Fourth Edition

Fundamentals of DIGITAL LOGIC with VHDL DESIGN

STEPHEN BROWN | ZVONKO VRANESIC

Mc Graw Hill

FUNDAMENTALS OF DIGITAL LOGIC WITH VHDL DESIGN

FOURTH EDITION

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

Mc Graw Hill

FUNDAMENTALS OF DIGITAL LOGIC WITH VHDL DESIGN, FOURTH EDITION

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 27 26 25 24 23 22

ISBN 978-0-07-338072-8 (bound edition)
MHID 0-07-338072-5 (bound edition)
ISBN 978-1-259-82897-3 (loose-leaf edition)
MHID 1-259-82897-2 (loose-leaf edition)

Portfolio Manager: *Beth Bettcher*
Product Developer: *Erin Kamm*
Marketing Manager: *Lisa Granger*
Content Project Managers: *Maria McGreal, Samantha Donisi*
Buyer: *Susan K. Culbertson*
Content Licensing Specialist: *Shawntel Schmitt*
Cover Image: *View Stock/Getty Images*
Compositor: *Aptara®, Inc.*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

Library of Congress Cataloging-in-Publication Data

Names: Brown, Stephen D., author. | Vranesic, Zvonko G., author.
Title: Fundamentals of digital logic with VHDL design / Stephen Brown and Zvonko Vranesic, Department of Electrical and Computer Engineering, University of Toronto.
Description: Fourth edition. | New York, NY : McGraw Hill, [2023] | Includes index.
Identifiers: LCCN 2021039775 (print) | LCCN 2021039776 (ebook) | ISBN 9780073380728 (hardcover ; acid-free paper) | ISBN 9781259828973 (spiral bound ; acid-free paper) | ISBN 9781266711381 (ebook) | ISBN 9780077496111 (ebook other)
Subjects: LCSH: Logic circuits–Design and construction–Data processing. | Logic design–Data processing. | VHDL (Computer hardware description language)
Classification: LCC TK7888.4 .B76 2023 (print) | LCC TK7888.4 (ebook) | DDC 621.39/5–dc23
LC record available at https://lccn.loc.gov/2021039775
LC ebook record available at https://lccn.loc.gov/2021039776

To Susan and Anne

About the Authors

Stephen Brown received the Ph.D. and M.A.Sc. degrees in Electrical Engineering from the University of Toronto, and his B.A.Sc. degree in Electrical Engineering from the University of New Brunswick. 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 FPGA Academic Programs for Intel Corporation.

His research interests include field-programmable VLSI technology, CAD algorithms, computer architecture, and applications of machine learning. He won the Canadian Natural Sciences and Engineering Research Council's 1992 Doctoral Prize for the best Ph.D. thesis in Canada, and the New Brunswick Governor-General's 1985 award for the highest academic standing in the Faculty of Engineering. He is a coauthor of more than 150 scientific research papers and two other textbooks: *Fundamentals of Digital Logic with Verilog Design* and *Field-Programmable Gate Arrays*.

He has won many awards for excellence in teaching electrical engineering, computer engineering, and computer science courses.

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 to 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 Department of Electrical & Computer Engineering. During the 1978–79 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984–85 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 a coauthor of four other books: *Computer Organization and Embedded Systems*, 6th ed.; *Fundamentals of Digital Logic with Verilog 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 the classical methods of logic design and a firm grasp of the modern design approach that relies on computer-aided design (CAD) tools.

The main goals of this book are (1) to teach students the fundamental concepts of classical manual digital design and (2) illustrate clearly the way in which digital circuits are designed today, using CAD tools. Even though current modern designers very rarely use manual techniques, our motivation for teaching such techniques is to give students an intuitive feeling for how digital circuits operate. The manual techniques also 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 simple design examples, which we perform using both manual techniques and modern CAD-tool-based methods. After basic concepts are established, more complex examples are provided using the CAD tools. Thus, to illustrate how digital design is presently carried out in practice, our emphasis is on modern design methodology.

Technology

This book discusses modern digital circuit implementation technologies. The emphasis is placed 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 the opportunity, in a laboratory setting, to implement this 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 (HDL) in specifying the logic circuits, because an HDL-based approach is the most efficient design method to use in practice. We describe in detail the IEEE Standard VHDL 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 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 VHDL, 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 VHDL can be used to specify a desired functionality and how CAD tools can provide a mechanism for developing the required circuits.

Chapter 4 presents combinational circuits such as multiplexers, encoders, and decoders that are used as building blocks in larger circuits. These circuits are very convenient for illustrating the application of many VHDL constructs, giving the reader the opportunity to discover more advanced features of VHDL.

Chapter 5 introduces storage elements. The use of flip-flops to realize regular structures, such as shift registers and counters, is discussed. VHDL-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 techniques for both manual and automated design.

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

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

Chapter 9 discusses asynchronous sequential circuits. While this treatment is not exhaustive, a clear indication of the main characteristics of such circuits is provided. Even though asynchronous circuits are not used extensively in practice, they provide an excellent vehicle for gaining a more complete understanding of the operation of digital circuits. Asynchronous circuits illustrate the importance of clearly understanding how the structure of a circuit affects the propagation delays of signals, and how these delays can impact a circuit's functionality.

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

Chapter 11 introduces the topic of testing. A designer of logic circuits must 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 VHDL features. Although use of VHDL is integrated throughout the book, this appendix provides a convenient reference that the reader may occasionally consult while writing VHDL code.

Appendix B presents the electronic aspects of digital circuits. This appendix shows how the basic gates are built using transistors and presents various factors that affect circuit performance. The most modern technologies are emphasized, with particular focus on CMOS technology and PLDs.

