

**Fundamentals**

**of**

**Digital Logic with Verilog Design THIRD EDITION**

**Stephen Brown and Zvonko Vranesic**

*Department of Electrical and Computer Engineering*

*University of Toronto*

**

**

FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN, THIRD EDITION

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2014 by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1234567890 DOC/DOC109876543

ISBN 978–0–07–338054–4

MHID 0–07–338054–7

Managing Director: *Thomas Timp*

Director: *Michael Lange*

Global Publisher: *Raghothaman Srinivasan*

Developmental Editor: *Vincent Bradshaw*

Marketing Manager: *Curt Reynolds*

Director, Content Production: *Terri Schiesl*

Senior Project Manager: *Melissa M. Leick*

Buyer: *Susan K. Culbertson*

Media Project Manager: *Prashanthi Nadipalli*

Cover Design: *Studio Montage, St. Louis, Missouri*

(USE) Cover Image: *Steven Brown and Zvonko Vranesic*

Compositor: *Techsetters, Inc.*

Typeface: *10/12 Times Roman*

Printer: *R. R. Donnelley, Crawfordsville, IN*

**Library of Congress Cataloging-in-Publication Data**

Brown, Stephen.

Fundamentals of digital logic with Verilog design / Stephen Brown and Zvonko Vranesic. — Third edition. pages cm

ISBN 978–0–07–338054–4 (alk. paper)

1. Logic circuits—Design and construction—Data processing. 2. Verilog (Computer hardware description language). 3. Computer-aided design. I. Vranesic, Zvonko G. II. Title.

TK7868.L6B76 2014

621.392–dc23 2012042163 www.mhhe.com

***To Susan and Anne***

*This page intentionally left blank*

**About the Authors**

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

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

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

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

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

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

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

**v**

**Preface**

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

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

**Technology**

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

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

**vi**

**Preface vii**

**Scope of the Book**

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

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

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

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

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

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

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

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

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

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

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

**viii Preface**

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

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

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

**What Can Be Covered in a Course**

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

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

**One-Semester Course**

The following material should be covered in lectures:

• Chapter 1—all sections.

• Chapter 2—all sections.

• Chapter 3—Sections 3.1 to 3.5.

• Chapter 4—all sections.

• Chapter 5—all sections.

• Chapter 6—all sections.

**One-Quarter Course**

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

• Chapter 1—all sections.

• Chapter 2—all sections.

**Preface ix**

• Chapter 3—Sections 3.1 to 3.3 and Section 3.5.

• Chapter 4—all sections.

• Chapter 5—all sections.

• Chapter 6—Sections 6.1 to 6.4.

**Verilog**

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

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

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

www.eecg.toronto.edu/∼brown/Verilog\_3e

**Solved Problems**

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

**Homework Problems**

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

**PowerPoint Slides and Solutions Manual**

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

www.mhhe.com/brownvranesic

**x Preface**

**CAD Tools**

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

**Acknowledgments**

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

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

Stephen Brown and Zvonko Vranesic

**Contents**

**Chapter 1**

**Introduction 1**

2.8.2 Multiplexer Circuit 60 2.8.3 Number Display 63

1.1 Digital Hardware 2

1.1.1 Standard Chips 4

1.1.2 Programmable Logic Devices 5

1.1.3 Custom-Designed Chips 5

1.2 The Design Process 6

1.3 Structure of a Computer 8

1.4 Logic Circuit Design in This Book 8 1.5 Digital Representation of Information 11 1.5.1 Binary Numbers 12

1.5.2 Conversion between Decimal and

Binary Systems 13

1.5.3 ASCII Character Code 14

1.5.4 Digital and Analog Information 16 1.6 Theory and Practice 16

Problems 18

References 19

**Chapter 2**

**Introduction to Logic**

**Circuits 21**

2.1 Variables and Functions 22

2.2 Inversion 25

2.3 Truth Tables 26

2.4 Logic Gates and Networks 27

2.4.1 Analysis of a Logic Network 29

2.5 Boolean Algebra 33

2.5.1 The Venn Diagram 37

2.5.2 Notation and Terminology 42

2.5.3 Precedence of Operations 43

2.6 Synthesis Using AND, OR, and NOT Gates 43

2.6.1 Sum-of-Products and Product-of-Sums Forms 48

2.7 NAND and NOR Logic Networks 54 2.8 Design Examples 59

2.8.1 Three-Way Light Control 59

**xi**

2.9 Introduction to CAD Tools 64

2.9.1 Design Entry 64

2.9.2 Logic Synthesis 66

2.9.3 Functional Simulation 67

2.9.4 Physical Design 67

2.9.5 Timing Simulation 67

2.9.6 Circuit Implementation 68

2.9.7 Complete Design Flow 68

2.10 Introduction to Verilog 68

2.10.1 Structural Specification of Logic

Circuits 70

2.10.2 Behavioral Specification of Logic Circuits 72

2.10.3 Hierarchical Verilog Code 76

2.10.4 How *NOT* to Write Verilog Code 78 2.11 Minimization and Karnaugh Maps 78 2.12 Strategy for Minimization 87

2.12.1 Terminology 87

2.12.2 Minimization Procedure 89

2.13 Minimization of Product-of-Sums Forms 91 2.14 Incompletely Specified Functions 94 2.15 Multiple-Output Circuits 96

2.16 Concluding Remarks 101

2.17 Examples of Solved Problems 101 Problems 111

References 120

**Chapter 3**

**Number Representation and Arithmetic Circuits 121**

3.1 Positional Number Representation 122 3.1.1 Unsigned Integers 122

3.1.2 Octal and Hexadecimal

Representations 123

3.2 Addition of Unsigned Numbers 125 3.2.1 Decomposed Full-Adder 129

3.2.2 Ripple-Carry Adder 129

3.2.3 Design Example 130

**xii Contents**

3.3 Signed Numbers 132

3.3.1 Negative Numbers 133

3.3.2 Addition and Subtraction 135

3.3.3 Adder and Subtractor Unit 138

3.3.4 Radix-Complement Schemes∗ 139 3.3.5 Arithmetic Overflow 143

3.3.6 Performance Issues 145

3.4 Fast Adders 145

3.4.1 Carry-Lookahead Adder 146

3.5 Design of Arithmetic Circuits Using CAD Tools 151

3.5.1 Design of Arithmetic Circuits Using Schematic Capture 151

3.5.2 Design of Arithmetic Circuits Using Verilog 152

3.5.3 Using Vectored Signals 155

3.5.4 Using a Generic Specification 156 3.5.5 Nets and Variables in Verilog 158 3.5.6 Arithmetic Assignment Statements 159 3.5.7 Module Hierarchy in Verilog Code 163 3.5.8 Representation of Numbers in Verilog Code 166

3.6 Multiplication 167

3.6.1 Array Multiplier for Unsigned

Numbers 167

3.6.2 Multiplication of Signed Numbers 169 3.7 Other Number Representations 170 3.7.1 Fixed-Point Numbers 170

3.7.2 Floating-Point Numbers 172

3.7.3 Binary-Coded-Decimal

Representation 174

3.8 Examples of Solved Problems 178 Problems 184

References 188

**Chapter 4**

**Combinational-Circuit**

**Building Blocks 189**

4.1 Multiplexers 190

4.1.1 Synthesis of Logic Functions Using Multiplexers 193

4.1.2 Multiplexer Synthesis Using Shannon’s Expansion 196

4.2 Decoders 201

4.2.1 Demultiplexers 203

4.3 Encoders 205

4.3.1 Binary Encoders 205

4.3.2 Priority Encoders 205

4.4 Code Converters 208

4.5 Arithmetic Comparison Circuits 208 4.6 Verilog for Combinational Circuits 210 4.6.1 The Conditional Operator 210

4.6.2 The If-Else Statement 212

4.6.3 The Case Statement 215

4.6.4 The For Loop 221

4.6.5 Verilog Operators 223

4.6.6 The Generate Construct 228

4.6.7 Tasks and Functions 229

4.7 Concluding Remarks 232

4.8 Examples of Solved Problems 233 Problems 243

References 246

**Chapter 5**

**Flip-Flops, Registers, and Counters 247**

5.1 Basic Latch 249

5.2 Gated SR Latch 251

5.2.1 Gated SR Latch with NAND Gates 253 5.3 Gated D Latch 253

5.3.1 Effects of Propagation Delays 255 5.4 Edge-Triggered D Flip-Flops 256 5.4.1 Master-Slave D Flip-Flop 256

5.4.2 Other Types of Edge-Triggered D Flip-Flops 258

5.4.3 D Flip-Flops with Clear and Preset 260 5.4.4 Flip-Flop Timing Parameters 263 5.5 T Flip-Flop 263

5.6 JK Flip-Flop 264

5.7 Summary of Terminology 266

5.8 Registers 267

5.8.1 Shift Register 267

5.8.2 Parallel-Access Shift Register 267 5.9 Counters 269

5.9.1 Asynchronous Counters 269

5.9.2 Synchronous Counters 272

5.9.3 Counters with Parallel Load 276 5.10 Reset Synchronization 278

5.11 Other Types of Counters 280

5.11.1 BCD Counter 280

5.11.2 Ring Counter 280

5.11.3 Johnson Counter 283

5.11.4 Remarks on Counter Design 283 5.12 Using Storage Elements with CAD Tools 284 5.12.1 Including Storage Elements in

Schematics 284

5.12.2 Using Verilog Constructs for Storage Elements 285

5.12.3 Blocking and Non-Blocking

Assignments 288

5.12.4 Non-Blocking Assignments for

Combinational Circuits 293

5.12.5 Flip-Flops with Clear Capability 293 5.13 Using Verilog Constructs for Registers and Counters 295

5.13.1 Flip-Flops and Registers with Enable Inputs 300

5.13.2 Shift Registers with Enable Inputs 302 5.14 Design Example 302

5.14.1 Reaction Timer 302

5.14.2 Register Transfer Level (RTL) Code 309 5.15 Timing Analysis of Flip-flop Circuits 310 5.15.1 Timing Analysis with Clock Skew 312 5.16 Concluding Remarks 314

