$ is a logic function with input variables x_1 and x_2. The AND and OR functions are two of the most important logic functions. Together with some other simple functions, they can be used as building blocks for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be

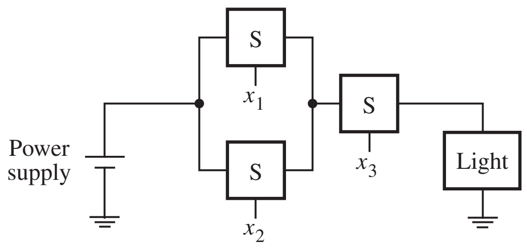


Figure 2.4 A series-parallel connection.

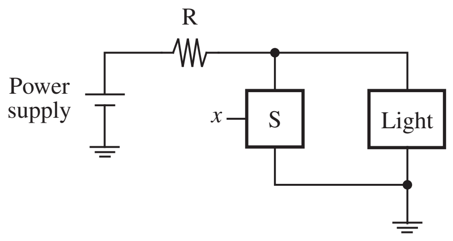


Figure 2.5 An inverting circuit.

used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

$$L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$$

The light is on if $x_3 = 1$ and, at the same time, at least one of the x_1 or x_2 inputs is equal to 1.

2.2 INVERSION

So far we have assumed that some positive action takes place when a switch is closed, such as turning the light on. It is equally interesting and useful to consider the possibility that a positive action takes place when a switch is opened. Suppose that we connect the light as shown in Figure 2.5. In this case the switch is connected in parallel with the light, rather than in series. Consequently, a closed switch will short-circuit the light and prevent the current from flowing through it. Note that we have included an extra resistor in this circuit to ensure that the closed switch does not short-circuit the power supply. The light will be turned on when the switch is opened. Formally, we express this functional behavior as

$$\begin{aligned} L(x) &= \overline{x} \\ \text{where} \quad L &= 1 \text{ if } x = 0, \\ L &= 0 \text{ if } x = 1 \end{aligned}$$

The value of this function is the inverse of the value of the input variable. Instead of using the word *inverse*, it is more common to use the term *complement*. Thus we say that $L(x)$ is a complement of x in this example. Another frequently used term for the same operation is the *NOT operation*. There are several commonly used notations for indicating the complementation. In the preceding expression we placed an overbar on top of x. This notation is probably the best from the visual point of view. However, when complements are needed in expressions that are typed using a computer keyboard, which is often done when using CAD tools, it is impractical to use overbars. Instead, either an apostrophe is placed after the variable, or an exclamation mark (!), the tilde character (∼), or the word NOT is placed in front of the variable to denote the complementation. Thus the following are equivalent:

$$\overline{x} = x' = !x = \sim x = \text{NOT } x$$

The complement operation can be applied to a single variable or to more complex operations. For example, if

$$f(x_1, x_2) = x_1 + x_2$$

then the complement of f is

$$\overline{f}(x_1, x_2) = \overline{x_1 + x_2}$$

This expression yields the logic value 1 only when neither x_1 nor x_2 is equal to 1, that is, when $x_1 = x_2 = 0$. Again, the following notations are equivalent:

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2) = \text{NOT } (x_1 + x_2)$$

2.3 Truth Tables

We have introduced the three most basic logic operations—AND, OR, and complement—by relating them to simple circuits built with switches. This approach gives these operations a certain "physical meaning." The same operations can also be defined in the form of a table, called a *truth table*, as shown in Figure 2.6. The first two columns (to the left of the double vertical line) give all four possible combinations of logic values that the variables x_1 and x_2

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 2.6 A truth table for the AND and OR operations.

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.7 Three-input AND and OR operations.

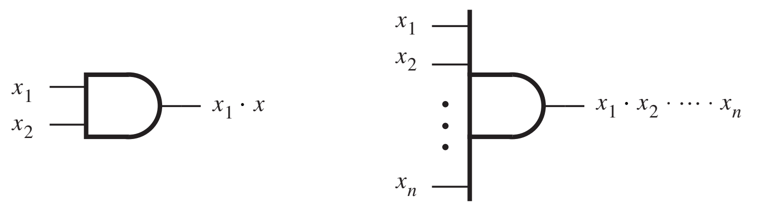
can have. The next column defines the AND operation for each combination of values of x_1 and x_2, and the last column defines the OR operation. Because we will frequently need to refer to "combinations of logic values" applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

The truth table is a useful aid for depicting information involving logic functions. We will use it in this book to define specific functions and to show the validity of certain functional relations. Small truth tables are easy to deal with. However, they grow exponentially in size with the number of variables. A truth table for three input variables has eight rows because there are eight possible valuations of these variables. Such a table is given in Figure 2.7, which defines three-input AND and OR functions. For four input variables the truth table has 16 rows, and so on. In general, for n input variables the truth table has 2^n rows.

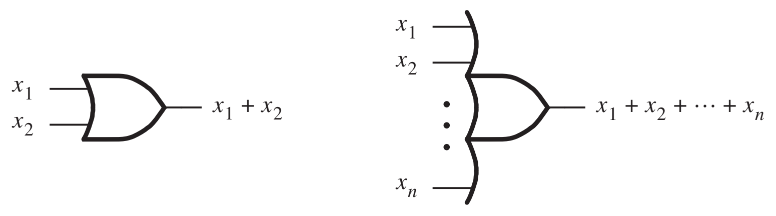
The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if one or more of the variables is equal to 1.

2.4 Logic Gates and Networks

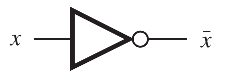
The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 2.8. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are



(a) AND gates



(b) OR gates



(c) NOT gate

Figure 2.8 The basic gates.

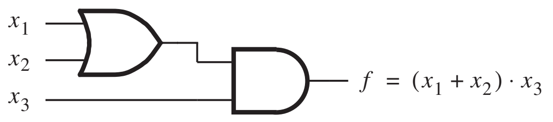


Figure 2.9 The function from Figure 2.4.

augmented to accommodate a greater number of inputs. We show how logic gates are built using transistors in Appendix B.

A larger circuit is implemented by a *network* of gates. For example, the logic function from Figure 2.4 can be implemented by the network in Figure 2.9. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible. We will see shortly that a given logic function can be implemented with a number of different networks. Some of these networks are simpler than others, hence searching for the solutions that entail minimum cost is prudent.

In technical jargon a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.

2.4.1 ANALYSIS OF A LOGIC NETWORK

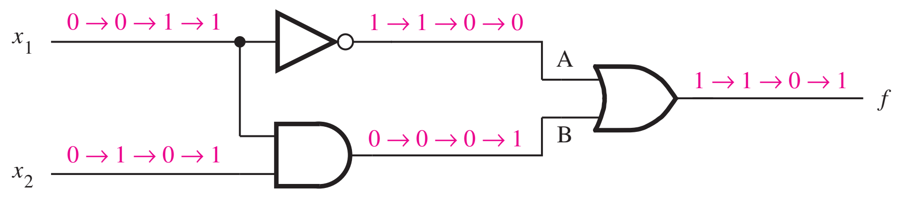
A designer of digital systems is faced with two basic issues. For an existing logic network, it must be possible to determine the function performed by the network. This task is referred to as the *analysis* process. The reverse task of designing a new network that implements a desired functional behavior is referred to as the *synthesis* process. The analysis process is rather straightforward and much simpler than the synthesis process.

Figure 2.10*a* shows a simple network consisting of three gates. To analyze its functional behavior, we can consider what happens if we apply all possible combinations of the input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, $f = 1$ if $x_1 = x_2 = 0$. If we then let $x_1 = 0$ and $x_2 = 1$, no change in the value of f will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply $x_1 = 1$ and $x_2 = 0$, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of f will be 0. Finally, let $x_1 = x_2 = 1$. Then the output of the AND gate goes to 1, which in turn causes f to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.

Timing Diagram

We have determined the behavior of the network in Figure 2.10*a* by considering the four possible valuations of the inputs x_1 and x_2. Suppose that the signals that correspond to these valuations are applied to the network in the order of our discussion; that is, $(x_1, x_2) = (0, 0)$ followed by $(0, 1)$, $(1, 0)$, and $(1, 1)$. Then changes in the signals at various points in the network would be as indicated in blue in the figure. The same information can be presented in graphical form, known as a *timing diagram*, as shown in Figure 2.10*c*. The time runs from left to right, and each input valuation is held for some fixed duration. The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled A and B.

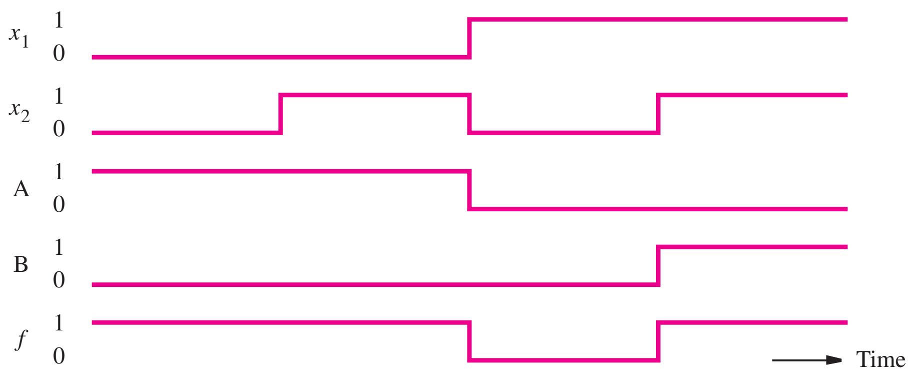
The timing diagram in Figure 2.10*c* shows that changes in the waveforms at points A and B and the output f take place instantaneously when the inputs x_1 and x_2 change their values. These idealized waveforms are based on the assumption that logic gates respond to changes on their inputs in zero time. Such timing diagrams are useful for indicating the *functional behavior* of logic circuits. However, practical logic gates are implemented using electronic circuits which need some time to change their states. Thus, there is a delay between a change in input values and a corresponding change in the output value of a gate. In chapters that follow we will use timing diagrams that incorporate such delays.



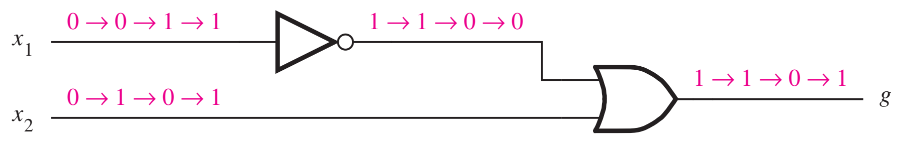
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

x_1	x_2	$f(x_1, x_2)$	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

(b) Truth table



(c) Timing diagram



(d) Network that implements $g = \bar{x}_1 + x_2$

Figure 2.10 An example of logic networks.

Timing diagrams are used for many purposes. They depict the behavior of a logic circuit in a form that can be observed when the circuit is tested using instruments such as logic analyzers and oscilloscopes. Also, they are often generated by CAD tools to show the designer how a given circuit is expected to behave before it is actually implemented electronically. We will introduce the CAD tools later in this chapter and will make use of them throughout the book.

Functionally Equivalent Networks

Now consider the network in Figure 2.10*d*. Going through the same analysis procedure, we find that the output g changes in exactly the same way as f does in part (*a*) of the figure. Therefore, $g(x_1, x_2) = f(x_1, x_2)$, which indicates that the two networks are functionally equivalent; the output behavior of both networks is represented by the truth table in Figure 2.10*b*. Since both networks realize the same function, it makes sense to use the simpler one, which is less costly to implement.

In general, a logic function can be implemented with a variety of different networks, probably having different costs. This raises an important question: How does one find the best implementation for a given function? We will discuss some of the main approaches for synthesizing logic functions later in this chapter. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in Figure 2.10*d*. Since $f(x_1, x_2) = \overline{x}_1 + x_1 \cdot x_2$ and $g(x_1, x_2) = \overline{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

$$\overline{x}_1 + x_1 \cdot x_2 = \overline{x}_1 + x_2$$

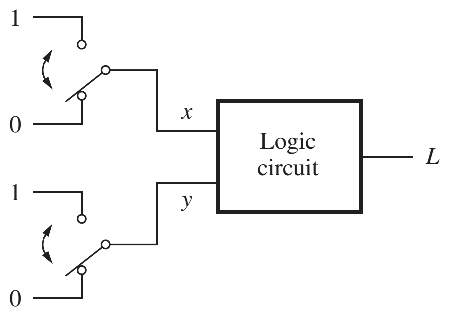
We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In Section 2.5 we will introduce a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

Example 2.1

As an example of a logic function, consider the diagram in Figure 2.11*a*. It includes two toggle switches that control the values of signals x and y. Each toggle switch can be pushed down to the bottom position or up to the top position. When a toggle switch is in the bottom position it makes a connection to logic value 0 (ground), and when in the top position it connects to logic value 1 (power supply level). Thus, these toggle switches can be used to set x and y to either 0 or 1.

The signals x and y are inputs to a logic circuit that controls a light L. The required behavior is that the light should be on only if one, but not both, of the toggle switches is in the top position. This specification leads to the truth table in part (*b*) of the figure. Since $L = 1$ when $x = 0$ and $y = 1$ or when $x = 1$ and $y = 0$, we can implement this logic function using the network in Figure 2.11*c*.

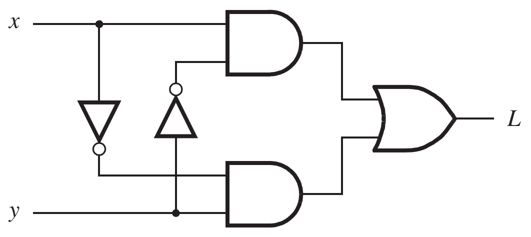
The reader may recognize the behavior of our light as being similar to that over a set of stairs in a house, where the light is controlled by two switches: one at the top of the stairs, and the other at the bottom. The light can be turned on or off by either switch because



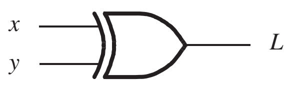
(a) Two switches that control a light

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Logic network



(d) XOR gate symbol

Figure 2.11 An example of a logic circuit.

it follows the truth table in Figure 2.11*b*. This logic function, which differs from the OR function only when both inputs are equal to 1, is useful for other applications as well. It is called the *exclusive-OR* (XOR) function and is indicated in logic expressions by the symbol $\oplus$. Thus, rather than writing $L = \overline{x} \cdot y + x \cdot \overline{y}$, we can write $L = x \oplus y$. The XOR function has the logic-gate symbol illustrated in Figure 2.11*d*.

Example 2.2 In Chapter 1 we showed how numbers are represented in computers by using binary digits. As another example of logic functions, consider the addition of two one-digit binary numbers a and b. The four possible valuations of a, b and the resulting sums are given in Figure 2.12*a* (in this figure the $+$ operator signifies *addition*). The sum $S = s_1 s_0$ has to be a two-digit binary number, because when $a = b = 1$ then $S = 10$.