What Can Be Covered in a Course

Much of the material in the book can be covered in 2 one-quarter courses. Thorough 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 discussing the intricacies of VHDL and CAD tools. To make this approach possible, we organized the VHDL material in a modular style that is conducive to self-study. Our experience in teaching students at the University of Toronto has shown that the instructor may only spend a total of three to four lecture hours on VHDL, describing how the code should be structured (including the use of design hierarchy) using scalar and vector variables, and the style of code needed to specify sequential circuits. The VHDL examples given in this book are largely self-explanatory, and students can understand them with ease.

The book is also suitable for a course in logic design that does not include exposure to VHDL. However, some knowledge of VHDL, even at a rudimentary level, is beneficial to the students, and is 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.

VHDL

VHDL is a complex language that some instructors feel is too hard for beginning students to grasp. We fully appreciate this issue and have attempted to solve it by presenting only the most important VHDL 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 VHDL material is integrated gradually, with more advanced features presented only at points where their use can be demonstrated in the design of relevant circuits.

The book includes more than 150 examples of VHDL code. These examples illustrate how VHDL 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.

CAD Tools

All of the examples of VHDL code presented in the book are provided on the authors' website at

www.eecg.toronto.edu/~brown

To gain a more thorough understanding of the VHDL code examples, the reader is encouraged to compile and simulate the code using a commercially-available simulation CAD tool. Toward this end each example of VHDL code on the Authors' website is accompanied by setup files for the widely-used *ModelSim* simulator available from Mentor Graphics. A detailed tutorial is provided that includes step-by-step instructions for obtaining, installing, and using this VHDL simulator.

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 VHDL should enable the reader to develop VHDL code that specifies logic circuits of varying degrees of complexity. To gain a proper appreciation of the design process, in addition to simulating the behavior of VHDL code (as mentioned above) it is highly beneficial to implement the designs in commercially-available PLDs by using CAD tools.

Some excellent CAD tools are available free of charge, for example, Intel Corporation's Quartus CAD software, which is widely used for implementing designs in programmable logic devices such as FPGAs. The Lite Edition of the Quartus software can be downloaded from Intel's website and used free-of-charge. Intel also provides a set of FPGA-based laboratory boards with accompanying tutorials and laboratory exercises that are appropriate for use with this book in a university setting. This material can be found on the Internet by searching for "Intel FPGA Academic Program".

Solved Problems

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

Homework Problems

More than 400 homework problems are provided in this book. Answers to selected problems are given at the back. Full solutions to all problems are available to instructors in the accompanying *Solutions Manual*.

PowerPoint Slides and Solutions Manual

A set of PowerPoint slides that contain all of the figures in the book is 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

Acknowledgments

We wish to express our thanks to the people who have helped during the preparation of this book. Dan Vranesic produced a substantial amount of artwork. He and Deshanand Singh also helped with the preparation of the solutions manual, and Tom Czajkowski helped in checking the answers to problems. Matthew Brown edited the text to improve readability, and Andrew Brown helped to create some examples of VHDL code. 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.

Nicholas Reeder, Sinclair Community College, Dayton, OH, provided excellent proof reading.

Stephen Brown and Zvonko Vranesic

CONTENTS

Instructors: Student Success Starts with You

Tools to enhance your unique voice

Want to build your own course? No problem. Prefer to use an OLC-aligned, prebuilt course? Easy. Want to make changes throughout the semester? Sure. And you'll save time with Connect's auto-grading too.

65%
Less Time Grading

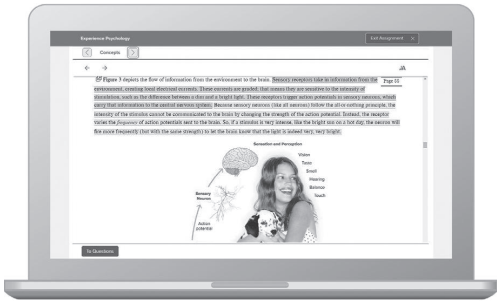


Figure 3 depicts the flow of information from the environment to the brain. Sensory receptors take in information from the environment, creating local electrical currents. These currents are graded; that means they are sensitive to the intensity of stimulation, such as the difference between a dim and a bright light. These receptors trigger action potentials in sensory neurons, which carry that information to the central nervous system. Because sensory neurons (like all neurons) follow the all-or-nothing principle, the intensity of the stimulus cannot be communicated to the brain by changing the strength of the action potential. Instead, the receptor varies the *frequency* of action potentials sent to the brain. So, if a stimulus is very intense, like the bright sun on a hot day, the neuron will fire more frequently (but with the same strength) to let the brain know that the light is indeed very, very bright.

Laptop: McGraw Hill; Woman/dog: George Doyle/Getty Images

Study made personal

Incorporate adaptive study resources like SmartBook® 2.0 into your course and help your students be better prepared in less time. Learn more about the powerful personalized learning experience available in SmartBook 2.0 at **www.mheducation.com/highered/connect/smartbook**

Affordable solutions, added value

Make technology work for you with LMS integration for single sign-on access, mobile access to the digital textbook, and reports to quickly show you how each of your students is doing. And with our Inclusive Access program you can provide all these tools at a discount to your students. Ask your McGraw Hill representative for more information.

Padlock: Jobalou/Getty Images

Solutions for your challenges

A product isn't a solution. Real solutions are affordable, reliable, and come with training and ongoing support when you need it and how you want it. Visit **www.supportateverystep.com** for videos and resources both you and your students can use throughout the semester.

Checkmark: Jobalou/Getty Images

SUPPORT AT every step

Students: Get Learning that Fits You

Effective tools for efficient studying

Connect is designed to help you be more productive with simple, flexible, intuitive tools that maximize your study time and meet your individual learning needs. Get learning that works for you with Connect.

