

M. Morris Mano • Michael D. Ciletti



DIGITAL DESIGN

*With An Introduction to
the Verilog HDL, VHDL,
and SystemVerilog*

Sixth Edition

 Pearson

Digital Design

With an Introduction to the Verilog HDL,
VHDL, and SystemVerilog



Digital Design

With an Introduction to the Verilog HDL,
VHDL, and SystemVerilog

SIXTH EDITION

M. Morris Mano

*Emeritus Professor of Computer Engineering
California State University, Los Angeles*

Michael D. Ciletti

*Emeritus Professor of Electrical and Computer Engineering
University of Colorado at Colorado Springs*



330 Hudson Street, NY NY 10013

Senior Vice President Courseware Portfolio Management: *Marcia J. Horton*

Director, Portfolio Management: *Engineering, Computer Science & Global Editions: Julian Partridge*

Higher Ed Portfolio Management: *Tracy Johnson (Dunkelberger)*

Portfolio Management Assistant: *Kristy Alaura*

Managing Content Producer: *Scott Disanno*

Content Producer: *Robert Engelhardt*

Web Developer: *Steve Wright*

Rights and Permissions Manager: *Ben Ferrini*

Manufacturing Buyer, Higher Ed, Lake Side

Communications Inc (LSC): *Maura Zaldivar-Garcia*

Inventory Manager: *Ann Lam*

Marketing Manager: *Demetrius Hall*

Product Marketing Manager: *Yvonne Vannatta*

Marketing Assistant: *Jon Bryant*

Cover Designer: *Marta Samsel*

Cover Photo: *The Mittens at Sunset – Monument Valley, Navaho Tribal Lands, Arizona, March 2015. Photograph courtesy of M. D. Ciletti and mdc Images LLC. Used with permission.*

Full-Service Project Management: *Vimala Vinayakam, SPi Global*

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

© 2018, 2013, 2007, 2002 by Pearson Education, Inc., Hoboken, New Jersey 07030. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, Pearson Education, Inc., Hoboken, New Jersey 07030.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Names: Mano, M. Morris, 1927- author. | Ciletti, Michael D., author.

Title: Digital design : with an introduction to the verilog HDL, VHDL, and system Verilog / M. Morris Mano, Emeritus Professor of Computer Engineering, California State University, Los Angeles, Michael D. Ciletti, Emeritus Professor of Electrical and Computer Engineering, University of Colorado at Colorado Springs.

Description: Sixth edition. | Upper Saddle River, New Jersey : Pearson Education, Inc., [2017] | Includes index.

Identifiers: LCCN 2017004488 | ISBN 9780134549897 (print : alk. paper)

Subjects: LCSH: Electronic digital computers—Circuits. | Logic circuits. | Logic design. | Digital integrated circuits.

Classification: LCC TK7888.3 .M343 2017 | DDC 621.39/5—dc23 LC record available at <https://lccn.loc.gov/2017004488>



Contents

Preface

ix

1 Digital Systems and Binary Numbers

1

| | | |
|-----|-------------------------------|----|
| 1.1 | Digital Systems | 1 |
| 1.2 | Binary Numbers | 4 |
| 1.3 | Number-Base Conversions | 6 |
| 1.4 | Octal and Hexadecimal Numbers | 9 |
| 1.5 | Complements of Numbers | 11 |
| 1.6 | Signed Binary Numbers | 17 |
| 1.7 | Binary Codes | 22 |
| 1.8 | Binary Storage and Registers | 31 |
| 1.9 | Binary Logic | 34 |

2 Boolean Algebra and Logic Gates

41

| | | |
|-----|--|----|
| 2.1 | Introduction | 42 |
| 2.2 | Basic Definitions | 42 |
| 2.3 | Axiomatic Definition of Boolean Algebra | 43 |
| 2.4 | Basic Theorems and Properties of Boolean Algebra | 47 |
| 2.5 | Boolean Functions | 50 |
| 2.6 | Canonical and Standard Forms | 56 |
| 2.7 | Other Logic Operations | 65 |
| 2.8 | Digital Logic Gates | 67 |
| 2.9 | Integrated Circuits | 73 |

v

3 Gate-Level Minimization 82

| | | |
|------|---------------------------------------|-----|
| 3.1 | Introduction | 83 |
| 3.2 | The Map Method | 83 |
| 3.3 | Four-Variable K-Map | 90 |
| 3.4 | Product-of-Sums Simplification | 95 |
| 3.5 | Don't-Care Conditions | 99 |
| 3.6 | NAND and NOR Implementation | 102 |
| 3.7 | Other Two-Level Implementations | 110 |
| 3.8 | Exclusive-OR Function | 115 |
| 3.9 | Hardware Description Languages (HDLs) | 121 |
| 3.10 | Truth Tables in HDLs | 138 |

4 Combinational Logic 147

| | | |
|------|--------------------------------------|-----|
| 4.1 | Introduction | 148 |
| 4.2 | Combinational Circuits | 148 |
| 4.3 | Analysis of Combinational Circuits | 149 |
| 4.4 | Design Procedure | 153 |
| 4.5 | Binary Adder–Subtractor | 156 |
| 4.6 | Decimal Adder | 168 |
| 4.7 | Binary Multiplier | 170 |
| 4.8 | Magnitude Comparator | 172 |
| 4.9 | Decoders | 175 |
| 4.10 | Encoders | 179 |
| 4.11 | Multiplexers | 182 |
| 4.12 | HDL Models of Combinational Circuits | 189 |
| 4.13 | Behavioral Modeling | 215 |
| 4.14 | Writing a Simple Testbench | 223 |
| 4.15 | Logic Simulation | 229 |

5 Synchronous Sequential Logic 245

| | | |
|-----|---|-----|
| 5.1 | Introduction | 246 |
| 5.2 | Sequential Circuits | 246 |
| 5.3 | Storage Elements: Latches | 248 |
| 5.4 | Storage Elements: Flip-Flops | 253 |
| 5.5 | Analysis of Clocked Sequential Circuits | 261 |
| 5.6 | Synthesizable HDL Models of Sequential Circuits | 275 |
| 5.7 | State Reduction and Assignment | 300 |
| 5.8 | Design Procedure | 305 |

6 Registers and Counters 326

| | | |
|-----|-----------------|-----|
| 6.1 | Registers | 326 |
| 6.2 | Shift Registers | 330 |

| | | |
|-----|--------------------------------------|-----|
| 6.3 | Ripple Counters | 338 |
| 6.4 | Synchronous Counters | 343 |
| 6.5 | Other Counters | 351 |
| 6.6 | HDL Models of Registers and Counters | 356 |

7 Memory and Programmable Logic 377

| | | |
|-----|---------------------------------|-----|
| 7.1 | Introduction | 378 |
| 7.2 | Random-Access Memory | 379 |
| 7.3 | Memory Decoding | 386 |
| 7.4 | Error Detection and Correction | 391 |
| 7.5 | Read-Only Memory | 394 |
| 7.6 | Programmable Logic Array | 400 |
| 7.7 | Programmable Array Logic | 404 |
| 7.8 | Sequential Programmable Devices | 408 |

8 Design at the Register Transfer Level 429

| | | |
|------|---|-----|
| 8.1 | Introduction | 430 |
| 8.2 | Register Transfer Level (RTL) Notation | 430 |
| 8.3 | RTL Descriptions | 432 |
| 8.4 | Algorithmic State Machines (ASMs) | 450 |
| 8.5 | Design Example (ASMD CHART) | 459 |
| 8.6 | HDL Description of Design Example | 469 |
| 8.7 | Sequential Binary Multiplier | 487 |
| 8.8 | Control Logic | 492 |
| 8.9 | HDL Description of Binary Multiplier | 498 |
| 8.10 | Design with Multiplexers | 513 |
| 8.11 | Race-Free Design (Software Race Conditions) | 529 |
| 8.12 | Latch-Free Design (Why Waste Silicon?) | 532 |
| 8.13 | SystemVerilog—An Introduction | 533 |