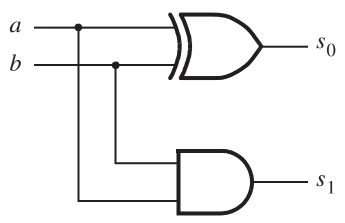
Figure 2.12*b* gives a truth table for the logic functions s_1 and s_0. From this table we can see that $s_1 = a \cdot b$ and $s_0 = a \oplus b$. The corresponding logic network is given in part (*c*) of the figure. This type of logic circuit, which adds binary numbers, is referred to as an *adder* circuit. We discuss circuits of this type in Chapter 3.

$$\begin{array}{r} a \\ +\,b \\ \hline s_1\,s_0 \end{array} \qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\;0 \end{array} \quad \begin{array}{r} 0 \\ +\,1 \\ \hline 0\;1 \end{array} \quad \begin{array}{r} 1 \\ +\,0 \\ \hline 0\;1 \end{array} \quad \begin{array}{r} 1 \\ +\,1 \\ \hline 1\;0 \end{array}$$

(a) Evaluation of $S = a + b$

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Logic network

Figure 2.12 Addition of binary numbers.

2.5 Boolean Algebra

In 1849 George Boole published a scheme for the algebraic description of processes involved in logical thought and reasoning [1]. Subsequently, this scheme and its further refinements became known as *Boolean algebra*. It was almost 100 years later that this algebra found application in the engineering sense. In the late 1930s Claude Shannon showed that Boolean algebra provides an effective means of describing circuits built with switches [2]. The algebra can, therefore, be used to describe logic circuits. We will show that this algebra is a powerful tool that can be used for designing and analyzing logic circuits. The reader will come to appreciate that it provides the foundation for much of our modern digital technology.

Axioms of Boolean Algebra

Like any algebra, Boolean algebra is based on a set of rules that are derived from a small number of basic assumptions. These assumptions are called *axioms*. Let us assume that Boolean algebra involves elements that take on one of two values, 0 and 1. Assume that the following axioms are true:

1*a*. $0 \cdot 0 = 0$

1*b*. $1 + 1 = 1$

2*a*. $1 \cdot 1 = 1$

2*b*. $0 + 0 = 0$

3*a*. $0 \cdot 1 = 1 \cdot 0 = 0$

3*b*. $1 + 0 = 0 + 1 = 1$

4*a*. If $x = 0$, then $\overline{x} = 1$

4*b*. If $x = 1$, then $\overline{x} = 0$

Single-Variable Theorems

From the axioms we can define some rules for dealing with single variables. These rules are often called *theorems*. If x is a Boolean variable, then the following theorems hold:

5*a*. $x \cdot 0 = 0$

5*b*. $x + 1 = 1$

6*a*. $x \cdot 1 = x$

6*b*. $x + 0 = x$

7*a*. $x \cdot x = x$

7*b*. $x + x = x$

8*a*. $x \cdot \overline{x} = 0$

8*b*. $x + \overline{x} = 1$

9. $\overline{\overline{x}} = x$

It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above. For example, in theorem 5*a*, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

Duality

Notice that we have listed the axioms and the single-variable theorems in pairs. This is done to reflect the important *principle of duality*. Given a logic expression, its *dual* is obtained by replacing all $+$ operators with $\cdot$ operators, and vice versa, and by replacing all 0s with 1s, and vice versa. The dual of any true statement (axiom or theorem) in Boolean algebra is also a true statement. At this point in the discussion, the reader might not appreciate why duality is a useful concept. However, this concept will become clear later in the chapter, when we will show that duality implies that at least two different ways exist to express every logic function with Boolean algebra. Often, one expression leads to a simpler physical implementation than the other and is thus preferable.

Two- and Three-Variable Properties

To enable us to deal with a number of variables, it is useful to define some two- and three-variable algebraic identities. For each identity, its dual version is also given. These identities are often referred to as *properties*. They are known by the names indicated below. If x, y, and z are Boolean variables, then the following properties hold:

10*a*.	$x \cdot y = y \cdot x$	*Commutative*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Associative*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
12*b*.	$x + y \cdot z = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absorption*
13*b*.	$x \cdot (x + y) = x$	
14*a*.	$x \cdot y + x \cdot \overline{y} = x$	*Combining*
14*b*.	$(x + y) \cdot (x + \overline{y}) = x$	
15*a*.	$\overline{x \cdot y} = \overline{x} + \overline{y}$	*DeMorgan's theorem*
15*b*.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	
16*a*.	$x + \overline{x} \cdot y = x + y$	
16*b*.	$x \cdot (\overline{x} + y) = x \cdot y$	
17*a*.	$x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$	*Consensus*
17*b*.	$(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$	

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. Figure 2.13 illustrates how perfect induction can be used to prove DeMorgan's theorem, using the format of a truth table. The evaluation of left-hand and right-hand sides of the identity in 15*a* gives the same result.

We have listed a number of axioms, theorems, and properties. Not all of these are necessary to define Boolean algebra. For example, assuming that the + and · operations are defined, it is sufficient to include theorems 5 and 8 and properties 10 and 12. These are sometimes referred to as Huntington's basic postulates [3]. The other identities can be derived from these postulates.

The preceding axioms, theorems, and properties provide the information necessary for performing algebraic manipulation of more complex expressions.

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\overline{y}$	$\overline{x} + \overline{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0
		LHS		RHS		

Figure 2.13 Proof of DeMorgan's theorem in 15*a*.

Example 2.3 Let us prove the validity of the logic equation

$$(x_1 + x_3) \cdot (\overline{x}_1 + \overline{x}_3) = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

The left-hand side can be manipulated as follows. Using the distributive property, 12*a*, gives

$$\text{LHS} = (x_1 + x_3) \cdot \overline{x}_1 + (x_1 + x_3) \cdot \overline{x}_3$$

Applying the distributive property again yields

$$\text{LHS} = x_1 \cdot \overline{x}_1 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + x_3 \cdot \overline{x}_3$$

Note that the distributive property allows ANDing the terms in parenthesis in a way analogous to multiplication in ordinary algebra. Next, according to theorem 8*a*, the terms $x_1 \cdot \overline{x}_1$ and $x_3 \cdot \overline{x}_3$ are both equal to 0. Therefore,

$$\text{LHS} = 0 + x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3 + 0$$

From 6*b* it follows that

$$\text{LHS} = x_3 \cdot \overline{x}_1 + x_1 \cdot \overline{x}_3$$

Finally, using the commutative property, 10*a* and 10*b*, this becomes

$$\text{LHS} = x_1 \cdot \overline{x}_3 + \overline{x}_1 \cdot x_3$$

which is the same as the right-hand side of the initial equation.

Example 2.4 Consider the logic equation

$$x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 = \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2$$

The left-hand side can be manipulated as follows

$$\begin{aligned}
\text{LHS} &= x_1 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot \overline{x}_3 + \overline{x}_2 \cdot x_3 && \text{using } 10b \\
&= x_1 \cdot (\overline{x}_3 + x_3) + \overline{x}_2 \cdot (\overline{x}_3 + x_3) && \text{using } 12a \\
&= x_1 \cdot 1 + \overline{x}_2 \cdot 1 && \text{using } 8b \\
&= x_1 + \overline{x}_2 && \text{using } 6a
\end{aligned}$$

The right-hand side can be manipulated as

$$\begin{aligned}
\text{RHS} &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot (x_2 + \overline{x}_2) && \text{using } 12a \\
&= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot 1 && \text{using } 8b \\
&= \overline{x}_1 \cdot \overline{x}_2 + x_1 && \text{using } 6a \\
&= x_1 + \overline{x}_1 \cdot \overline{x}_2 && \text{using } 10b \\
&= x_1 + \overline{x}_2 && \text{using } 16a
\end{aligned}$$

Being able to manipulate both sides of the initial equation into identical expressions establishes the validity of the equation. Note that the same logic function is represented by either the left- or the right-hand side of the above equation; namely

$$\begin{aligned} f(x_1, x_2, x_3) &= x_1 \cdot \overline{x}_3 + \overline{x}_2 \cdot \overline{x}_3 + x_1 \cdot x_3 + \overline{x}_2 \cdot x_3 \\ &= \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_2 + x_1 \cdot \overline{x}_2 \end{aligned}$$

As a result of manipulation, we have found a much simpler expression

$$f(x_1, x_2, x_3) = x_1 + \overline{x}_2$$

which also represents the same function. This simpler expression would result in a lower-cost logic circuit that could be used to implement the function.

Examples 2.3 and 2.4 illustrate the purpose of the axioms, theorems, and properties as a mechanism for algebraic manipulation. Even these simple examples suggest that it is impractical to deal with highly complex expressions in this way. However, these theorems and properties provide the basis for automating the synthesis of logic functions in CAD tools. To understand what can be achieved using these tools, the designer needs to be aware of the fundamental concepts.

2.5.1 The Venn Diagram

We have suggested that perfect induction can be used to verify the theorems and properties. This procedure is quite tedious and not very informative from the conceptual point of view. A simple visual aid that can be used for this purpose also exists. It is called the Venn diagram, and the reader is likely to find that it provides for a more intuitive understanding of how two expressions may be equivalent.

The Venn diagram has traditionally been used in mathematics to provide a graphical illustration of various operations and relations in the algebra of sets. A set s is a collection of elements that are said to be the members of s. In the Venn diagram the elements of a set are represented by the area enclosed by a contour such as a square, a circle, or an ellipse. For example, in a universe N of integers from 1 to 10, the set of even numbers is $E = \{2, 4, 6, 8, 10\}$. A contour representing E encloses the even numbers. The odd numbers form the complement of E; hence the area outside the contour represents $\overline{E} = \{1, 3, 5, 7, 9\}$.

Since in Boolean algebra there are only two values (elements) in the universe, $B = \{0, 1\}$, we will say that the area within a contour corresponding to a set s denotes that $s = 1$, while the area outside the contour denotes $s = 0$. In the diagram we will shade the area where $s = 1$. The concept of the Venn diagram is illustrated in Figure 2.14. The universe B is represented by a square. Then the constants 1 and 0 are represented as shown in parts (a) and (b) of the figure. A variable, say, x, is represented by a circle, such that the area inside the circle corresponds to $x = 1$, while the area outside the circle corresponds to $x = 0$. This is illustrated in part (c). An expression involving one or more variables is depicted by

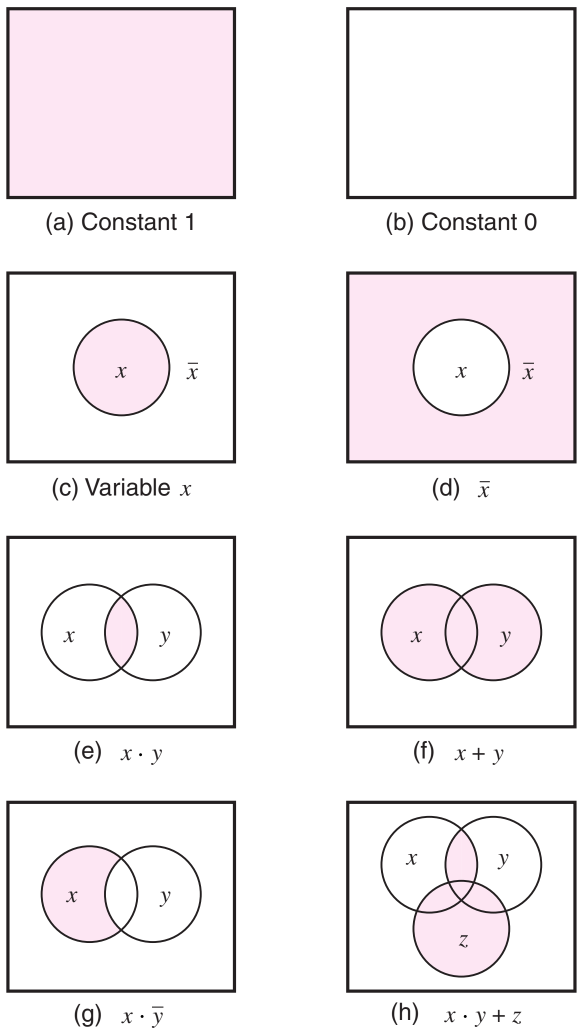


Figure 2.14 The Venn diagram representation.

shading the area where the value of the expression is equal to 1. Part (*d*) indicates how the complement of x is represented.

To represent two variables, x and y, we draw two overlapping circles. Then the area where the circles overlap represents the case where $x = y = 1$, namely, the AND of x and y, as shown in part (*e*). Since this common area consists of the intersecting portions of x and y, the AND operation is often referred to formally as the *intersection* of x and y. Part (*f*) illustrates the OR operation, where $x + y$ represents the total area within both circles,

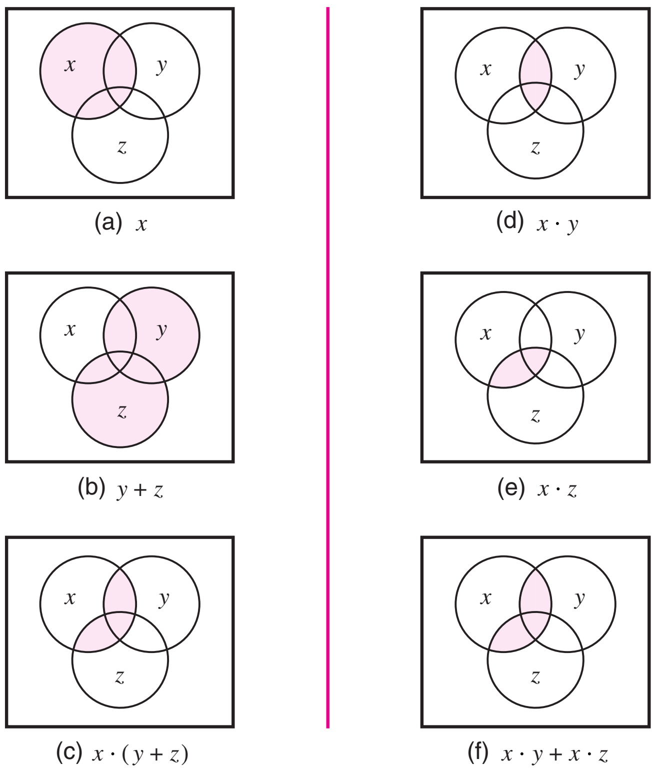


Figure 2.15 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

namely, where at least one of x or y is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of x and y.

Part (g) depicts the term $x \cdot \overline{y}$, which is represented by the intersection of the area for x with that for $\overline{y}$. Part (h) gives a three-variable example; the expression $x \cdot y + z$ is the union of the area for z with that of the intersection of x and y.