Study anytime, anywhere

Download the free ReadAnywhere app and access your online eBook, SmartBook 2.0, or Adaptive Learning Assignments when it's convenient, even if you're offline. And since the app automatically syncs with your Connect account, all of your work is available every time you open it. Find out more at **www.mheducation.com/readanywhere**

"I really liked this app—it made it easy to study when you don't have your text-book in front of you."

- Jordan Cunningham,
  Eastern Washington University

Calendar: owattaphotos/Getty Images

Everything you need in one place

Your Connect course has everything you need—whether reading on your digital eBook or completing assignments for class, Connect makes it easy to get your work done.

Learning for everyone

McGraw Hill works directly with Accessibility Services Departments and faculty to meet the learning needs of all students. Please contact your Accessibility Services Office and ask them to email accessibility@mheducation.com, or visit **www.mheducation.com/about/accessibility** for more information.

Top: Jenner Images/Getty Images, Left: Hero Images/Getty Images, Right: Hero Images/Getty Images

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 *VHDL*. Design with VHDL is first introduced in Chapter 2, and usage of VHDL 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 *digital hardware*. The reason that the name *digital* 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.

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 is informally known as *Moore's law*. Thus in the early 1990s microprocessors could be

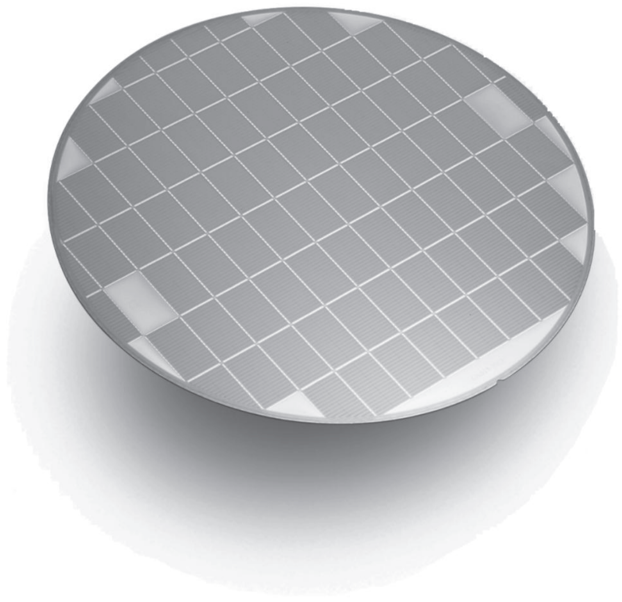


Figure 1.1 A silicon wafer.

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 Roadmap for Devices and Systems (IRDS)*, this forecast discusses many aspects of technology, including the maximum number of transistors that can be manufactured in a given chip area. An example of data that can be derived from the roadmap is given in Figure 1.2. It indicates how the maximum number of transistors that can be manufactured on a single chip has changed over time—chips with about 10 million transistors could be successfully manufactured in 1995, and this number has steadily increased, leading to modern chips that have over a billion transistors. The roadmap predicts that chips with more than 100 billion transistors will be possible by the year 2025. 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 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,

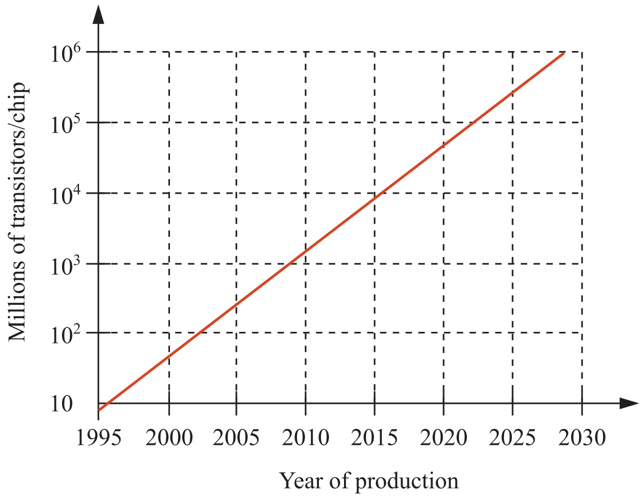


Figure 1.2 An estimate of the maximum number of transistors per chip over time.

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 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.