5.17 Examples of Solved Problems 315 Problems 321

References 329

**Chapter 6**

**Synchronous Sequential**

**Circuits 331**

6.1 Basic Design Steps 333

6.1.1 State Diagram 333

6.1.2 State Table 335

6.1.3 State Assignment 336

6.1.4 Choice of Flip-Flops and Derivation of Next-State and Output Expressions 337

6.1.5 Timing Diagram 339

6.1.6 Summary of Design Steps 340

6.2 State-Assignment Problem 344

6.2.1 One-Hot Encoding 347

6.3 Mealy State Model 349

6.4 Design of Finite State Machines Using CAD Tools 354

6.4.1 Verilog Code for Moore-Type FSMs 355 6.4.2 Synthesis of Verilog Code 356

6.4.3 Simulating and Testing the Circuit 358

**Contents xiii**

6.4.4 Alternative Styles of Verilog Code 359 6.4.5 Summary of Design Steps When Using CAD Tools 360

6.4.6 Specifying the State Assignment in Verilog Code 361

6.4.7 Specification of Mealy FSMs Using Verilog 363

6.5 Serial Adder Example 363

6.5.1 Mealy-Type FSM for Serial Adder 364 6.5.2 Moore-Type FSM for Serial Adder 367 6.5.3 Verilog Code for the Serial Adder 370 6.6 State Minimization 372

6.6.1 Partitioning Minimization

Procedure 374

6.6.2 Incompletely Specified FSMs 381 6.7 Design of a Counter Using the Sequential Circuit Approach 383

6.7.1 State Diagram and State Table for a Modulo-8 Counter 383

6.7.2 State Assignment 384

6.7.3 Implementation Using D-Type

Flip-Flops 385

6.7.4 Implementation Using JK-Type

Flip-Flops 386

6.7.5 Example—A Different Counter 390 6.8 FSM as an Arbiter Circuit 393

6.9 Analysis of Synchronous Sequential Circuits 397

6.10 Algorithmic State Machine (ASM) Charts 401

6.11 Formal Model for Sequential Circuits 405 6.12 Concluding Remarks 407

6.13 Examples of Solved Problems 407 Problems 416

References 420

**Chapter 7**

**Digital System Design 421**

7.1 Bus Structure 422

7.1.1 Using Tri-State Drivers to Implement a Bus 422

7.1.2 Using Multiplexers to Implement a Bus 424

7.1.3 Verilog Code for Specification of Bus Structures 426

7.2 Simple Processor 429

**xiv Contents**

7.3 A Bit-Counting Circuit 441

7.4 Shift-and-Add Multiplier 446

7.5 Divider 455

7.6 Arithmetic Mean 466

7.7 Sort Operation 470

7.8 Clock Synchronization and Timing Issues 478

7.8.1 Clock Distribution 478

7.8.2 Flip-Flop Timing Parameters 481 7.8.3 Asynchronous Inputs to Flip-Flops 482 7.8.4 Switch Debouncing 483

7.9 Concluding Remarks 485

Problems 485

References 489

**Chapter 8**

**Optimized Implementation of Logic Functions 491**

8.1 Multilevel Synthesis 492

8.1.1 Factoring 493

8.1.2 Functional Decomposition 496

8.1.3 Multilevel NAND and NOR

Circuits 502

8.2 Analysis of Multilevel Circuits 504 8.3 Alternative Representations of Logic Functions 510

8.3.1 Cubical Representation 510

8.3.2 Binary Decision Diagrams 514

8.4 Optimization Techniques Based on Cubical Representation 520

8.4.1 A Tabular Method for Minimization 521 8.4.2 A Cubical Technique for

Minimization 529

8.4.3 Practical Considerations 536

8.5 Concluding Remarks 537

8.6 Examples of Solved Problems 537 Problems 546

References 549

**Chapter 9**

**Asynchronous Sequential Circuits 551**

9.1 Asynchronous Behavior 552

9.2 Analysis of Asynchronous Circuits 556

9.3 Synthesis of Asynchronous Circuits 564 9.4 State Reduction 577

9.5 State Assignment 592

9.5.1 Transition Diagram 595

9.5.2 Exploiting Unspecified Next-State Entries 598

9.5.3 State Assignment Using Additional State Variables 602

9.5.4 One-Hot State Assignment 607 9.6 Hazards 608

9.6.1 Static Hazards 609

9.6.2 Dynamic Hazards 613

9.6.3 Significance of Hazards 614

9.7 A Complete Design Example 616 9.7.1 The Vending-Machine Controller 616 9.8 Concluding Remarks 621

9.9 Examples of Solved Problems 623 Problems 631

References 635

**C h a p t e r 10**

**Computer Aided Design**

**Tools 637**

10.1 Synthesis 638

10.1.1 Netlist Generation 638

10.1.2 Gate Optimization 638

10.1.3 Technology Mapping 640

10.2 Physical Design 644

10.2.1 Placement 646

10.2.2 Routing 647

10.2.3 Static Timing Analysis 648

10.3 Concluding Remarks 650

References 651

**C h a p t e r 11**

**Testing of Logic Circuits 653**

11.1 Fault Model 654

11.1.1 Stuck-at Model 654

11.1.2 Single and Multiple Faults 655 11.1.3 CMOS Circuits 655

11.2 Complexity of a Test Set 655

11.3 Path Sensitizing 657

11.3.1 Detection of a Specific Fault 659 11.4 Circuits with Tree Structure 661 11.5 Random Tests 662

11.6 Testing of Sequential Circuits 665 11.6.1 Design for Testability 665

11.7 Built-in Self-Test 669

11.7.1 Built-in Logic Block Observer 673 11.7.2 Signature Analysis 675

11.7.3 Boundary Scan 676

11.8 Printed Circuit Boards 676

11.8.1 Testing of PCBs 678

11.8.2 Instrumentation 679

11.9 Concluding Remarks 680

Problems 680

References 683

**Appendix A**

**Verilog Reference 685**

A.1 Documentation in Verilog Code 686 A.2 White Space 686

A.3 Signals in Verilog Code 686

A.4 Identifier Names 687

A.5 Signal Values, Numbers, and Parameters 687 A.5.1 Parameters 688

A.6 Net and Variable Types 688

A.6.1 Nets 688

A.6.2 Variables 689

A.6.3 Memories 690

A.7 Operators 690

A.8 Verilog Module 692

A.9 Gate Instantiations 694

A.10 Concurrent Statements 696

A.10.1 Continuous Assignments 696

A.10.2 Using Parameters 697

A.11 Procedural Statements 698

A.11.1 Always and Initial Blocks 698

A.11.2 The If-Else Statement 700

A.11.3 Statement Ordering 701

A.11.4 The Case Statement 702

A.11.5 Casez and Casex Statements 703 A.11.6 Loop Statements 704

A.11.7 Blocking versus Non-blocking

Assignments for Combinational

Circuits 708

A.12 Using Subcircuits 709

A.12.1 Subcircuit Parameters 710

A.12.2 The Generate Capability 712

A.13 Functions and Tasks 713

**Contents xv**

A.14 Sequential Circuits 716

A.14.1 A Gated D Latch 717

A.14.2 D Flip-Flop 717

A.14.3 Flip-Flops with Reset 718

A.14.4 Registers 718

A.14.5 Shift Registers 720

A.14.6 Counters 721

A.14.7 An Example of a Sequential Circuit 722 A.14.8 Moore-Type Finite State Machines 723 A.14.9 Mealy-Type Finite State Machines 724

A.15 Guidelines for Writing Verilog Code 725 A.16 Concluding Remarks 731

References 731

**Appendix B**

**Implementation**

**Technology 733**

B.1 Transistor Switches 734

B.2 NMOS Logic Gates 736

B.3 CMOS Logic Gates 739

B.3.1 Speed of Logic Gate Circuits 746 B.4 Negative Logic System 747

B.5 Standard Chips 749

B.5.1 7400-Series Standard Chips 749 B.6 Programmable Logic Devices 753 B.6.1 Programmable Logic Array (PLA) 754 B.6.2 Programmable Array Logic (PAL) 757 B.6.3 Programming of PLAs and PALs 759 B.6.4 Complex Programmable Logic Devices (CPLDs) 761

B.6.5 Field-Programmable Gate Arrays 764 B.7 Custom Chips, Standard Cells, and Gate Arrays 769

B.8 Practical Aspects 771

B.8.1 MOSFET Fabrication and Behavior 771 B.8.2 MOSFET On-Resistance 775

B.8.3 Voltage Levels in Logic Gates 776 B.8.4 Noise Margin 778

B.8.5 Dynamic Operation of Logic Gates 779 B.8.6 Power Dissipation in Logic Gates 782 B.8.7 Passing 1s and 0s Through Transistor Switches 784

B.8.8 Transmission Gates 786

B.8.9 Fan-in and Fan-out in Logic Gates 788 B.8.10 Tri-state Drivers 792

**xvi Contents**

B.9 Static Random Access Memory (SRAM) 794 B.9.1 SRAM Blocks in PLDs 797

B.10 Implementation Details for SPLDs, CPLDs, and FPGAs 797

B.10.1 Implementation in FPGAs 804

B.11 Concluding Remarks 806

B.12 Examples of Solved Problems 807 Problems 814

References 823

**Answers 825**

**Index 839**

**chapter**

**1**

**Introduction**

**Chapter Objectives**

In this chapter you will be introduced to: • Digital hardware components • An overview of the design process • Binary numbers

• Digital representation of information**1**

**2 CHAPTER 1** • **Introduction**

**T**his book is about logic circuits—the circuits from which computers are built. Proper understanding of logic circuits is vital for today’s electrical and computer engineers. These circuits are the key ingredient of computers and are also used in many other applications. They are found in commonly-used products like music and video players, electronic games, digital watches, cameras, televisions, printers, and many household appliances, as well as in large systems, such as telephone networks, Internet equipment, television broadcast equipment, industrial control units, and medical instruments. In short, logic circuits are an important part of almost all modern products.

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

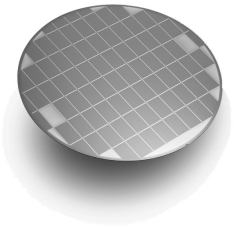
Logic circuits are implemented electronically, using transistors on an integrated circuit chip. Commonly available chips that use modern technology may contain more than a billion transistors, as in the case of some computer processors. The basic building blocks for such circuits are easy to understand,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

**1.1 Digital Hardware**

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

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

**1.1 Digital Hardware 3**

****

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

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

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

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

**4 CHAPTER 1** • **Introduction**

Millions of transistors/chip

105

104

103

102

10

20001995 2005 2010 2015 20252020 Year of production

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

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

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

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

**1.1.1 Standard Chips**

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

**1.1 Digital Hardware 5**

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

**1.1.2 Programmable Logic Devices**

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

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

**1.1.3 Custom-Designed Chips**

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

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

**6 CHAPTER 1** • **Introduction**

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

**1.2 The Design Process**

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

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

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

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

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

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

**1.2 The Design Process 7**

Required product

Define specifications

Initial design

Simulation

Design correct?

Yes

Prototype implementation Testing

Meets specifications? Yes

Finished product

No No

Redesign

Make corrections

Yes

Minor errors?