To see how we can use Venn diagrams to verify the equivalence of two expressions, let us demonstrate the validity of the distributive property, 12a, in Section 2.5. Figure 2.15 gives the construction of the left and right sides of the identity that defines the property

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

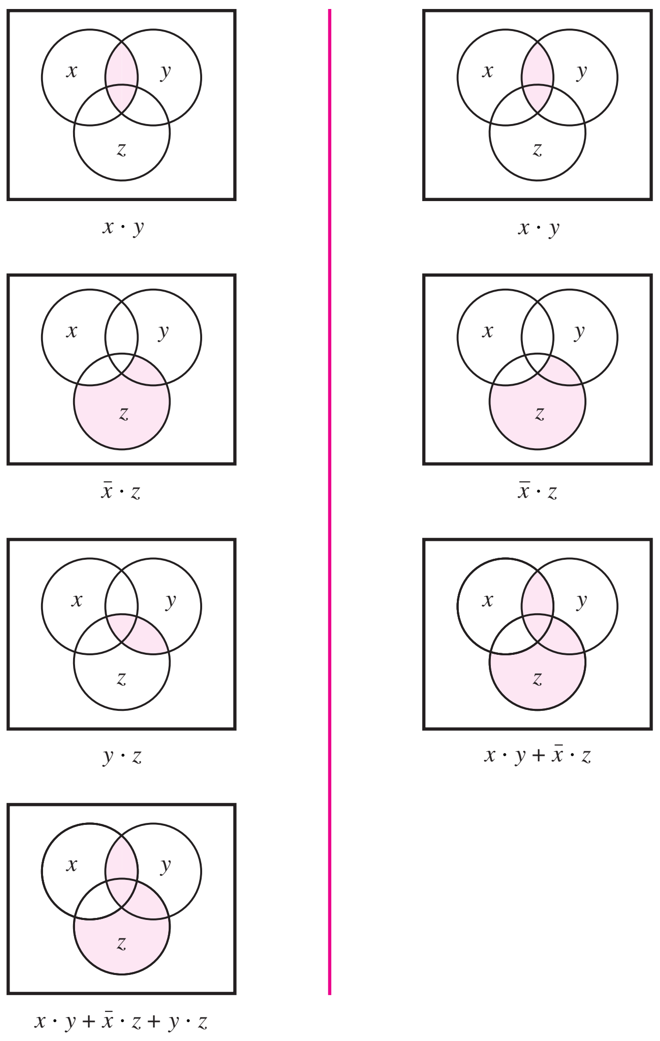
Part (a) shows the area where $x = 1$. Part (b) indicates the area for $y + z$. Part (c) gives the diagram for $x \cdot (y + z)$, the intersection of shaded areas in parts (a) and (b). The right-hand side is constructed in parts (d), (e), and (f). Parts (d) and (e) describe the terms $x \cdot y$ and $x \cdot z$, respectively. The union of the shaded areas in these two diagrams then corresponds to the expression $x \cdot y + x \cdot z$, as seen in part (f). Since the shaded areas in parts (c) and (f) are identical, it follows that the distributive property is valid.

As another example, consider the identity

$$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$$

which is illustrated in Figure 2.16. Notice that this identity states that the term $y \cdot z$ is fully covered by the terms $x \cdot y$ and $\overline{x} \cdot z$; therefore, this term can be omitted. This identity, which we listed earlier as property 17a, is often referred to as *consensus*.

The reader should use the Venn diagram to prove some other identities. The examples below prove the distributive property 12b, and DeMorgan's theorem, 15a.



$x \cdot y$
$\bar{x} \cdot z$
$y \cdot z$
$x \cdot y + \bar{x} \cdot z + y \cdot z$
$x \cdot y$
$\bar{x} \cdot z$
$x \cdot y + \bar{x} \cdot z$
x
y
z

Figure 2.16 Verification of $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$.

Example 2.5

The distributive property 12*a* in Figure 2.15 will look familiar to the reader, because it is valid both for Boolean variables and for variables that are real numbers. In the case of real-number variables, the operations involved would be multiplication and addition, rather than logical AND and OR. However, the dual form 12*b* of this property, $x + y \cdot z = (x + y) \cdot (x + z)$, does not hold for real-number variables involving multiplication and addition operations. To prove that this identity is valid in Boolean algebra we can use the Venn diagrams in Figure 2.17. Parts (*a*) and (*b*) of the figure depict the terms x and $y \cdot z$, respectively, and part (*c*) gives the union of parts (*a*) and (*b*). Parts (*d*) and (*e*) depict the sum terms $(x + y)$ and $(x + z)$, and part (*f*) shows the intersection of (*d*) and (*e*). Since the diagrams in (*c*) and (*f*) are the same, this proves the identity.

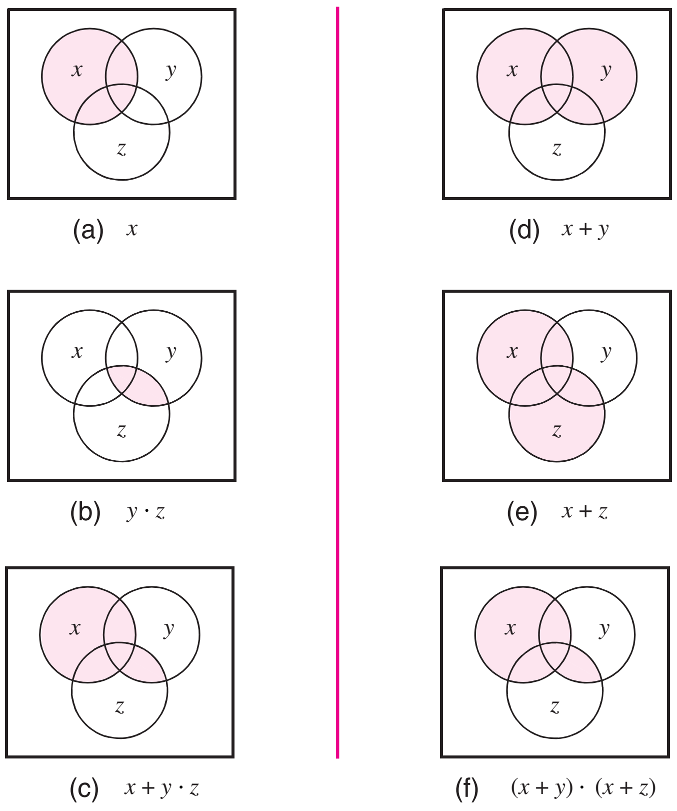


Figure 2.17 Proof of the distributive property 12*b*.

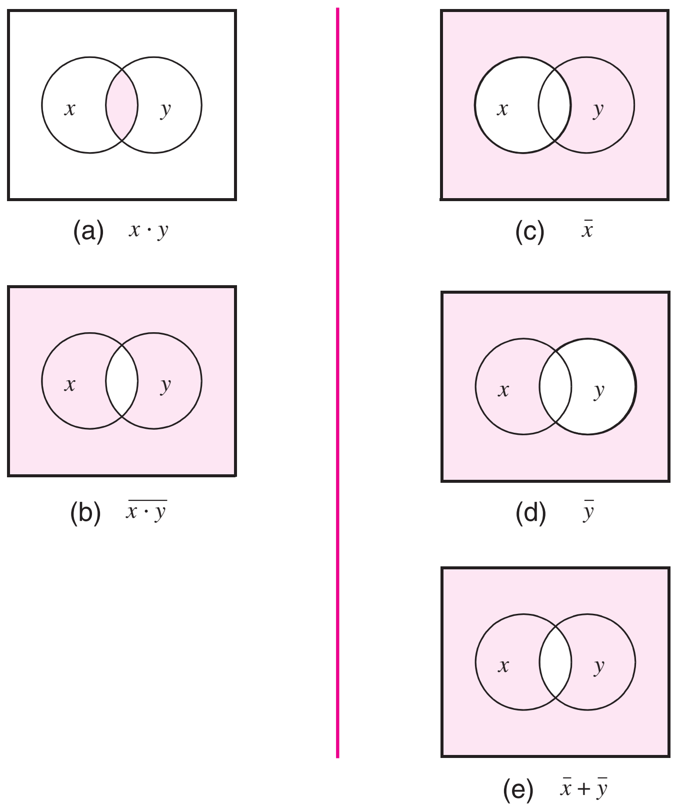


Figure 2.18 Proof of DeMorgan's theorem 15*a*.

Example 2.6 A proof of DeMorgan's theorem 15*a* by using Venn diagrams is illustrated in Figure 2.18. The diagram in part (*b*) of the figure, which is the complement of $x \cdot y$, is the same as the diagram in part (*e*), which is the union of part (*c*) with part (*d*), thus proving the theorem. We leave it as an exercise for the reader to prove the dual form of DeMorgan's theorem, 15*b*.

2.5.2 Notation and Terminology

Boolean algebra is based on the AND and OR operations, for which we have adopted the symbols $\cdot$ and $+$, respectively. These are also the standard symbols for the familiar arithmetic multiplication and addition operations. Considerable similarity exists between the Boolean operations and the arithmetic operations, which is the main reason why the same symbols are used. In fact, when single digits are involved there is only one significant difference; the result of $1 + 1$ is equal to 2 in ordinary arithmetic, whereas it is equal to 1 in Boolean algebra as defined by theorem 7*b* in Section 2.5.

Because of the similarity with the arithmetic addition and multiplication operations, the OR and AND operations are often called the *logical sum* and *product* operations. Thus $x_1 + x_2$ is the logical sum of x_1 and x_2, and $x_1 \cdot x_2$ is the logical product of x_1 and x_2. Instead of saying "logical product" and "logical sum," it is customary to say simply "product" and "sum." Thus we say that the expression

$$x_1 \cdot \overline{x}_2 \cdot x_3 + \overline{x}_1 \cdot x_4 + x_2 \cdot x_3 \cdot \overline{x}_4$$

is a sum of three product terms, whereas the expression

$$(\overline{x}_1 + x_3) \cdot (x_1 + \overline{x}_3) \cdot (\overline{x}_2 + x_3 + x_4)$$

is a product of three sum terms.

2.5.3 Precedence of Operations

Using the three basic operations—AND, OR, and NOT—it is possible to construct an infinite number of logic expressions. Parentheses can be used to indicate the order in which the operations should be performed. However, to avoid an excessive use of parentheses, another convention defines the precedence of the basic operations. It states that in the absence of parentheses, operations in a logic expression must be performed in the order: NOT, AND, and then OR. Thus in the expression

$$x_1 \cdot x_2 + \overline{x}_1 \cdot \overline{x}_2$$

it is first necessary to generate the complements of x_1 and x_2. Then the product terms $x_1 \cdot x_2$ and $\overline{x}_1 \cdot \overline{x}_2$ are formed, followed by the sum of the two product terms. Observe that in the absence of this convention, we would have to use parentheses to achieve the same effect as follows:

$$(x_1 \cdot x_2) + ((\overline{x}_1) \cdot (\overline{x}_2))$$

Finally, to simplify the appearance of logic expressions, it is customary to omit the $\cdot$ operator when there is no ambiguity. Therefore, the preceding expression can be written as

$$x_1x_2 + \overline{x}_1\overline{x}_2$$

We will use this style throughout the book.

2.6 Synthesis Using AND, OR, and NOT Gates

Armed with some basic ideas, we can now try to implement arbitrary functions using the AND, OR, and NOT gates. Suppose that we wish to design a logic circuit with two inputs, x_1 and x_2. Assume that x_1 and x_2 represent the states of two switches, either of which may

x_1	x_2	$f(x_1, x_2)$
0	0	1
0	1	1
1	0	0
1	1	1

Figure 2.19 A function to be synthesized.

produce a 0 or 1. The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches (x_1, x_2) are in states (0, 0), (0, 1), or (1, 1). If the state of the switches is (1, 0), the output should be 0. We can express the required behavior using a truth table, as shown in Figure 2.19.

A possible procedure for designing a logic circuit that implements this truth table is to create a product term that has a value of 1 for each valuation for which the output function f has to be 1. Then we can take a logical sum of these product terms to realize f. Let us begin with the fourth row of the truth table, which corresponds to $x_1 = x_2 = 1$. The product term that is equal to 1 for this valuation is $x_1 \cdot x_2$, which is just the AND of x_1 and x_2. Next consider the first row of the table, for which $x_1 = x_2 = 0$. For this valuation the value 1 is produced by the product term $\overline{x}_1 \cdot \overline{x}_2$. Similarly, the second row leads to the term $\overline{x}_1 \cdot x_2$. Thus f may be realized as

$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

The logic network that corresponds to this expression is shown in Figure 2.20*a*.

Although this network implements f correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and properties from Section 2.5. According to theorem 7*b*, we can replicate any term in a logical sum expression. Replicating the third product term, the above expression becomes

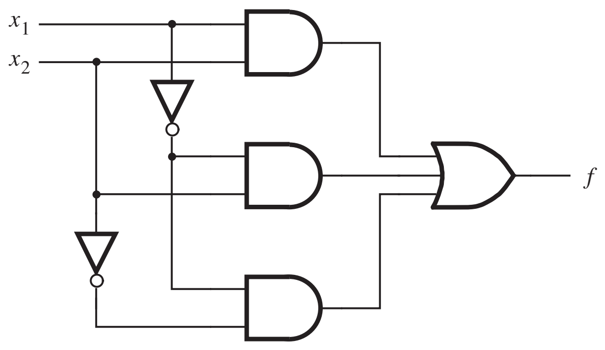
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + \overline{x}_1x_2$$

Using the commutative property 10*b* to interchange the second and third product terms gives

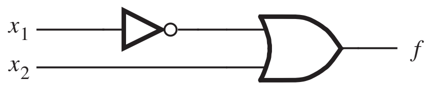
$$f(x_1, x_2) = x_1x_2 + \overline{x}_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_1x_2$$

Now the distributive property 12*a* allows us to write

$$f(x_1, x_2) = (x_1 + \overline{x}_1)x_2 + \overline{x}_1(\overline{x}_2 + x_2)$$



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.20 Two implementations of the function in Figure 2.19.

Applying theorem 8*b* we get

$$f(x_1, x_2) = 1 \cdot x_2 + \overline{x}_1 \cdot 1$$

Finally, theorem 6*a* leads to

$$f(x_1, x_2) = x_2 + \overline{x}_1$$

The network described by this expression is given in Figure 2.20*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

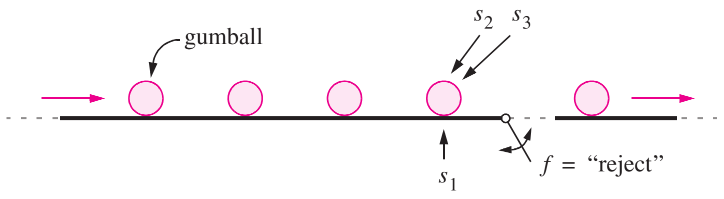
This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$ in that row, then $\overline{x}_i$ is entered. The sum of these product terms realizes the desired function. Second, there are many different networks that can realize a given function. Some of these networks may be simpler than others. Algebraic manipulation can be used to derive simplified logic expressions and thus lower-cost networks.