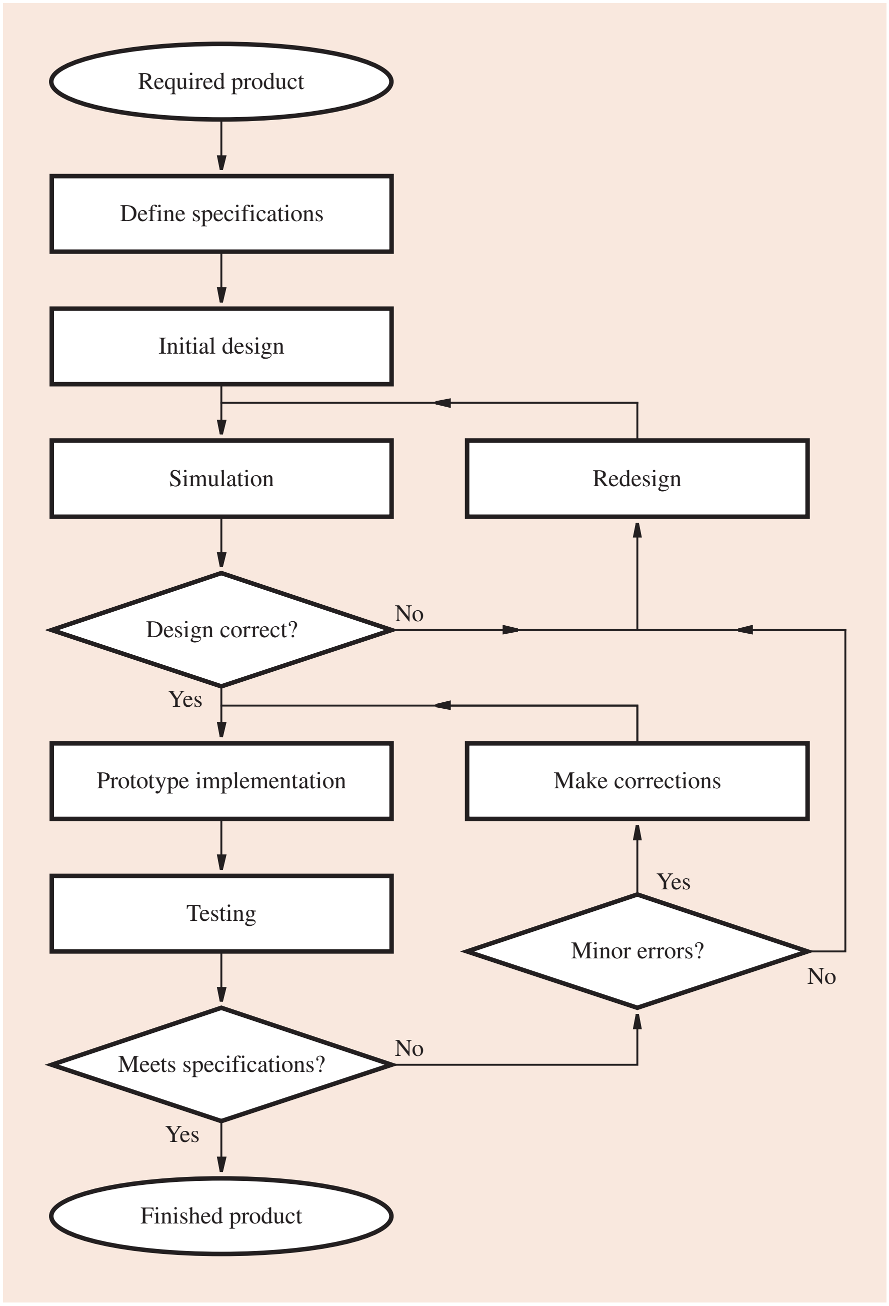


Figure 1.3 The development process.

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.

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.

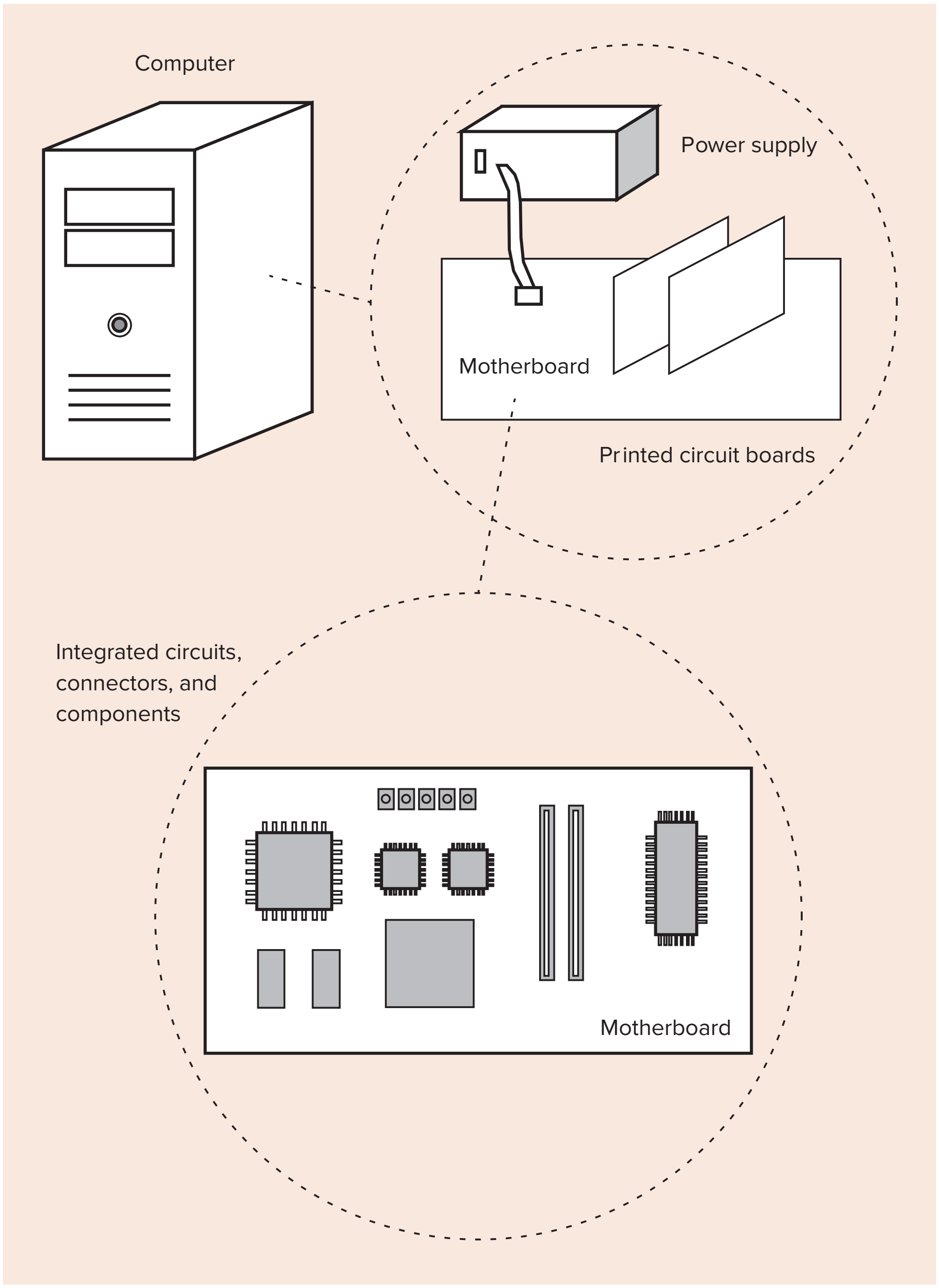


Figure 1.4 A digital hardware system (Part *a*).