9 Laboratory Experiments with Standard ICs and FPGAs 555

| | | |
|------|---|-----|
| 9.1 | Introduction to Experiments | 555 |
| 9.2 | Experiment 1: Binary and Decimal Numbers | 560 |
| 9.3 | Experiment 2: Digital Logic Gates | 563 |
| 9.4 | Experiment 3: Simplification of Boolean Functions | 565 |
| 9.5 | Experiment 4: Combinational Circuits | 567 |
| 9.6 | Experiment 5: Code Converters | 568 |
| 9.7 | Experiment 6: Design with Multiplexers | 570 |
| 9.8 | Experiment 7: Adders and Subtractors | 572 |
| 9.9 | Experiment 8: Flip-Flops | 575 |
| 9.10 | Experiment 9: Sequential Circuits | 577 |
| 9.11 | Experiment 10: Counters | 579 |

viii Contents

| | | |
|------|--|-----|
| 9.12 | Experiment 11: Shift Registers | 580 |
| 9.13 | Experiment 12: Serial Addition | 584 |
| 9.14 | Experiment 13: Memory Unit | 585 |
| 9.15 | Experiment 14: Lamp Handball | 587 |
| 9.16 | Experiment 15: Clock-Pulse Generator | 591 |
| 9.17 | Experiment 16: Parallel Adder and Accumulator | 593 |
| 9.18 | Experiment 17: Binary Multiplier | 595 |
| 9.19 | HDL Simulation Experiments and Rapid Prototyping with FPGAs | 599 |

10 Standard Graphic Symbols 605

| | | |
|------|------------------------------------|-----|
| 10.1 | Rectangular-Shape Symbols | 605 |
| 10.2 | Qualifying Symbols | 608 |
| 10.3 | Dependency Notation | 610 |
| 10.4 | Symbols for Combinational Elements | 612 |
| 10.5 | Symbols for Flip-Flops | 614 |
| 10.6 | Symbols for Registers | 616 |
| 10.7 | Symbols for Counters | 619 |
| 10.8 | Symbol for RAM | 621 |

Appendix 624

Answers to Selected Problems 638

Index 683

Preface

The speed, density, and complexity of today's digital devices are made possible by advances in physical processing technology and digital design methodology. Aside from semiconductor technology, the design of leading-edge devices depends critically on hardware description languages (HDLs) and synthesis tools. Three public-domain languages, *Verilog*, *VHDL*, and *SystemVerilog*, all play a role in design flows for today's digital devices. HDLs, together with fundamental knowledge of digital logic circuits, provide an entry point to the world of digital design for students majoring in computer science, computer engineering, and electrical engineering.

In the not-too-distant past, it would be unthinkable for an electrical engineering student to graduate without having used an oscilloscope. Today, the needs of industry demand that undergraduate students become familiar with the use of at least one hardware description language. Their use of an HDL as a student will better prepare them to be productive members of a design team after they graduate.

Given the presence of three HDLs in the design arena, we have expanded our presentation of HDLs in *Digital Design* to treat *Verilog* and *VHDL*, and to provide an introduction to *SystemVerilog*. Our intent is not to require students to learn three, or even two, languages, but to *provide the instructor with a choice between Verilog and VHDL while teaching a systematic methodology for design, regardless of the language, and an optional introduction to SystemVerilog*. Certainly, *Verilog* and *VHDL* are widely used and taught, dominate the design space, and have common underlying concepts supporting combinational and sequential logic design, and both are essential to the synthesis of high-density integrated circuits. **Our text offers *parallel tracks of presentation of both languages, but allows concentration on a single language*.** The level of treatment of *Verilog* and *VHDL* is essentially equal, without emphasizing one language over the other. **A language-neutral presentation of digital design is a common**

thread through the treatment of both languages. A large set of problems, which are stated in language-neutral terms, at the end of each chapter can be worked with either Verilog or VHDL.

The emphasis in our presentation is on digital design, with HDLs in a supporting role. Consequently, we present only those details of *Verilog*, *VHDL*, and *SystemVerilog* that are needed to support our treatment of an *introduction* to digital design. Moreover, although we present examples using each language, we identify and segregate the treatment of topics and examples so that *the instructor can choose a path of presentation for a single language*—either *Verilog* or *VHDL*. Naturally, a path that emphasizes *Verilog* can conclude with *SystemVerilog*, but it can be skipped without compromising the objectives. The introduction to *SystemVerilog* is selective—we present only topics and examples that are extensions of *Verilog*, and well within the scope of an introductory treatment. To be clear, we are *not* advocating simultaneous presentation of the languages. The instructor can choose either *Verilog/SystemVerilog* or *VHDL* as the core language supporting an introductory course in digital design. **Regardless of the language, our focus is on digital design.**

The language-based examples throughout the book are not just about the details of an HDL. We emphasize and demonstrate the modeling and verification of digital circuits having specified behavior. **Neither Verilog or VHDL are covered in their entirety.** *Some details of the languages will be left to the reader's continuing education and use of web resources.* Regardless of language, our examples introduce a design methodology based on the concept of computer-aided modeling of digital systems by means of a mainstream, IEEE-standardized, hardware description language.

This revision of *Digital Design* begins each chapter with a statement of its objectives. Problems at the end of each chapter are combined with in-chapter examples, and with in-chapter Practice Exercises. Together, these encounters with the subject matter bring the student closer to achieving the stated objectives and becoming skilled in digital design. Answers are given to selected problems at the end of each chapter. A Solution Manual gives detailed solutions to all of the problems at the end of the chapters. The level of detail of the solutions is such that an instructor can use individual problems to support classroom instruction.

MULTIMODAL LEARNING

Like the previous editions, this edition of *Digital Design* supports a multimodal approach to learning. The so-called VARK^{1,2} characterization of learning modalities identifies four major modes by which we learn: (V) visual, (A) aural (hearing), (R) reading, and (K) kinesthetic. The relatively high level of illustrations and graphical content of our text

¹ Kolb, David A. (2015) [1984]. *Experiential learning: Experience as the source of learning and development* (2nd ed.). Upper Saddle River, NJ: Pearson Education. ISBN 9780133892406. OCLC 909815841.

² Fleming, Neil D. (2014). "The VARK modalities" *vark-learn.com*.

addresses the visual (V) component of VARK; discussions and numerous examples address the reading (R) component. Students who exploit the availability of free Verilog, VHDL and SystemVerilog simulators and synthesis tools to work assignments are led through a kinesthetic learning experience, including the delight of designing a digital circuit that actually works. The remaining element of VARK, the aural/auditory (A) experience depends on the instructor and the attentiveness of the student (Put away the smart phone!). We have provided an abundance of materials and examples to support classroom lectures. Thus, a course using *Digital Design*, can provide a rich, balanced, learning experience and address all the modes identified by VARK.

For skeptics who might still question the need to present and use HDLs in a first course in digital design, we note that industry does not rely on schematic-based design methods. Schematic entry creates a representation of functionality that is implicit in the constructs and layout of the schematic. Unfortunately, it is difficult for anyone in a reasonable amount of time to determine the functionality represented by the schematic of a logic circuit without having been instrumental in its construction, or without having additional documentation expressing the design intent. Consequently, industry today relies almost exclusively on HDLs to describe the functionality of a design and to serve as a basis for documenting, simulating, testing, and synthesizing the hardware implementation of the design in a standard cell-based ASIC or an FPGA. The utility of a schematic depends on the detailed documentation of a carefully constructed hierarchy of design units. In the past, designers relied on their years of experience to create a schematic of a circuit to implement functionality. Today's designers using HDLs, can express functionality directly and explicitly, without years of accumulated experience, and use synthesis tools to generate the schematic as a byproduct, automatically. Industry adopted HDL-based design flows because schematic entry dooms us to inefficiency, if not failure, in understanding and designing large, complex, ICs.