The process whereby we begin with a description of the desired functional behavior and then generate a circuit that realizes this behavior is called *synthesis*. Thus we can say that we "synthesized" the networks in Figure 2.20 from the truth table in Figure 2.19. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

Example 2.7 Figure 2.21*a* depicts a part of a factory that makes bubble gumballs. The gumballs travel on a conveyor that has three associated sensors s_1, s_2, and s_3. The sensor s_1 is connected to a scale that weighs each gumball, and if a gumball is not heavy enough to be acceptable then the sensor sets $s_1 = 1$. Sensors s_2 and s_3 examine the diameter of each gumball. If a gumball is too small to be acceptable, then $s_2 = 1$, and if it is too large, then $s_3 = 1$. If a gumball is of an acceptable weight and size, then the sensors give $s_1 = s_2 = s_3 = 0$. The conveyor pushes the gumballs over a "trap door" that it used to reject the ones that are not properly formed. A gumball should be rejected if it is too large, or both too small and too light. The trap door is opened by setting the logic function f to the value 1. By inspection, we can see that an appropriate logic expression is $f = s_1s_2 + s_3$. We will use Boolean algebra to derive this logic expression from the truth table.

The truth table for f is given in Figure 2.21*b*. It sets f to 1 for each row in the table where s_3 has the value 1 (too large), as well as for each row where $s_1 = s_2 = 1$ (too light and too small). As described previously, a logic expression for f can be formed by including a product term for each row where $f = 1$. Thus, we can write

$$f = \bar{s}_1\bar{s}_2s_3 + \bar{s}_1s_2s_3 + s_1\bar{s}_2s_3 + s_1s_2\bar{s}_3 + s_1s_2s_3$$



(a) Conveyor and sensors

s_1	s_2	s_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure 2.21 A bubble gumball factory.

We can use algebraic manipulation to simplify this expression in a number of ways. For example, as shown below, we can first use rule 7*b* to repeat the term $s_1 s_2 s_3$, and then use the distributive property 12*a* and rule 8*b* to simplify the expression

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 s_3(\bar{s}_2 + s_2) + s_1 s_3(\bar{s}_2 + s_2) + s_1 s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1 s_3 + s_1 s_3 + s_1 s_2 \end{aligned}$$

Now, using the combining property 14*a* on the first two product terms gives

$$f = s_3 + s_1 s_2$$

The observant reader will notice that using the combining property 14*a* is really just a short form of first using the distributive property 12*a* and then applying rule 8*b*, as we did in the previous step. Our simplified expression for f is the same as the one that we determined earlier, by inspection.

Example 2.8

There are different ways in which we can simplify the logic expression produced from the truth table in Figure 2.21*b*. Another approach is to first repeat the term $s_1 s_2 s_3$, as we did in Example 2.7, and then proceed as follows

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= s_3(\bar{s}_1\bar{s}_2 + \bar{s}_1 s_2 + s_1\bar{s}_2 + s_1 s_2) + s_1 s_2(\bar{s}_3 + s_3) \\ &= s_3 \cdot 1 + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

Here, we used the distributive property 12*a* to produce the expression $(\bar{s}_1\bar{s}_2 + \bar{s}_1 s_2 + s_1\bar{s}_2 + s_1 s_2)$. Since this expression includes all possible valuations of s_1, s_2, it is equal to 1, leading to the same expression for f that we derived before.

Example 2.9

Yet another way of producing the symplified logic expression is shown below.

$$\begin{aligned} f &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1\bar{s}_2 s_3 + \bar{s}_1 s_2 s_3 + s_1\bar{s}_2 s_3 + \bar{s}_1\bar{s}_2 s_3 + s_1 s_2 \bar{s}_3 + s_1 s_2 s_3 \\ &= \bar{s}_1 s_3(\bar{s}_2 + s_2) + \bar{s}_2 s_3(s_1 + \bar{s}_1) + s_1 s_2(\bar{s}_3 + s_3) \\ &= \bar{s}_1 s_3 + \bar{s}_2 s_3 + s_1 s_2 \\ &= s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2 \\ &= s_3(\overline{s_1 s_2}) + s_1 s_2 \\ &= s_3 + s_1 s_2 \end{aligned}$$

In this solution, we first repeat the term $\bar{s}_1\bar{s}_2 s_3$, and then symplify to generate the expression $s_3(\bar{s}_1 + \bar{s}_2) + s_1 s_2$. Using DeMorgan's theorem 15*a* we can replace $(\bar{s}_1 + \bar{s}_2)$ with $(\overline{s_1 s_2})$, which can then be deleted by applying property 16*a*.

As illustrated by Examples 2.7 to 2.9, there are multiple ways in which a logic expression can be minimized by using Boolean algebra. This process can be daunting, because it is not obvious which rules, identities, and properties should be applied, and in what order. Later in this chapter, in Section 2.11, we will introduce a graphical technique, called the Karnaugh map, that clarifies this process by providing a systematic way of generating a minimal-cost logic expression for a function.

2.6.1 Sum-of-Products and Product-of-Sums Forms

Having introduced the synthesis process by means of simple examples, we will now present it in more formal terms using the terminology that is encountered in the technical literature. We will also show how the principle of duality, which was introduced in Section 2.5, applies broadly in the synthesis process.

If a function f is specified in the form of a truth table, then an expression that realizes f can be obtained by considering either the rows in the table for which $f = 1$, as we have already done, or by considering the rows for which $f = 0$, as we will explain shortly.

Minterms

For a function of n variables, a product term in which each of the n variables appears once is called a *minterm*. The variables may appear in a minterm either in uncomplemented or complemented form. For a given row of the truth table, the minterm is formed by including x_i if $x_i = 1$ and by including $\overline{x}_i$ if $x_i = 0$.

To illustrate this concept, consider the truth table in Figure 2.22. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. From the discussion of the binary number representation in Section 1.5, we can observe that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2, and x_3. The figure shows all minterms for the three-variable table. For example, in the first row the variables

Row number	x_1	x_2	x_3	Minterm	Maxterm
0	0	0	0	$m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$	$M_0 = x_1 + x_2 + x_3$
1	0	0	1	$m_1 = \overline{x}_1\overline{x}_2x_3$	$M_1 = x_1 + x_2 + \overline{x}_3$
2	0	1	0	$m_2 = \overline{x}_1x_2\overline{x}_3$	$M_2 = x_1 + \overline{x}_2 + x_3$
3	0	1	1	$m_3 = \overline{x}_1x_2x_3$	$M_3 = x_1 + \overline{x}_2 + \overline{x}_3$
4	1	0	0	$m_4 = x_1\overline{x}_2\overline{x}_3$	$M_4 = \overline{x}_1 + x_2 + x_3$
5	1	0	1	$m_5 = x_1\overline{x}_2x_3$	$M_5 = \overline{x}_1 + x_2 + \overline{x}_3$
6	1	1	0	$m_6 = x_1x_2\overline{x}_3$	$M_6 = \overline{x}_1 + \overline{x}_2 + x_3$
7	1	1	1	$m_7 = x_1x_2x_3$	$M_7 = \overline{x}_1 + \overline{x}_2 + \overline{x}_3$

Figure 2.22 Three-variable minterms and maxterms.

have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\overline{x}_1\overline{x}_2\overline{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\overline{x}_1\overline{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation m_i to denote the minterm for row number i. Thus $m_0 = \overline{x}_1\overline{x}_2\overline{x}_3$, $m_1 = \overline{x}_1\overline{x}_2x_3$, and so on.

Sum-of-Products Form

A function f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of f for the corresponding valuation of input variables. For example, the two-variable minterms are $m_0 = \overline{x}_1\overline{x}_2$, $m_1 = \overline{x}_1x_2$, $m_2 = x_1\overline{x}_2$, and $m_3 = x_1x_2$. The function in Figure 2.19 can be represented as

$$\begin{aligned} f &= m_0 \cdot 1 + m_1 \cdot 1 + m_2 \cdot 0 + m_3 \cdot 1 \\ &= m_0 + m_1 + m_3 \\ &= \overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_1x_2 \end{aligned}$$

which is the form that we derived in the previous section using an intuitive approach. Only the minterms that correspond to the rows for which $f = 1$ appear in the resulting expression.

Any function f can be represented by a sum of minterms that correspond to the rows in the truth table for which $f = 1$. The resulting implementation is functionally correct and unique, but it is not necessarily the lowest-cost implementation of f. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be in the *sum-of-products* (*SOP*) form. If each product term is a minterm, then the expression is called a for the function f. As we have seen in the example of Figure 2.20, the first step in the synthesis process is to derive a canonical sum-of-products expression for the given function. Then we can manipulate this expression, using the theorems and properties of Section 2.5, with the goal of finding a functionally equivalent sum-of-products expression that has a lower cost.

As another example, consider the three-variable function $f(x_1, x_2, x_3)$, specified by the truth table in Figure 2.23. To synthesize this function, we have to include the minterms m_1,

Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.23 A three-variable function.

m_4, m_5, and m_6. Copying these minterms from Figure 2.22 leads to the following canonical sum-of-products expression for f

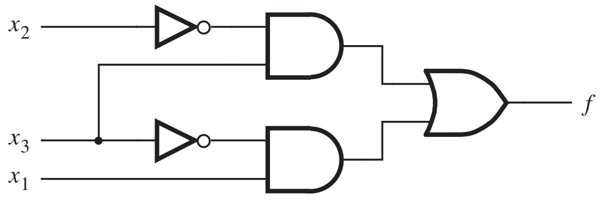
$$f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

This expression can be manipulated as follows

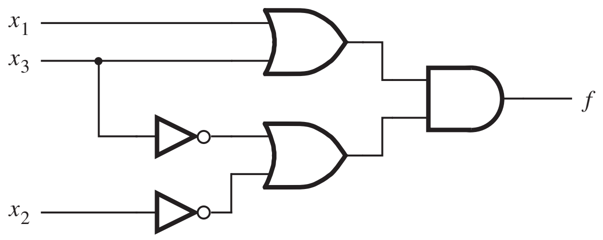
$$\begin{aligned} f &= (\bar{x}_1 + x_1)\bar{x}_2x_3 + x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= 1 \cdot \bar{x}_2x_3 + x_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_2x_3 + x_1\bar{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in Figure 2.24*a*. A good indication of the *cost* of a logic circuit is the total number of gates plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 2.24*a* is 13, because there are five gates and eight inputs to the gates. By comparison, the network implemented on the basis of the canonical sum-of-products would have a cost of 27; from the preceding expression, the OR gate has four inputs, each of the four AND gates has three inputs, and each of the three NOT gates has one input.

Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form. For example, the function in Figure 2.23 can be specified



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.24 Two realizations of the function in Figure 2.23.

as

$$f(x_1, x_2, x_3) = \sum(m_1, m_4, m_5, m_6)$$

or even more simply as

$$f(x_1, x_2, x_3) = \sum m(1, 4, 5, 6)$$

The $\sum$ sign denotes the logical sum operation. This shorthand notation is often used in practice.

Example 2.10

Consider the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

The canonical SOP expression for the function is derived using minterms

$$\begin{aligned} f &= m_2 + m_3 + m_4 + m_6 + m_7 \\ &= \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3 \end{aligned}$$

This expression can be simplified using the identities in Section 2.5 as follows

$$\begin{aligned} f &= \overline{x}_1 x_2 (\overline{x}_3 + x_3) + x_1 (\overline{x}_2 + x_2) \overline{x}_3 + x_1 x_2 (\overline{x}_3 + x_3) \\ &= \overline{x}_1 x_2 + x_1 \overline{x}_3 + x_1 x_2 \\ &= (\overline{x}_1 + x_1) x_2 + x_1 \overline{x}_3 \\ &= x_2 + x_1 \overline{x}_3 \end{aligned}$$

Example 2.11

Suppose that a four-variable function is defined by

$$f(x_1, x_2, x_3, x_4) = \sum m(3, 7, 9, 12, 13, 14, 15)$$

The canonical SOP expression for this function is

$$f = \overline{x}_1 \overline{x}_2 x_3 x_4 + \overline{x}_1 x_2 x_3 x_4 + x_1 \overline{x}_2 \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 \overline{x}_4 + x_1 x_2 \overline{x}_3 x_4 + x_1 x_2 x_3 \overline{x}_4 + x_1 x_2 x_3 x_4$$

A simpler SOP expression can be obtained as follows

$$\begin{aligned} f &= \overline{x}_1 (\overline{x}_2 + x_2) x_3 x_4 + x_1 (\overline{x}_2 + x_2) \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 (\overline{x}_4 + x_4) + x_1 x_2 x_3 (\overline{x}_4 + x_4) \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 \overline{x}_3 + x_1 x_2 x_3 \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 (\overline{x}_3 + x_3) \\ &= \overline{x}_1 x_3 x_4 + x_1 \overline{x}_3 x_4 + x_1 x_2 \end{aligned}$$

Maxterms

The principle of duality suggests that if it is possible to synthesize a function f by considering the rows in the truth table for which $f = 1$, then it should also be possible to synthesize f by considering the rows for which $f = 0$. This alternative approach uses the complements of minterms, which are called *maxterms*. All possible maxterms for three-variable functions are listed in Figure 2.22. We will refer to a maxterm M_j by the same row number as its corresponding minterm m_j as shown in the figure.

Product-of-Sums Form

If a given function f is specified by a truth table, then its complement $\overline{f}$ can be represented by a sum of minterms for which $\overline{f} = 1$, which are the rows where $f = 0$. For example, for the function in Figure 2.19

$$\begin{aligned}\overline{f}(x_1, x_2) &= m_2\\ &= x_1\overline{x}_2\end{aligned}$$

If we complement this expression using DeMorgan's theorem, the result is

$$\begin{aligned}\overline{\overline{f}} = f &= \overline{x_1\overline{x}_2}\\ &= \overline{x}_1 + x_2\end{aligned}$$

Note that we obtained this expression previously by algebraic manipulation of the canonical sum-of-products form for the function f. The key point here is that

$$f = \overline{m}_2 = M_2$$

where M_2 is the maxterm for row 2 in the truth table.

As another example, consider again the function in Figure 2.23. The complement of this function can be represented as

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= m_0 + m_2 + m_3 + m_7\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1x_2x_3\end{aligned}$$

Then f can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7}\\ &= \overline{m}_0 \cdot \overline{m}_2 \cdot \overline{m}_3 \cdot \overline{m}_7\\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_7\\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)\end{aligned}$$

This expression represents f as a product of maxterms.

A logic expression consisting of sum (OR) terms that are the factors of a logical product (AND) is said to be of the *product-of-sums* (*POS*) form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function f can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which $f = 0$ and forming a product of these maxterms.

Returning to the preceding example, we can attempt to reduce the complexity of the derived expression that comprises a product of maxterms. Using the commutative property 10*b* and the associative property 11*b* from Section 2.5, this expression can be written as

$$f = ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(x_1 + (\overline{x}_2 + \overline{x}_3))(\overline{x}_1 + (\overline{x}_2 + \overline{x}_3))$$

Then, using the combining property 14*b*, the expression reduces to

$$f = (x_1 + x_3)(\overline{x}_2 + \overline{x}_3)$$

The corresponding network is given in Figure 2.24*b*. The cost of this network is 13. While this cost happens to be the same as the cost of the sum-of-products version in Figure 2.24*a*, the reader should not assume that the cost of a network derived in the sum-of-products form will in general be equal to the cost of a corresponding circuit derived in the product-of-sums form.