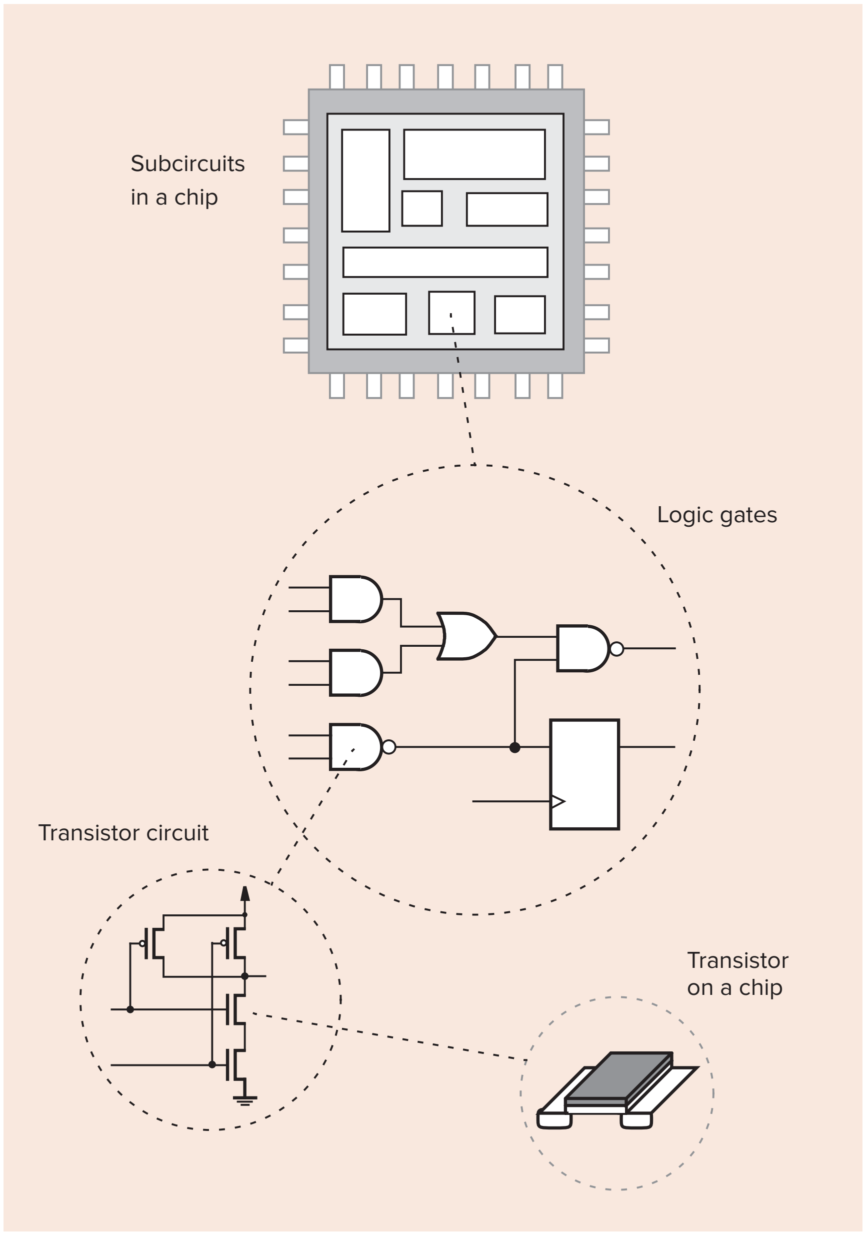


Figure 1.4 A digital hardware system (Part *b*).

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 VHDL 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 Intel, Cadence, Mentor Graphics, Synopsys, and Xilinx. The CAD systems offered by any of these companies can be used equally well with this book. Two examples of CAD systems that are particularly well-suited for use with this book are those provided by Intel and Xilinx. The CAD tools from both of these corporations 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 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. 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

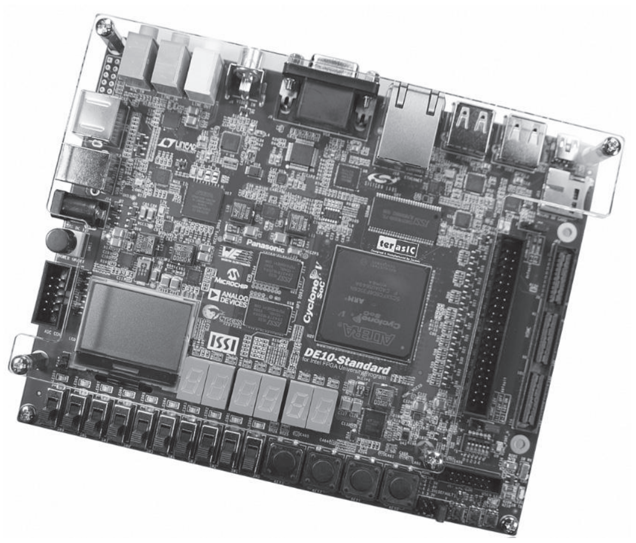


Figure 1.5 An FPGA board.