No

**Figure 1.3** The development process.

**8 CHAPTER 1** • **Introduction**

**1.3 Structure of a Computer**

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

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

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

**1.4 Logic Circuit Design in This Book**

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

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

Computer

Integrated circuits, connectors, and components

**1.4 Logic Circuit Design in This Book 9** Power supply

Motherboard

Printed circuit boards

Motherboard

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

**10 CHAPTER 1** • **Introduction**

Subcircuits

in a chip

Transistor circuit

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

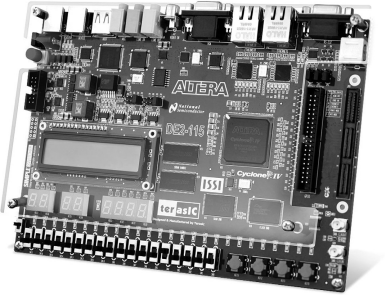
Logic gates

Transistor on a chip

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

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

**1.5 Digital Representation of Information 11**

**Figure 1.5** An FPGA board.

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

**1.5 Digital Representation of Information**

In Section 1.1 we mentioned that information is represented in logic circuits as electronic signals. Each of these signals can be thought of as representing one *digit* of information. To make the design of logic circuits easier, each digit is allowed to take on only two possible values, usually denoted as 0 and 1. These logic values are implemented as voltage levels in a circuit; the value 0 is usually represented as 0 V (ground), and the value 1 is the voltage

**12 CHAPTER 1** • **Introduction**

level of the circuit’s power supply. As we discuss in Appendix B, typical power-supply voltages in logic circuits range from 1 V DC to 5 V DC.

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

**1.5.1 Binary Numbers**

In the familiar decimal system, a number consists of digits that have 10 possible values, from 0 to 9, and each digit represents a multiple of a power of 10. For example, the number 8547 represents 8 × 103 + 5 × 102 + 4 × 101 + 7 × 100. We do not normally write the powers of 10 with the number, because they are implied by the positions of the digits. In general, a decimal integer is expressed by an *n*-tuple comprising *n* decimal digits

*D* = *dn*−1*dn*−2 ··· *d*1*d*0

which represents the value

*V(D)* = *dn*−1 × 10*n*−1 + *dn*−2 × 10*n*−2 +···+ *d*1 × 101 + *d*0 × 100

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

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

*B* = *bn*−1*bn*−2 ··· *b*1*b*0

represents an integer that has the value

*V(B)* = *bn*−1 × 2*n*−1 + *bn*−2 × 2*n*−2 +···+ *b*1 × 21 + *b*0 × 20 **[1.1]**

*n*−1

=

*i*=0

*bi* × 2*i*

For example, the binary number 1101 represents the value

*V* = 1 × 23 + 1 × 22 + 0 × 21 + 1 × 20

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

*(*1101*)*2 = *(*13*)*10

**1.5 Digital Representation of Information 13**

**Table 1.1** Numbers in decimal

and binary.

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

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

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

**1.5.2 Conversion between Decimal and Binary Systems**

A binary number is converted into a decimal number simply by applying Equation 1.1 and evaluating it using decimal arithmetic. Converting a decimal number into a binary number is not quite as straightforward, because we need to construct the number by using powers of 2. For example, the number *(*17*)*10 is 24 + 20 = *(*10001*)*2, and the number *(*50*)*10 is 25 + 24 + 21 = *(*110010*)*2. In general, the conversion can be performed by successively dividing the decimal number by 2 as follows. Suppose that a decimal number *D* = *dk*−1 ··· *d*1*d*0, with a value *V*, is to be converted into a binary number *B* = *bn*−1 ··· *b*2*b*1*b*0. Then, we can write *V* in the form

*V* = *bn*−1 × 2*n*−1 +···+ *b*2 × 22 + *b*1 × 21 + *b*0

**14 CHAPTER 1** • **Introduction**

Convert (857)10

Remainder

857 ⎟ 2 = 428 1 LSB

428 ⎟ 2 = 214 0

214 ⎟ 2 = 107 0

107 ⎟ 2 = 53 1

53 ⎟ 2 = 26 1

26 ⎟ 2 = 13 0

13 ⎟ 2=6 1

6 ⎟ 2=3 0

3 ⎟ 2=1 1

1 ⎟ 2 = 0 1 MSB

Result is (1101011001)2

**Figure 1.6** Conversion from decimal to binary.

If we now divide *V* by 2, the result is

*V*

2 = *bn*−1 × 2*n*−2 +···+ *b*2 × 21 + *b*1 +*b*02

The quotient of this integer division is *bn*−1 × 2*n*−2 +···+ *b*2 × 2 + *b*1, and the remainder is *b*0. If the remainder is 0, then *b*0 = 0; if it is 1, then *b*0 = 1. Observe that the quotient is just another binary number, which comprises *n* − 1 bits, rather than *n* bits. Dividing this number by 2 yields the remainder *b*1. The new quotient is

*bn*−1 × 2*n*−3 +···+ *b*2

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

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

**1.5.3 ASCII Character Code**

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

**1.5 Digital Representation of Information 15**

**Table 1.2** The seven-bit ASCII code.

**Bit**

**positions**

**Bit positions 654**

**3210** 000 001 010 011 100 101 110 111 0000 NUL DLE SPACE 0 @ P ´ p 0001 SOH DC1 ! 1 A Q a q 0010 STX DC2 ” 2 B R b r 0011 ETX DC3 # 3 C S c s 0100 EOT DC4 $ 4 D T d t 0101 ENQ NAK % 5 E U e u 0110 ACK SYN & 6 F V f v 0111 BEL ETB ’ 7 G W g w 1000 BS CAN ( 8 H X h x 1001 HT EM ) 9 I Y i y 1010 LF SUB \* : J Z j z 1011 VT ESC + ; K [ k { 1100 FF FS , *<* L \ 1 | 1101 CR GS - = M ] m } 1110 SO RS . *>* Nˆn ˜

1111 SI US / ? O — o DEL NUL Null/Idle SI Shift in SOH Start of header DLE Data link escape STX Start of text DC1-DC4 Device control ETX End of text NAK Negative acknowledgement EOT End of transmission SYN Synchronous idle ENQ Enquiry ETB End of transmitted block ACQ Acknowledgement CAN Cancel (error in data) BEL Audible signal EM End of medium BS Back space SUB Special sequence HT Horizontal tab ESC Escape LF Line feed FS File separator VT Vertical tab GS Group separator FF Form feed RS Record separator CR Carriage return US Unit separator SO Shift out DEL Delete/Idle

Bit positions of code format =

|  | 6 5 4 3 2 |  |  |  | 1 | 0 |
| --- | --- | --- | --- | --- | --- | --- |

**16 CHAPTER 1** • **Introduction**

The ASCII code uses seven-bit patterns to denote 128 different characters. Ten of the characters are decimal digits 0 to 9. As the table shows, the high-order bits have the same pattern, *b*6*b*5*b*4 = 011, for all 10 digits. Each digit is identified by the low-order four bits, *b*3−0, using the binary patterns for these digits. Capital and lowercase letters are encoded in a way that makes sorting of textual information easy. The codes for A to Z are in ascending numerical sequence, which means that the task of sorting letters (or words) can be accomplished by a simple arithmetic comparison of the codes that represent the letters.

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

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

The ASCII standard uses seven bits to encode a character. In computer systems a more natural size is eight bits, or one byte. There are two common ways of fitting an ASCII encoded character into a byte. One is to set the eighth bit, *b*7, to 0. Another is to use this bit to indicate the *parity* of the other seven bits, which means showing whether the number of 1s in the seven-bit code is even or odd. We discuss parity in Chapter 4.

**1.5.4 Digital and Analog Information**

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

**1.6 Theory and Practice**

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

Headphones

|  |
| --- |

|  |  |  |
| --- | --- | --- |

| D/A |
| --- |

**1.6 Theory and Practice 17**

Memory

11000100110

10010001000

11111000101

00101001010

11001001011 **...**

**Figure 1.7** Using digital technology to represent music.

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

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

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

**18 CHAPTER 1** • **Introduction**

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

**Problems**

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

**\*1.1** Convert the following decimal numbers into binary, using the method shown in Figure 1.6. (a) *(*20*)*10

(b) *(*100*)*10

(c) *(*129*)*10

(d) *(*260*)*10

(e) *(*10240*)*10

**1.2** Convert the following decimal numbers into binary, using the method shown in Figure 1.6. (a) *(*30*)*10

(b) *(*110*)*10

(c) *(*259*)*10

(d) *(*500*)*10

(e) *(*20480*)*10

**1.3** Convert the following decimal numbers into binary, using the method shown in Figure 1.6. (a) *(*1000*)*10

(b) *(*10000*)*10

(c) *(*100000*)*10

(c) *(*1000000*)*10

**\*1.4** In Figure 1.6 we show how to convert a decimal number into binary by successively dividing by 2. Another way to derive the answer is to constuct the number by using powers of 2. For example, if we wish to convert the number *(*23*)*10, then the largest power of 2 that is not larger than 23 is 24 = 16. Hence, the binary number will have five bits and the most significant bit is *b*4 = 1. We then perform the subtraction 23 − 16 = 7. Now, the largest power of 2 that is not larger than 7 is 22 = 4. Hence, *b*3 = 0 (because 23 = 8 is larger than 7) and *b*2 = 1. Continuing this process gives

23 = 16 + 4 + 2 + 1

= 24 + 22 + 21 + 20

= 10000 + 00100 + 00010 + 00001

= 10111

**References 19**

Using this method, convert the following decimal numbers into binary.

(a) *(*17*)*10

(b) *(*33*)*10

(c) *(*67*)*10

(d) *(*130*)*10

(e) *(*2560*)*10

(f) *(*51200*)*10

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

**\*1.6** Convert the following binary numbers into decimal.

(a) *(*1001*)*2

(b) *(*11100*)*2

(c) *(*111111*)*2

(d) *(*101010101010*)*2

**1.7** Convert the following binary numbers into decimal.

(a) *(*110010*)*2

(b) *(*1100100*)*2

(c) *(*11001000*)*2

(d) *(*110010000*)*2

**\*1.8** What is the minimum number of bits needed to represent the following decimal numbers in binary?

(a) *(*270*)*10

(b) *(*520*)*10

(c) *(*780*)*10

(d) *(*1029*)*10

**1.9** Repeat Problem 8 for the following decimal numbers:

(a) *(*111*)*10

(b) *(*333*)*10

(c) *(*555*)*10

(d) *(*1111*)*10

**References**

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

3. Xilinx Corporation, “Xilinx Field Programmable Gate Arrays Product Literature,” http://www.xilinx.com

*This page intentionally left blank*

**chapter**

**2**

**Introduction to Logic Circuits**

**Chapter Objectives**

In this chapter you will be introduced to:

• Logic functions and circuits

• Boolean algebra for dealing with logic functions

• Logic gates and synthesis of simple circuits

• CAD tools and the Verilog hardware description language

• Minimization of functions and Karnaugh maps

**21**

**22 CHAPTER 2** • **Introduction to Logic Circuits**

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

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

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

**2.1 Variables and Functions**

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

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

*x* 0= *x* 1=

(a) Two states of a switch

S

*x*

(b) Symbol for a switch

**Figure 2.1** A binary switch.

**2.1 Variables and Functions 23** S

Battery Light *x*

(a) Simple connection to a battery S

*x* Power supply

Light

(b) Using a ground connection as the return path

**Figure 2.2** A light controlled by a switch.

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

*L(x)* = *x*

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

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

Consider now the possibility of using two switches to control the state of the light. Let *x*1 and *x*2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches are closed. If either switch is open, the light will be off.