Introduction of HDLs in a first course in digital design is not intended to replace fundamental understanding of the building blocks of such circuits, or to eliminate a discussion of manual methods of design. It is still essential for students to understand *how hardware works*. Thus, this edition of *Digital Design* retains a thorough treatment of combinational and sequential logic design and a foundation in Boolean algebra. Manual design practices are presented, and their results are compared with those obtained using HDLs. What we are presenting, however, is an emphasis on how hardware is designed today, to better prepare a student for a career in today's industry, where HDL-based design practices are dominant.

FLEXIBILITY

We include both manual and HDL-based design examples. Our end-of-chapter problems cross-reference problems that access a manual design task with a companion problem that uses an HDL to accomplish the assigned task. We also link the manual and HDL-based approaches by presenting annotated results of simulations in the text, in answers to selected problems at the end of the text, and extensively in the solution manual.

NEW TO THIS EDITION

This edition of *Digital Design* uses the latest features of IEEE Standard 1364, but only insofar as they support our pedagogical objectives. The revisions and updates to the text include:

- Elimination of specialized circuit-level content not typically covered in a first course in logic circuits and digital design (e.g., RTL, DTL, and emitter-coupled logic circuits)
- Addition of “Web Search Topics” at the end of each chapter to point students to additional subject matter available on the web
- Revision of approximately one-third of the problems at the end of the chapters
- A solution manual for the entire text, including all new problems
- Streamlining of the discussion of Karnaugh maps
- Integration of treatment of basic CMOS technology with treatment of logic gates
- Inclusion of an appendix introducing semiconductor technology
- Treatment of digital design with VHDL and SystemVerilog

DESIGN METHODOLOGY

A highlight of our presentation is a systematic methodology for designing a state machine to control the data path of a digital system. The framework in which this material is presented treats the realistic situation in which status signals from the datapath are used by the controller, i.e., the system has feedback. Thus, our treatment provides a foundation for designing complex and interactive digital systems. Although it is presented with an emphasis on HDL-based design, the methodology is also applicable to manual-based approaches to design and is language-neutral.

JUST ENOUGH HDL

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

VERIFICATION

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

HDL CONTENT

We have ensured that all examples in the text and all answers in the solution manual conform to accepted industry practices for modeling digital hardware. As in the previous edition, HDL material is inserted in separate sections so that it can be covered or skipped as desired, does not diminish treatment of manual-based design, and does not dictate the sequence of presentation. The treatment is at a level suitable for beginning students who are learning digital circuits and an HDL at the same time. The text prepares students to work on significant independent design projects and to succeed in a later course in computer architecture and advanced digital design.

Instructor Resources

Instructors can obtain the following classroom-ready resources from the publisher:

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

HDL Simulators

Two free simulators can be downloaded from www.Syncad.com. The first simulator is *VeriLogger Pro*, a traditional Verilog simulator that can be used to simulate the HDL examples in the book and to verify the solutions of HDL problems. This simulator accepts the syntax of the IEEE-1995 standard and will be useful to those who have legacy models. As an interactive simulator, *VeriLogger Extreme* accepts the syntax of IEEE-2001 as well as IEEE-1995, allowing the designer to simulate and analyze design

ideas before a complete simulation model or schematic is available. This technology is particularly useful for students because they can quickly enter Boolean and *D* flip-flop or latch input equations to check equivalency or to experiment with flip-flops and latch designs. Free design tools that support design entry, simulation and synthesis (of FPGAs) are available from www.altera.com and from www.xilinx.com.

Chapter Summary

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

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

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

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

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

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

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

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

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

Chapter 9 presents experiments that can be performed in the laboratory with hardware that is readily available commercially. The operation of the ICs used in the experiments is explained by referring to diagrams of similar components introduced in previous chapters. Each experiment is presented informally and the student is expected to design the circuit and formulate a procedure for checking its operation in the laboratory. The lab experiments can be used in a stand-alone manner too and can be accomplished by a traditional approach, with a breadboard and TTL circuits, or with an HDL/synthesis approach using FPGAs. Today, software for synthesizing an HDL model and implementing a circuit with an FPGA is available at no cost from vendors of FPGAs, allowing students to conduct a significant amount of work in their personal environment before using prototyping boards and other resources in a lab. Circuit boards for rapid prototyping circuits with FPGAs are available at a nominal cost, and typically include push buttons, switches, seven-segment displays, LCDs, keypads, and other I/O devices. With these resources, students can work prescribed lab exercises or their own projects and get results immediately.

Chapter 10 presents the standard graphic symbols for logic functions recommended by an ANSI/IEEE standard. These graphic symbols have been developed for small-scale integration (SSI) and medium-scale integration (MSI) components so that the user can recognize each function from the unique graphic symbol assigned. The chapter shows the standard graphic symbols of the ICs used in the laboratory experiments.

Acknowledgments

We are grateful to the reviewers of *Digital Design*, 6e. Their expertise, careful reviews, and suggestions helped shape this edition.

Vijay Madiseti, Georgia Tech
 Dmitri Donetski, SUNY Stony Brook
 David Potter, Northeastern
 Xiaolong Wu, California State-Long Beach
 Avinash Kodi, Ohio University
 Lee Belfore, Old Dominion University

We also wish to express our gratitude to the editorial and publication team at Pearson Education for supporting this edition of our text. We are grateful, too, for the ongoing support and encouragement of our wives, Sandra and Jerilynn.

M. MORRIS MANO
Emeritus Professor of Computer Engineering
California State University, Los Angeles

MICHAEL D. CILETTI
Emeritus Professor of Electrical and Computer Engineering
University of Colorado at Colorado Springs

Chapter 1

Digital Systems and Binary Numbers

CHAPTER OBJECTIVES

1. Understand binary number system.
2. Know how to convert between binary, octal, decimal, and hexadecimal numbers.
3. Know how to take the complement and reduced radix complement of a number.
4. Know how to form the code of a number.
5. Know how to form the parity bit of a word.

1.1 DIGITAL SYSTEMS

Digital systems have such a prominent role in everyday life that we refer to the present technological period as the *digital age*. Digital systems are used in communication, business transactions, traffic control, spacecraft guidance, medical treatment, weather monitoring, the Internet, and many other commercial, industrial, and scientific enterprises. We have digital telephones, digital televisions, digital versatile discs (DVDs), digital cameras, personal, handheld, touch-screen devices, and, of course, digital computers. We enjoy music downloaded to our portable media player (e.g., iPod Touch®) and other handheld devices having high-resolution displays and touch-screen graphical user interfaces (GUIs). GUIs enable them to execute commands that appear to the user to be simple, but which, in fact, involve precise execution of a sequence of complex internal instructions. Most, if not all, of these devices have a special-purpose digital computer, or

2 Chapter 1 Digital Systems and Binary Numbers

processor, embedded within them. The most striking property of the digital computer is its generality. It can follow a sequence of instructions, called a program, which operates on given data. The user can specify and change the program or the data according to the specific need. Because of this flexibility, general-purpose digital computers can perform a variety of information-processing tasks that range over a wide spectrum of applications and provide unprecedented access to massive repositories of information and media.

One characteristic of digital systems is their ability to represent and manipulate discrete elements of information. Any set that is restricted to a finite number of elements contains discrete information. Examples of discrete sets are the 10 decimal digits, the 26 letters of the alphabet, the 52 playing cards, and the 64 squares of a chessboard. Early digital computers were used for numeric computations. In this case, the discrete elements were the digits. From this application, the term *digital* computer emerged.