Using the shorthand notation, an alternative way of specifying our sample function is

$$f(x_1, x_2, x_3) = \Pi(M_0, M_2, M_3, M_7)$$

or more simply

$$f(x_1, x_2, x_3) = \Pi M(0, 2, 3, 7)$$

The Π sign denotes the logical product operation.

The preceding discussion has shown how logic functions can be realized in the form of logic circuits, consisting of networks of gates that implement basic functions. A given function may be realized with various different circuit structures, which usually implies a difference in cost. An important objective for a designer is to minimize the cost of the designed circuit. We will discuss strategies for finding minimum-cost implementations in Section 2.11.

Example 2.12

Consider again the function in Example 2.10. Instead of using the minterms, we can specify this function as a product of maxterms for which $f = 0$, namely

$$f(x_1, x_2, x_3) = \Pi M(0, 1, 5)$$

Then, the canonical POS expression is derived as

$$\begin{aligned} f &= M_0 \cdot M_1 \cdot M_5 \\ &= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3) \end{aligned}$$

A simplified POS expression can be derived as

$$\begin{aligned} f &= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3) \\ &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \overline{x}_3)(x_1 + (x_2 + \overline{x}_3))(\overline{x}_1 + (x_2 + \overline{x}_3)) \\ &= ((x_1 + x_2) + x_3\overline{x}_3)(x_1\overline{x}_1 + (x_2 + \overline{x}_3)) \\ &= (x_1 + x_2)(x_2 + \overline{x}_3) \end{aligned}$$

Another way of deriving this product-of-sums expression is to use the sum-of-products form of $\overline{f}$. Thus,

$$\begin{aligned}\overline{f}(x_1, x_2, x_3) &= \sum m(0, 1, 5)\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3\\ &= \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2x_3 + \overline{x}_1\overline{x}_2x_3 + x_1\overline{x}_2x_3\\ &= \overline{x}_1\overline{x}_2(\overline{x}_3 + x_3) + \overline{x}_2x_3(\overline{x}_1 + x_1)\\ &= \overline{x}_1\overline{x}_2 + \overline{x}_2x_3\end{aligned}$$

Now, first applying DeMorgan's theorem 15*b*, and then applying 15*a* (twice) gives

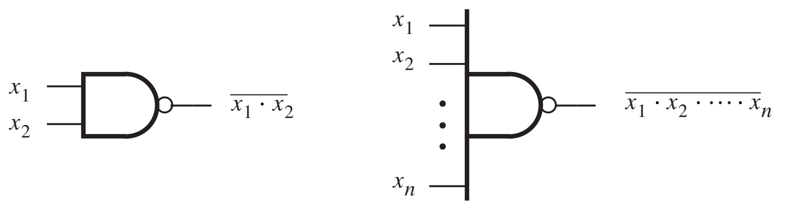
$$\begin{aligned}f &= \overline{\overline{f}}\\ &= (\overline{\overline{x}_1\overline{x}_2 + \overline{x}_2x_3})\\ &= (\overline{\overline{x}_1\overline{x}_2})(\overline{\overline{x}_2x_3})\\ &= (x_1 + x_2)(x_2 + \overline{x}_3)\end{aligned}$$

To see that this product-of-sums expression for f is equivalent to the sum-of-products expression that we derived in Example 2.10, we can slightly rearrange our expression as $f = (x_2 + x_1)(x_2 + \overline{x}_3)$. Now, recognizing that this expression has the form of the righthand side of the distributive property 12*b*, we have the sum-of-products expression $f = x_2 + x_1\overline{x}_3$.

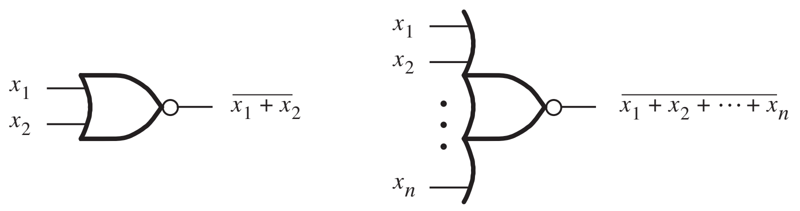
2.7 NAND AND NOR LOGIC NETWORKS

We have discussed the use of AND, OR, and NOT gates in the synthesis of logic circuits. There are other basic logic functions that are also used for this purpose. Particularly useful are the NAND and NOR functions which are obtained by complementing the output generated by AND and OR operations, respectively. These functions are attractive because they are implemented with simpler electronic circuits than the AND and OR functions, as we discuss in Appendix B. Figure 2.25 gives the graphical symbols for the NAND and NOR gates. A bubble is placed on the output side of the AND and OR gate symbols to represent the complemented output signal.

If NAND and NOR gates are realized with simpler circuits than AND and OR gates, then we should ask whether these gates can be used directly in the synthesis of logic circuits. In Section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 2.26. Identity 15*a* is interpreted in part (*a*) of the figure. It specifies that a NAND of variables x_1 and x_2 is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as bubbles, which denote inversion of the logic value at that point. The other half of DeMorgan's theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.

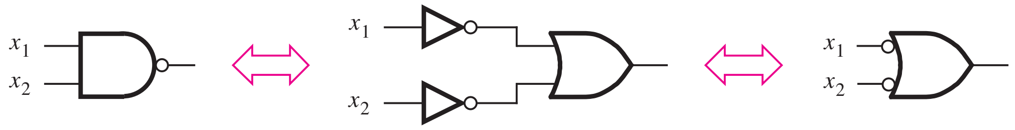

x_1
x_2
$\overline{x_1 \cdot x_2}$
x_1
x_2
x_n
$\overline{x_1 \cdot x_2 \cdot \cdots \cdot x_n}$

(a) NAND gates

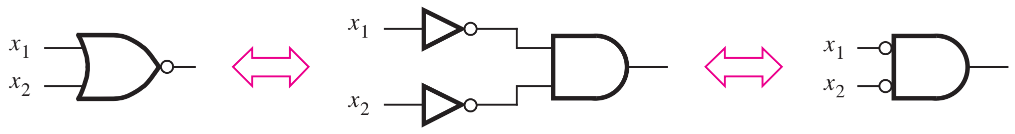


(b) NOR gates

Figure 2.25 NAND and NOR gates.



(a) $\overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$



(b) $\overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$

Figure 2.26 DeMorgan's theorem in terms of logic gates.

In Section 2.6 we explained how any logic function can be implemented either in sum-of-products or product-of-sums form, which leads to logic networks that have either an AND-OR or an OR-AND structure, respectively. We will now show that such networks can be implemented using only NAND gates or only NOR gates.

Consider the network in Figure 2.27 as a representative of general AND-OR networks. This network can be transformed into a network of NAND gates as shown in the figure. First, each connection between the AND gate and an OR gate is replaced by a connection that includes two inversions of the signal: one inversion at the output of the AND gate and the other at the input of the OR gate. Such double inversion has no effect on the behavior of the network, as stated formally in theorem 9 in Section 2.5. According to Figure 2.26*a*, the OR gate with inversions at its inputs is equivalent to a NAND gate. Thus we can redraw the network using only NAND gates, as shown in Figure 2.27. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.

Figure 2.28 gives a similar construction for a product-of-sums network, which can be transformed into a circuit with only NOR gates. The procedure is exactly the same as the one described for Figure 2.27 except that now the identity in Figure 2.26*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

Example 2.13 Let us implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

using NOR gates only. In Example 2.12 we showed that the function can be represented by the POS expression

$$f = (x_1 + x_2)(x_2 + \overline{x}_3)$$

An OR-AND circuit that corresponds to this expression is shown in Figure 2.29*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 2.29*b*. Note that x_3 is inverted by a NOR gate that has its inputs tied together.

Example 2.14 Let us now implement the function

$$f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$$

using NAND gates only. In Example 2.10 we derived the SOP expression

$$f = x_2 + x_1\overline{x}_3$$

which is realized using the circuit in Figure 2.30*a*. We can again use the same structure to obtain a circuit with NAND gates, but with one difference. The signal x_2 passes only through an OR gate, instead of passing through an AND gate and an OR gate. If we simply

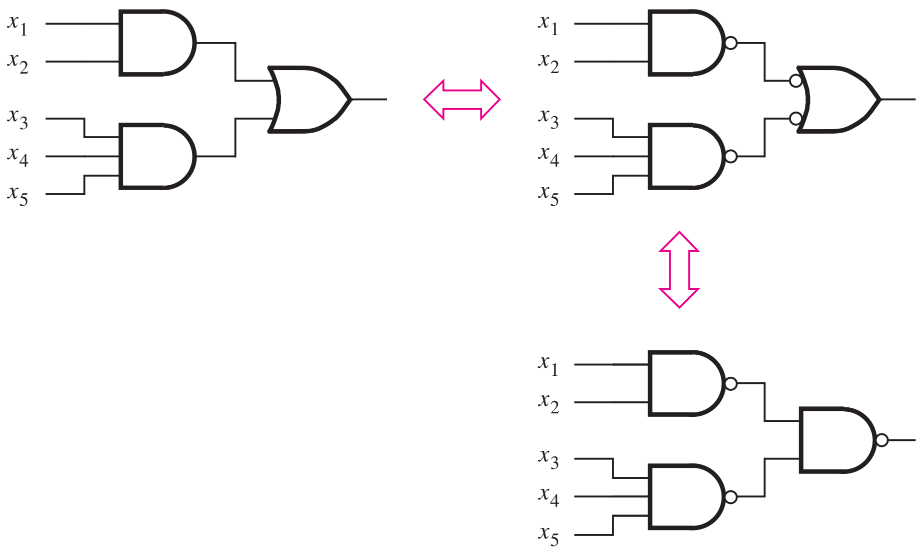


Figure 2.27 Using NAND gates to implement a sum-of-products.

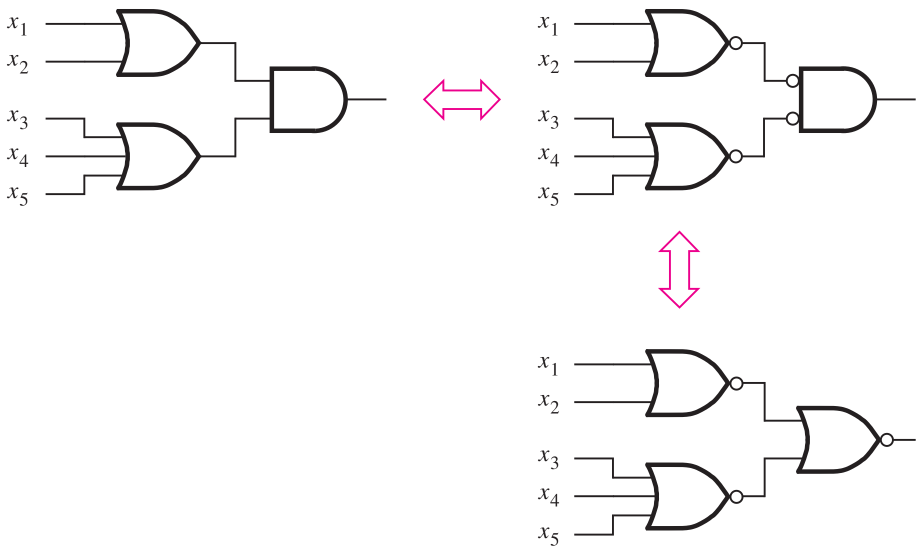
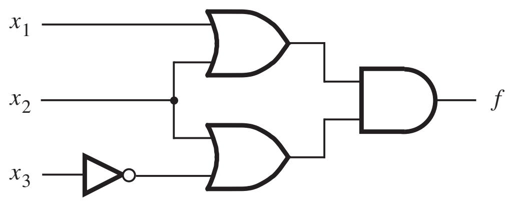
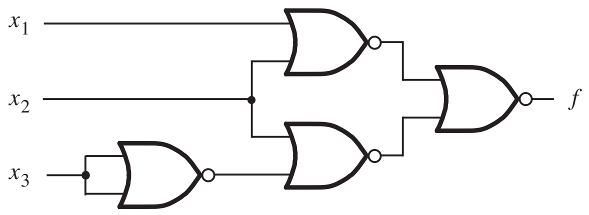


Figure 2.28 Using NOR gates to implement a product-of-sums.

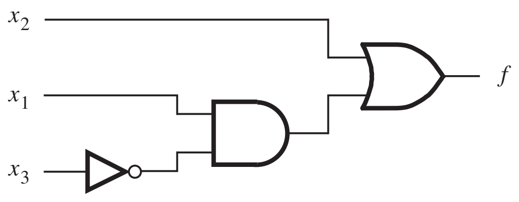


(a) POS implementation

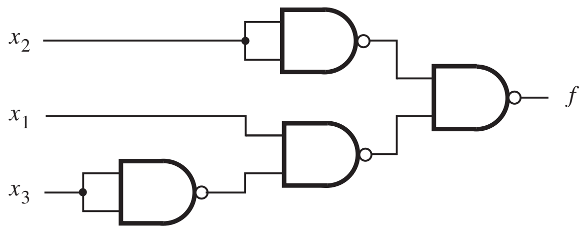


(b) NOR implementation

Figure 2.29 NOR-gate realization of the function in Example 2.13.



(a) SOP implementation



(b) NAND implementation

Figure 2.30 NAND-gate realization of the function in Example 2.10.

replace the OR gate with a NAND gate, this signal would be inverted which would result in a wrong output value. Since x_2 must either not be inverted, or it can be inverted twice, we can pass it through two NAND gates as depicted in Figure 2.30*b*. Observe that for this circuit the output f is

$$f = \overline{\overline{x}_2 \cdot \overline{x_1 \overline{x}_3}}$$

Applying DeMorgan's theorem, this expression becomes

$$f = x_2 + x_1 \overline{x}_3$$

2.8 Design Examples

Logic circuits provide a solution to a problem. They implement functions that are needed to carry out specific tasks. Within the framework of a computer, logic circuits provide complete capability for execution of programs and processing of data. Such circuits are complex and difficult to design. But regardless of the complexity of a given circuit, a designer of logic circuits is always confronted with the same basic issues. First, it is necessary to specify the desired behavior of the circuit. Second, the circuit has to be synthesized and implemented. Finally, the implemented circuit has to be tested to verify that it meets the specifications. The desired behavior is often initially described in words, which then must be turned into a formal specification. In this section we give three simple examples of design.

2.8.1 Three-Way Light Control

Assume that a large room has three doors and that a switch near each door controls a light in the room. It has to be possible to turn the light on or off by changing the state of any one of the switches.