**24 CHAPTER 2** • **Introduction to Logic Circuits**

Power supply

S

*x*1

S

*x*2

Light

(a) The logical AND function (series connection)

S

*x*1

Power

supply S *x*2

Light

(b) The logical OR function (parallel connection)

**Figure 2.3** Two basic functions.

This behavior can be described by the expression

*L(x*1*, x*2*)* = *x*1 · *x*2

where *L* = 1 if *x*1 = 1 and *x*2 = 1*,*

*L* = 0 otherwise*.*

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

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either the *x*1 or *x*2 switch is closed. The light will also be on if both switches are closed. The light will be off only if both switches are open. This behavior can be stated as

*L(x*1*, x*2*)* = *x*1 + *x*2

where *L* = 1 if *x*1 = 1 or *x*2 = 1 or if *x*1 = *x*2 = 1*,*

*L* = 0 if *x*1 = *x*2 = 0*.*

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

In the above expressions for AND and OR, the output *L(x*1*, x*2*)* is a logic function with input variables *x*1 and *x*2. The AND and OR functions are two of the most important logic functions. Together with some other simple functions, they can be used as building blocks for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be

**2.2 Inversion 25**

S

*x*1

Power

supply S *x*2

S

*x*3

Light

**Figure 2.4** A series-parallel connection.

R

Power

supply

*x* S Light

**Figure 2.5** An inverting circuit.

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

*L(x*1*, x*2*, x*3*)* = *(x*1 + *x*2*)* · *x*3

The light is on if *x*3 = 1 and, at the same time, at least one of the *x*1 or *x*2 inputs is equal to 1.

**2.2 Inversion**

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

*L(x)* = *~~x~~*

where *L* = 1 if *x* = 0*,*

*L* = 0 if *x* = 1

**26 CHAPTER 2** • **Introduction to Logic Circuits**

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

*~~x~~* = *x*= !*x* = ∼*x* = NOT *x*

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

*f (x*1*, x*2*)* = *x*1 + *x*2

then the complement of *f* is

*f (x*1*, x*2*)* = *~~x~~*1 ~~+~~ *~~x~~*2

This expression yields the logic value 1 only when neither *x*1 nor *x*2 is equal to 1, that is, when *x*1 = *x*2 = 0. Again, the following notations are equivalent:

*~~x~~*1 ~~+~~ *~~x~~*2 = *(x*1 + *x*2*)*= !*(x*1 + *x*2*)* = ∼*(x*1 + *x*2*)* = NOT *(x*1 + *x*2*)*

**2.3 Truth Tables**

We have introduced the three most basic logic operations—AND, OR, and complement—by relating them to simple circuits built with switches. This approach gives these operations a certain “physical meaning.” The same operations can also be defined in the form of a table, called a *truth table*, as shown in Figure 2.6. The first two columns (to the left of the double vertical line) give all four possible combinations of logic values that the variables *x*1 and *x*2

| *x*1 *x*2 · |
| --- |
| 0  0  0  1 |

*x*1 *x*2 *x*1 + *x*2

0 0 0

0 1 1

1 0 1

1 1 1

AND OR

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

**2.4 Logic Gates and Networks 27**

| *x*1 *x*2 *x*3 · · |
| --- |
| 0  0  0  0  0  0  0  1 |

*x*1 *x*2 *x*3 *x*1 + *x*2 + *x*3

000 0

001 1

010 1

011 1

100 1

101 1

110 1

111 1

**Figure 2.7** Three-input AND and OR operations.

can have. The next column defines the AND operation for each combination of values of *x*1 and *x*2, and the last column defines the OR operation. Because we will frequently need to refer to “combinations of logic values” applied to some variables, we will adopt a shorter term, *valuation*, to denote such a combination of logic values.

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

The AND and OR operations can be extended to *n* variables. An AND function of variables *x*1*, x*2*,..., xn* has the value 1 only if all *n* variables are equal to 1. An OR function of variables *x*1*, x*2*,..., xn* has the value 1 if one or more of the variables is equal to 1.

**2.4 Logic Gates and Networks**

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

**28 CHAPTER 2** • **Introduction to Logic Circuits** *x*1

*x*1

*x*2

*x*2*x*1 *x x*1 *x*2 … *xn* ⋅ ⋅ · ·

*xn*

(a) AND gates

*x*1

*x*2

*x*1 *x*2 … *xn* +++ *x*1 *x*2*x*1 *x*2 +

*xn*

(b) OR gates

*x x*

(c) NOT gate

**Figure 2.8** The basic gates.

*x x*

1

2

*x*3*f x*1 *x*2 ( ) + *x*3 = ⋅

**Figure 2.9** The function from Figure 2.4.

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

A larger circuit is implemented by a *network* of gates. For example, the logic function from Figure 2.4 can be implemented by the network in Figure 2.9. The complexity of a given network has a direct impact on its cost. Because it is always desirable to reduce the cost of any manufactured product, it is important to find ways for implementing logic circuits as inexpensively as possible. We will see shortly that a given logic function can

**2.4 Logic Gates and Networks 29**

be implemented with a number of different networks. Some of these networks are simpler than others, hence searching for the solutions that entail minimum cost is prudent. In technical jargon a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.

**2.4.1 Analysis of a Logic Network**

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

Figure 2.10*a* shows a simple network consisting of three gates. To analyze its functional behavior, we can consider what happens if we apply all possible combinations of the input signals to it. Suppose that we start by making *x*1 = *x*2 = 0. This forces the output of the NOT gate to be equal to 1 and the output of the AND gate to be 0. Because one of the inputs to the OR gate is 1, the output of this gate will be 1. Therefore, *f* = 1 if *x*1 = *x*2 = 0. If we then let *x*1 = 0 and *x*2 = 1, no change in the value of *f* will take place, because the outputs of the NOT and AND gates will still be 1 and 0, respectively. Next, if we apply *x*1 = 1 and *x*2 = 0, then the output of the NOT gate changes to 0 while the output of the AND gate remains at 0. Both inputs to the OR gate are then equal to 0; hence the value of *f* will be 0. Finally, let *x*1 = *x*2 = 1. Then the output of the AND gate goes to 1, which in turn causes *f* to be equal to 1. Our verbal explanation can be summarized in the form of the truth table shown in Figure 2.10*b*.

**Timing Diagram**

We have determined the behavior of the network in Figure 2.10*a* by considering the four possible valuations of the inputs *x*1 and *x*2. Suppose that the signals that correspond to these valuations are applied to the network in the order of our discussion; that is, *(x*1*, x*2*)* = *(*0*,* 0*)* followed by *(*0*,* 1*)*, *(*1*,* 0*)*, and *(*1*,* 1*)*. Then changes in the signals at various points in the network would be as indicated in blue in the figure. The same information can be presented in graphical form, known as a *timing diagram*, as shown in Figure 2.10*c*. The time runs from left to right, and each input valuation is held for some fixed duration. The figure shows the waveforms for the inputs and output of the network, as well as for the internal signals at the points labeled *A* and *B*.

The timing diagram in Figure 2.10*c* shows that changes in the waveforms at points *A* and *B* and the output *f* take place instantaneously when the inputs *x*1 and *x*2 change their values. These idealized waveforms are based on the assumption that logic gates respond to changes on their inputs in zero time. Such timing diagrams are useful for indicating the *functional behavior* of logic circuits. However, practical logic gates are implemented using electronic circuits which need some time to change their states. Thus, there is a delay between a change in input values and a corresponding change in the output value of a gate. In chapters that follow we will use timing diagrams that incorporate such delays.

**30 CHAPTER 2** • **Introduction to Logic Circuits**

*x*1

0011 →→→

1100 →→→

A

*f*

1101 →→→

*x*2

0101 →→→

0001 →→→

B

(a) Network that implements *f x*1 *x*1 *x*2 ~~+=~~ ⋅

*x*1 *x*2 *f x*1 *x*2 ( ) ,

A B

0 0 1

1

0 1

0 1

1

1

0 1

1

1

0 0

0 0 0 1

*x*1 *x*2 A B

(b) Truth table

1

0

1

0

1

0

1

0

1

*f* Time 0

(c) Timing diagram

0011 →→→ 1100 →→→

*x*1

1101 →→→

0101 →→→ *g x*2

(d) Network that implements *g x*1 *x*2 ~~+=~~

**Figure 2.10** An example of logic networks.

**2.4 Logic Gates and Networks 31**

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

**Functionally Equivalent Networks**

Now consider the network in Figure 2.10*d*. Going through the same analysis procedure, we find that the output *g* changes in exactly the same way as *f* does in part (*a*) of the figure. Therefore, *g(x*1*, x*2*)* = *f (x*1*, x*2*)*, which indicates that the two networks are functionally equivalent; the output behavior of both networks is represented by the truth table in Figure 2.10*b*. Since both networks realize the same function, it makes sense to use the simpler one, which is less costly to implement.

In general, a logic function can be implemented with a variety of different networks, probably having different costs. This raises an important question: How does one find the best implementation for a given function? We will discuss some of the main approaches for synthesizing logic functions later in this chapter. For now, we should note that some manipulation is needed to transform the more complex network in Figure 2.10*a* into the network in Figure 2.10*d*. Since *f (x*1*, x*2*)* = *~~x~~*1 + *x*1 · *x*2 and *g(x*1*, x*2*)* = *~~x~~*1 + *x*2, there must exist some rules that can be used to show the equivalence

*~~x~~*1 + *x*1 · *x*2 = *~~x~~*1 + *x*2

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

**A**s an example of a logic function, consider the diagram in Figure 2.11*a*. It includes two **Example 2.1** toggle switches that control the values of signals *x* and *y*. Each toggle switch can be pushed down to the bottom position or up to the top position. When a toggle switch is in the bottom position it makes a connection to logic value 0 (ground), and when in the top position it connects to logic value 1 (power supply level). Thus, these toggle switches can be used to set *x* and *y* to either 0 or 1.

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

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

**32 CHAPTER 2** • **Introduction to Logic Circuits** 1

*x y*

*L*

0 1

0

*x y*

| Logic  circuit |
| --- |

*L*

0 0 1

1

0 1

0 1

0 1

1

0

(a) Two switches that control a light *x*

*L*

|  |
| --- |

*y*

(c) Logic network

(b) Truth table

*x*

*yL* (d) XOR gate symbol

**Figure 2.11** An example of a logic circuit.

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

**Example 2.2 I**n Chapter 1 we showed how numbers are represented in computers by using binary digits. As another example of logic functions, consider the addition of two one-digit binary numbers *a* and *b*. The four possible valuations of *a, b* and the resulting sums are given in Figure 2.12*a* (in this figure the + operator signifies *addition*). The sum *S* = *s*1*s*0 has to be a two-digit binary number, because when *a* = *b* = 1 then *S* = 10.

Figure 2.12*b* gives a truth table for the logic functions *s*1 and *s*0. From this table we can see that *s*1 = *a* · *b* and *s*0 = *a* ⊕ *b*. The corresponding logic network is given in part (*c*) of the figure. This type of logic circuit, which adds binary numbers, is referred to as an *adder* circuit. We discuss circuits of this type in Chapter 3.

*a*

+ *b*

0

0+

0

1+

1

0+

**2.5 Boolean Algebra 33**

1

1+

*s*0 *s*1

1000

10

01

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

*a b*

*s*1

*s*0

*a*

*s*0

0 0 1

1

0 1

0 1

0 0 0 1

0 1

1

0

*b*

*s*1

(b) Truth table

(c) Logic network

**Figure 2.12** Addition of binary numbers.

**2.5 Boolean Algebra**

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

**Axioms of Boolean Algebra**

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

1*a*. 0 · 0 = 0

1*b*. 1 + 1 = 1

2*a*. 1 · 1 = 1

**34 CHAPTER 2** • **Introduction to Logic Circuits**

2*b*. 0 + 0 = 0

3*a*. 0 · 1 = 1 · 0 = 0

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

4*a*. If *x* = 0, then *~~x~~* = 1

4*b*. If *x* = 1, then *~~x~~* = 0

**Single-Variable Theorems**

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

5*a*. *x* · 0 = 0

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

6*a*. *x* · 1 = *x*

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

7*a*. *x* · *x* = *x*

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

8*a*. *x* · *~~x~~* = 0

8*b*. *x* + *~~x~~* = 1

9. *~~x~~* = *x*

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

**Duality**

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

**Two- and Three-Variable Properties**

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

**2.5 Boolean Algebra 35**

10*a*. *x* · *y* = *y* · *x Commutative*

10*b*. *x* + *y* = *y* + *x*

11*a*. *x* · *( y* · *z)* = *(x* · *y)* · *z Associative*

11*b*. *x* + *( y* + *z)* = *(x* + *y)* + *z*

12*a*. *x* · *( y* + *z)* = *x* · *y* + *x* · *z Distributive*

12*b*. *x* + *y* · *z* = *(x* + *y)* · *(x* + *z)*

13*a*. *x* + *x* · *y* = *x Absorption*

13*b*. *x* · *(x* + *y)* = *x*

14*a*. *x* · *y* + *x* · *~~y~~* = *x Combining*

14*b*. *(x* + *y)* · *(x* + *~~y)~~* = *x*

15*a*. *~~x~~* ~~·~~ *~~y~~* = *~~x~~* + *~~y~~ DeMorgan’s theorem*

15*b*. *~~x~~* ~~+~~ *~~y~~* = *~~x~~* · *~~y~~*

16*a*. *x* + *~~x~~* · *y* = *x* + *y*

16*b*. *x* · *(~~x~~* + *y)* = *x* · *y*

17*a*. *x* · *y* + *y* · *z* + *~~x~~* · *z* = *x* · *y* + *~~x~~* · *z Consensus*

17*b*. *(x* + *y)* · *(y* + *z)* · *(~~x~~* + *z)* = *(x* + *y)* · *(~~x~~* + *z)*

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

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

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

| *x y* · | *x ~~y~~*  ~~·~~ | *x* | *y* |
| --- | --- | --- | --- |

*x y x*+ *y*

0 0 0 1 1 1 1

0 1 0 1 1 0 1

1 0 0 1 0 1 1

1 1 1 0 0 0 0

LHS RHS

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

**36 CHAPTER 2** • **Introduction to Logic Circuits**

**Example 2.3 L**et us prove the validity of the logic equation

*(x*1 + *x*3*)* · *(~~x~~*1 + *~~x~~*3*)* = *x*1 · *~~x~~*3 + *~~x~~*1 · *x*3

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

LHS = *(x*1 + *x*3*)* · *~~x~~*1 + *(x*1 + *x*3*)* · *~~x~~*3

Applying the distributive property again yields

LHS = *x*1 · *~~x~~*1 + *x*3 · *~~x~~*1 + *x*1 · *~~x~~*3 + *x*3 · *~~x~~*3

Note that the distributive property allows ANDing the terms in parenthesis in a way analo gous to multiplication in ordinary algebra. Next, according to theorem 8*a*, the terms *x*1 · *~~x~~*1 and *x*3 · *~~x~~*3 are both equal to 0. Therefore,

LHS = 0 + *x*3 · *~~x~~*1 + *x*1 · *~~x~~*3 + 0

From 6*b* it follows that

LHS = *x*3 · *~~x~~*1 + *x*1 · *~~x~~*3

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

LHS = *x*1 · *~~x~~*3 + *~~x~~*1 · *x*3

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

**Example 2.4 C**onsider the logic equation

*x*1 · *~~x~~*3 + *~~x~~*2 · *~~x~~*3 + *x*1 · *x*3 + *~~x~~*2 · *x*3 = *~~x~~*1 · *~~x~~*2 + *x*1 · *x*2 + *x*1 · *~~x~~*2

The left-hand side can be manipulated as follows

LHS = *x*1 · *~~x~~*3 + *x*1 · *x*3 + *~~x~~*2 · *~~x~~*3 + *~~x~~*2 · *x*3 using 10*b*

= *x*1 · *(~~x~~*3 + *x*3*)* + *~~x~~*2 · *(~~x~~*3 + *x*3*)* using 12*a*

= *x*1 · 1 + *~~x~~*2 · 1 using 8*b*

= *x*1 + *~~x~~*2 using 6*a*

The right-hand side can be manipulated as

RHS = *~~x~~*1 · *~~x~~*2 + *x*1 · *(x*2 + *~~x~~*2*)* using 12*a*

= *~~x~~*1 · *~~x~~*2 + *x*1 · 1 using 8*b*

= *~~x~~*1 · *~~x~~*2 + *x*1 using 6*a*

= *x*1 + *~~x~~*1 · *~~x~~*2 using 10*b*

= *x*1 + *~~x~~*2 using 16*a*

**2.5 Boolean Algebra 37**

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

*f (x*1*, x*2*, x*3*)* = *x*1 · *~~x~~*3 + *~~x~~*2 · *~~x~~*3 + *x*1 · *x*3 + *~~x~~*2 · *x*3

= *~~x~~*1 · *~~x~~*2 + *x*1 · *x*2 + *x*1 · *~~x~~*2

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

*f (x*1*, x*2*, x*3*)* = *x*1 + *~~x~~*2

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

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

**2.5.1 The Venn Diagram**

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

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

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

**38 CHAPTER 2** • **Introduction to Logic Circuits**

(a) Constant 1 (b) Constant 0

*x x xx*

(c) Variable *x* (d) *x*

*x y xy*

(e) (f)

·

*x y x y* +

*x y*

*x y*

*z*

(g) (h)

· ·

*x ~~y~~ xy z* +

**Figure 2.14** The Venn diagram representation.

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

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

*x y z*

**2.5 Boolean Algebra 39**

*x y*

*z*

·

(a) (d)

*x x y*

*x y z*

*x y*

*z*

·

(b) *y z* + (e) *x z*

*x y z*

*x y z*

· · ·

(c) (f)

*x yz* ( ) + *xy x* + *z*

**Figure 2.15** Verification of the distributive property *x* · *( y* + *z)* = *x* · *y* + *x* · *z*.

namely, where at least one of *x* or *y* is equal to 1. Since this combines the areas in the circles, the OR operation is formally often called the *union* of *x* and *y*. Part (*g*) depicts the term *x* · *~~y~~*, which is represented by the intersection of the area for *x* with that for *~~y~~*. Part (*h*) gives a three-variable example; the expression *x* · *y* + *z* is the union of the area for *z* with that of the intersection of *x* and *y*.

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

*x* · *( y* + *z)* = *x* · *y* + *x* · *z*

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

**40 CHAPTER 2** • **Introduction to Logic Circuits**

As another example, consider the identity

*x* · *y* + *~~x~~* · *z* + *y* · *z* = *x* · *y* + *~~x~~* · *z*

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

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

*x y z*

*x y z*

· ·

*x y*

*x y z*

*x y*

*x y z*

· ·

*x z*

*yx z*

*x z*

*x y z*

· · ·

*y z xy x* + *z*

*x y*

*z*

···

*xy x* + *z yz* +

**Figure 2.16** Verification of *x* · *y* + *~~x~~* · *z* + *y* · *z* = *x* · *y* + *~~x~~* · *z*.

**2.5 Boolean Algebra 41**

**T**he distributive property 12*a* in Figure 2.15 will look familiar to the reader, because it is valid **Example 2.5** both for Boolean variables and for variables that are real numbers. In the case of real-number variables, the operations involved would be multiplication and addition, rather than logical AND and OR. However, the dual form 12*b* of this property, *x* + *y* · *z* = *(x* + *y)* · *(x* + *z)*, does not hold for real-number variables involving multiplication and addition operations. To prove that this identity is valid in Boolean algebra we can use the Venn diagrams in Figure 2.17. Parts (*a*) and (*b*) of the figure depict the terms *x* and *y* · *z*, respectively, and part (*c*) gives the union of parts (*a*) and (*b*). Parts (*d*) and (*e*) depict the sum terms *(x* + *y)* and *(x* + *z)*, and part ( *f* ) shows the intersection of (*d*) and (*e*). Since the diagrams in (*c*) and ( *f* ) are the same, this proves the identity.

| *yx*  *z* |
| --- |

(a) *x*

| *yx*  *z* |
| --- |

·

(b) *y z*

| *yx*  *z* |
| --- |

| *yx*  *z* |
| --- |

(d) *x y* +

| *yx*  *z* |
| --- |

(e) *x z* +

| *yx*  *z* |
| --- |

· ·

(c) *x y* + *z*

(f) (*x y* + ( ) *x z*)+

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

**42 CHAPTER 2** • **Introduction to Logic Circuits**

| *x y* |
| --- |

(a) *x y*·

| *x y* |
| --- |

(b) *x ~~y~~*~~·~~

| *x y* |
| --- |

(c) *x*

| *x y* |
| --- |

(d) *y*

| *x y* |
| --- |

(e) *x y*+

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

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

**2.5.2 Notation and Terminology**

Boolean algebra is based on the AND and OR operations, for which we have adopted the symbols · and +, respectively. These are also the standard symbols for the familiar arithmetic multiplication and addition operations. Considerable similarity exists between the Boolean operations and the arithmetic operations, which is the main reason why the

**2.6 Synthesis Using AND, OR, and NOT Gates 43**

same symbols are used. In fact, when single digits are involved there is only one significant difference; the result of 1 + 1 is equal to 2 in ordinary arithmetic, whereas it is equal to 1 in Boolean algebra as defined by theorem 7*b* in Section 2.5.

Because of the similarity with the arithmetic addition and multiplication operations, the OR and AND operations are often called the *logical sum* and *product* operations. Thus *x*1 + *x*2 is the logical sum of *x*1 and *x*2, and *x*1 · *x*2 is the logical product of *x*1 and *x*2. Instead of saying “logical product” and “logical sum,” it is customary to say simply “product” and “sum.” Thus we say that the expression

*x*1 · *~~x~~*2 · *x*3 + *~~x~~*1 · *x*4 + *x*2 · *x*3 · *~~x~~*4

is a sum of three product terms, whereas the expression

*(~~x~~*1 + *x*3*)* · *(x*1 + *~~x~~*3*)* · *(~~x~~*2 + *x*3 + *x*4*)*

is a product of three sum terms.

**2.5.3 Precedence of Operations**

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

*x*1 · *x*2 + *~~x~~*1 · *~~x~~*2

it is first necessary to generate the complements of *x*1 and *x*2. Then the product terms *x*1 · *x*2 and *~~x~~*1 · *~~x~~*2 are formed, followed by the sum of the two product terms. Observe that in the absence of this convention, we would have to use parentheses to achieve the same effect as follows:

*(x*1 · *x*2*)* + *((~~x~~*1*)* · *(~~x~~*2*))*

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

*x*1*x*2 + *~~x~~*1*~~x~~*2

We will use this style throughout the book.

**2.6 Synthesis Using AND, OR, and NOT Gates**

Armed with some basic ideas, we can now try to implement arbitrary functions using the AND, OR, and NOT gates. Suppose that we wish to design a logic circuit with two inputs, *x*1 and *x*2. Assume that *x*1 and *x*2 represent the states of two switches, either of which may

**44 CHAPTER 2** • **Introduction to Logic Circuits**

*x*1 *x*2 *f x*1 *x*2

( , )

0 0 1

0 1 1

1 0 0

1 1 1

**Figure 2.19** A function to be synthesized.

produce a 0 or 1. The function of the circuit is to continuously monitor the state of the switches and to produce an output logic value 1 whenever the switches *(x*1*, x*2*)* are in states *(*0*,* 0*)*, *(*0*,* 1*)*, or *(*1*,* 1*)*. If the state of the switches is *(*1*,* 0*)*, the output should be 0. We can express the required behavior using a truth table, as shown in Figure 2.19.

A possible procedure for designing a logic circuit that implements this truth table is to create a product term that has a value of 1 for each valuation for which the output function *f* has to be 1. Then we can take a logical sum of these product terms to realize *f* . Let us begin with the fourth row of the truth table, which corresponds to *x*1 = *x*2 = 1. The product term that is equal to 1 for this valuation is *x*1 · *x*2, which is just the AND of *x*1 and *x*2. Next consider the first row of the table, for which *x*1 = *x*2 = 0. For this valuation the value 1 is produced by the product term *~~x~~*1 · *~~x~~*2. Similarly, the second row leads to the term *~~x~~*1 · *x*2. Thus *f* may be realized as

*f (x*1*, x*2*)* = *x*1*x*2 + *~~x~~*1*~~x~~*2 + *~~x~~*1*x*2

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

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

*f (x*1*, x*2*)* = *x*1*x*2 + *~~x~~*1*~~x~~*2 + *~~x~~*1*x*2 + *~~x~~*1*x*2

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

*f (x*1*, x*2*)* = *x*1*x*2 + *~~x~~*1*x*2 + *~~x~~*1*~~x~~*2 + *~~x~~*1*x*2

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

*f (x*1*, x*2*)* = *(x*1 + *~~x~~*1*)x*2 + *~~x~~*1*(~~x~~*2 + *x*2*)*

*x*1

*x*2

*x*1

*x*2

**2.6 Synthesis Using AND, OR, and NOT Gates 45**

*f*

(a) Canonical sum-of-products

*f*

(b) Minimal-cost realization

**Figure 2.20** Two implementations of the function in Figure 2.19.

Applying theorem 8*b* we get

*f (x*1*, x*2*)* = 1 · *x*2 + *~~x~~*1 · 1

Finally, theorem 6*a* leads to

*f (x*1*, x*2*)* = *x*2 + *~~x~~*1

The network described by this expression is given in Figure 2.20*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure. This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables, and it is formed such that if the input variable *xi* is equal to 1 in the given row, then *xi* is entered in the term; if *xi* = 0 in that row, then *~~x~~i* is entered. The sum of these product terms realizes the desired function. Second, there are many different networks that can realize a given function. Some of these networks may be simpler than others. Algebraic manipulation can be used to derive simplified logic expressions and thus lower-cost networks. The process whereby we begin with a description of the desired functional behavior and then generate a circuit that realizes this behavior is called *synthesis*. Thus we can say that we “synthesized” the networks in Figure 2.20 from the truth table in Figure 2.19. Generation of AND-OR expressions from a truth table is just one of many types of synthesis techniques that we will encounter in this book.

**46 CHAPTER 2** • **Introduction to Logic Circuits**

**Example 2.7 F**igure 2.21*a* depicts a part of a factory that makes bubble gumballs. The gumballs travel on a conveyor that has three associated sensors *s*1, *s*2, and *s*3. The sensor *s*1 is connected to a scale that weighs each gumball, and if a gumball is not heavy enough to be acceptable then the sensor sets *s*1 = 1. Sensors *s*2 and *s*3 examine the diameter of each gumball. If a gumball is too small to be acceptable, then *s*2 = 1, and if it is too large, then *s*3 = 1. If a gumball is of an acceptable weight and size, then the sensors give *s*1 = *s*2 = *s*3 = 0. The conveyor pushes the gumballs over a “trap door” that it used to reject the ones that are not properly formed. A gumball should be rejected if it is too large, or both too small and too light. The trap door is opened by setting the logic function *f* to the value 1. By inspection, we can see that an appropriate logic expression is *f* = *s*1*s*2 + *s*3. We will use Boolean algebra to derive this logic expression from the truth table.

The truth table for *f* is given in Figure 2.21*b*. It sets *f* to 1 for each row in the table where *s*3 has the value 1 (too large), as well as for each row where *s*1 = *s*2 = 1 (too light and too small). As described previously, a logic expression for *f* can be formed by including a product term for each row where *f* = 1. Thus, we can write

*f* = *~~s~~*1*~~s~~*2*s*3 + *~~s~~*1*s*2*s*3 + *s*1*~~s~~*2*s*3 + *s*1*s*2*~~s~~*3 + *s*1*s*2*s*3

gumball

*s*1

*s*2 *s*3

*f* = “reject”

(a) Conveyor and sensors

*s*1 *s*2 *s*3

*f*

0 0 0 0 1

1

1

1

0 0 1

1

0 0 1

1

0 1

0 1

0 1

0 1

0 1

0 1

0 1

1

1

(b) Truth table

**Figure 2.21** A bubble gumball factory.

**2.6 Synthesis Using AND, OR, and NOT Gates 47**

We can use algebraic manipulation to simplify this expression in a number of ways. For example, as shown below, we can first use rule 7*b* to repeat the term *s*1*s*2*s*3, and then use the distributive property 12*a* and rule 8*b* to simplify the expression

*f* = *~~s~~*1*~~s~~*2*s*3 + *~~s~~*1*s*2*s*3 + *s*1*~~s~~*2*s*3 + *s*1*s*2*s*3 + *s*1*s*2*~~s~~*3 + *s*1*s*2*s*3

= *~~s~~*1*s*3*(~~s~~*2 + *s*2*)* + *s*1*s*3*(~~s~~*2 + *s*2*)* + *s*1*s*2*(~~s~~*3 + *s*3*)*

= *~~s~~*1*s*3 + *s*1*s*3 + *s*1*s*2

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

*f* = *s*3 + *s*1*s*2

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

**T**here are different ways in which we can simplify the logic expression produced from the **Example 2.8** truth table in Figure 2.21*b*. Another approach is to first repeat the term *s*1*s*2*s*3, as we did in Example 2.7, and then proceed as follows

*f* = *~~s~~*1*~~s~~*2*s*3 + *~~s~~*1*s*2*s*3 + *s*1*~~s~~*2*s*3 + *s*1*s*2*s*3 + *s*1*s*2*~~s~~*3 + *s*1*s*2*s*3

= *s*3*(~~s~~*1*~~s~~*2 + *~~s~~*1*s*2 + *s*1*~~s~~*2 + *s*1*s*2*)* + *s*1*s*2*(~~s~~*3 + *s*3*)*

= *s*3 · 1 + *s*1*s*2

= *s*3 + *s*1*s*2

Here, we used the distributive property 12*a* to produce the expression *(~~s~~*1*~~s~~*2 + *~~s~~*1*s*2 + *s*1*~~s~~*2 + *s*1*s*2*)*. Since this expression includes all possible valuations of *s*1*,s*2, it is equal to 1, leading to the same expression for *f* that we derived before.

**Y**et another way of producing the symplified logic expression is shown below. **Example 2.9**

*f* = *~~s~~*1*~~s~~*2*s*3 + *~~s~~*1*s*2*s*3 + *s*1*~~s~~*2*s*3 + *s*1*s*2*~~s~~*3 + *s*1*s*2*s*3

= *~~s~~*1*~~s~~*2*s*3 + *~~s~~*1*s*2*s*3 + *s*1*~~s~~*2*s*3 + *~~s~~*1*~~s~~*2*s*3 + *s*1*s*2*~~s~~*3 + *s*1*s*2*s*3

= *~~s~~*1*s*3*(~~s~~*2 + *s*2*)* + *~~s~~*2*s*3*(s*1 + *~~s~~*1*)* + *s*1*s*2*(~~s~~*3 + *s*3*)*

= *~~s~~*1*s*3 + *~~s~~*2*s*3 + *s*1*s*2

= *s*3*(~~s~~*1 + *~~s~~*2*)* + *s*1*s*2

= *s*3*(~~s~~*1*~~s~~*2*~~)~~* + *s*1*s*2

= *s*3 + *s*1*s*2

In this solution, we first repeat the term *~~s~~*1*~~s~~*2*s*3, and then symplify to generate the expression *s*3*(~~s~~*1 + *~~s~~*2*)* + *s*1*s*2. Using DeMorgan’s theorem 15*a* we can replace *(~~s~~*1 + *~~s~~*2*)* with *(~~s~~*1*~~s~~*2*~~)~~*, which can then be deleted by applying property 16*a*.