Discrete elements of information are represented in a digital system by physical quantities called *signals*. Electrical signals such as voltages and currents are the most common. Electronic devices called transistors predominate in the circuitry that implement, represent, and manipulate these signals. The signals in most present-day electronic digital systems use just two discrete values and are therefore said to be *binary*. A binary digit, called a *bit*, has two numerical values: 0 and 1. Discrete elements of information are represented with groups of bits called *binary codes*. For example, the decimal digits 0 through 9 are represented in a digital system with a code of four bits (e.g., the number 7 is represented by 0111). How a pattern of bits is interpreted as a number depends on the code system in which it resides. To make this distinction, we could write $(0111)_2$ to indicate that the pattern 0111 is to be interpreted in a binary system, and $(0111)_{10}$ to indicate that the reference system is decimal. Then $0111_2 = 7_{10}$, which is not the same as 0111_{10} , or one hundred eleven. The subscript indicating the base for interpreting a pattern of bits will be used only when clarification is needed. Through various techniques, groups of bits can be made to represent discrete symbols, not necessarily numbers, which are then used to develop the system in a digital format. Thus, a digital system is a system that manipulates discrete elements of information represented internally in binary form. In today's technology, binary systems are most practical because, as we will see, they can be implemented with electronic components.

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

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

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

A digital system is an interconnection of digital modules. **To understand the operation of each digital module, it is necessary to have a basic knowledge of digital circuits and their logical function.** The first seven chapters of this book present the basic tools of digital design, such as logic gate structures, combinational and sequential circuits, and programmable logic devices. Chapter 8 introduces digital design at the register transfer level (RTL) using a modern, public-domain hardware description language (HDL). Chapter 9 concludes the text with laboratory exercises using digital circuits.

Today's array of inexpensive digital devices is made possible by the convergence of fabrication technology and computer-based design methodology. Today's "best practice" in digital design methodology uses HDLs to describe and simulate the functionality of a digital circuit. An HDL resembles a programming language and is suitable for describing digital circuits in textual form. It is used to simulate a digital system to verify its operation before hardware is built. It is also used in conjunction with logic synthesis tools to automate the design process. Because **it is important that students become familiar with an HDL-based design methodology**, HDL descriptions of digital circuits are presented throughout the book. While these examples help illustrate the features of an HDL, they

4 Chapter 1 Digital Systems and Binary Numbers

also demonstrate the best practices used by industry to exploit HDLs. Ignorance of these practices will lead to cute, but worthless, HDL models that may simulate a phenomenon, but that cannot be synthesized by design tools, or to models which waste silicon area or synthesize to hardware that does not operate correctly.

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

1.2 BINARY NUMBERS

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

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

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

$$a_5 a_4 a_3 a_2 a_1 a_0 \cdot a_{-1} a_{-2} a_{-3}$$

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

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

with $a_3 = 7$, $a_2 = 3$, $a_1 = 9$, and $a_0 = 2$, and the other coefficients equal to zero.

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

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

There are many different number systems. In general, a number expressed in a base- r system has coefficients multiplied by powers of r :

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

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

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

The coefficient values for base 5 can be only 0, 1, 2, 3, and 4. The octal number system is a base-8 system that has eight digits: 0, 1, 2, 3, 4, 5, 6, 7. An example of an octal number is $(127.4)_8$. To determine its equivalent decimal value, we expand the number in a power series with a base of 8:

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

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

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

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

The hexadecimal system is used commonly by designers to represent long strings of bits in the addresses, instructions, and data in digital systems. For example, B65F is used to represent 1011011001011111.

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

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

There are four 1's in the binary number. The corresponding decimal number is the sum of the four powers of two. Zero and the first 24 numbers obtained from 2 to the power of n are listed in Table 1.1. In computer work, 2^{10} is referred to as K (kilo), 2^{20} as M (mega), 2^{30} as G (giga), and 2^{40} as T (tera). Thus, $4K = 2^{12} = 4,096$ and $16M = 2^{24} = 16,777,216$. Computer memory capacity and word size are usually given in bytes. A *byte* is equal to eight bits and can accommodate (i.e., represent the code of) one keyboard character. A computer hard disk with four gigabytes of storage has a capacity of $4G = 2^{32}$ bytes (approximately 4 billion bytes). A terabyte is 1024 gigabytes, approximately 1 trillion bytes.

6 Chapter 1 Digital Systems and Binary Numbers

Table 1.1
Powers of Two

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

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

augend: 101101

addend: +100111

sum: 1010100

minuend: 101101

subtrahend: −100111

difference: 000110

multiplicand: 1011

multiplier: × 101

partial product: 0000

product: 110111

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

Practice Exercise 1.1

What is the decimal value of $1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$?

Answer: 21

1.3 NUMBER-BASE CONVERSIONS

Representations of a number in a different radix are said to be equivalent if they have the same decimal representation. For example, $(0011)_8$ and $(1001)_2$ are equivalent—both

have decimal value 9. The conversion of a number in base r to decimal is done by expanding the number in a power series and adding all the terms as shown previously. We now present a general procedure for the reverse operation of converting a decimal number to a number in base r . If the number includes a radix point, it is necessary to separate the number into an integer part and a fraction part, since each part must be converted differently. The conversion of a decimal integer to a number in base r is done by *dividing the number and all successive quotients by r and accumulating the remainders*. This procedure is best illustrated by example.

EXAMPLE 1.1

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

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

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

The arithmetic process can be manipulated more conveniently as follows:

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

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



EXAMPLE 1.2

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

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

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

EXAMPLE 1.3

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

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

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

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

EXAMPLE 1.4

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

$$0.513 \times 8 = 4.104$$

$$0.104 \times 8 = 0.832$$

$$0.832 \times 8 = 6.656$$

$$0.656 \times 8 = 5.248$$

$$0.248 \times 8 = 1.984$$

$$0.984 \times 8 = 7.872$$

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

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

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

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

From Examples 1.2 and 1.4, we have

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

Practice Exercise 1.2

Convert $(117.23)_{10}$ to octal.

Answer: $(117.23)_{10} = (165.1656)_8$

1.4 OCTAL AND HEXADECIMAL NUMBERS

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

The conversion from binary to octal is easily accomplished by partitioning the binary number into groups of three digits each, starting from the binary point and proceeding

10 Chapter 1 Digital Systems and Binary Numbers

Table 1.2
Numbers with Different Bases

| Decimal (base 10) | Binary (base 2) | Octal (base 8) | Hexadecimal (base 16) |
|----------------------|--------------------|-------------------|--------------------------|
| 00 | 0000 | 00 | 0 |
| 01 | 0001 | 01 | 1 |
| 02 | 0010 | 02 | 2 |
| 03 | 0011 | 03 | 3 |
| 04 | 0100 | 04 | 4 |
| 05 | 0101 | 05 | 5 |
| 06 | 0110 | 06 | 6 |
| 07 | 0111 | 07 | 7 |
| 08 | 1000 | 10 | 8 |
| 09 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |

to the left and to the right. The corresponding octal digit is then assigned to each group. The following example illustrates the procedure:

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

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

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

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

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

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

and

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

Binary numbers are difficult to work with because they require three or four times as many digits as their decimal equivalents. For example, the binary number 1111111111 is equivalent to decimal 4095. However, digital computers use binary representation of numbers, and it is sometimes necessary for the human operator or user to communicate directly with the machine by means of such numbers. One scheme that retains the binary system in the computer, but reduces the number of digits the human must consider,¹ utilizes the relationship between the binary number system and the octal or hexadecimal system. By this method, the human thinks in terms of octal or hexadecimal numbers and performs the required conversion by inspection when direct communication with the machine is necessary. Thus, the binary number 1111111111 has 12 digits and is expressed in octal as 7777 (4 digits) or in hexadecimal as FFF (3 digits). During communication between people (about binary numbers in the computer), the octal or hexadecimal representation is more desirable because it can be expressed more compactly with a third or a quarter of the number of digits required for the equivalent binary number. Thus, **most computer manuals use either octal or hexadecimal numbers to specify instructions and other binary quantities.** The choice between them is arbitrary, although hexadecimal tends to win out, since it can represent a byte with two digits.

Practice Exercise 1.3

Find the binary representation of 135_{10} .

Answer: $135_{10} = 1110\ 0001_2$

Practice Exercise 1.4

Find the octal representation of $(135)_{10}$.

Answer: $135_{10} = 702_8$

1.5 COMPLEMENTS OF NUMBERS

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

¹ Machines having a word length of 64 bits are common.

Diminished Radix Complement

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

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

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

For binary numbers, $r = 2$ and $r - 1 = 1$, so the 1's complement of N is $(2^n - 1) - N$. Again, 2^n is represented by a binary number that consists of a 1 followed by n 0's. $2^n - 1$ is a binary number represented by n 1's. For example, if $n = 4$, we have $2^4 = (10000)_2$ and $2^4 - 1 = (1111)_2$. Thus, the 1's complement of a binary number is obtained by subtracting each digit from 1. However, when subtracting binary digits from 1, we can have either $1 - 0 = 1$ or $1 - 1 = 0$, which causes the bit to change from 0 to 1 or from 1 to 0, respectively. Therefore, **the 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's**. The following are some numerical examples:

The 1's complement of 1011000 is 0100111.

The 1's complement of 0101101 is 1010010.

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

Radix Complement

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

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

the 10's complement of 012398 is 987602

and

the 10's complement of 246700 is 753300

The 10's complement of the first number (012398) is obtained by subtracting 8 from 10 in the least significant position and subtracting all other digits from 9. The 10's complement of the second number (246700) is obtained by leaving the two least significant 0's unchanged, subtracting 7 from 10, and subtracting the other three digits from 9.

Practice Exercise 1.5

Find (a) the diminished radix (9's) complement and (b) the radix (10's) complement of 135_{10} .

Answer:

(a) 9's complement: 864_{10}

(b) 10's complement: 865_{10}

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

the 2's complement of 1101100 is 0010100

and

the 2's complement of 0110111 is 1001001

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

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

Subtraction with Complements

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

14 Chapter 1 Digital Systems and Binary Numbers

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

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

The following examples illustrate the procedure:

EXAMPLE 1.5

Using 10's complement, subtract $72532 - 3250$.

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

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

EXAMPLE 1.6

Using 10's complement, subtract $3250 - 72532$.

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

There is no end carry. Therefore, the answer is written with a minus sign as $-(10's \text{ complement of } 30718) = -69282$.

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

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

EXAMPLE 1.7

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

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

$$\begin{array}{rcl}
 \text{(b)} & Y = & 1000011 \\
 & 2\text{'s complement of } X = & +0101100 \\
 & \text{Sum} = & 1101111
 \end{array}$$

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

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

EXAMPLE 1.8

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

$$\begin{array}{rcl}
 \text{(a)} & X - Y = 1010100 - 1000011 \\
 & X = & 1010100 \\
 & 1\text{'s complement of } Y = & +0111100 \\
 & \text{Sum} = & 10010000 \\
 & \text{End-around carry} = & + \quad 1 \\
 & \text{Answer: } X - Y = & 0010001
 \end{array}$$

$$\begin{array}{rcl}
 \text{(b)} & Y - X = 1000011 - 1010100 \\
 & Y = & 1000011 \\
 & 1\text{'s complement of } X = & +0101011 \\
 & \text{Sum} = & 1101110
 \end{array}$$

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

16 Chapter 1 Digital Systems and Binary Numbers

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

Practice Exercise 1.6

Given $X = (1101010)_2$ and $Y = (0101011)_2$, (a) find $X - Y$ and (b) find $Y - X$ using 2's complements.

Answer: (a) $X = (1101010)_2 = 106_{10}$, $Y = (0101011)_2 = 43_{10}$
 $X - Y = 106_{10} - 43_{10} = 63_{10}$
 2's complement of Y : 1010101_2
 $X - Y = (1101010)_2 + (1010101)_2 = (0111111)_2 = 63_{10}$

(b) $Y - X = 43_{10} - 106_{10} = -63_{10}$
 2's complement of X : $(1\ 0010110)_2$
 $Y - X = (010\ 1011)_2 + (001\ 0110)_2 = (100\ 0001)_2$ No end carry
 $Y - X = -2$'s complement of $(100\ 0001)_2$
 $Y - X = -(011\ 1111)_2 = -63_{10}$

Practice Exercise 1.7

Repeat Practice Exercise 1.5 using 1's complements.

Answer: (a) $X = (1101010)_2 = 106_{10}$, $Y = (0101011)_2 = 43_{10}$
 $X - Y = 106_{10} - 43_{10} = 63_{10}$
 1's complement of Y : 1010100_2
 $X - Y = (1101010)_2$
 $\quad + (1010100)_2$
 $\quad (10111110)_2$ End-around carry
 $X - Y = 0111110_2 + 0000001_2 = 0111111_2 = 63_{10}$

(b) $X = (1101010)_2 = 106_{10}$, $Y = (0101011)_2 = 43_{10}$
 $Y - X = 43_{10} - 106_{10} = -63_{10}$
 1's complement of X : $(0010101)_2$
 $Y - X = (0100011)_2$
 $\quad + (0010101)_2$
 $\quad (011\ 1000)_2$
 No end-around carry
 $Y - X = -1$'s complement of $((011\ 1000)_2 + (000\ 0001)_2)$
 $Y - X = -1$'s complement of $(100\ 0001)_2 = (011\ 1110)_2 = -63_{10}$

1.6 SIGNED BINARY NUMBERS

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

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

Practice Exercise 1.8

Which bit of a signed binary number represents the sign?

Answer: The leftmost bit

Practice Exercise 1.9

What unsigned binary number is represented by the string of bits 11001?

Answer: 25_{10}

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

As an example, consider the number 9, represented in binary with eight bits. +9 is represented with a sign bit of 0 in the leftmost position, followed by the binary equivalent

18 Chapter 1 Digital Systems and Binary Numbers

of 9, which gives 00001001. Note that all eight bits must have a value; therefore, 0's are inserted following the sign bit up to the first 1. Although there is only one way to represent +9, there are three different ways to represent -9 with eight bits:

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

Practice Exercise 1.10

What decimal number does the signed-magnitude binary number $N = 10011$ represent?

Answer: $N = -3_{10}$

Practice Exercise 1.11

Convert the signed-magnitude binary number $N = 01100$ to a negative value having the same magnitude.

Answer: $N = 11100$

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

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

Practice Exercise 1.12

Represent -5 three ways with 8 bits: (a) signed-magnitude (b) signed 1's complement, and (c) signed 2's complement.

Answer: (a) 10000101, (b) 11111010, and (c) 11111011

Table 1.3
Signed Binary Numbers

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

Practice Exercise 1.13

In the signed-2's-complement system, negate the number 7_{10} , represented with 8 bits.

Answer: $N = 0000\ 0111_2$
 $1\text{'s comp} = 1111\ 1000_2$
 $2\text{'s comp} = 1111\ 1001_2$

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

Arithmetic Addition

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

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

Numerical examples for addition follow:

| | | | |
|-------|----------|-----|----------|
| + 6 | 00000110 | − 6 | 11111010 |
| +13 | 00001101 | +13 | 00001101 |
| +19 | 00010011 | +7 | 00000111 |
| <hr/> | | | |
| + 6 | 00000110 | − 6 | 11111010 |
| −13 | 11110011 | −13 | 11110011 |
| − 7 | 11111001 | −19 | 11101101 |

Note that negative numbers must be initially in 2's-complement form and that if the sum obtained after the addition is negative, it is in 2's-complement form. For example, -7 is represented as 11111001, which is the 2's complement of $+7$.

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

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

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

Arithmetic Subtraction

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

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

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

$$\begin{aligned}(\pm A) - (+B) &= (\pm A) + (-B); \\ (\pm A) - (-B) &= (\pm A) + (+B).\end{aligned}$$

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

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

Practice Exercise 1.14 – Using 2's complements, find the following sums:

$$\begin{array}{r} \text{(a)} \quad +4 \\ \quad +11 \end{array}$$

$$\begin{array}{r} \text{(b)} \quad -4 \\ \quad +11 \end{array}$$

$$\begin{array}{r} \text{(c)} \quad +4 \\ \quad -11 \end{array}$$

$$\begin{array}{r} \text{(d)} \quad -4 \\ \quad -11 \end{array}$$

Answer:

$$\begin{array}{r} +4 \quad 0000 \ 0100 \\ \text{(a)} \quad +11 \quad 0000 \ 1011 \\ \hline +15 \quad 0000 \ 1111 \end{array}$$

$$\begin{array}{r} -4 \quad 1111 \ 1100 \\ \text{(b)} \quad +11 \quad 0000 \ 1011 \\ \hline +7 \quad 0000 \ 0111 \end{array}$$

$$\begin{array}{r} +4 \quad 0000 \ 0100 \\ \text{(c)} \quad -11 \quad 1111 \ 0101 \\ \hline -7 \quad 1111 \ 1001 \end{array}$$

$$\begin{array}{r} -4 \quad 1111 \ 1100 \\ \text{(d)} \quad -11 \quad 1111 \ 0101 \\ \hline -15 \quad 1111 \ 0001 \end{array}$$

1.7 BINARY CODES

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

An n -bit binary code is a group of n bits that assumes up to 2^n distinct combinations of 1's and 0's, with each combination representing one element of the set that is being coded. A set of four elements can be coded with two bits, with each element assigned one of the following bit combinations: 00, 01, 10, and 11. A set of eight elements requires a three-bit code and a set of 16 elements requires a four-bit code. The bit combination of an n -bit code is determined from the count in binary from 0 to $2^n - 1$. Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise, the code assignment will be ambiguous.

Although the *minimum* number of bits required to code 2^n distinct quantities is n , there is no *maximum* number of bits that may be used for a binary code. For example, the 10 decimal digits can be coded with 10 bits, and each decimal digit can be assigned a bit combination of nine 0's and a 1. In this particular binary code, the digit 6 is assigned the bit combination 000100000.

Binary-Coded Decimal Code

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

A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2. The 10 decimal digits form such a set. A binary

code that distinguishes among 10 elements must contain at least four bits, but 6 out of the 16 possible combinations remain unassigned. Different binary codes can be obtained by arranging four bits into 10 distinct combinations. The code most commonly used for the decimal digits is the straight binary assignment listed in Table 1.4. This scheme is called *binary-coded decimal* and is commonly referred to as BCD. Other decimal codes are possible and a few of them are presented later in this section.

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

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

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

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

Table 1.4
Binary-Coded Decimal (BCD)

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

Practice Exercise 1.15

Find the BCD representation of 84_{10} .

Answer: $84_{10} = 1000\ 0100_{\text{BCD}}$

BCD Addition

Consider the addition of two decimal digits in BCD, together with a possible carry from a previous less significant pair of digits. Since each digit does not exceed 9, the sum cannot be greater than $9 + 9 + 1 = 19$, with 1 being a previous carry. Suppose we add the BCD digits as if they were binary numbers. Then the binary sum will produce a result in the range from 0 to 19. In binary, this range will be from 0000 to 10011, but in BCD, it is from 0000 to 11001, with the first (i.e., leftmost) 1 being a carry and the next four bits being the BCD sum. When the binary sum is equal to or less than 1001 (without a carry), the corresponding BCD digit is correct. However, when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit. The addition of $6 = (0110)_2$ to the binary sum converts it to the correct digit and also produces a carry as required. This is because a carry in the most significant bit position of the binary sum and a decimal carry differ by $16 - 10 = 6$. Consider the following three BCD additions:

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

In each case, the two BCD digits are added as if they were two binary numbers. If the binary sum is greater than or equal to 1010, we add 0110 to obtain the correct BCD sum and a carry. In the first example, the sum is equal to 9 and is the correct BCD sum. In the second example, the binary sum produces an invalid BCD digit (1100). The addition of 0110 produces the correct BCD sum, 0010 (i.e., the number 2), and a carry. In the third example, the binary sum (10001) produces a carry. This condition occurs when the sum is greater than or equal to 16. Although the other four bits are less than 1001, the binary sum requires a correction because of the carry. Adding 0110, we obtain the required BCD sum 0111 (i.e., the number 7) and a BCD carry.

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

$$\begin{array}{r}
 \begin{array}{cc} \text{BCD} & \begin{array}{cc} 1 & 1 \end{array} \\ & \begin{array}{cc} 0001 & 1000 \end{array} \\ & +0101 & 0111 \\ \hline \text{Binary sum} & 0111 & 10000 & 1010 \\ \text{Add 6} & & 0110 & 0110 \\ \hline \text{BCD sum} & 0111 & 0110 & 0000 & 760 \end{array}
 \end{array}$$

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

Practice Exercise 1.16

Find the BCD sum of $4 + 6$.

Answer: 10000

Decimal Arithmetic

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

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

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

$$\begin{array}{r} 0 \ 375 \\ +9 \ 760 \\ \hline 0 \ 135 \end{array}$$

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

Practice Exercise 1.17

Find the BCD sum

(a) $370 + (-250)$

Answer: 0120

(b) $250 + (-370)$

Answer: 9880, -120

26 Chapter 1 Digital Systems and Binary Numbers

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

Other Decimal Codes

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

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

Table 1.5
Four Different Binary Codes for the Decimal Digits

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

$2 \times 1 + 4 \times 1 + 2 \times 0 + 1 \times 1 = 7$. Note that some digits can be coded in two possible ways in the 2421 code. For instance, decimal 4 can be assigned to bit combination 0100 or 1010, since both combinations add up to a total weight of 4.

BCD adders add BCD values directly, digit by digit, without converting the numbers to binary. However, it is necessary to add 6 to the result if it is greater than 9. BCD adders require significantly more hardware and no longer have a speed advantage of conventional binary adders [5].

The 2421 and the excess-3 codes are examples of self-complementing codes. Such codes have the property that the 9's complement of a decimal number is obtained directly by changing 1's to 0's and 0's to 1's (i.e., by complementing each bit in the pattern). For example, the codes in Table 1.5 indicate that decimal 395 is represented in the excess-3 code as 0110 1100 1000. Its 9's complement, 604, is represented as 1001 0011 0111, which is obtained simply by complementing each bit of the code for 395 (as with the 1's complement of binary numbers).

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

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

Gray Code

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

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

A typical application of the Gray code is the representation of analog data by a continuous change in the angular position of a shaft. The shaft is partitioned into segments,

Table 1.6
Gray Code

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

and each segment is assigned a number. If adjacent segments are made to correspond with the Gray-code sequence, ambiguity is eliminated between the angle of the shaft and the value encoded by the sensor.

ASCII Character Code

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

The standard binary code for the alphanumeric characters is the American Standard Code for Information Interchange (ASCII), which uses seven bits to code 128 characters, as shown in Table 1.7. The seven bits of the code are designated by b_1 through b_7 , with b_7 being the most significant bit. The letter *A*, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code also contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions. The

Table 1.7
American Standard Code for Information Interchange (ASCII)

| $b_4b_3b_2b_1$ | $b_7b_6b_5$ | | | | | | | |
|----------------|-------------|-----|-----|-----|-----|-----|-----|-----|
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NUL | DLE | SP | 0 | @ | P | ' | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | (| 8 | H | X | h | x |
| 1001 | HT | EM |) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [| k | { |
| 1100 | FF | FS | , | < | L | | l | |
| 1101 | CR | GS | - | = | M |] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | - | o | DEL |

Control Characters

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

graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters, such as %, *, and \$.

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

ASCII is a seven-bit code, but most computers manipulate an eight-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize eight-bit ASCII characters with the most significant bit set to 0. An additional 128 eight-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font.

Error-Detecting Code

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

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

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

The parity bit is helpful in detecting errors during the transmission of information from one location to another. This function is handled by generating an even parity bit at the sending end for each character. The eight-bit characters that include parity bits are transmitted to their destination. The parity of each character is then checked at the receiving end. If the parity of the received character is not even, then at least one bit has changed value during the transmission. This method detects one, three, or any odd

combination of errors in each character that is transmitted. An even combination of errors, however, goes undetected, and additional error detection codes may be needed to take care of that possibility.

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

Practice Exercise 1.18

What is the even parity bit of $A = 0101100$?

Answer: 1

1.8 BINARY STORAGE AND REGISTERS

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

Registers

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

1100001111001001

A register with 16 cells can be in one of 2^{16} possible states. If one assumes that the content of the register represents a binary integer, then the register can store any binary number from 0 to $2^{16} - 1$. For the particular example shown, the content of the register is the binary equivalent of the decimal number 50,121. If one assumes instead that the register stores alphanumeric characters of an eight-bit code, then the content of the register is any two meaningful characters. For the ASCII code with an even parity placed in the eighth most significant bit position, the register contains the two characters C (the leftmost eight bits) and I (the rightmost eight bits). If, however, one interprets the content

of the register to be four decimal digits represented by a four-bit code, then the content of the register is a four-digit decimal number. In the excess-3 code, the register holds the decimal number 9,096. The content of the register is meaningless in BCD, because the bit combination 1100 is not assigned to any decimal digit. From this example, it is clear that a register can store discrete elements of information and that the same bit configuration may be interpreted differently for different types of data depending on the application.

Register Transfer

A digital system is characterized by its registers and the components that perform data processing. In digital systems, a *register transfer* operation is a basic operation that consists of a transfer of binary information from one set of registers into another set of registers. The transfer may be direct, from one register to another, or may pass through data-processing circuits to perform an operation. Figure 1.1 illustrates the transfer of information among registers and demonstrates pictorially the transfer of binary information from a keyboard into a register in the memory unit. The input unit is assumed to have a keyboard, a control circuit, and an input register. Each time a key is struck, the

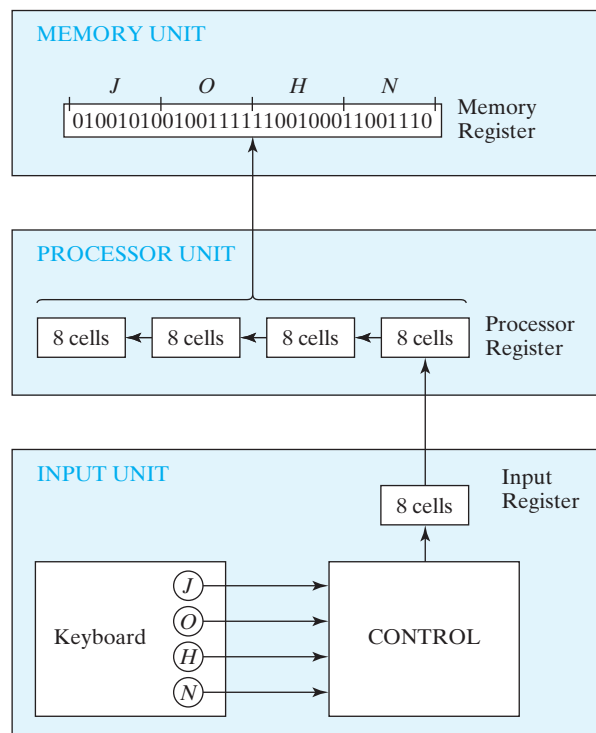


FIGURE 1.1
Transfer of information among registers

control circuit enters an equivalent eight-bit alphanumeric character code into the input register. We shall assume that the code used is the ASCII code with an odd-parity bit. The information from the input register is transferred into the eight least significant cells of a processor register. After every transfer, the input register is cleared to enable the control to insert a new eight-bit code when the keyboard is struck again. Each eight-bit character transferred to the processor register is preceded by a shift of the previous character to the next eight cells on its left. When a transfer of four characters is completed, the processor register is full, and its contents are transferred into a memory register. The content stored in the memory register shown in Fig. 1.1 came from the transfer of the characters “J,” “O,” “H,” and “N” after the four appropriate keys were struck.

To process discrete quantities of information in binary form, a computer must be provided with devices that hold the data to be processed and with circuit elements that manipulate individual bits of information. **The device most commonly used for holding data is a register.** Binary variables are manipulated by means of digital logic circuits. Figure 1.2 illustrates the process of adding two 10-bit binary numbers. The memory unit,

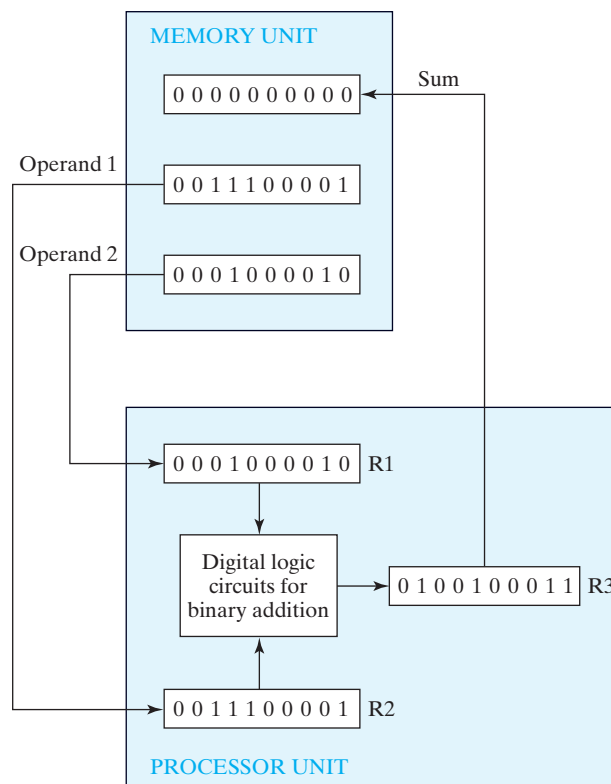


FIGURE 1.2
Example of registers in binary information processing

which normally consists of millions of registers, is shown with only three of its registers. The part of the processor unit shown consists of three registers— $R1$, $R2$, and $R3$ —together with digital logic circuits that manipulate the bits of $R1$ and $R2$ and transfer into $R3$ a binary number equal to their arithmetic sum. Memory registers store information and are incapable of processing the two operands. However, the information stored in memory can be transferred to processor registers, and the results obtained in processor registers can be transferred back into a memory register for storage until needed again. The diagram shows the contents of two operands transferred from two memory registers into $R1$ and $R2$. The digital logic circuits produce the sum, which is transferred to register $R3$. The contents of $R3$ can now be transferred back to one of the memory registers.

The last two examples demonstrated the information-flow capabilities of a digital system in a simple manner. The registers of the system are the basic elements for storing and holding the binary information. Digital logic circuits process the binary information stored in the registers. Digital logic circuits and registers are covered in Chapters 2 through 6. The memory unit is explained in Chapter 7. The description of register operations at the register transfer level and the design of digital systems are covered in Chapter 8.

1.9 BINARY LOGIC

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

Definition of Binary Logic

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

1. **AND:** This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $xy = z$ is read “ x AND y is equal to z .” The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$. (Remember that x , y , and z are binary variables and can be equal either to 1 or 0, and nothing else.) The result of the operation $x \cdot y$ is z .
2. **OR:** This operation is represented by a plus sign. For example, $x + y = z$ is read “ x OR y is equal to z ,” meaning that $z = 1$ if $x = 1$ or if $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

Table 1.8
Truth Tables of Logical Operations

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

3. NOT: This operation is represented by a prime (sometimes by an overbar). For example, $x' = z$ (or $\bar{x} = z$) is read “not x is equal to z ,” meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$, but if $x = 0$, then $z = 1$. The NOT operation is also referred to as the complement operation, since it changes a 1 to 0 and a 0 to 1, that is, the result of complementing 1 is 0, and vice versa.

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

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

Logic Gates

Logic gates are electronic circuits that operate on one or more physical input signals to produce an output signal. Electrical signals such as voltages or currents exist as analog signals having values over a given continuous range, say, 0–3 V, but in a digital system these voltages are interpreted to be either of two recognizable values, 0 or 1. Voltage-operated logic circuits respond to two separate voltage levels that represent a binary variable equal to logic 1 or logic 0. For example, a particular digital system may define logic 0 as a signal equal to 0 V and logic 1 as a signal equal to 3 V. In practice, each voltage level has an acceptable range, as shown in Fig. 1.3. The input terminals

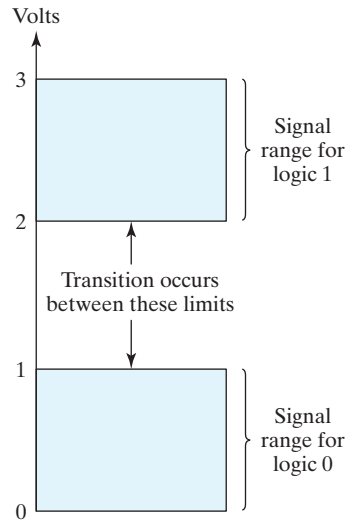


FIGURE 1.3
Signal levels for binary logic values

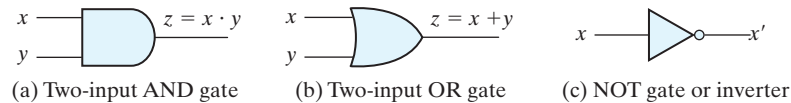


FIGURE 1.4
Symbols for digital logic circuits

of digital circuits accept binary signals within the allowable range and respond at the output terminals with binary signals that fall within the specified range. The intermediate region between the allowed regions is crossed only during a state transition. Any desired information for computing or control can be operated on by passing binary signals through various combinations of logic gates, with each signal representing a particular binary variable. When the physical signal is in a particular range it is interpreted to be either a 0 or a 1.

The graphic symbols used to designate the three types of gates are shown in Fig. 1.4. The gates are blocks of hardware that produce the equivalent of logic-1 or logic-0 output signals if input logic requirements are satisfied. The input signals x and y in the AND and OR gates may exist in one of four possible states: 00, 10, 11, or 01. These input signals are shown in Fig. 1.5 together with the corresponding output signal for each gate. The timing diagrams illustrate the idealized response of each gate to the four input signal combinations. The horizontal axis of the timing diagram represents the time, and the vertical axis shows the signal as it changes between the two possible voltage levels. In reality, the transitions between logic values occur quickly, but not

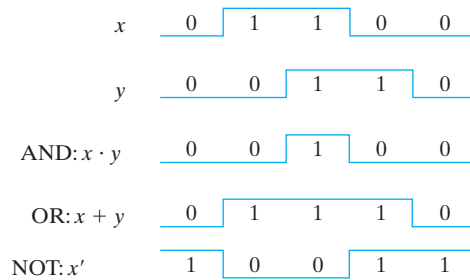


FIGURE 1.5
Input–output signals for gates

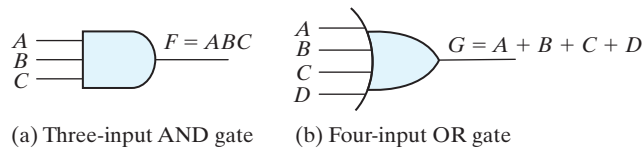


FIGURE 1.6
Gates with multiple inputs

instantaneously. The low level represents logic 0 and the high level logic 1. The AND gate responds with a logic 1 output signal when both input signals are logic 1. The OR gate responds with a logic 1 output signal if any input signal is logic 1. The NOT gate is commonly referred to as an *inverter*. The reason for this name is apparent from the signal response in the timing diagram, which shows that the output signal inverts the logic sense of the input signal.

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

PROBLEMS

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

- 1.1** (a) List the octal and hexadecimal numbers from 14_{10} to 32_{10} . Using A and B for the last two digits, list the numbers from 8_{10} to 28_{10} in base 12.
- 1.2*** What is the exact number of bytes in a system that contains (a) 32K bytes, (b) 64M bytes, and (c) 6.4G bytes?
- 1.3** Convert the following numbers with the indicated bases to decimal:

| | |
|-----------------|-------------------|
| (a)* $(4310)_5$ | (b)* $(198)_{12}$ |
| (c) $(445)_8$ | (d) $(345)_6$ |

38 Chapter 1 Digital Systems and Binary Numbers

- 1.4** What is the largest binary number that can be expressed with 16 bits? What are the equivalent decimal and hexadecimal numbers?
- 1.5*** Determine the base of the numbers in each case for the following operations to be correct:
(a) $14/2 = 5$ (b) $56/4 = 15$ (c) $32 + 12 = 28$.
- 1.6*** The solutions to the quadratic equation $x^2 - 11x + 22 = 0$ are $x = 3$ and $x = 6$. What is the base of the numbers?
- 1.7*** Convert the hexadecimal number 64CD to binary, and then convert it from binary to octal.
- 1.8** Convert the decimal number 431 to binary in two ways: (a) convert directly to binary; (b) convert first to hexadecimal and then from hexadecimal to binary. Which method is faster?
- 1.9** Express the following numbers in decimal:
(a)* $(10110.0101)_2$ (b)* $(16.5)_{16}$
(c)* $(26.24)_8$ (d) $(DABA.B)_{16}$
(e) $(1011.1001)_2$
- 1.10** Convert the following binary numbers to hexadecimal and to decimal: (a) 1.10010 (b) 110.010. Explain why the decimal answer in (b) is four times that in (a).
- 1.11** Perform the following division in binary: $111011 \div 101$.
- 1.12*** Add and multiply the following numbers without converting them to decimal:
(a) Binary numbers 1011 and 101.
(b) Hexadecimal numbers 2E and 34.
- 1.13** Do the following conversion problems:
(a) Convert decimal 27.315 to binary.
(b) Calculate the binary equivalent of $2/3$ out to eight places. Then convert from binary to decimal. How close is the result to $2/3$?
(c) Convert the binary result in (b) into hexadecimal. Then convert the result to decimal. Is the answer the same?
- 1.14** Obtain the 1's and 2's complements of the following binary numbers:
(a) 10010000 (b) 00000000
(c) 11011010 (d) 10101010
(e) 10100101 (f) 11111111.
- 1.15** Find the 9's and the 10's complement of the following decimal numbers:
(a) 25,478,036 (b) 63,325,600
(c) 25,000,000 (d) 00,000,000.
- 1.16** (a) Find the 16's complement of C3AF.
(b) Convert C3AF to binary.
(c) Find the 2's complement of the result in (b).
(d) Convert the answer in (c) to hexadecimal and compare with the answer in (a).
- 1.17** Perform subtraction on the given unsigned numbers using the 10's complement of the subtrahend. Where the result should be negative, find its 10's complement and affix a minus sign. Verify your answers.
(a) $6,473 - 5,297$ (b) $125 - 1,800$
(c) $1,076 - 3,217$ (d) $1,631 - 745$