As a first step, let us turn this word statement into a formal specification using a truth table. Let x_1, x_2, and x_3 be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If $f(x_1, x_2, x_3)$ represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 2.31. The canonical sum-of-products expression for the specified function is

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \overline{x}_1 \overline{x}_2 x_3 + \overline{x}_1 x_2 \overline{x}_3 + x_1 \overline{x}_2 \overline{x}_3 + x_1 x_2 x_3 \end{aligned}$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.32*a*.

x_1	x_2	x_3	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

Figure 2.31 Truth table for the three-way light control.

An alternative realization for this function is in the product-of-sums form. The canonical expression of this type is

$$\begin{aligned} f &= M_0 \cdot M_3 \cdot M_5 \cdot M_6 \\ &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3) \end{aligned}$$

The resulting circuit is depicted in Figure 2.32*b*. It has the same cost as the circuit in part (*a*) of the figure.

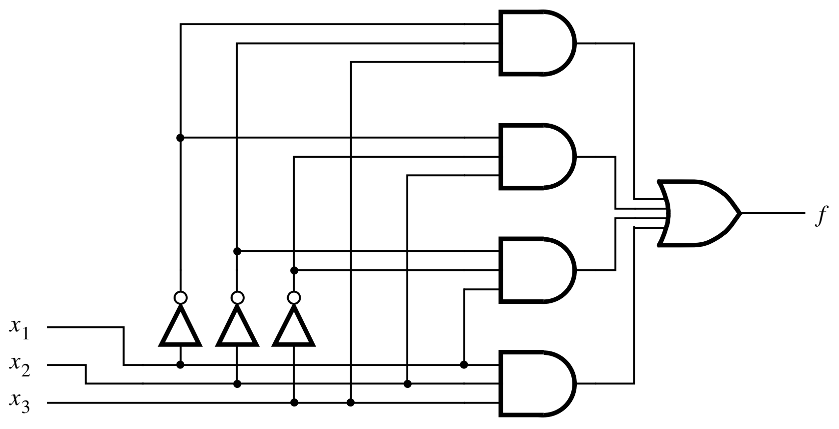
When the designed circuit is implemented, it can be tested by applying the various input valuations to the circuit and checking whether the output corresponds to the values specified in the truth table. A straightforward approach is to check that the correct output is produced for all eight possible input valuations.

2.8.2 MULTIPLEXER CIRCUIT

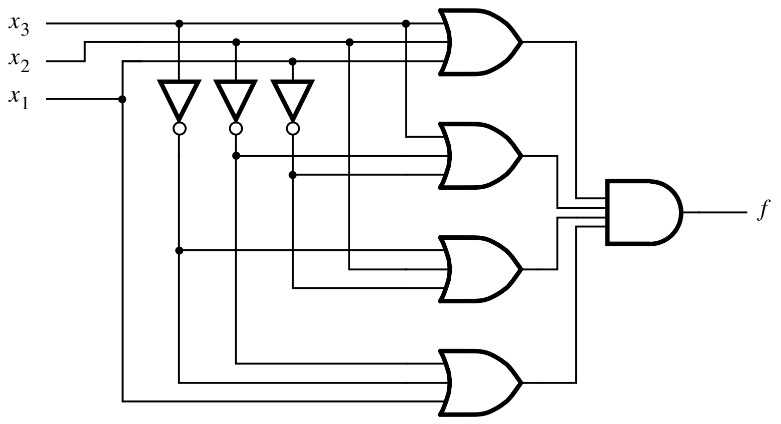
In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals x_1 and x_2. The values of these signals change in time, perhaps at regular intervals. Thus sequences of 0s and 1s are applied on each of the inputs x_1 and x_2. We want to design a circuit that produces an output that has the same value as either x_1 or x_2, dependent on the value of a selection control signal s. Therefore, the circuit should have three inputs: x_1, x_2, and s. Assume that the output of the circuit will be the same as the value of input x_1 if $s = 0$, and it will be the same as x_2 if $s = 1$.

Based on these requirements, we can specify the desired circuit in the form of a truth table given in Figure 2.33*a*. From the truth table, we derive the canonical sum of products

$$f(s, x_1, x_2) = \overline{s}x_1\overline{x}_2 + \overline{s}x_1x_2 + s\overline{x}_1x_2 + sx_1x_2$$



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.32 Implementation of the function in Figure 2.31.

Using the distributive property, this expression can be written as

$$f = \bar{s}x_1(\bar{x}_2 + x_2) + s(\bar{x}_1 + x_1)x_2$$

Applying theorem 8*b* yields

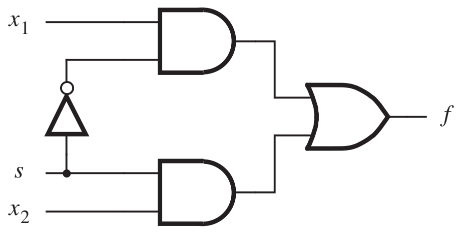
$$f = \bar{s}x_1 \cdot 1 + s \cdot 1 \cdot x_2$$

Finally, theorem 6*a* gives

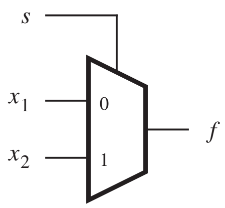
$$f = \bar{s}x_1 + sx_2$$

$s\ x_1\ x_2$	$f(s, x_1, x_2)$
0 0 0	0
0 0 1	0
0 1 0	1
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

(a) Truth table



(b) Circuit



(c) Graphical symbol

s	$f(s, x_1, x_2)$
0	x_1
1	x_2

(d) More compact truth-table representation

Figure 2.33 Implementation of a multiplexer.

A circuit that implements this function is shown in Figure 2.33*b*. Circuits of this type are used so extensively that they are given a special name. A circuit that generates an output that exactly reflects the state of one of a number of data inputs, based on the value of one or more selection control inputs, is called a *multiplexer*. We say that a multiplexer circuit "multiplexes" input signals onto a single output.

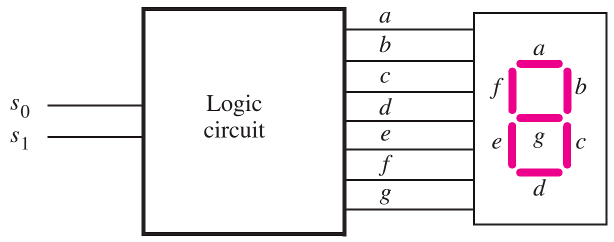
In this example we derived a multiplexer with two data inputs, which is referred to as a “2-to-1 multiplexer.” A commonly used graphical symbol for the 2-to-1 multiplexer is shown in Figure 2.33*c*. The same idea can be extended to larger circuits. A 4-to-1 multiplexer has four data inputs and one output. In this case two selection control inputs are needed to choose one of the four data inputs that is transmitted as the output signal. An 8-to-1 multiplexer needs eight data inputs and three selection control inputs, and so on.

Note that the statement “$f = x_1$ if $s = 0$, and $f = x_2$ if $s = 1$” can be presented in a more compact form of a truth table, as indicated in Figure 2.33*d*. In later chapters we will have occasion to use such representation.

We showed how a multiplexer can be built using AND, OR, and NOT gates. The same circuit structure can be used to implement the multiplexer using NAND gates, as explained in Section 2.7. In Appendix B we will show other possibilities for constructing multiplexers. In Chapter 4 we will discuss the use of multiplexers in considerable detail.

2.8.3 Number Display

In Example 2.2 we designed an adder circuit that generates the arithmetic sum $S = a + b$, where a and b are one-bit numbers and $S = s_1 s_0$ provides the resulting two-bit sum, which is either 00, 01, or 10. In this design example we wish to create a logic circuit that drives a familiar seven-segment display, as illustrated in Figure 2.34*a*. This display allows us to show the value of S as a decimal number, either 0, 1, or 2. The display includes seven



(a) Logic circuit and 7-segment display

	s_1	s_0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1

(b) Truth table

Figure 2.34 Display of numbers.

segments, labeled $a, b, \ldots, g$ in the figure, where each segment is a light-emitting diode (LED). Our logic circuit has the two inputs s_1 and s_0. It produces seven outputs, one for each segment in the display. Setting an output to the value 1 illuminates the corresponding segment in the display. By illuminating specific segments for each valuation of $s_1 s_0$ we can make the display's appearance correspond to the shape of the appropriate decimal digit.

Part (b) of Figure 2.34 shows a truth table for the possible valuations of $s_1 s_0$ and indicates on the lefthand side how the display should appear in each case. The truth table specifies the logic values needed for each of the seven functions. For example, segment a in the 7-segment display needs to be turned on when S has the decimal values 0 or 2, but has to be off when S has the value 1. Hence, the corresponding logic function is set to 1 for minterms m_0 and m_2, giving $a = \overline{s}_1\overline{s}_0 + s_1\overline{s}_0 = \overline{s}_0$. Logic expressions for each of the seven functions are:

$$
\begin{aligned}
a &= d = e = \overline{s}_0 \\
b &= 1 \\
c &= \overline{s}_1 \\
f &= \overline{s}_1\overline{s}_0 \\
g &= s_1\overline{s}_0
\end{aligned}
$$

Designers of logic circuits rely heavily on CAD tools. We want to encourage the reader to become familiar with CAD tools as soon as possible. We have reached a point where an introduction to these tools is useful. The next section presents some basic concepts that are needed to use these tools. We will also introduce, in Section 2.10, a special language for describing logic circuits, called Verilog. This language is used to describe the circuits as an input to the CAD tools, which then proceed to derive a suitable implementation.

2.9 Introduction to CAD Tools

The preceding sections introduced a basic approach for synthesis of logic circuits. A designer could use this approach manually for small circuits. However, logic circuits found in complex systems, such as today's computers, cannot be designed manually—they are designed using sophisticated CAD tools that automatically implement the synthesis techniques.

To design a logic circuit, a number of CAD tools are needed. They are usually packaged together into a *CAD system*, which typically includes tools for the following tasks: design entry, logic synthesis and optimization, simulation, and physical design. We will introduce some of these tools in this section and will provide additional discussion in later chapters.

2.9.1 Design Entry

The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure. This step is done manually by the designer because it requires design experience and intuition. The rest of the design process is done with the aid of CAD tools. The first stage of this process involves entering into the CAD system a description of the circuit being designed. This stage is called *design entry*. We will describe two design entry methods: using schematic capture and writing source code in a hardware description language.

Schematic Capture

A logic circuit can be defined by drawing logic gates and interconnecting them with wires. A CAD tool for entering a designed circuit in this way is called a *schematic capture* tool. The word *schematic* refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines.

A schematic capture tool uses the graphics capabilities of a computer and a computer mouse to allow the user to draw a schematic diagram. To facilitate inclusion of gates in the schematic, the tool provides a collection of graphical symbols that represent gates of various types with different numbers of inputs. This collection of symbols is called a *library*. The gates in the library can be imported into the user's schematic, and the tool provides a graphical way of interconnecting the gates to create a logic network.

Any subcircuits that have been previously created can be represented as graphical symbols and included in the schematic. In practice it is common for a CAD system user to create a circuit that includes within it other smaller circuits. This methodology is known as *hierarchical design* and provides a good way of dealing with the complexities of large circuits.

The schematic-capture method is simple to use, but becomes awkward when large circuits are involved. A better method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

Hardware Description Languages

A *hardware description language (HDL)* is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer. Many commercial HDLs are available. Some are proprietary, meaning that they are provided by a particular company and can be used to implement circuits only in the technology offered by that company. We will not discuss the proprietary HDLs in this book. Instead, we will focus on a language that is supported by virtually all vendors that provide digital hardware technology and is officially endorsed as an *Institute of Electrical and Electronics Engineers (IEEE)* standard. The IEEE is a worldwide organization that promotes technical activities to the benefit of society in general. One of its activities involves the development of standards that define how certain technological concepts can be used in a way that is suitable for a large body of users.

Two HDLs are IEEE standards: *Verilog HDL* and *VHDL (Very High Speed Integrated Circuit Hardware Description Language)*. Both languages are in widespread use in the industry. We use Verilog in this book, but a VHDL version of the book is also available from the same publisher [4]. Although the two languages differ in many ways, the choice of using one or the other when studying logic circuits is not particularly important, because both offer similar features. Concepts illustrated in this book using Verilog can be directly applied when using VHDL.

In comparison to performing schematic capture, using Verilog offers a number of advantages. Because it is supported by most organizations that offer digital hardware technology, Verilog provides design *portability*. A circuit specified in Verilog can be implemented in different types of chips and with CAD tools provided by different companies, without having to change the Verilog specification. Design portability is an important advantage because digital circuit technology changes rapidly. By using a standard language, the designer can focus on the functionality of the desired circuit without being overly concerned about the details of the technology that will eventually be used for implementation.

Design entry of a logic circuit is done by writing Verilog code. Signals in the circuit can be represented as variables in the source code, and logic functions are expressed by assigning values to these variables. Verilog source code is plain text, which makes it easy for the designer to include within the code documentation that explains how the circuit works. This feature, coupled with the fact that Verilog is widely used, encourages sharing and reuse of Verilog-described circuits. This allows faster development of new products in cases where existing Verilog code can be adapted for use in the design of new circuits.

Similar to the way in which large circuits are handled in schematic capture, Verilog code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in Verilog code.

Verilog design entry can be combined with other methods. For example, a schematic-capture tool can be used in which a subcircuit in the schematic is described using Verilog. We will introduce Verilog in Section 2.10.

2.9.2 Logic Synthesis

Synthesis is the process of generating a logic circuit from an initial specification that may be given in the form of a schematic diagram or code written in a hardware description language. Synthesis CAD tools generate efficient implementations of circuits from such specifications.

The process of translating, or *compiling*, Verilog code into a network of logic gates is part of synthesis. The output is a set of logic expressions that describe the logic functions needed to realize the circuit.

Regardless of what type of design entry is used, the initial logic expressions produced by the synthesis tools are not likely to be in an optimal form because they reflect the designer's input to the CAD tools. It is impossible for a designer to manually produce optimal results for large circuits. So, one of the important tasks of the synthesis tools is to manipulate the user's design to automatically generate an equivalent, but better circuit.

The measure of what makes one circuit better than another depends on the particular needs of a design project and the technology chosen for implementation. Earlier in this chapter we suggested that a good circuit might be one that has the lowest cost. There are other possible optimization goals, which are motivated by the type of hardware technology used for implementation of the circuit. We discuss implementation technologies in Appendix B.

The performance of a synthesized circuit can be assessed by physically constructing the circuit and testing it. But, its behavior can also be evaluated by means of simulation.

2.9.3 Functional Simulation

A circuit represented in the form of logic expressions can be simulated to verify that it will function as expected. The tool that performs this task is called a *functional simulator*. It uses the logic expressions (often referred to as equations) generated during synthesis, and assumes that these expressions will be implemented with perfect gates through which signals propagate instantaneously. The simulator requires the user to specify valuations of the circuit's inputs that should be applied during simulation. For each valuation, the simulator evaluates the outputs produced by the expressions. The results of simulation are usually provided in the form of a timing diagram which the user can examine to verify that the circuit operates as required.