**48 CHAPTER 2** • **Introduction to Logic Circuits**

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

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

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

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

**Minterms**

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

To illustrate this concept, consider the truth table in Figure 2.22. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. From the discussion of the binary number representation in Section 1.5, we can observe that the row numbers chosen are just the numbers represented by the bit patterns of variables *x*1, *x*2, and *x*3. The figure shows all minterms for the three-variable table. For example, in the first row the variables

Row

| *x*1 *x*2 *x*3 | Minterm |
| --- | --- |
| 000  001  010  011  100  101  110  111 | *m*0 = *x* 1 *x* 2 *x* 3 *m*1 = *x*1 *x* 2 *x*3 *m*2 = *x* 1 *x*2 *x* 3 *m*3 = *x* 1 *x*2 *x*3 *m*4 = *x*1 *x* 2 *x* 3 *m*5 = *x*1 *x* 2 *x*3 *m*6 = *x*1 *x*2 *x* 3 *m*7 = *x*1 *x*2 *x*3 |

number Maxterm

0 *M*0 = *x*1 + *x*2 + *x*3 1 *M*1 = *x*1 + *x*2 + *x* 3 2 *M*2 = *x*1 + *x* 2 + *x*3 3 *M*3 = *x*1 + *x* 2 + *x* 3 4 *M*4 = *x* 1 + *x*2 + *x*3 5 *M*5 = *x* 1 + *x*2 + *x* 3 6 *M*6 = *x* 1 + *x* 2 + *x*3 7 *M*7 = *x* 1 + *x* 2 + *x* 3

**Figure 2.22** Three-variable minterms and maxterms.

**2.6 Synthesis Using AND, OR, and NOT Gates 49**

have the values *x*1 = *x*2 = *x*3 = 0, which leads to the minterm *~~x~~*1*~~x~~*2*~~x~~*3. In the second row *x*1 = *x*2 = 0 and *x*3 = 1, which gives the minterm *~~x~~*1*~~x~~*2*x*3, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation *mi* to denote the minterm for row number *i*. Thus *m*0 = *~~x~~*1*~~x~~*2*~~x~~*3, *m*1 = *~~x~~*1*~~x~~*2*x*3, and so on.

**Sum-of-Products Form**

A function *f* can be represented by an expression that is a sum of minterms, where each minterm isANDed with the value of*f* for the corresponding valuation of input variables. For example, the two-variable minterms are *m*0 = *~~x~~*1*~~x~~*2, *m*1 = *~~x~~*1*x*2, *m*2 = *x*1*~~x~~*2, and *m*3 = *x*1*x*2. The function in Figure 2.19 can be represented as

*f* = *m*0 · 1 + *m*1 · 1 + *m*2 · 0 + *m*3 · 1

= *m*0 + *m*1 + *m*3

= *~~x~~*1*~~x~~*2 + *~~x~~*1*x*2 + *x*1*x*2

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

As another example, consider the three-variable function *f (x*1*, x*2*, x*3*)*, specified by the truth table in Figure 2.23. To synthesize this function, we have to include the minterms *m*1,

Row

| *x*1 *x*2 *x*3 |
| --- |
| 000  001  010  011  100  101  110  111 |

number *f x*1 *x*2 *x*3 ( , , )

0 0

1 1

2 0

3 0

4 1

5 1

6 1

7 0

**Figure 2.23** A three-variable function.

**50 CHAPTER 2** • **Introduction to Logic Circuits**

*m*4, *m*5, and *m*6. Copying these minterms from Figure 2.22 leads to the following canonical sum-of-products expression for *f*

*f (x*1*, x*2*, x*3*)* = *~~x~~*1*~~x~~*2*x*3 + *x*1*~~x~~*2*~~x~~*3 + *x*1*~~x~~*2*x*3 + *x*1*x*2*~~x~~*3

This expression can be manipulated as follows

*f* = *(~~x~~*1 + *x*1*)~~x~~*2*x*3 + *x*1*(~~x~~*2 + *x*2*)~~x~~*3

= 1 · *~~x~~*2*x*3 + *x*1 · 1 · *~~x~~*3

= *~~x~~*2*x*3 + *x*1*~~x~~*3

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

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

*x*2

*f*

*x*3

*x*1

(a) A minimal sum-of-products realization

*x*1

*x*3

*f*

*x*2

(b) A minimal product-of-sums realization

**Figure 2.24** Two realizations of the function in Figure 2.23.

**2.6 Synthesis Using AND, OR, and NOT Gates 51**

as

*f (x*1*, x*2*, x*3*)* = *(m*1*, m*4*, m*5*, m*6*)*

or even more simply as

*f (x*1*, x*2*, x*3*)* = *m(*1*,* 4*,* 5*,* 6*)*

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

**C**onsider the function **Example 2.10** *f (x*1*, x*2*, x*3*)* = *m(*2*,* 3*,* 4*,* 6*,* 7*)*

The canonical SOP expression for the function is derived using minterms

*f* = *m*2 + *m*3 + *m*4 + *m*6 + *m*7

= *~~x~~*1*x*2*~~x~~*3 + *~~x~~*1*x*2*x*3 + *x*1*~~x~~*2*~~x~~*3 + *x*1*x*2*~~x~~*3 + *x*1*x*2*x*3

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

*f* = *~~x~~*1*x*2*(~~x~~*3 + *x*3*)* + *x*1*(~~x~~*2 + *x*2*)~~x~~*3 + *x*1*x*2*(~~x~~*3 + *x*3*)*

= *~~x~~*1*x*2 + *x*1*~~x~~*3 + *x*1*x*2

= *(~~x~~*1 + *x*1*)x*2 + *x*1*~~x~~*3

= *x*2 + *x*1*~~x~~*3

**S**uppose that a four-variable function is defined by **Example 2.11** *f (x*1*, x*2*, x*3*, x*4*)* = *m(*3*,* 7*,* 9*,* 12*,* 13*,* 14*,* 15*)*

The canonical SOP expression for this function is

*f* = *~~x~~*1*~~x~~*2*x*3*x*4 + *~~x~~*1*x*2*x*3*x*4 + *x*1*~~x~~*2*~~x~~*3*x*4 + *x*1*x*2*~~x~~*3*~~x~~*4 + *x*1*x*2*~~x~~*3*x*4 + *x*1*x*2*x*3*~~x~~*4 + *x*1*x*2*x*3*x*4 A simpler SOP expression can be obtained as follows

*f* = *~~x~~*1*(~~x~~*2 + *x*2*)x*3*x*4 + *x*1*(~~x~~*2 + *x*2*)~~x~~*3*x*4 + *x*1*x*2*~~x~~*3*(~~x~~*4 + *x*4*)* + *x*1*x*2*x*3*(~~x~~*4 + *x*4*)*

= *~~x~~*1*x*3*x*4 + *x*1*~~x~~*3*x*4 + *x*1*x*2*~~x~~*3 + *x*1*x*2*x*3

= *~~x~~*1*x*3*x*4 + *x*1*~~x~~*3*x*4 + *x*1*x*2*(~~x~~*3 + *x*3*)*

= *~~x~~*1*x*3*x*4 + *x*1*~~x~~*3*x*4 + *x*1*x*2

**52 CHAPTER 2** • **Introduction to Logic Circuits**

**Maxterms**

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

**Product-of-Sums Form**

If a given function *f* is specified by a truth table, then its complement *f* can be rep resented by a sum of minterms for which *f* = 1, which are the rows where *f* = 0. For example, for the function in Figure 2.19

*f (x*1*, x*2*)* = *m*2

= *x*1*~~x~~*2

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

*f* = *f* = *x*1*~~x~~*2

= *~~x~~*1 + *x*2

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

*f* = *~~m~~*2 = *M*2

where *M*2 is the maxterm for row 2 in the truth table.

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

*f (x*1*, x*2*, x*3*)* = *m*0 + *m*2 + *m*3 + *m*7

= *~~x~~*1*~~x~~*2*~~x~~*3 + *~~x~~*1*x*2*~~x~~*3 + *~~x~~*1*x*2*x*3 + *x*1*x*2*x*3

Then *f* can be expressed as

*f* = *~~m~~*0 ~~+~~ *~~m~~*2 ~~+~~ *~~m~~*3 ~~+~~ *~~m~~*7

= *~~m~~*0 · *~~m~~*2 · *~~m~~*3 · *~~m~~*7

= *M*0 · *M*2 · *M*3 · *M*7

= *(x*1 + *x*2 + *x*3*)(x*1 + *~~x~~*2 + *x*3*)(x*1 + *~~x~~*2 + *~~x~~*3*)(~~x~~*1 + *~~x~~*2 + *~~x~~*3*)*

This expression represents *f* as a product of maxterms.

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

**2.6 Synthesis Using AND, OR, and NOT Gates 53**

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

*f* = *((x*1 + *x*3*)* + *x*2*)((x*1 + *x*3*)* + *~~x~~*2*)(x*1 + *(~~x~~*2 + *~~x~~*3*))(~~x~~*1 + *(~~x~~*2 + *~~x~~*3*))*

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

*f* = *(x*1 + *x*3*)(~~x~~*2 + *~~x~~*3*)*

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

Using the shorthand notation, an alternative way of specifying our sample function is *f (x*1*, x*2*, x*3*)* = *(M*0*, M*2*, M*3*, M*7*)*

or more simply

*f (x*1*, x*2*, x*3*)* = *M (*0*,* 2*,* 3*,* 7*)*

The sign denotes the logical product operation.

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

**C**onsider again the function in Example 2.10. Instead of using the minterms, we can specify **Example 2.12** this function as a product of maxterms for which *f* = 0, namely

*f (x*1*, x*2*, x*3*)* = *M (*0*,* 1*,* 5*)*

Then, the canonical POS expression is derived as

*f* = *M*0 · *M*1 · *M*5

= *(x*1 + *x*2 + *x*3*)(x*1 + *x*2 + *~~x~~*3*)(~~x~~*1 + *x*2 + *~~x~~*3*)*

A simplified POS expression can be derived as

*f* = *(x*1 + *x*2 + *x*3*)(x*1 + *x*2 + *~~x~~*3*)(x*1 + *x*2 + *~~x~~*3*)(~~x~~*1 + *x*2 + *~~x~~*3*)*

= *((x*1 + *x*2*)* + *x*3*)((x*1 + *x*2*)* + *~~x~~*3*)(x*1 + *(x*2 + *~~x~~*3*))(~~x~~*1 + *(x*2 + *~~x~~*3*))*

= *((x*1 + *x*2*)* + *x*3*~~x~~*3*)(x*1*~~x~~*1 + *(x*2 + *~~x~~*3*))*

= *(x*1 + *x*2*)(x*2 + *~~x~~*3*)*

**54 CHAPTER 2** • **Introduction to Logic Circuits**

Another way of deriving this product-of-sums expression is to use the sum-of-products form of *f* . Thus,

*f (x*1*, x*2*, x*3*)* = *m(*0*,* 1*,* 5*)*

= *~~x~~*1*~~x~~*2*~~x~~*3 + *~~x~~*1*~~x~~*2*x*3 + *x*1*~~x~~*2*x*3

= *~~x~~*1*~~x~~*2*~~x~~*3 + *~~x~~*1*~~x~~*2*x*3 + *~~x~~*1*~~x~~*2*x*3 + *x*1*~~x~~*2*x*3

= *~~x~~*1*~~x~~*2*(~~x~~*3 + *x*3*)* + *~~x~~*2*x*3*(~~x~~*1 + *x*1*)*

= *~~x~~*1*~~x~~*2 + *~~x~~*2*x*3

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

*f* = *f*

= *(~~x~~*1*~~x~~*2 + *~~x~~*2*x*3*)*

= *(~~x~~*1*~~x~~*2*)(~~x~~*2*x*3*)*

= *(x*1 + *x*2*)(x*2 + *~~x~~*3*)*

To see that this product-of-sums expression for *f* is equivalent to the sum-of-products expression that we derived in Example 2.10, we can slightly rearrange our expression as *f* = *(x*2 + *x*1*)(x*2 + *~~x~~*3*)*. Now, recognizing that this expression has the form of the righthand side of the distributive property 12*b*, we have the sum-of-products expression *f* = *x*2 + *x*1*~~x~~*3.

**2.7 NAND and NOR Logic Networks**

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

If NAND and NOR gates are realized with simpler circuits than AND and OR gates, then we should ask whether these gates can be used directly in the synthesis of logic circuits. In Section 2.5 we introduced DeMorgan’s theorem. Its logic gate interpretation is shown in Figure 2.26. Identity 15*a* is interpreted in part (*a*) of the figure. It specifies that a NAND of variables *x*1 and *x*2 is equivalent to first complementing each of the variables and then ORing them. Notice on the far-right side that we have indicated the NOT gates simply as bubbles, which denote inversion of the logic value at that point. The other half of DeMorgan’s theorem, identity 15*b*, appears in part (*b*) of the figure. It states that the NOR function is equivalent to first inverting the input variables and then ANDing them.

*x*1

**2.7 NAND and NOR Logic Networks 55**

*x*1

*x*2

*x*2*~~x~~*1 *~~x~~*2 *~~x~~*1 *~~x~~*2 ~~…~~ *~~x~~n* ~~· ···~~

*xn*

(a) NAND gates

*x*1

*x*2

*x*1 *~~x~~*2 ~~…~~ *~~x~~n* ~~+++~~ *x*1 *x*2*x*1 *~~x~~*2 ~~+~~

*xn*

(b) NOR gates

**Figure 2.25** NAND and NOR gates.

*x*1 *x*2

*x*1 *x*2

*x*1 *x*2

*x*1 *x*2

*x*1 *~~x~~*2 *x*1 *x* ~~+=~~ 2 (a) *x*1 *~~x~~* ~~+~~ 2 *x*1 *x* = 2 (b)

*x*1 *x*2

*x*1 *x*2

**Figure 2.26** DeMorgan’s theorem in terms of logic gates.

**56 CHAPTER 2** • **Introduction to Logic Circuits**

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

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

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

**Example 2.13 L**et us implement the function

*f (x*1*, x*2*, x*3*)* = *m(*2*,* 3*,* 4*,* 6*,* 7*)*

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

*f* = *(x*1 + *x*2*)(x*2 + *~~x~~*3*)*

An OR-AND circuit that corresponds to this expression is shown in Figure 2.29*a*. Using the same structure of the circuit, a NOR-gate version is given in Figure 2.29*b*. Note that *x*3 is inverted by a NOR gate that has its inputs tied together.

**Example 2.14 L**et us now implement the function

*f (x*1*, x*2*, x*3*)* = *m(*2*,* 3*,* 4*,* 6*,* 7*)*

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

*f* = *x*2 + *x*1*~~x~~*3

which is realized using the circuit in Figure 2.30*a*. We can again use the same structure to obtain a circuit with NAND gates, but with one difference. The signal *x*2 passes only through an OR gate, instead of passing through an AND gate and an OR gate. If we simply

**2.7 NAND and NOR Logic Networks 57**

*x*1

*x*1

*x*2

*x*2

*x*3

*x*3

*x*4

*x*4

*x*5

*x*5

*x*1

*x*2

*x*3

*x*4

*x*5

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

*x*1 *x*2

*x*3 *x*4 *x*5

*x*1 *x*2

*x*3 *x*4 *x*5

*x*1 *x*2

*x*3 *x*4 *x*5

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

**58 CHAPTER 2** • **Introduction to Logic Circuits** *x*1

*x*2

*f*

*x*3

(a) POS implementation

*x*1

*x*2

*f*

*x*3

(b) NOR implementation

**Figure 2.29** NOR-gate realization of the function in Example 2.13. *x*2

*x*1

*x*3

*x*2

*f*

(a) SOP implementation

*x*1*f x*3

(b) NAND implementation

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

**2.8 Design Examples 59**

replace the OR gate with a NAND gate, this signal would be inverted which would result in a wrong output value. Since *x*2 must either not be inverted, or it can be inverted twice, we can pass it through two NAND gates as depicted in Figure 2.30*b*. Observe that for this circuit the output *f* is

*f* = *~~x~~*2 · *x*1*~~x~~*3

Applying DeMorgan’s theorem, this expression becomes

*f* = *x*2 + *x*1*~~x~~*3

**2.8 Design Examples**

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

**2.8.1 Three-Way Light Control**

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

As a first step, let us turn this word statement into a formal specification using a truth table. Let *x*1*, x*2, and *x*3 be the input variables that denote the state of each switch. Assume that the light is off if all switches are open. Closing any one of the switches will turn the light on. Then turning on a second switch will have to turn off the light. Thus the light will be on if exactly one switch is closed, and it will be off if two (or no) switches are closed. If the light is off when two switches are closed, then it must be possible to turn it on by closing the third switch. If *f (x*1*, x*2*, x*3*)* represents the state of the light, then the required functional behavior can be specified as shown in the truth table in Figure 2.31. The canonical sum-of-products expression for the specified function is

*f* = *m*1 + *m*2 + *m*4 + *m*7

= *~~x~~*1*~~x~~*2*x*3 + *~~x~~*1*x*2*~~x~~*3 + *x*1*~~x~~*2*~~x~~*3 + *x*1*x*2*x*3

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

**60 CHAPTER 2** • **Introduction to Logic Circuits**

*x*1 *x*2 *x*3 *f*

000 0

001 1

010 1

011 0

100 1

101 0

110 0

111 1

**Figure 2.31** Truth table for the three-way light

control.

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

*f* = *M*0 · *M*3 · *M*5 · *M*6

= *(x*1 + *x*2 + *x*3*)(x*1 + *~~x~~*2 + *~~x~~*3*)(~~x~~*1 + *x*2 + *~~x~~*3*)(~~x~~*1 + *~~x~~*2 + *x*3*)*

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

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

**2.8.2 Multiplexer Circuit**

In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals *x*1 and *x*2. The values of these signals change in time, perhaps at regular intervals. Thus sequences of 0s and 1s are applied on each of the inputs *x*1 and *x*2. We want to design a circuit that produces an output that has the same value as either *x*1 or *x*2, dependent on the value of a selection control signal *s*. Therefore, the circuit should have three inputs: *x*1, *x*2, and *s*. Assume that the output of the circuit will be the same as the value of input *x*1 if *s* = 0, and it will be the same as *x*2 if *s* = 1.

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

*f (s, x*1*, x*2*)* = *~~sx~~*1*~~x~~*2 + *~~sx~~*1*x*2 + *s~~x~~*1*x*2 + *sx*1*x*2

*x*1

*x*2

*x*3

*x*3

*x*2

*x*1

**2.8 Design Examples 61**

*f*

(a) Sum-of-products realization

|  | |
| --- | --- |
|  |  |

*f*

(b) Product-of-sums realization

**Figure 2.32** Implementation of the function in Figure 2.31.

Using the distributive property, this expression can be written as *f* = *~~sx~~*1*(~~x~~*2 + *x*2*)* + *s(~~x~~*1 + *x*1*)x*2

Applying theorem 8*b* yields

*f* = *~~sx~~*1 · 1 + *s* · 1 · *x*2

Finally, theorem 6*a* gives

*f* = *~~sx~~*1 + *sx*2

**62 CHAPTER 2** • **Introduction to Logic Circuits**

*s x*1 *x*2

*fsx*1 *x*2 ( , , )

*x*1

*s*

*x*2

000 0 001 0 010 1 011 1 100 0 101 1 110 0 111 1

(a) Truth table *f*

*s*

*x*1 *x*2

0 1

*f*

(b) Circuit (c) Graphical symbol

*s fsx*1 *x*2

( , , )

0 *x*1

1 *x*2

(d) More compact truth-table representation

**Figure 2.33** Implementation of a multiplexer.

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

**2.8 Design Examples 63**

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

Note that the statement “*f* = *x*1 if *s* = 0, and *f* = *x*2 if *s* = 1” can be presented in a more compact form of a truth table, as indicated in Figure 2.33*d*. In later chapters we will have occasion to use such representation.

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

**2.8.3 Number Display**

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

| Logic  circuit |
| --- |

*a*

| *a*  *bf*  *g*  *ce*  *d* |
| --- |

| *b* |
| --- |
| *c* |
| *d* |
| *e* |
| *f* |
| *g* |

*s*0

*s*1

(a) Logic circuit and 7-segment display

*s*1

*s*0 *a*

*b*

*c*

*d*

*e*

*f*

*g*

|  |
| --- |

0 0 1

0 1

0

1

0 1

1 1 1

1

1

0

1

0 1

1

0 1

1

0 0

0 0 1

(b) Truth table

**Figure 2.34** Display of numbers.