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

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

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 rightmost 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

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

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

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

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

Convert $(857)_{10}$

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

Result is $(1101011001)_2$

Figure 1.6 Conversion from decimal to binary.

$(857)_{10} = (1101011001)_2$. 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.

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 leftmost 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.

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 = $\boxed{6}\boxed{5}\boxed{4}\boxed{3}\boxed{2}\boxed{1}\boxed{0}$

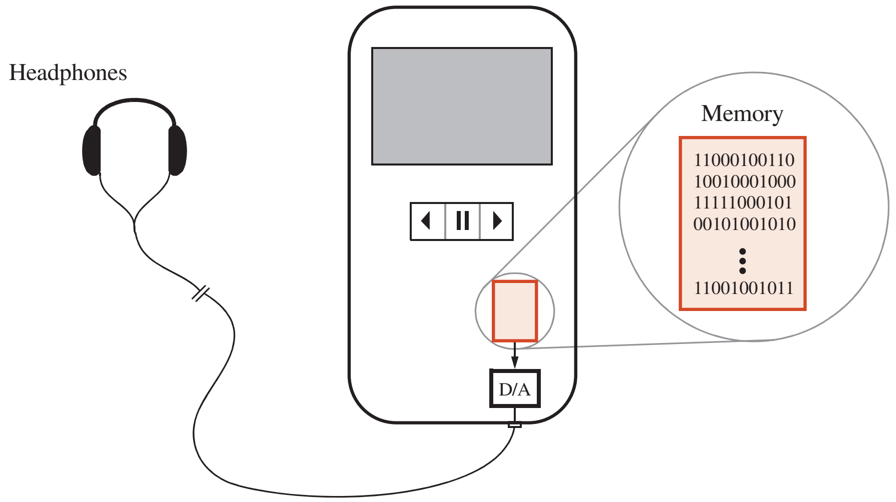


Figure 1.7 Using digital technology to represent music.

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!

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) $(100000)_{10}$
(d) $(1000000)_{10}$

***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 construct 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 1.3 using the method described in Problem 1.4.

***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 1.8 for the following decimal numbers:
(a) $(111)_{10}$
(b) $(333)_{10}$
(c) $(555)_{10}$
(d) $(1111)_{10}$

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 VHDL 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 *logic circuit*. 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 they are represented in mathematical notation, and how they are 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. The current flows when the switch is closed, that is, when $x = 1$. In this example the input

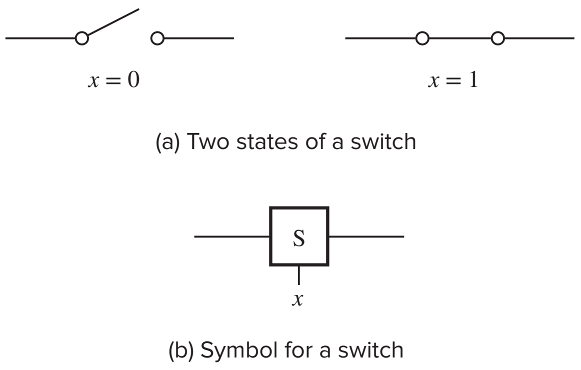
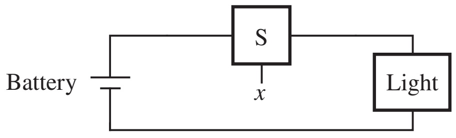
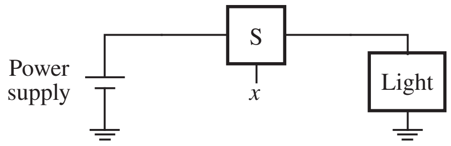


Figure 2.1 A binary switch.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

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*.

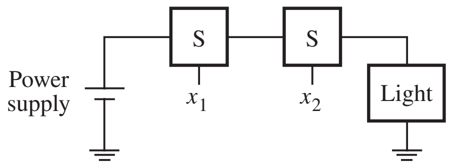
The circuit in Figure 2.2*a* can be found in an ordinary flashlight, where the switch is a simple mechanical device. In an electronic circuit the switch is implemented as a transistor and the light may be a light-emitting diode (LED). An electronic circuit is powered by a power supply of a certain voltage, 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. This behavior can be described by the expression

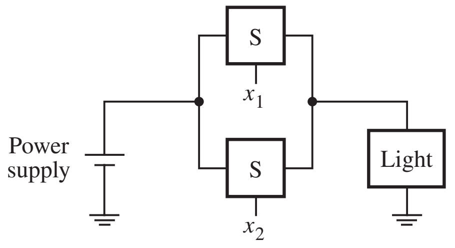
$$L(x_1, x_2) = x_1 \cdot x_2$$

where

$$L = 1 \text{ if } x_1 = 1 \text{ and } x_2 = 1,$$
$$L = 0 \text{ otherwise.}$$



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

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

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

where $L = 1$ if $x_1 = 1$ or $x_2 = 1$ or if $x_1 = x_2 = 1$,
$L = 0$ if $x_1 = x_2 = 0$.

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 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.

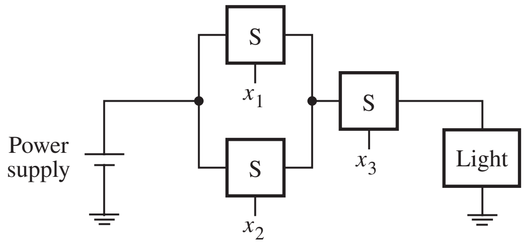


Figure 2.4 A series-parallel connection.

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

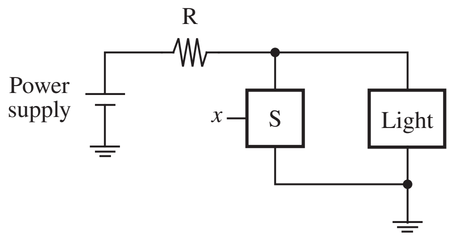


Figure 2.5 An inverting circuit.

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 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

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.

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.

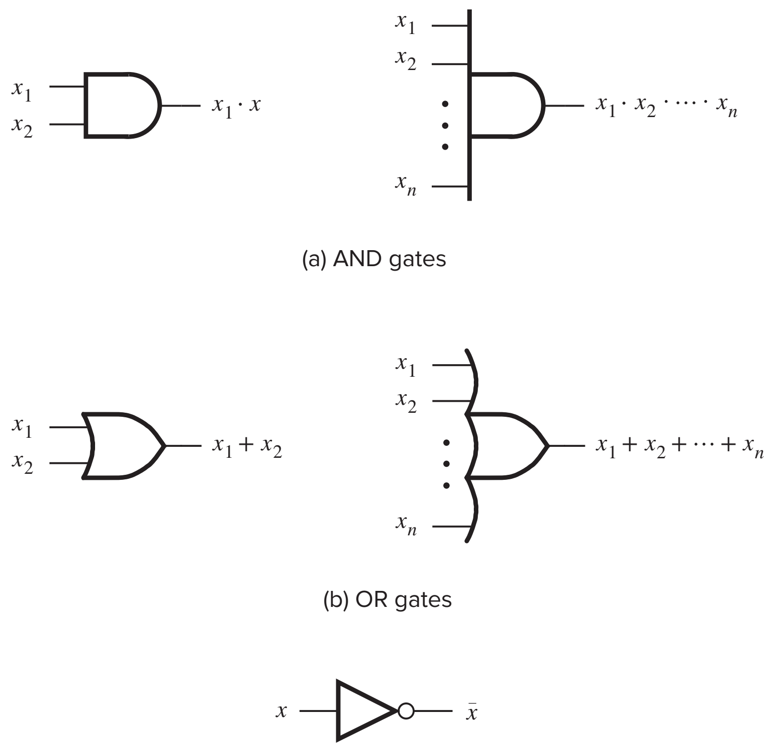
The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if one or more of the variables is equal to 1.

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 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.



(a) AND gates

(b) OR gates

(c) NOT gate

Figure 2.8 The basic gates.

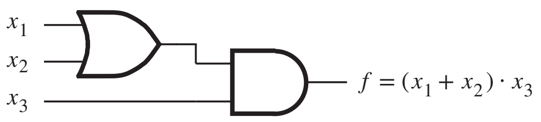
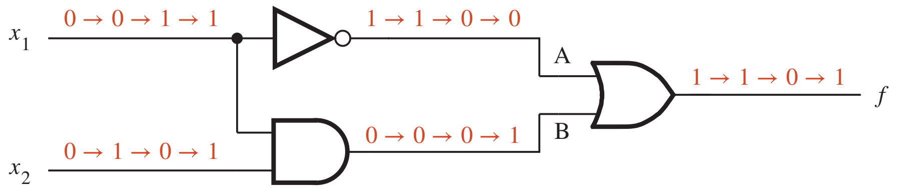


Figure 2.9 The function from Figure 2.4.

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.

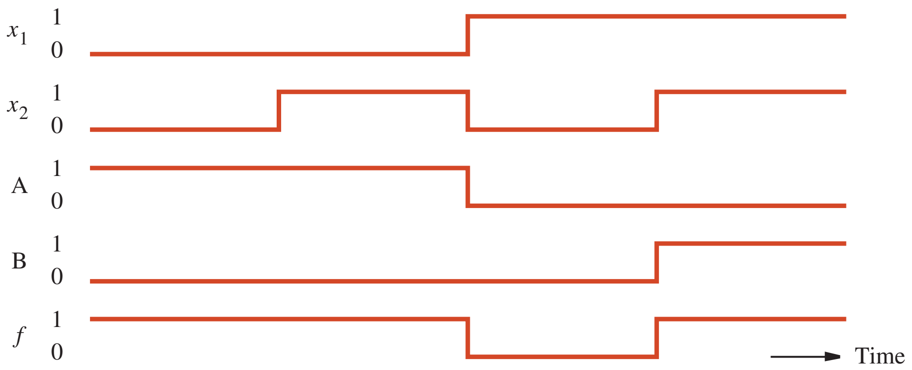
Figure 2.10*a* shows a simple network consisting of three gates. To analyze its functional behavior, we can consider what happens if we apply all possible combinations of the input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the



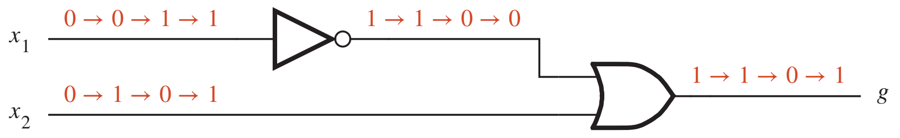
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

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.

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 red 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.

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

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

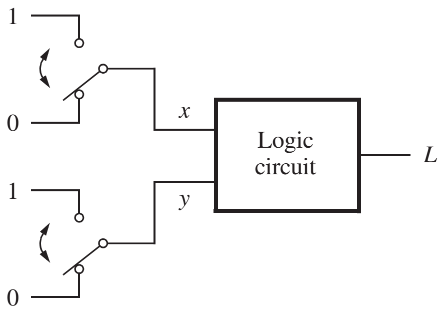
We have already established this equivalence through detailed analysis of the two circuits and construction of the truth table. But the same outcome can be achieved through algebraic manipulation of logic expressions. In Section 2.5 we will introduce a mathematical approach for dealing with logic functions,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

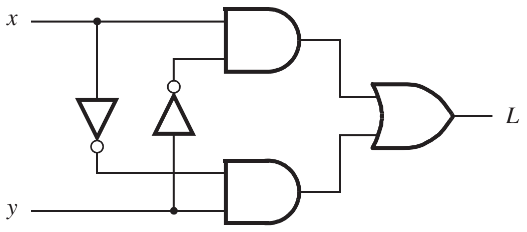
The reader may recognize the behavior of our light as being similar to that over a set of stairs in a house, where the light is controlled by two switches: one at the top of the stairs,



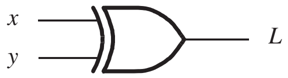
(a) Two switches that control a light

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

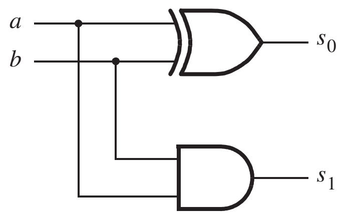
Figure 2.11 An example of a logic circuit.

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

(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.

and the other at the bottom. The light can be turned on or off by either switch because 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.

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:

1a. $0 \cdot 0 = 0$

1b. $1 + 1 = 1$

2a. $1 \cdot 1 = 1$

2b. $0 + 0 = 0$

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

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

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

4b. 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:

5a. $x \cdot 0 = 0$

5b. $x + 1 = 1$

6a. $x \cdot 1 = x$

6b. $x + 0 = x$

7a. $x \cdot x = x$

7b. $x + x = x$

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

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

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 5a, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true according to axiom 1a. Similarly, if $x = 1$, then theorem 5a states that $1 \cdot 0 = 0$, which is also true according to axiom 3a. The reader should verify that theorems 5a 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 · 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

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*.

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.

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

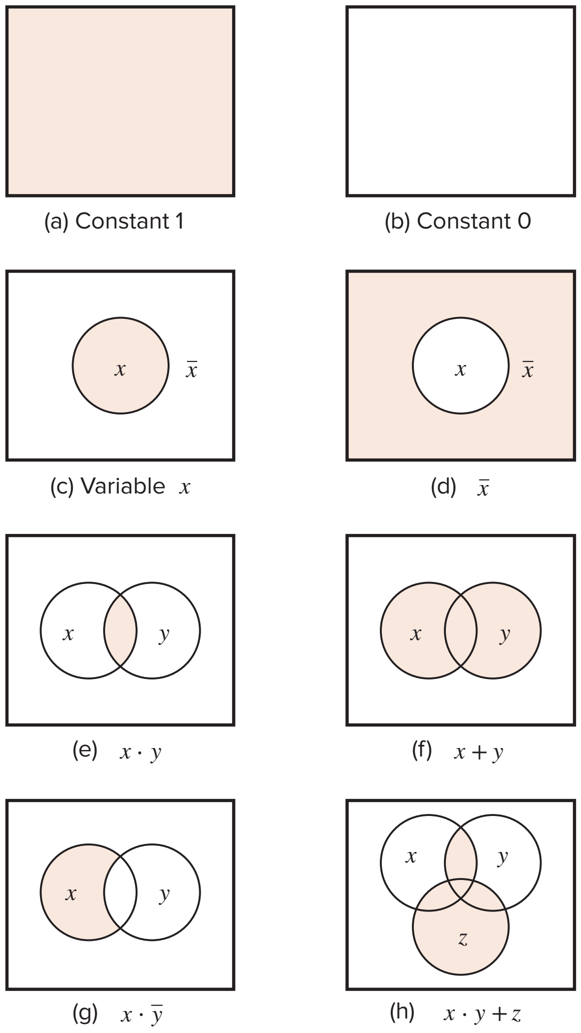


Figure 2.14 The Venn diagram representation.

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 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*)

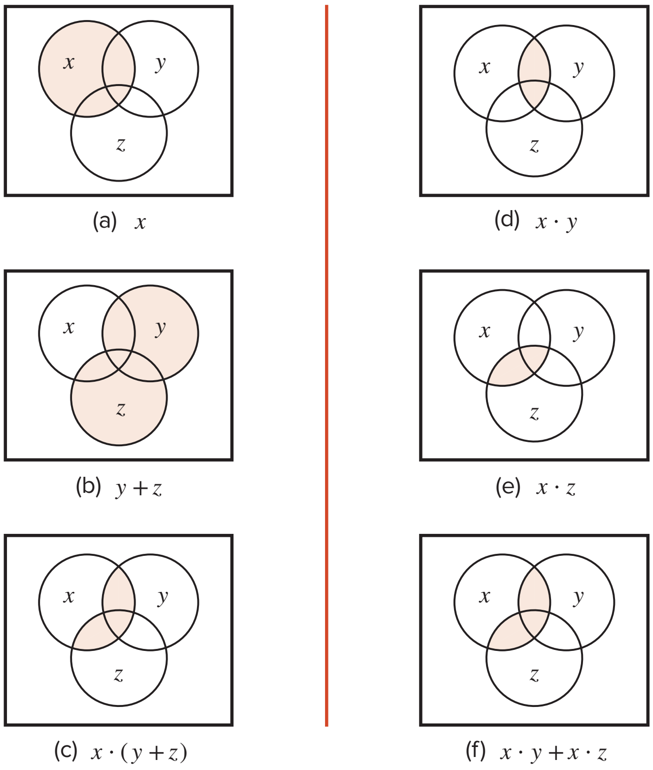


Figure 2.15 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

illustrates the OR operation, where $x+y$ represents the total area within both circles, 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

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