2.9.4 Physical Design

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit on a given chip. This step is often called *physical design*. As we discuss in Appendix B, there are several different technologies that may be used to implement logic circuits. The physical design tools map a circuit specified in the form of logic expressions into a realization that makes use of the resources available on the target chip. They determine the placement of specific logic elements, which are not necessarily simple gates of the type we have encountered so far. They also determine the wiring connections that have to be made between these elements to implement the desired circuit.

2.9.5 Timing Simulation

Logic gates and other logic elements are implemented with electronic circuits, and these circuits cannot perform their function with zero delay. When the values of inputs to the circuit change, it takes a certain amount of time before a corresponding change occurs at the output. This is called a *propagation delay* of the circuit. The propagation delay consists of two kinds of delays. Each logic element needs some time to generate a valid output signal whenever there are changes in the values of its inputs. In addition to this delay, there is a delay caused by signals that must propagate along wires that connect various logic elements. The combined effect is that real circuits exhibit delays, which has a significant impact on their speed of operation.

A *timing simulator* evaluates the expected delays of a designed logic circuit. Its results can be used to determine if the generated circuit meets the timing requirements of the specification for the design. If the requirements are not met, the designer can ask the physical design tools to try again by indicating specific timing constraints that have to be met. If this does not succeed, then the designer has to try different optimizations in the synthesis step, or else improve the initial design that is presented to the synthesis tools.

2.9.6 Circuit Implementation

Having ascertained that the designed circuit meets all requirements of the specification, the circuit is implemented on an actual chip. If a custom-manufactured chip is created for this design, then this step is called *chip fabrication*. But if a programmable hardware device is used, then this step is called chip *configuration* or *programming*. Various types of chip technologies are described in Appendix B.

2.9.7 Complete Design Flow

The CAD tools discussed above are the essential parts of a CAD system. The complete design flow that we discussed is illustrated in Figure 2.35. This has been just a brief introductory discussion. A full presentation of the CAD tools is given in Chapter 10.

At this point the reader should have some appreciation for what is involved when using CAD tools. However, the tools can be fully appreciated only when they are used firsthand. We strongly encourage the reader to obtain access to suitable CAD tools and implement some examples of circuits by using these tools. Two examples of commonly-used CAD tools are the Quartus II tools available from Altera Corporation and the ISE tools provided by Xilinx Corporation. Both of these CAD systems can be obtained free-of-charge for educational use from their respective corporations' websites.

2.10 Introduction to Verilog

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design Systems. In 1990 Verilog was put into the public domain, and it has since become one of the most popular languages for describing digital circuits. In 1995 Verilog was adopted as an official IEEE Standard, called 1364-1995. An enhanced version of Verilog, called Verilog 2001, was adopted as IEEE Standard 1364-2001 in 2001. While this version introduced a number of new features, it also supports all of the features in the original Verilog standard.

Verilog was originally intended for simulation and verification of digital circuits. Subsequently, with the addition of synthesis capability, Verilog has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the Verilog code into a hardware implementation of the described circuit. In this book our main use of Verilog will be for synthesis.

Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. To simplify the presentation, we will focus the discussion on the features of the Verilog language that are actually used in the examples in the book. The material presented is sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete Verilog language can refer to one of the specialized texts [5–11].

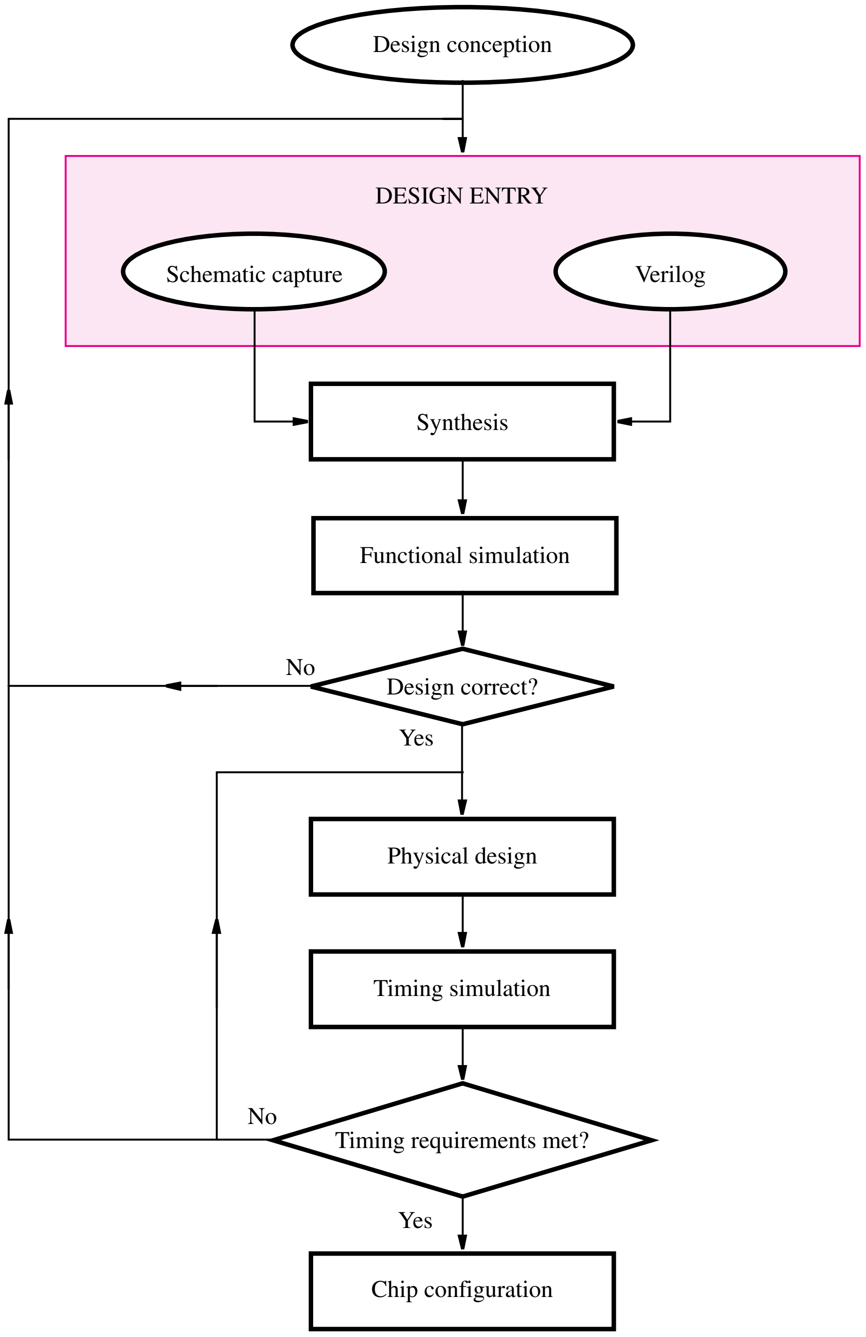


Figure 2.35 A typical CAD system.

Verilog is introduced in several stages throughout the book. Our general approach will be to introduce particular features only when they are relevant to the design topics covered in that part of the text. In Appendix A we provide a concise summary of the Verilog features covered in the book. The reader will find it convenient to refer to that material from time to time. In the remainder of this chapter we discuss the most basic concepts needed to write simple Verilog code.

Representation of Digital Circuits in Verilog

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in the previous section. One efficient way is to write this description in the form of Verilog source code. The Verilog compiler translates this code into a logic circuit.

Verilog allows the designer to describe a desired circuit in a number of ways. One possibility is to use Verilog constructs that describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and Verilog programming constructs that define the desired behavior of the circuit, but not its actual structure in terms of gates. This is called the *behavioral* representation.

2.10.1 STRUCTURAL SPECIFICATION OF LOGIC CIRCUITS

Verilog includes a set of *gate-level primitives* that correspond to commonly-used logic gates. A gate is represented by indicating its functional name, output, and inputs. For example, a two-input AND gate, with output y and inputs x_1 and x_2, is denoted as

and (y, x1, x2);

A four-input OR gate is specified as

or (y, x1, x2, x3, x4);

The keywords **nand** and **nor** are used to define the NAND and NOR gates in the same way. The NOT gate given by

not (y, x);

implements $y = \overline{x}$. The gate-level primitives can be used to specify larger circuits. All of the available Verilog gate-level primitives are listed in Table A.2 in Appendix A.

A logic circuit is specified in the form of a *module* that contains the statements that define the circuit. A module has inputs and outputs, which are referred to as its *ports*. The word port is a commonly-used term that refers to an input or output connection to an electronic circuit. Consider the multiplexer circuit from Figure 2.33*b*, which is reproduced in Figure 2.36. This circuit can be represented by the Verilog code in Figure 2.37. The first statement gives the module a name, *example1*, and indicates that there are four port signals. The next two statements declare that x_1, x_2, and s are to be treated as **input** signals,

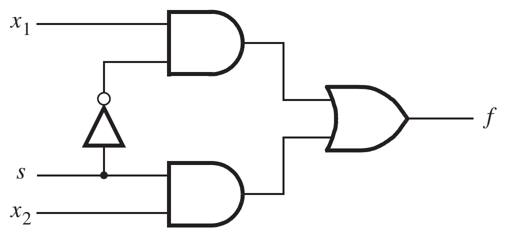


Figure 2.36 The logic circuit for a multiplexer.

```
module example1 (x1, x2, s, f);
   input x1, x2, s;
   output f;

   not (k, s);
   and (g, k, x1);
   and (h, s, x2);
   or (f, g, h);

endmodule
```

Figure 2.37 Verilog code for the circuit in Figure 2.36.

while f is the **output**. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives $k = \bar{s}$. The AND gates produce $g = \bar{s}x_1$ and $h = sx_2$. The outputs of AND gates are combined in the OR gate to form

$$\begin{aligned} f &= g + h \\ &= \bar{s}x_1 + sx_2 \end{aligned}$$

The module ends with the **endmodule** statement. We have written the Verilog keywords in bold type to make the text easier to read. We will continue this practice throughout the book.

A second example of Verilog code is given in Figure 2.38. It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h. It implements the logic functions

$$\begin{aligned} g &= x_1x_3 + x_2x_4 \\ h &= (x_1 + \bar{x}_3)(\bar{x}_2 + x_4) \\ f &= g + h \end{aligned}$$

```
module example2 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   and (z1, x1, x3);
   and (z2, x2, x4);
   or (g, z1, z2);
   or (z3, x1, ~x3);
   or (z4, ~x2, x4);
   and (h, z3, z4);
   or (f, g, h);

endmodule
```

Figure 2.38 Verilog code for a four-input circuit.

Instead of using explicit NOT gates to define $\overline{x}_2$ and $\overline{x}_3$, we have used the Verilog operator "~" (tilde character on the keyboard) to denote complementation. Thus, $\overline{x}_2$ is indicated as ~x2 in the code. The circuit produced by the Verilog compiler for this example is shown in Figure 2.39.

Verilog Syntax

The names of modules and signals in Verilog code follow two simple rules: the name must start with a letter, and it can contain any letter or number plus the "_" underscore and "$" characters. Verilog is case sensitive. Thus, the name *k* is not the same as *K* and *Example1* is not the same as *example1*. The Verilog syntax does not enforce a particular style of code. For example, multiple statements can appear on a single line. White space characters, such as SPACE and TAB, and blank lines are ignored. As a matter of good style, code should be formatted in such a way that it is easy to read. Indentation and blank lines can be used to make separate parts of the code easily recognizable, as we have done in Figures 2.37 and 2.38. Comments may be included in the code to improve its readability. A comment begins with the double slash "//" and continues to the end of the line.

2.10.2 Behavioral Specification of Logic Circuits

Using gate-level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a logic circuit. One possibility is to define the circuit using logic expressions. Figure 2.40 shows how the circuit in Figure 2.36 can be defined with the expression

$$f = \overline{s}x_1 + sx_2$$

The AND and OR operations are indicated by the "&" and "|" Verilog operators, respectively. The *assign* keyword provides a *continuous assignment* for the signal *f*. The word continuous

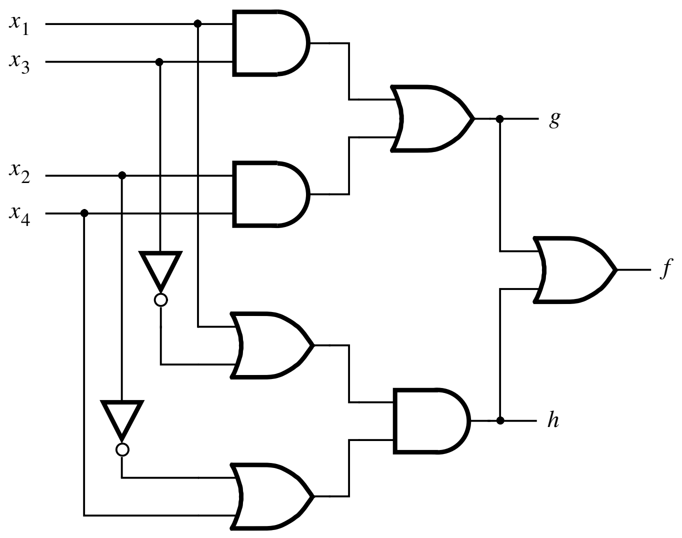


Figure 2.39 Logic circuit for the code in Figure 2.38.

```
module example3 (x1, x2, s, f);
  input x1, x2, s;
  output f;

  assign f = (~s & x1) | (s & x2);

endmodule
```

Figure 2.40 Using the continuous assignment to specify the circuit in Figure 2.36.

stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated. The effect is equivalent to using the gate-level primitives in Figure 2.37. Following this approach, the circuit in Figure 2.39 can be specified as shown in Figure 2.41.

Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage. Consider again the multiplexer circuit of Figure 2.36. The circuit can be described in words by saying that $f = x_1$ if $s = 0$ and $f = x_2$ if $s = 1$. In Verilog, this behavior can be defined with the **if-else** statement

```
if (s == 0)
  f = x1;
else
  f = x2;
```

```
module example4 (x1, x2, x3, x4, f, g, h);
   input x1, x2, x3, x4;
   output f, g, h;

   assign g = (x1 & x3) | (x2 & x4);
   assign h = (x1 | ~x3) & (~x2 | x4);
   assign f = g | h;

endmodule
```

Figure 2.41 Using the continuous assignment to specify the circuit in Figure 2.39.

```
// Behavioral specification
module example5 (x1, x2, s, f);
   input x1, x2, s;
   output f;
   reg f;

   always @(x1 or x2 or s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

Figure 2.42 Behavioral specification of the circuit in Figure 2.36.

The complete code is given in Figure 2.42. The first line illustrates how a comment can be inserted. The **if-else** statement is an example of a Verilog *procedural* statement. We will introduce other procedural statements, such as loop statements, in Chapters 3 and 4.

Verilog syntax requires that procedural statements be contained inside a construct called an **always** block, as shown in Figure 2.42. An **always** block can contain a single statement, as in this example, or it can contain multiple statements. A typical Verilog design module may include several **always** blocks, each representing a part of the circuit being modeled. An important property of the **always** block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.

The part of the **always** block after the @ symbol, in parentheses, is called the *sensitivity list*. This list has its roots in the use of Verilog for simulation. The statements inside an **always** block are executed by the simulator only when one or more of the signals in the sensitivity list changes value. In this way, the complexity of a simulation process is simplified, because it is not necessary to execute every statement in the code at all times. When Verilog is being employed for synthesis of circuits, as in this book, the sensitivity list simply tells the Verilog compiler which signals can directly affect the outputs produced by the **always** block.

If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a *variable*; this is accomplished by using the keyword **reg** in Figure 2.42. This term also derives from the simulation jargon: It means that, once a variable's value is assigned with a procedural statement, the simulator "registers" this value and it will not change until the **always** block is executed again. We will discuss this issue in detail in Chapter 3.

Instead of using a separate statement to declare that the variable f is of **reg** type in Figure 2.42, we can alternatively use the syntax

output reg f;

which combines these two statements. Also, Verilog 2001 adds the ability to declare a signal's direction and type directly in the module's list of ports. This style of code is illustrated in Figure 2.43. In the sensitivity list of the **always** statement we can use commas instead of the word **or**, which is also illustrated in Figure 2.43. Moreover, instead of listing the relevant signals in the sensitivity list, it is possible to write simply

always @(*)

or even more simply

always @*

assuming that the compiler will figure out which signals need to be considered.

Behavioral specification of a logic circuit defines only its behavior. CAD synthesis tools use this specification to construct the actual circuit. The detailed structure of the synthesized circuit will depend on the technology used.

```
// Behavioral specification
module example5 (input x1, x2, s, output reg f);

   always @(x1, x2, s)
      if (s == 0)
         f = x1;
      else
         f = x2;

endmodule
```

Figure 2.43 A more compact version of the code in Figure 2.42.

2.10.3 HIERARCHICAL VERILOG CODE

The examples of Verilog code given so far include just a single module. For larger designs, it is often convenient to create a hierarchical structure in the Verilog code, in which there is a *top-level* module that includes multiple instances of *lower-level* modules. To see how hierarchical Verilog code can be written consider the circuit in Figure 2.44. This circuit comprises two lower-level modules: the adder module that we described in Figure 2.12, and the module that drives a 7-segment display which we showed in Figure 2.34. The purpose of the circuit is to generate the arithmetic sum of the two inputs x and y, using the *adder* module, and then to show the resulting decimal value on the 7-segment display.

Verilog code for the *adder* module from Figure 2.12 and the *display* module from Figure 2.34 is given in Figures 2.45 and 2.46, respectively. For the *adder* module continuous assignment statements are used to specify the two-bit sum $s_1 s_0$. The assignment statement for s_0 uses the Verilog XOR operator, which is specified as $s_0 = a \wedge b$. The code for the *display* module includes continuous assignment statements that correspond to the

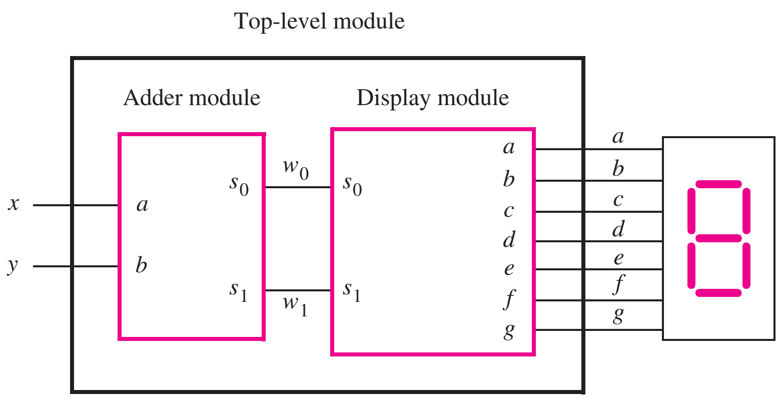


Figure 2.44 A logic circuit with two modules.

```
// An adder module
module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;

    assign s1 = a & b;
    assign s0 = a ^ b;

endmodule
```

Figure 2.45 Verilog specification of the circuit in Figure 2.12.

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
   input s1, s0;
   output a, b, c, d, e, f, g;

   assign a = ~s0;
   assign b = 1;
   assign c = ~s1;
   assign d = ~ s0;
   assign e = ~s0;
   assign f = ~s1 & ~s0;
   assign g = s1 & ~s0;

endmodule
```

Figure 2.46 Verilog specification of the circuit in Figure 2.34.

```
module adder_display (x, y, a, b, c, d, e, f, g);
   input x, y;
   output a, b, c, d, e, f, g;
   wire w1, w0;

   adder U1 (x, y, w1, w0);
   display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

Figure 2.47 Hierarchical Verilog code for the circuit in Figure 2.44.

logic expressions for each of the seven outputs of the display circuit, which are given in Section 2.8.3. The statement

assign b = 1;

assigns the output b of the display module to have the constant value 1. We discuss the specification of numbers in Verilog code in Chapter 3.

The top-level Verilog module, named *adder_display*, is given in Figure 2.47. This module has the inputs x and y, and the outputs $a, \dots, g$. The statement

wire w1, w0;

is needed because the signals w_1 and w_0 are neither inputs nor outputs of the circuit in Figure 2.44. Since these signals cannot be declared as input or output ports in the Verilog code, they have to be declared as (internal) *wires*. The statement

```
adder U1 (x, y, w1, w0);
```

instantiates the *adder* module from Figure 2.45 as a submodule. The submodule is given a name, U1, which can be any valid Verilog name. In this instantiation statement the signals attached to the ports of the *adder* submodule are listed in the same order as those in Figure 2.45. Thus, the input ports x and y of the top-level module in Figure 2.47 are connected to the first two ports of *adder*, which are named a and b. The order in which signals are listed in the instantiation statement determines which signal is connected to each port in the submodule. The instantiation statement also attaches the last two ports of the *adder* submodule, which are its outputs, to the wires w1 and w0 in the top-level module. The statement

```
display U2 (w1, w0, a, b, c, d, e, f, g);
```

instantiates the other submodule in our circuit. Here, the wires w1 and w0, which have already been connected to the outputs of the *adder* submodule, are attached to the corresponding input ports of the *display* submodule. The *display* submodule's output ports are attached to the $a, \ldots, g$ output ports of the top-level module.

2.10.4 How *Not* to Write Verilog Code

When learning how to use Verilog or other hardware description languages, the tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. This book contains more than 100 examples of complete Verilog code that represent a wide range of logic circuits. In these examples the code is easily related to the described logic circuit. The reader is advised to adopt the same style of code. A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the Verilog code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to model.

Once complete Verilog code is written for a particular design, the reader is encouraged to analyze the resulting circuit produced by the CAD synthesis tools; typical CAD systems provide graphical viewing tools that can display a logic circuit that corresponds to the output produced by the Verilog compiler. Much can be learned about Verilog, logic circuits, and logic synthesis through this process. We provide additional guidelines for writing Verilog code in Appendix A.

2.11 Minimization and Karnaugh Maps

In a number of our examples we have used algebraic manipulation to find a reduced-cost implementation of a function in either sum-of-products or product-of-sums form. In these examples, we made use of the rules, theorems, and properties of Boolean algebra that

Row number	x_1	x_2	x_3	f
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.48 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

were introduced in Section 2.5. For example, we often used the distributive property, DeMorgan's theorem, and the combining property. In general, it is not obvious when to apply these theorems and properties to find a minimum-cost circuit, and it is often tedious and impractical to do so. This section introduces a more manageable approach, call the *Karnaugh map*, which provides a systematic way of producing a minimum-cost logic expression.

The key to the Karnaugh map approach is that it allows the application of the combining property 14*a*, or 14*b*, as judiciously as possible. To understand how it works consider the function f in Figure 2.48. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3$$

The combining property 14*a* allows us to replace two minterms that differ in the value of only one variable with a single product term that does not include that variable at all. For example, both m_0 and m_2 include $\overline{x}_1$ and $\overline{x}_3$, but they differ in the value of x_2 because m_0 includes $\overline{x}_2$ while m_2 includes x_2. Thus

$$\begin{aligned}\overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2\overline{x}_3 &= \overline{x}_1(\overline{x}_2 + x_2)\overline{x}_3\\ &= \overline{x}_1 \cdot 1 \cdot \overline{x}_3\\ &= \overline{x}_1\overline{x}_3\end{aligned}$$

Hence m_0 and m_2 can be replaced by the single product term $\overline{x}_1\overline{x}_3$. Similarly, m_4 and m_6 differ only in the value of x_2 and can be combined using

$$\begin{aligned}x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 &= x_1(\overline{x}_2 + x_2)\overline{x}_3\\ &= x_1 \cdot 1 \cdot \overline{x}_3\\ &= x_1\overline{x}_3\end{aligned}$$

Now the two newly-generated terms, $\overline{x}_1\overline{x}_3$ and $x_1\overline{x}_3$, can be combined further as

$$\begin{aligned}\overline{x}_1\overline{x}_3 + x_1\overline{x}_3 &= (\overline{x}_1 + x_1)\overline{x}_3\\ &= 1 \cdot \overline{x}_3\\ &= \overline{x}_3\end{aligned}$$

These optimization steps indicate that we can replace the four minterms m_0, m_2, m_4, and m_6 with the single product term $\overline{x}_3$. In other words, the minterms m_0, m_2, m_4, and m_6 are all *included* in the term $\overline{x}_3$. The remaining minterm in f is m_5. It can be combined with m_4, which gives

$$x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 = x_1\overline{x}_2$$

Recall that theorem 7b in Section 2.5 indicates that

$$m_4 = m_4 + m_4$$

which means that we can use the minterm m_4 twice—to combine with minterms m_0, m_2, and m_6 to yield the term $\overline{x}_3$ as explained above and also to combine with m_5 to yield the term $x_1\overline{x}_2$.

We have now accounted for all the minterms in f; hence all five input valuations for which $f = 1$ are covered by the minimum-cost expression

$$f = \overline{x}_3 + x_1\overline{x}_2$$

The expression has the product term $\overline{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of x_1 and x_2. The four minterms m_0, m_2, m_4, and m_6 represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables x_1 and x_2. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 2.48, but it is obvious if we write the corresponding valuations grouped together:

	x_1	x_2	x_3
m_0	0	0	0
m_2	0	1	0
m_4	1	0	0
m_6	1	1	0

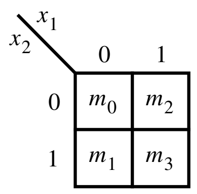
In a similar way, if we look at m_4 and m_5 as a group of two

	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

it is clear that when $x_1 = 1$ and $x_2 = 0$, then $f = 1$ regardless of the value of x_3.

x_1	x_2	
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

(a) Truth table



(b) Karnaugh map

Figure 2.49 Location of two-variable minterms.

The preceding discussion suggests that it would be advantageous to devise a method that allows easy discovery of groups of minterms for which $f = 1$ that can be combined into single terms. The Karnaugh map is a useful vehicle for this purpose.

The *Karnaugh map* [1] is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 2.49. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_1, and the rows are labeled by x_2. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from Section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms m_2 and m_3 can be combined as

$$\begin{aligned} m_2 + m_3 &= x_1\overline{x}_2 + x_1x_2 \\ &= x_1(\overline{x}_2 + x_2) \\ &= x_1 \cdot 1 \\ &= x_1 \end{aligned}$$

The Karnaugh map is not just useful for combining pairs of minterms. As we will see in several larger examples, the Karnaugh map can be used directly to derive a minimum-cost circuit for a logic function.

Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 2.50. It corresponds to the function f of Figure 2.19. The value of f for each valuation of the variables x_1 and x_2 is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause f to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell.

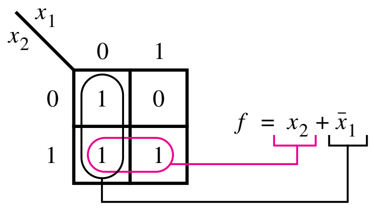


Figure 2.50 The function of Figure 2.19.

Thus if $x_2 = 1$, then $f = 1$ regardless of whether x_1 is equal to 0 or 1. The product term representing the two cells is simply x_2.

Similarly, $f = 1$ for both cells in the first column. These cells are identified by $x_1 = 0$. Therefore, they lead to the product term $\bar{x}_1$. Since this takes care of all instances where $f = 1$, it follows that the minimum-cost realization of the function is

$$f = x_2 + \bar{x}_1$$

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where $f = 1$. Moreover, the cost of these product terms should be as low as possible. Note that a product term that covers two adjacent cells is cheaper to implement than a term that covers only a single cell. For our example once the two cells in the bottom row have been covered by the product term x_2, only one cell (top left) remains. Although it could be covered by the term $\bar{x}_1\bar{x}_2$, it is better to combine the two cells in the left column to produce the product term $\bar{x}_1$ because this term is cheaper to implement.

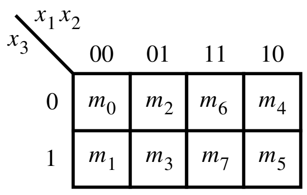
Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 2.51*a* lists all of the three-variable minterms, and part (*b*) of the figure indicates the locations of these minterms in the Karnaugh map. In this case each valuation of x_1 and x_2 identifies a column in the map, while the value of x_3 distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of (x_1, x_2) values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable x_1. Also, the first and the fourth columns differ only in variable x_1, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 2.51*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)

Figure 2.52*a* represents the function of Figure 2.23 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row,

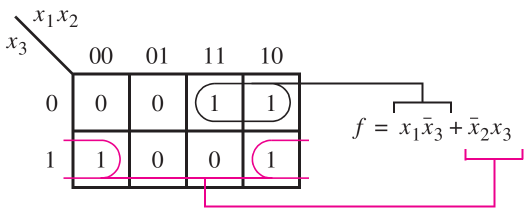
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

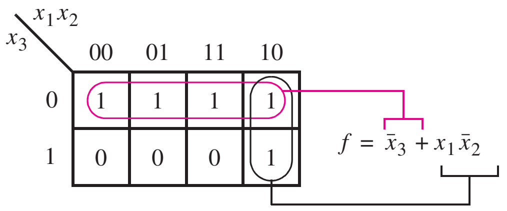


(b) Karnaugh map

Figure 2.51 Location of three-variable minterms.



(a) The function of Figure 2.23



(b) The function of Figure 2.48

Figure 2.52 Examples of three-variable Karnaugh maps.

which are represented by the term $x_1\bar{x}_3$. The second term is $\bar{x}_2 x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

$$f = x_1\bar{x}_3 + \bar{x}_2 x_3$$

which describes the circuit obtained in Figure 2.24*a*.

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization