FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN

STEPHEN BROWN ZVONKO VRANESIC

TATA McGRAW-HILL EDITION

FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN

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

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN

Copyright © 2003 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.

Tata McGraw-Hill Edition 2002

Third reprint 2004
RXXQCRZIDRLCR

Reprinted in India by arrangement with The McGraw-Hill Companies Inc.,
New York

Sales territories: India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan

Library of Congress Cataloging-in-Publication Data

Brown, Stephen D.

Fundamentals of digital logic with Verilog design / Stephen D. Brown, Zvonko G. Vranesic.—1st ed.
p. cm. (McGraw-Hill Series in electrical and computer engineering)
Includes index.
ISBN 0-07-282315-1
1. Logic circuits—Design and construction—Data processing. 2. Verilog (Computer hardware description language). 3. Computer-aided design. I. Vranesic, Zvonko G. II. Title. III. Series.

TK7868.L6 B76 2003
621.39'2—dc21

2002071439
CIP

ISBN 0-07-052899-3

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, and printed at
Krishna Offset, Delhi 110 032

To Susan and Anne

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 an Associate Professor in the Department of Electrical & Computer Engineering. He is also Director of Software Development at the Altera Toronto Technology Center.

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

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

Zvonko Vranesic received his B.A.Sc., M.A.Sc., and Ph.D. degrees, all in Electrical Engineering, from the University of Toronto. From 1963–1965, he worked as a design engineer with the Northern Electric Co. Ltd. in Bramalea, Ontario. In 1968 he joined the University of Toronto, where he is now a Professor in the Departments of Electrical & Computer Engineering and Computer Science. During the 1978–79 academic year, he was a Senior Visitor at the University of Cambridge, England, and during 1984–85 he was at the University of Paris, 6. From 1995 to 2000 he served as Chair of the Division of Engineering Science at the University of Toronto. He is also involved in research and development at the Altera Toronto Technology Center.

His current research interests include computer architecture, field-programmable VLSI technology, and multiple-valued logic systems.

He is a coauthor of four other books: *Computer Organization*, 5th ed.; *Fundamentals of Digital Logic with VHDL Design*; *Microcomputer Structures*; and *Field-Programmable Gate Arrays*. In 1990, he received the Wighton Fellowship for "innovative and distinctive contributions to undergraduate laboratory instruction."

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

McGraw-Hill Series in Electrical and Computer Engineering

Brown, Vranesic: *Fundamentals of Digital Logic with VHDL Design*
Givone: *Digital Principles and Design*
Ham, Kostanic: *Principles of Neurocomputing for Science and Engineering*
Hamacher, Vranesic, and Zaky: *Computer Organization*
Hayes: *Computer Architecture and Organization*
Hwang: *Advanced Computer Architecture: Parallelism, Scalability, Programmability*
Hwang: *Scalable Parallel Computing: Technology, Architecture, Programming*
Leon-Garcia, Widjaja: *Communication Networks*
Marcovitz: *Inroduction to Logic Design*
Navabi: *VHDL: Analysis and Modeling of Digital Systems*
Patt, Patel: *Introduction to Computing Systems: From Bits & Gates to C & Beyond*
Schalkoff: *Artificial Neural Networks*
Shen, Lipasti: *Modern Processor Design*

FOREWORD

Chess is a game that provides a splendid vehicle for displaying human intelligence in a competitive environment. During the past 30 years, it has also served as a platform for determining the extent to which machines can emulate intelligent behavior. Many chess programs are available for today's computers. Chess machines, comprising a computer and a chess-playing program, are now capable of defeating even the strongest human players.

The ultimate challenge took place in 1997, when IBM's Deep Blue chess machine defeated the World Champion Garry Kasparov in a six-game match. The essence of this machine are logic circuits, algorithms, and software—coupled with people who know how to use these resources. Although all of these factors are crucial, the greatest leap forward, in terms of chess-playing strength, was made when extremely powerful logic circuits were developed. Most of these circuits are used in general purpose computers, but some are specialized for the chess-playing application. A key reason why the Deep Blue machine is so strong is that it can evaluate about 200 million chess position in one second.

This textbook deals with logic circuits and explains how they are designed. We have included in the book the moves from the decisive sixth game of the 1997 match to remind the reader of the incredible possibilities that are attainable with well-designed logic circuits. Deep Blue played with the white pieces.

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 computer-aided design (CAD) tools. The purpose of our book is to provide the desirable balance between teaching the basic concepts and practical application through CAD tools. To facilitate the learning process, the necessary CAD software is included as an integral part of the book package.

A serious drawback of many books on digital logic design is that they cover too much material. A book that covers a large number of topics is not easy to use in a classroom, particularly if the topics are not covered in sufficient depth. Also, in their desire to provide a vast amount of practical advice, the authors often make the text difficult to follow by the students who are still struggling with the fundamental concepts. Our aim is to avoid both of these problems.

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 AND CAD SUPPORT

The book discusses modern digital circuit implementation technologies. We briefly discuss SSI, as well as semi-custom and full-custom technologies. However, the emphasis is on programmable logic devices (PLDs). This 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).

Our CAD support is based on Altera MAX+plusII software. MAX+plusII provides automatic mapping of a design into Altera CPLDs and FPGAs, which are among the most widely used PLDs in the industry. The features of MAX+plusII that are particularly attractive for our purposes are:

- It is a commercial product. The version included with the book supports all major features of the product. Students will be able to easily enter a design into the CAD system, compile the design into a selected device (the choice of device can be changed at any time and the design retargeted to a different device), simulate the functionality and detailed timing of the resulting circuit, and if laboratory facilities are provided at the student's school, implement the designs in actual devices.
- It provides for design entry using both hardware description languages (HDLs) and schematic capture. In the book, we provide examples of design using schematic capture, but we emphasize the HDL-based design because it is the most efficient design method to use in practice. We describe in detail the IEEE Standard Verilog language and use it extensively in examples. The CAD system included with the book has a Verilog compiler, which allows the student to automatically create circuits from the Verilog code and implement these circuits in real chips.
- It can automatically target a design to various types of devices. This feature allows us to illustrate the ways in which the architecture of the target device affects a designer's circuit.
- It can be used on most types of popular computers. We expect that most students will use the version of the software that runs on IBM-compatible computers (running any version of Microsoft windows), which is provided with the book. However, through Altera's university program the software is also available for other machines, such as SUN or HP workstations.

A MAX+plusII CD-ROM is included with each copy of the book. Use of the software is fully integrated into the book so that students can try, firsthand, all design examples. To teach the students how to use this software, the book includes three, progressively advanced, hands-on tutorials.

SCOPE OF THE BOOK

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.

Chapter 2 introduces the basic aspects of logic circuits. It shows how Boolean algebra is used to represent such 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.

The electronic aspects of digital circuits are presented in Chapter 3. This chapter 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.

Chapter 4 deals with the synthesis of combinational circuits. It covers all aspects of the synthesis process, starting with an initial design and performing the optimization steps needed to generate a desired final circuit. It shows how CAD tools are used for this purpose.

Chapter 5 concentrates on circuits that perform arithmetic operations. It begins with a discussion of how numbers are represented in digital systems and then shows how such numbers 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. We chose to introduce the number representations at this point, rather than in the very beginning of the book, to make the discussion more meaningful and interesting, because we can immediately provide examples of how numerical information may be processed by actual circuits.

Chapter 6 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 7. 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 8 gives a detailed presentation of synchronous sequential circuits (finite state machines). It explains the behavior of these circuits and develops practical design techniques for both manual and automated design.

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 should be studied because 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 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 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.

Appendices B, C, and D contain a sequence of tutorials on the MAX+plusII CAD tools. This material is suitable for self-study; it shows the student in a step-by-step manner how to use the CAD software provided with the book.

Appendix E gives detailed information about the devices used in illustrative examples. It also includes a brief discussion of TTL technology.

WHAT CAN BE COVERED IN A COURSE

All 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 2 to 3 lecture hours on Verilog, concentrating mostly on the specification of sequential circuits. The Verilog examples given in the book are largely self-explanatory, and students can understand them easily. Moreover, the instructor need not teach how to use the CAD tools, because the MAX+plusII tutorials in Appendices B, C, and D are suitable for self-study.

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

A natural starting point for formal lectures is Chapter 2. The material in Chapter 1 is a general introduction that serves as a motivation for why logic circuits are important and interesting; students can read and understand this material easily.

The following material should be covered in lectures:

- Chapter 2—all sections.
- Chapter 3—sections 3.1 to 3.7. Also, it is useful to cover sections 3.8 and 3.9 if the students have some basic knowledge of electrical circuits.
- Chapter 4—sections 4.1 to 4.7 and section 4.12.
- Chapter 5—sections 5.1 to 5.5.
- Chapter 6—all sections.
- Chapter 7—all sections.
- Chapter 8—sections 8.1 to 8.9.

If time permits, it would also be very useful to cover sections 9.1 to 9.3 and section 9.6 in Chapter 9, as well as one or two examples in Chapter 10.

One-Quarter Course

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

- Chapter 2—all sections.
- Chapter 3—sections 3.1 to 3.3.
- Chapter 4—sections 4.1 to 4.5 and section 4.12.
- Chapter 5—sections 5.1 to 5.3 and section 5.5.
- Chapter 6—all sections.
- Chapter 7—sections 7.1 to 7.10 and section 7.13.
- Chapter 8—Sections 8.1 to 8.5.

A MORE TRADITIONAL APPROACH

The material in Chapters 2 and 4 introduces Boolean algebra, combinational logic circuits, and basic minimization techniques. Chapter 2 provides initial exposure to these topics using only AND, OR, NOT, NAND, and NOR gates. Then Chapter 3 discusses the implementation technology details, before proceeding with the synthesis techniques and other types of gates

in Chapter 4. The material in Chapter 4 is appreciated better if students understand the technological reasons for the existence of NAND, NOR, and XOR gates, and the various programmable logic devices.

An instructor who favors a more traditional approach may cover Chapters 2 and 4 in succession. To understand the use of NAND, NOR, and XOR gates, it is necessary only that the instructor provide a functional definition of these gates.

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 140 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.

HOMEWORK PROBLEMS

More than 400 homework problems are provided in the book. Solutions to these problems are available to instructors in the *Solutions Manual* that accompanies the book.

LABORATORY

The book can be used for a course that does not include laboratory exercises, in which case students can get useful practical experience by simulating the operation of their designed circuits by using the CAD tools provided with the book. If there is an accompanying laboratory, then a number of design examples in the book are suitable for laboratory experiments. Additional experiments are available on the authors' website.

ACKNOWLEDGMENTS

We wish to express our thanks to the people who have helped during the preparation of the book. Kelly Chan helped with the technical preparation of the manuscript. Dan Vranesic produced a substantial amount of artwork. He and Deshanand Singh also helped with the preparation of the solutions manual. The reviewers, William Barnes, New Jersey Institute of Technology; James Clark, McGill University; Stephen DeWeerth, Georgia Institute of Technology; Clay Gloster, Jr., North Carolina State University (Raleigh); Carl Hamacher, Queen's University; Wei-Ming Lin, University of Texas (Austin); Wayne Loucks, University of Waterloo; Chris Myers, University of Utah; 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; Charles Silio, Jr., University of Maryland; Scott Smith, University of Missouri (Rolla); Arun Somani, Iowa State University; and Zeljko Zilic, McGill University provided constructive criticism and made numerous suggestions for improvements. We are grateful to the Altera Corporation for providing the MAX+plusII CAD system. The support of McGraw-Hill people has been exemplary. We truly appreciate the help of Kelley Butcher, Catherine Fields Shultz, Michaela Graham, Betsy Jones, Kara Kudronowicz, Carlise Paulson, Jill Peter, John Wannemacher, and Michelle Whitaker.

Stephen Brown and Zvonko Vranesic

CONTENTS

Chapter 1

DESIGN CONCEPTS 1

1.1 Digital Hardware 2
- 1.1.1 Standard Chips 4
- 1.1.2 Programmable Logic Devices 4
- 1.1.3 Custom-Designed Chips 5

1.2 The Design Process 6

1.3 Design of Digital Hardware 8
- 1.3.1 Basic Design Loop 8
- 1.3.2 Design of a Digital Hardware Unit 9

1.4 Logic Circuit Design in This Book 11

1.5 Theory and Practice 14

References 15

Chapter 2

INTRODUCTION TO LOGIC CIRCUITS 17

2.1 Variables and Functions 18

2.2 Inversion 21

2.3 Truth Tables 22

2.4 Logic Gates and Networks 23
- 2.4.1 Analysis of a Logic Network 24

2.5 Boolean Algebra 27
- 2.5.1 The Venn Diagram 30
- 2.5.2 Notation and Terminology 34
- 2.5.3 Precedence of Operations 34

2.6 Synthesis Using AND, OR, and NOT Gates 35
- 2.6.1 Sum-of-Products and Product-of-Sums Forms 37

2.7 NAND and NOR Logic Networks 41

2.8 Design Examples 44
- 2.8.1 Three-Way Light Control 44
- 2.8.2 Multiplexer Circuit 45

2.9 Introduction to CAD Tools 48
- 2.9.1 Design Entry 48
- 2.9.2 Synthesis 51
- 2.9.3 Functional Simulation 52
- 2.9.4 Summary 52

2.10 Introduction to Verilog 54
- 2.10.1 Structural Specification of Logic Circuits 55
- 2.10.2 Behavioral Specification of Logic Cirucits 58
- 2.10.3 How *Not* to Write Verilog Code 60

2.11 Concluding Remarks 60

Problems 61

References 65

Chapter 3

IMPLEMENTATION TECHNOLOGY 67

3.1 Transistor Switches 69

3.2 NMOS Logic Gates 71

3.3 CMOS Logic Gates 74
- 3.3.1 Speed of Logic Gate Circuits 81

3.4 Negative Logic System 82

3.5 Standard Chips 83
- 3.5.1 7400-Series Standard Chips 83

3.6 Programmable Logic Devices 87
- 3.6.1 Programmable Logic Array (PLA) 87
- 3.6.2 Programmable Array Logic (PAL) 90
- 3.6.3 Programming of PLAs and PALs 92
- 3.6.4 Complex Programmable Logic Devices (CPLDs) 94
- 3.6.5 Field-Programmable Gate Arrays 98
- 3.6.6 Using CAD Tools to Implement Circuits in CPLDs and FPGAs 102

3.7 Custom Chips, Standard Cells, and Gate Arrays 103

3.8 Practical Aspects 106
- 3.8.1 MOSFET Fabrication and Behavior 106
- 3.8.2 MOSFET On-Resistance 110
- 3.8.3 Voltage Levels in Logic Gates 111
- 3.8.4 Noise Margin 113
- 3.8.5 Dynamic Operation of Logic Gates 114
- 3.8.6 Power Dissipation in Logic Gates 117

3.8.7 Passing 1s and 0s Through Transistor Switches 118
3.8.8 Fan-in and Fan-out in Logic Gates 120
3.9 Transmission Gates 126
3.9.1 Exclusive-OR Gates 127
3.9.2 Multiplexer Circuit 128
3.10 Implementation Details for SPLDs, CPLDs, and FPGAs 129
3.10.1 Implementation in FPGAs 135
3.11 Concluding Remarks 137
Problems 138
References 147

Chapter 4

OPTIMIZED IMPLEMENTATION OF LOGIC FUNCTIONS 149

4.1 Karnaugh Map 150
4.2 Strategy for Minimization 158
4.2.1 Terminology 159
4.2.2 Minimization Procedure 160
4.3 Minimization of Product-of-Sums Forms 164
4.4 Incompletely Specified Functions 166
4.5 Multiple-Output Circuits 167
4.6 Multilevel Synthesis 171
4.6.1 Factoring 172
4.6.2 Functional Decomposition 175
4.6.3 Multilevel NAND and NOR Circuits 181
4.7 Analysis of Multilevel Circuits 184
4.8 Cubical Representation 189
4.8.1 Cubes and Hypercubes 189
4.9 A Tabular Method for Minimization 193
4.9.1 Generation of Prime Implicants 193
4.9.2 Determination of a Minimum Cover 195
4.9.3 Summary of the Tabular Method 200
4.10 A Cubical Technique for Minimization 201
4.10.1 Determination of Essential Prime Implicants 204
4.10.2 Complete Procedure for Finding a Minimal Cover 206
4.11 Practical Considerations 208
4.12 CAD Tools 209
4.12.1 Logic Synthesis and Optimization 210
4.12.2 Physical Design 211
4.12.3 Timing Simulation 213
4.12.4 Summary of Design Flow 213
4.12.5 Examples of Circuits Synthesized from Verilog Code 216
4.13 Concluding Remarks 220
Problems 221
References 226

Chapter 5

NUMBER REPRESENTATION AND ARITHMETIC CIRCUITS 229

5.1 Positional Number Representation 230
5.1.1 Unsigned Integers 230
5.1.2 Conversion Between Decimal and Binary Systems 231
5.1.3 Octal and Hexadecimal Representations 232
5.2 Addition of Unsigned Numbers 234
5.2.1 Decomposed Full-Adder 238
5.2.2 Ripple-Carry Adder 239
5.2.3 Design Example 240
5.3 Signed Numbers 240
5.3.1 Negative Numbers 240
5.3.2 Addition and Subtraction 224
5.3.3 Adder and Subtractor Unit 248
5.3.4 Radix-Complement Schemes 249
5.3.5 Arithmetic Overflow 253
5.3.6 Performance Issues 254
5.4 Fast Adders 255
5.4.1 Carry-Lookahead Adder 255
5.5 Design of Arithmetic Circuits Using CAD Tools 262
5.5.1 Design of Arithmetic Circuits Using Schematic Capture 262
5.5.2 Design of Arithmetic Circuits Using Verilog 265
5.5.3 Using Vectored Signals 268
5.5.4 Using a Generic Specification 269
5.5.5 Nets and Variables in Verilog 270
5.5.6 Arithmetic Assignment Statements 271
5.5.7 Representation of Numbers in Verilog Code 275
5.6 Multiplication 277
5.6.1 Array Multiplier for Unsigned Numbers 279
5.6.2 Multiplication of Signed Numbers 279

5.7 Other Number Representations 282
5.7.1 Fixed-Point Numbers 282
5.7.2 Floating-Point Numbers 282
5.7.3 Binary-Coded-Decimal Representation 284
5.8 ASCII Character Code 289
Problems 291
References 295

Chapter 6
COMBINATIONAL-CIRCUIT BUILDING BLOCKS 297

6.1 Multiplexers 298
6.1.1 Synthesis of Logic Functions Using Multiplexers 303
6.1.2 Multiplexer Synthesis Using Shannon's Expansion 304
6.2 Decoders 311
6.2.1 Demultiplexers 314
6.3 Encoders 316
6.3.1 Binary Encoders 316
6.3.2 Priority Encoders 317
6.4 Code Converters 318
6.5 Arithmetic Comparison Circuits 320
6.6 Verilog for Combinational Circuits 320
6.6.1 The Conditional Operator 321
6.6.2 The If-Else Statement 323
6.6.3 The Case Statement 326
6.6.4 The For Loop 331
6.6.5 Verilog Operators 333
6.6.6 The Generate Construct 338
6.6.7 Tasks and Functions 339
6.7 Concluding Remarks 343
Problems 343
References 347

Chapter 7
FLIP-FLOPS, REGISTERS, COUNTERS, AND A SIMPLE PROCESSOR 349

7.1 Basic Latch 351
7.2 Gated SR Latch 353
7.2.1 Gated SR Latch with NAND Gates 355
7.3 Gated D Latch 356
7.3.1 Effects of Propagation Delays 358
7.4 Master-Slave and Edge-Triggered D Flip-Flops 359
7.4.1 Master-Slave D Flip-Flop 359
7.4.2 Edge-Triggered D Flip-Flop 360
7.4.3 D Flip-Flops with Clear and Preset 362
7.5 T Flip-Flop 364
7.5.1 Configurable Flip-Flops 367
7.6 JK Flip-Flop 367
7.7 Summary of Terminology 368
7.8 Registers 368
7.8.1 Shift Register 369
7.8.2 Parallel-Access Shift Register 370
7.9 Counters 371
7.9.1 Asynchronous Counters 371
7.9.2 Synchronous Counters 374
7.9.3 Counters with Parallel Load 378
7.10 Reset Synchronization 378
7.11 Other Types of Counters 382
7.11.1 BCD Counter 382
7.11.2 Ring Counter 383
7.11.3 Johnson Counter 384
7.11.4 Remarks on Counter Design 385
7.12 Using Storage Elements with CAD Tools 385
7.12.1 Including Storage Elements in Schematics 385
7.12.2 Using Verilog Constructs for Storage Elements 388
7.12.3 Blocking and Non-blocking Assignments 390
7.12.4 Non-blocking Assignments for Combinational Circuits 394
7.12.5 Flip-Flops with Clear Capability 395
7.13 Using Registers and Counters with CAD Tools 396
7.13.1 Including Registers and Counters in Schematics 396
7.13.2 Using Library Modules in Verilog Code 399
7.13.3 Using Verilog Constructs for Registers and Counters 400
7.14 Design Examples 405
7.14.1 Bus Structure 405
7.14.2 Simple Processor 417
7.14.3 Reaction Timer 429
7.14.4 Register Transfer Level (RTL) Code 433
7.15 Concluding Remarks 434

Problems 434
References 442

Chapter 8
SYNCHRONOUS SEQUENTIAL CIRCUITS 445

8.1 Basic Design Steps 447
8.1.1 State Diagram 447
8.1.2 State Table 449
8.1.3 State Assignment 449
8.1.4 Choice of Flip-Flops and Derivation of Next-State and Output Expressions 451
8.1.5 Timing Diagram 453
8.1.6 Summary of Design Steps 454
8.2 State-Assignment Problem 458
8.2.1 One-Hot Encoding 460
8.3 Mealy State Model 462
8.4 Design of Finite State Machines Using CAD Tools 467
8.4.1 Verilog Code for Moore-Type FSMs 467
8.4.2 Synthesis of Verilog Code 468
8.4.3 Simulating and Testing the Circuit 470
8.4.4 Alternative Styles of Verilog Code 471
8.4.5 Summary of Design Steps When Using CAD Tools 473
8.4.6 Specifying the State Assignment in Verilog Code 474
8.4.7 Specification of Mealy FSMs Using Verilog 475
8.5 Serial Adder Example 477
8.5.1 Mealy-Type FSM for Serial Adder 477
8.5.2 Moore-Type FSM for Serial Adder 479
8.5.3 Verilog Code for the Serial Adder 480
8.6 State Minimization 486
8.6.1 Partitioning Minimization Procedure 486
8.6.2 Incompletely Specified FSMs 493
8.7 Design of a Counter Using the Sequential Circuit Approach 495
8.7.1 State Diagram and State Table for a Modulo-8 Counter 495
8.7.2 State Assignment 496
8.7.3 Implementation Using D-Type Flip-Flops 497
8.7.4 Implementation Using JK-Type Flip-Flops 498
8.7.5 Example—A Different Counter 502
8.8 FSM as an Arbiter Circuit 505
8.8.1 Implementation of the Arbiter Circuit 508
8.8.2 Minimizing the Output Delays for an FSM 511
8.8.3 Summary 511
8.9 Analysis of Synchronous Sequential Circuits 512
8.10 Algorithmic State Machine (ASM) Charts 516
8.11 Formal Model for Sequential Circuits 519
8.12 Concluding Remarks 521
Problems 521
References 525

Chapter 9
ASYNCHRONOUS SEQUENTIAL CIRCUITS 527

9.1 Asynchronous Behavior 528
9.2 Analysis of Asynchronous Circuits 531
9.3 Synthesis of Asynchronous Circuits 540
9.4 State Reduction 553
9.5 State Assignment 568
9.5.1 Transition Diagram 571
9.5.2 Exploiting Unspecified Next-State Entries 574
9.5.3 State Assignment Using Additional State Variables 578
9.5.4 One-Hot State Assignment 582
9.6 Hazards 584
9.6.1 Static Hazards 585
9.6.2 Dynamic Hazards 590
9.6.3 Significance of Hazards 592
9.7 A Complete Design Example 592
9.7.1 The Vending-Machine Controller 592
9.8 Concluding Remarks 599
Problems 599
References 604

Chapter 10
DIGITAL SYSTEM DESIGN 605

10.1 Building Block Circuits 606
10.1.1 Flip-Flops and Registers with Enable Inputs 606
10.1.2 Shift Registers with Enable Inputs 607

10.1.3 Static Random Access Memory (SRAM) 609
10.1.4 SRAM Blocks in PLDs 611
10.2 Design Examples 612
10.2.1 A Bit-Counting Circuit 612
10.2.2 ASM-Chart-Implied Timing Information 613
10.2.3 Shift-and-Add Multiplier 618
10.2.4 Divider 623
10.2.5 Arithmetic Mean 631
10.2.6 Sort Operation 641
10.3 Clock Synchronization 653
10.3.1 Clock Skew 653
10.3.2 Flip-Flop Timing Parameters 655
10.3.3 Asynchronous Inputs to Flip-Flops 656
10.3.4 Switch Debouncing 657
10.4 Concluding Remarks 659
Problems 659
References 663

Chapter 11
TESTING OF LOGIC CIRCUITS 665

11.1 Fault Model 666
11.1.1 Stuck-at Model 666
11.1.2 Single and Multiple Faults 667
11.1.3 CMOS Circuits 667
11.2 Complexity of a Test Set 667
11.3 Path Sensitizing 669
11.3.1 Detection of a Specific Fault 671
11.4 Circuits with Tree Structure 673
11.5 Random Tests 674
11.6 Testing of Sequential Circuits 677
11.6.1 Design for Testability 677
11.7 Built-in Self-Test 681
11.7.1 Built-in Logic Block Observer 685
11.7.2 Signature Analysis 687
11.7.3 Boundary Scan 688
11.8 Printed Circuit Boards 688
11.8.1 Testing of PCBs 690
11.8.2 Instrumentation 691
11.9 Concluding Remarks 692
Problems 692
References 695

Appendix A
VERILOG REFERENCE 697

A.1 Documentation in Verilog Code 699
A.2 White Space 699
A.3 Signals in Verilog Code 699
A.4 Identifier Names 699
A.5 Signal Values, Numbers, and Parameters 700
A.6 Net and Variable Types 701
A.6.1 Nets 701
A.6.2 Variables 702
A.6.3 Memories 703
A.7 Operators 703
A.8 Verilog Module 705
A.9 Gate Instantiations 706
A.10 Concurrent Statements 708
A.10.1 Continuous Assignments 709
A.10.2 Using Parameters 710
A.11 Procedural Statements 711
A.11.1 Always and Initial Blocks 711
A.11.2 The If-Else Statement 713
A.11.3 Statement Ordering 714
A.11.4 The Case Statement 715
A.11.5 Casex and Casez Statements 717
A.11.6 Loop Statements 717
A.11.7 Blocking versus Non-blocking Assignments for Combinational Circuits 721
A.12 Using Subcircuits 721
A.12.1 Subcircuit Parameters 723
A.12.2 Verilog 2001 Generate Capability 725
A.13 Functions and Tasks 726
A.14 Sequential Circuits 730
A.14.1 A Gated D Latch 730
A.14.2 D Flip-Flop 730
A.14.3 Flip-Flops with Reset 731
A.14.4 Instantiating a Flip-Flop from a Library 732
A.14.5 Registers 733
A.14.6 Shift Registers 734
A.14.7 Counters 735
A.14.8 An Example of a Sequential Circuit 736
A.14.9 Moore-Type Finite State Machines 737
A.14.10 Mealy-Type Finite State Machines 739
A.15 Guidelines for Writing Verilog Code 742
A.16 MAX+PlusII Verilog Support 745
A.16.1 Limitations in MAX+PlusII 745
A.17 Concluding Remarks 746
References 746

Appendix B
TUTORIAL 1 747

B.1 Introduction 748
B.1.1 Getting Started 748
B.2 Design Entry Using Schematic Capture 751
B.2.1 Specifying the Project Name 752
B.2.2 Using the Graphic Editor 752
B.2.3 Synthesizing a Circuit from the Schematic 758
B.2.4 Performing Functional Simulation 759
B.2.5 Using the Message Processor to Locate and Fix Errors 763
B.3 Design Entry Using Verilog 765
B.3.1 Specifying the Project Name 765
B.3.2 Using the Text Editor 765
B.3.3 Synthesizing a Circuit from the Verilog Code 767
B.3.4 Performing Functional Simulation 767
B.3.5 Using the Message Processor to Debug Verilog Code 768
B.4 Design Entry Using Truth Tables 768
B.4.1 Using the Waveform Editor 769
B.4.2 Create the Timing Diagram 769
B.4.3 Synthesizing a Circuit from the Waveforms 770
B.5 Mixing Design-Entry Methods 772
B.5.1 Creating a Schematic that Includes a Truth Table 772
B.5.2 Synthesizing and Simulating a Circuit from the Schematic 774
B.5.3 Using the Hierarchy Display 775
B.5.4 Concluding Remarks 775

Appendix C
TUTORIAL 2 777

C.1 Implementing a Circuit in a MAX 7000 CPLD 778
C.1.1 Using the Compiler 779
C.1.2 Selecting a Chip 780
C.1.3 Viewing the Logic Synthesis Options 781
C.1.4 Examining the Implemented Circuit 782
C.1.5 Running the Timing Simulator 783
C.1.6 Using the Floorplan Editor 784
C.2 Implementing a Circuit in a FLEX 10K FPGA 787
C.3 Downloading a Circuit into a Device 789
C.4 Making Pin Assignments 790
C.4.1 Assigning Signals to Pins in the Floorplan Editor 792
C.4.2 Making Pin Assignments Permanent 794
C.5 Concluding Remarks 795

Appendix D
TUTORIAL 3 797

D.1 Design Using Verilog Code 798
D.1.1 The Ripple-Carry Adder Code 798
D.1.2 Using the Timing Analyzer Module 801
D.2 Using an LPM Module 802
D.3 Design of a Sequential Circuit 806
D.3.1 Using the Graphic Editor 806
D.3.2 Synthesizing a Circuit and Using the Timing Simulator 812
D.3.3 Using the Timing Analyzer 813
D.3.4 Using Verilog Code 814
D.4 Design of a Finite State Machine 815
D.4.1 Implementation in a CPLD 815
D.4.2 Implementation in an FPGA 816
D.5 Concluding Remarks 819

Appendix E
COMMERCIAL DEVICES 821

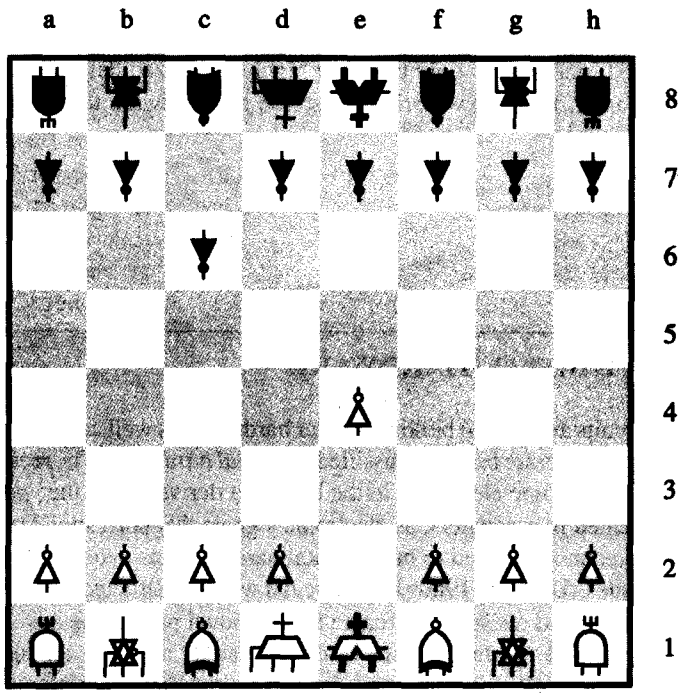
E.1 Simple PLDs 822
E.1.1 The 22V10 PAL Device 822
E.2 Complex PLDs 824
E.2.1 Altera MAX 7000 825
E.3 Field-Programmable Gate Arrays 826
E.3.1 Altera FLEX 10K 827
E.3.2 Xilinx XC4000 830
E.3.3 Altera APEX 20K 831
E.3.4 Altera Stratix 832
E.3.5 Xilix Virtex 834
E.4 Transistor-Transistor Logic 835
E.4.1 TTL Circuit Families 836
References 837

INDEX 838

chapter

1

DESIGN CONCEPTS



1. e2–e4, c7–c6

This book is about logic circuits—the circuits from which computers are built. Proper understanding of logic circuits is vital for today's electrical and computer engineers. These circuits are the key ingredient of computers and are also used in many other applications. They are found in commonly used products, such as digital watches, various household appliances, CD players, and electronic games, as well as in large systems, such as the equipment for telephone and television networks.

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 in great depth 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 Verilog. 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. With modern technology it is possible to fabricate chips that contain tens of millions of transistors, as in the case of computer processors. The basic building blocks for such circuits are easy to understand, but there is nothing simple about a circuit that contains tens of millions of transistors. The complexity that comes with the large size of logic 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 become clear later in the book—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 four 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 larger. Integrated circuit chips are manufactured on a silicon wafer, such as the one shown in Figure 1.1. The wafer is cut to produce the individual chips, which are then placed inside a special type of chip package. By 1970 it was possible to implement all circuitry needed to realize a microprocessor on a single chip. Although early microprocessors had modest computing capability by today's standards, they opened the door for the information processing revolution by providing the means for implementation of affordable personal computers. About 30 years ago Gordon Moore, chairman of Intel Corporation, observed that integrated circuit technology was progressing at an astounding rate, doubling the number of transistors that could be placed on a chip every 1.5 to 2 years.

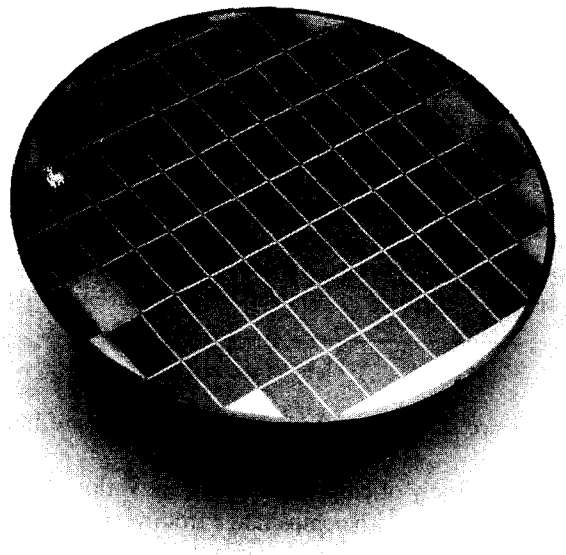


Figure 1.1 A silicon wafer (courtesy of Altera Corp.).

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 contain more than 10 million transistors. Presently chips can have a few hundreds of millions of transistors.

Moore's law is expected to continue to hold true for at least the next decade. A consortium of integrated circuit manufacturers called the Semiconductor Industry Association (SIA) produces an estimate of how the technology is expected to evolve. Known as the *SIA Roadmap* [1], this estimate predicts the minimum size of a transistor that can be fabricated on an integrated circuit chip. The size of a transistor is measured by a parameter called its *gate length*, which we will discuss in Chapter 3. A sample of the SIA Roadmap is given in Table 1.1. In 2002 the minimum possible gate length that can be reliably manufactured is 0.13 μm. The first row of the table indicates that the minimum gate length is expected to reduce steadily to about 0.035 μm by the year 2012. The size of a transistor determines how many transistors can be placed in a given amount of chip area, with the current maximum being about 20 million transistors per cm^2. This number is expected to grow to 100 million transistors by the year 2012. The largest chip size is expected to be about 1300 mm^2 at that time; thus chips with up to 1.3 billion transistors will be possible! 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, more likely, designing circuits that involve a number of chips placed on a *printed circuit board (PCB)*. Frequently, some of the logic circuits can be realized in existing chips that are readily available. This situation simplifies the design task and shortens the time needed to develop the final product. Before we discuss the design

Table 1.1 A sample of the SIA Roadmap

	Year					
	1999	**2001**	**2003**	**2006**	**2009**	**2012**
Transistor gate length	0.14 μm	0.12 μm	0.10 μm	0.07 μm	0.05 μm	0.035 μm
Transistors per cm^2	14 million	16 million	24 million	40 million	64 million	100 million
Chip size	800 mm^2	850 mm^2	900 mm^2	1000 mm^2	1100 mm^2	1300 mm^2

process in more 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 very simple chips with low functionality to extremely complex chips. For example, a digital hardware product may require a microprocessor to perform some arithmetic operations, memory chips to provide storage capability, and interface chips that allow easy connection to input and output devices. Such chips are available from various vendors.

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

1.1.1 STANDARD CHIPS

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

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

1.1.2 PROGRAMMABLE LOGIC DEVICES

In contrast to standard chips that have fixed functionality, it is possible to construct chips that contain circuitry that 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 collec-

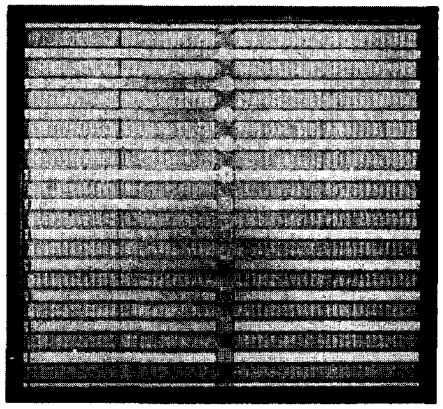


Figure 1.2 A field-programmable gate array chip (courtesy of Altera Corp.).

tion 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 needed for a particular application by choosing an appropriate configuration of the switches. The switches are programmed by the end user, rather than when the chip is manufactured. Such chips are known as *programmable logic devices (PLDs)*. We will introduce them in Chapter 3.

Most types of PLDs can be programmed multiple times. This capability is advantageous because a designer who is developing a prototype of a product can program a PLD to perform some function, but later, when the prototype hardware is being tested, can make corrections by reprogramming the PLD. Reprogramming might be necessary, for instance, if a designed function is not quite as intended or if new functions are needed that were not contemplated in the original design.

PLDs are available in a wide range of sizes. They can be used to realize much larger logic circuits than a typical standard chip can realize. Because of their size and the fact that they can be tailored to meet the requirements of a specific application, PLDs are widely used today. One of the most sophisticated types of PLD is known as a *field-programmable gate array (FPGA)*. FPGAs that contain more than 100 million transistors are now available [2, 3]. A photograph of an FPGA chip that has 10 million transistors is shown in Figure 1.2. The chip consists of a large number of small logic circuit elements, which can be connected together using the programmable switches. The logic circuit elements are arranged in a regular two-dimensional structure.

1.1.3 CUSTOM-DESIGNED CHIPS

PLDs are available as off-the-shelf components that can be purchased from different suppliers. Because they are programmable, they can be used to implement most logic circuits found in digital hardware. However, PLDs 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 PLDs 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 an appropriate technology is chosen to implement the chip. Finally, the chip is manufactured by a company that has the fabrication facilities. This approach is known as *custom* or *semi-custom design*, and such chips are called *custom* or *semi-custom chips*. Such chips are intended for use in specific applications and are sometimes called *application-specific integrated circuits (ASICs)*.

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

A disadvantage of the custom-design approach is that manufacturing a custom chip often takes a considerable amount of time, on the order of months. In contrast, if a PLD 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 design 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. We will start by introducing a typical development cycle in the most general terms. Then we will focus on the particular aspects that pertain to the design of logic circuits.

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

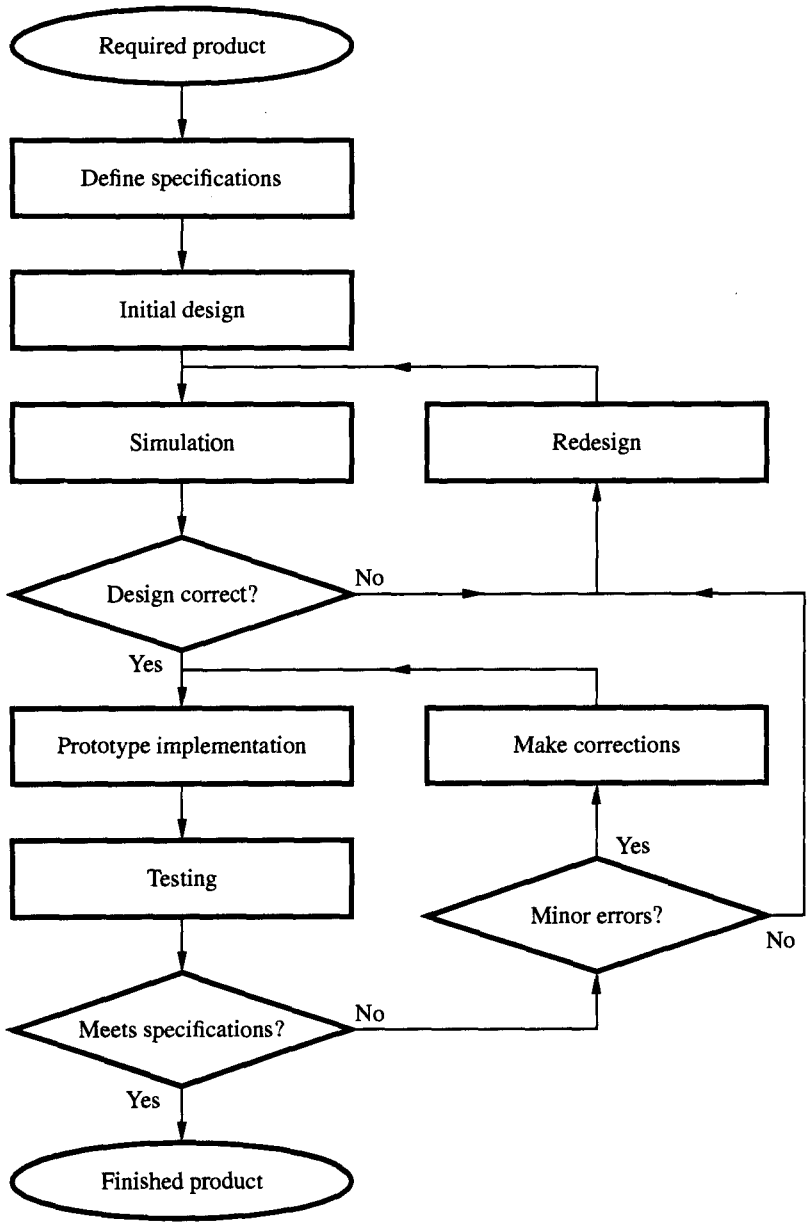


Figure 1.3 The development process.

of individual parts of the system to those that allow the entire system's structure to be represented in a computer. When the initial design is finished, the results must be verified against the original specifications. Traditionally, before the advent of CAD tools, this step involved constructing a physical model of the designed product, usually including just the key parts. Today it is seldom necessary to build a physical model. CAD tools enable

designers to simulate the behavior of incredibly complex products, and such simulations are used to determine whether the obtained design meets the required specifications. If errors are found, then appropriate changes are made and the verification of the new design is repeated through simulation. Although some design flaws may escape detection via simulation, usually all but the most subtle problems are discovered in this way.

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

1.3 DESIGN OF DIGITAL HARDWARE

Our previous discussion of the development process is relevant in a most general way. The steps outlined in Figure 1.3 are fully applicable in the development of digital hardware. Before we discuss the complete sequence of steps in this development environment, we should emphasize the iterative nature of the design process.

1.3.1 BASIC DESIGN LOOP

Any design process comprises a basic sequence of tasks that are performed in various situations. This sequence is presented in Figure 1.4. Assuming that we have an initial concept about what should be achieved in the design process, the first step is to generate an initial design. This step often requires a lot of manual effort because most designs have some specific goals that can be reached only through the designer's knowledge, skill, and intuition. The next step is the simulation of the design at hand. There exist excellent CAD tools to assist in this step. To carry out the simulation successfully, it is necessary to have adequate input conditions that can be applied to the design that is being simulated and later to the final product that has to be tested. Applying these input conditions, the simulator tries to verify that the designed product will perform as required under the original product specifications. If the simulation reveals some errors, then the design must be changed to overcome the problems. The redesigned version is again simulated to determine whether the errors have disappeared. This loop is repeated until the simulation indicates a successful design. A prudent designer expends considerable effort to remedy errors during simulation because errors are typically much harder to fix if they are discovered late in the design process. Even so, some errors may not be detected during simulation, in which case they have to be dealt with in later stages of the development cycle.

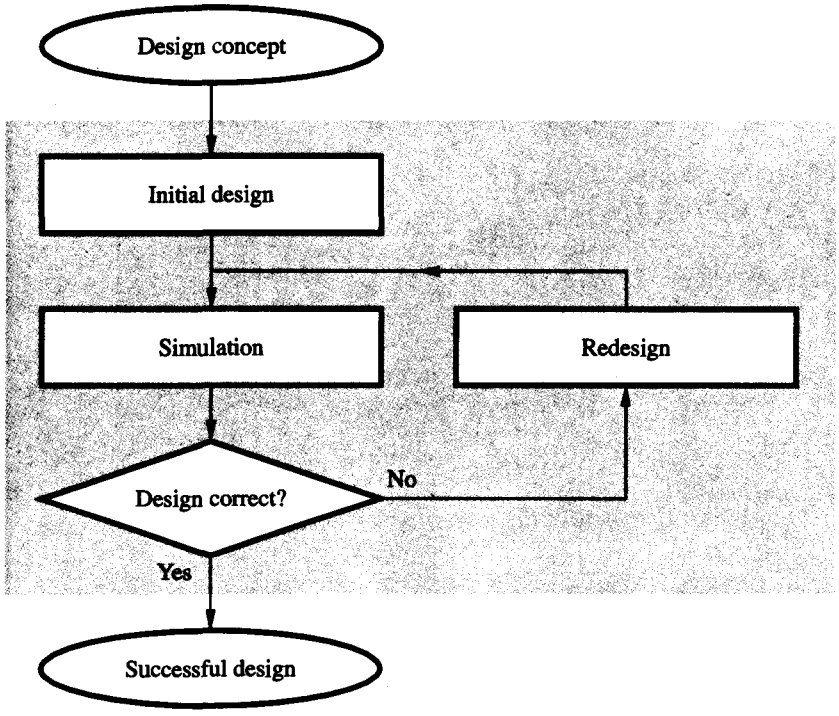


Figure 1.4 The basic design loop.

1.3.2 DESIGN OF A DIGITAL HARDWARE UNIT

Digital hardware products usually involve one or more PCBs that contain many chips and other components. Development of such products starts with the definition of the overall structure. Then the required integrated circuit chips are selected, and the PCBs that house and connect the chips together are designed. If the selected chips include PLDs or custom chips, then these chips must be designed before the PCB-level design is undertaken. Since the complexity of circuits implemented on individual chips and on the circuit boards is usually very high, it is essential to make use of good CAD tools.

An example of a PCB is given in Figure 1.5. The PCB is a part of a large computer system designed at the University of Toronto. This computer, called *NUMAchine* [4,5], is a *multiprocessor*, which means that it contains many processors that can be used together to work on a particular task. The PCB in the figure contains one processor chip and various memory and support chips. Complex logic circuits are needed to form the interface between the processor and the rest of the system. A number of PLDs are used to implement these logic circuits.

To illustrate the complete development cycle in more detail, we will consider the steps needed to produce a digital hardware unit that can be implemented on a PCB. This hardware

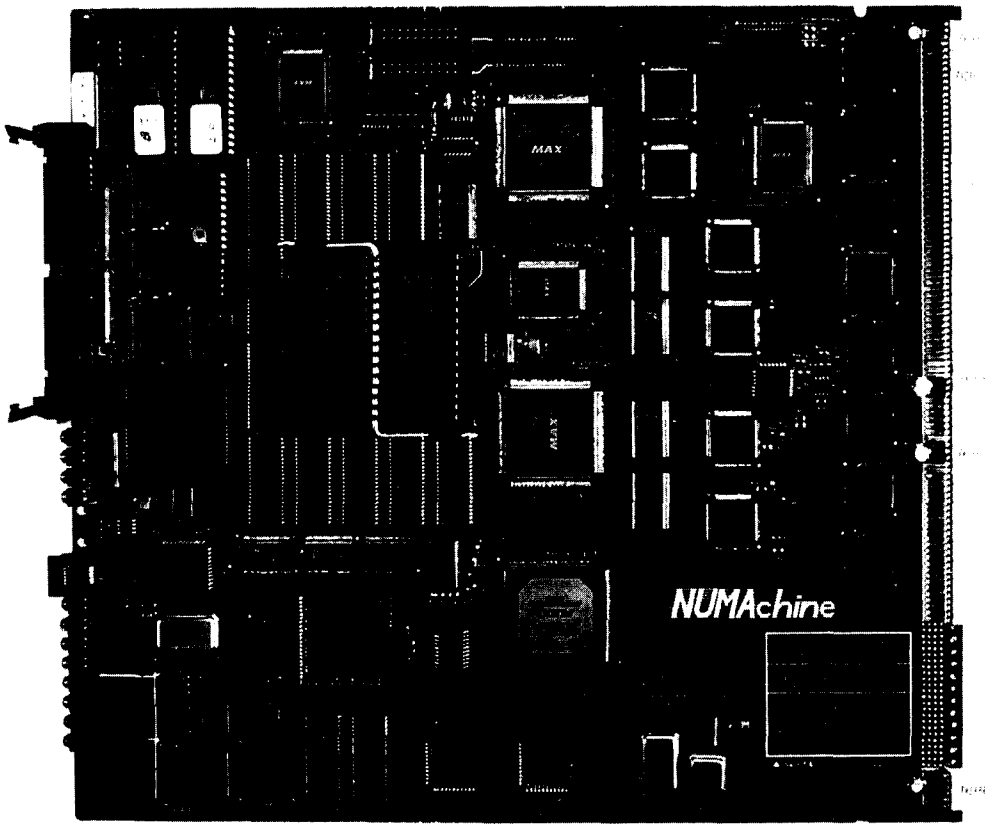


Figure 1.5 A printed circuit board.

could be viewed as a very complex logic circuit that performs the functions defined by the product specifications. Figure 1.6 shows the design flow, assuming that we have a design concept that defines the expected behavior and characteristics of this large circuit.

An orderly way of dealing with the complexity involved is to partition the circuit into smaller blocks and then to design each block separately. Breaking down a large task into more manageable smaller parts is known as the divide-and-conquer approach. The design of each block follows the procedure outlined in Figure 1.4. The circuitry in each block is defined, and the chips needed to implement it are chosen. The operation of this circuitry is simulated, and any necessary corrections are made.

Having successfully designed all blocks, the interconnection between the blocks must be defined, which effectively combines these blocks into a single large circuit. Now it is necessary to simulate this complete circuit and correct any errors. Depending on the errors encountered, it may be necessary to go back to the previous steps as indicated by the paths A, B, and C in the flowchart. Some errors may be caused by incorrect connections

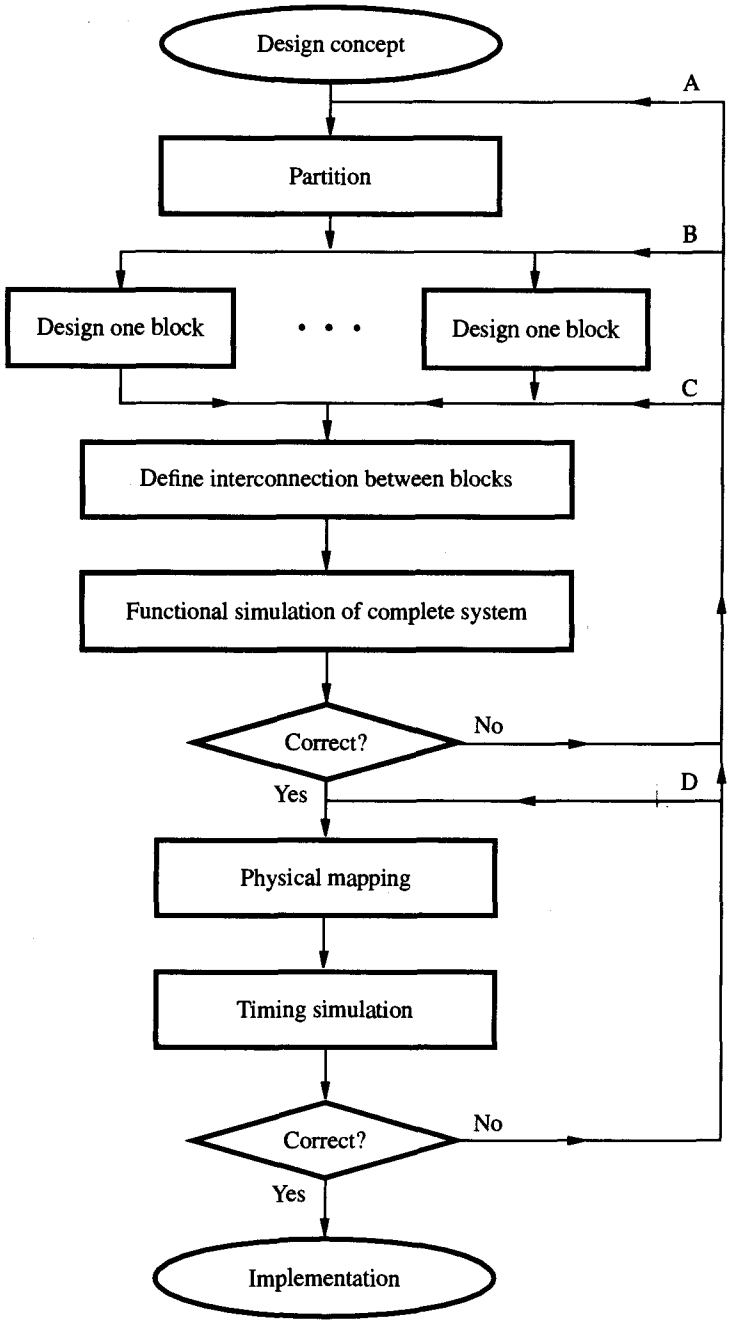


Figure 1.6 Design flow for logic circuits.

between the blocks, in which case these connections have to be redefined, following path C. Some blocks may not have been designed correctly, in which case path B is followed and the erroneous blocks are redesigned. Another possibility is that the very first step of partitioning the overall large circuit into blocks was not done well, in which case path A is followed. This may happen, for example, if none of the blocks implement some functionality needed in the complete circuit.

Successful completion of functional simulation suggests that the designed circuit will correctly perform all of its functions. The next step is to decide how to realize this circuit on a PCB. The physical location of each chip on the board has to be determined, and the wiring pattern needed to make connections between the chips has to be defined. We refer to this step as the *physical design* of the PCB. CAD tools are relied on heavily to perform this task automatically.

Once the placement of chips and the actual wire connections on the PCB have been established, it is desirable to see how this physical layout will affect the performance of the circuit on the finished board. It is reasonable to assume that if the previous functional simulation indicated that all functions will be performed correctly, then the CAD tools used in the physical design step will ensure that the required functional behavior will not be corrupted by placing the chips on the board and wiring them together to realize the final circuit. However, even though the functional behavior may be correct, the realized circuit may operate more slowly than desired and thus lead to inadequate performance. This condition occurs because the physical wiring on the PCB involves metal traces that present resistance and capacitance to electrical signals and thus may have a significant impact on the speed of operation. To distinguish between simulation that considers only the functionality of the circuit and simulation that also considers timing behavior, it is customary to use the terms *functional simulation* and *timing simulation*. A timing simulation may reveal potential performance problems, which can then be corrected by using the CAD tools to make changes in the physical design of the PCB.

Having completed the design process, the designed circuit is ready for physical implementation. The steps needed to implement a prototype board are indicated in Figure 1.7. A first version of the board is built and tested. Most minor errors that are detected can usually be corrected by making changes directly on the prototype board. This may involve changes in wiring or perhaps reprogramming some PLDs. Larger problems require a more substantial redesign. Depending on the nature of the problem, the designer may have to return to any of the points A, B, C, or D in the design process of Figure 1.6.

We have described the development process where the final circuit is implemented using many chips on a PCB. The material presented in this book is directly applicable to this type of design problem. However, for practical reasons the design examples that appear in the book are relatively small and can be realized in a single integrated circuit, either a custom-designed chip or a PLD. All the steps in Figure 1.6 are relevant in this case as well, with the understanding that the circuit blocks to be designed are on a smaller scale.

1.4 LOGIC CIRCUIT DESIGN IN THIS BOOK

In this book we use PLDs extensively to illustrate many aspects of logic circuit design. We selected this technology because it is widely used in real digital hardware products

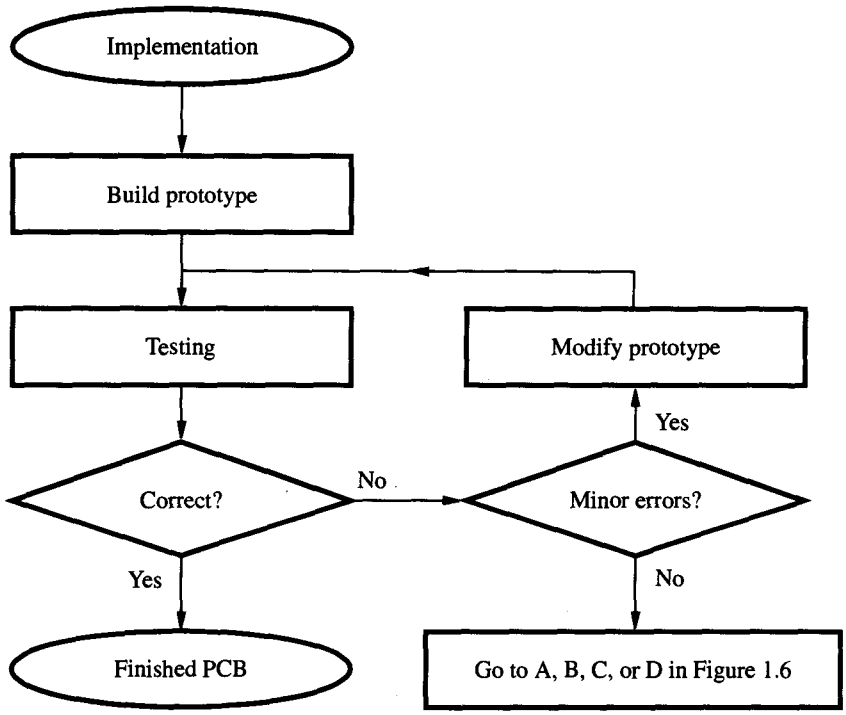


Figure 1.7 Completion of PCB development.

and because the chips are user programmable. PLD technology is particularly well suited for educational purposes because many readers have access to facilities for programming PLDs, which enables the reader to actually implement the sample circuits. To illustrate practical design issues, in this book we use two types of PLDs—they are the two types of devices that are widely used in digital hardware products today. One type is known as *complex programmable logic devices* (CPLDs) and the other as *field-programmable gate arrays* (FPGAs). These chips are introduced in Chapter 3.

To gain practical experience and a deeper understanding of logic circuits, we advise the reader to implement the examples in this book using CAD tools. Most of the major vendors of CAD systems provide their tools through university programs for educational use. Some examples are Altera, Cadence, Mentor Graphics, Synopsys, Synplicity, and Xilinx. The CAD systems offered by any of these companies can be used equally well with this book. For those who do not already have access to CAD tools, we include Altera's MAX+plusII CAD system on a CD-ROM. This industrial-quality software supports all phases of the design cycle and is powerful and easy to use. The software is easily installed on a personal computer, and we provide a sequence of complete step-by-step tutorials in Appendices B, C, and D to illustrate the use of CAD tools in concert with the book.

For educational purposes, some PLD manufacturers provide laboratory development printed circuit boards that include one or more PLD chips and an interface to a personal computer. Once a logic circuit has been designed using the CAD tools, the circuit can be

downloaded into a PLD on the board. Inputs can then be applied to the PLD by way of simple switches, and the generated outputs can be examined. These laboratory boards are described on the World Wide Web pages of the PLD suppliers.

1.5 THEORY AND PRACTICE

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

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

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

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

On a final note, 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.

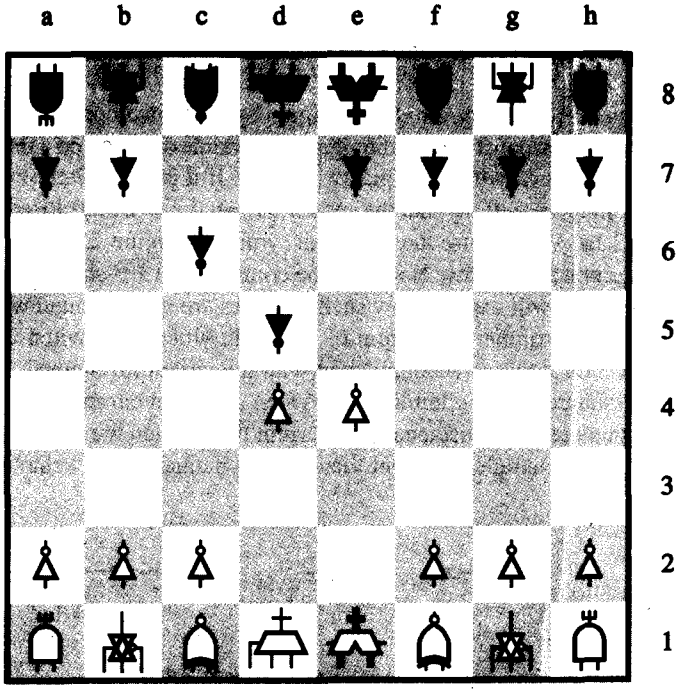
REFERENCES

1. Semiconductor Industry Association, "National Technology Roadmap for Semiconductors," http://www.semichips.org/
2. Altera Corporation, "APEX II Programmable Logic Devices," http://www.altera.com
3. Xilinx Corporation, "Virtex II Field Programmable Gate Arrays," http://www.xilinx.com
4. S. Brown, N. Manjikian, Z. Vranesic, S. Caranci, A. Grbic, R. Grindley, M. Gusat, K. Loveless, Z. Zilic, and S. Srbljic, "Experience in Designing a Large-Scale Multiprocessor Using Field-Programmable Devices and Advanced CAD Tools," 33rd IEEE Design Automation Conference, Las Vegas, June 1996.
5. A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic, "The Design and Implementation of the NUMAchine Multiprocessor," IEEE Design Automation Conference, San Francisco, June 1998.

chapter

2

INTRODUCTION TO LOGIC CIRCUITS



2. d2–d4, d7–d5

The study of logic circuits is motivated mostly by their use in digital computers. But such circuits also form the foundation of many other digital systems where performing arithmetic operations on numbers is not of primary interest. For example, in a myriad of control applications actions are determined by some simple logical operations on input information, without having to do extensive numerical computations.

Logic circuits perform operations on digital signals and are usually implemented as electronic circuits where the signal values are restricted to a few discrete values. In *binary* logic circuits there are only two values, 0 and 1. In *decimal* logic circuits there are 10 values, from 0 to 9. Since each signal value is naturally represented by a digit, such logic circuits are referred to as *digital circuits*. In contrast, there exist *analog circuits* where the signals may take on a continuous range of values between some minimum and maximum levels.

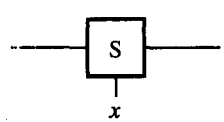
In this book we deal with binary circuits, which have the dominant role in digital technology. We hope to provide the reader with an understanding of how these circuits work, how are they represented in mathematical notation, and how are they designed using modern design automation techniques. We begin 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 Chapter 3 we will explain how such switches are implemented with transistors.

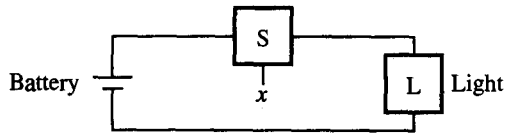
Consider a simple application of a switch, where the switch turns a small lightbulb on or off. This action is accomplished with the circuit in Figure 2.2*a*. A battery provides the power source. The lightbulb glows when sufficient current passes through its filament, which is an electrical resistance.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

(a) Two states of a switch

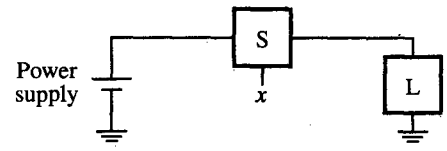


(b) Symbol for a switch

Figure 2.1 A binary switch.



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 2.2 A light controlled by a switch.

switch control x. The output is defined as the state (or condition) of the light L. If the light is on, we will say that $L = 1$. If the the light is off, we will say that $L = 0$. Using this convention, we can describe the state of the light L as a function of the input variable x. Since $L = 1$ if $x = 1$ and $L = 0$ if $x = 0$, we can say that

$$L(x) = x$$

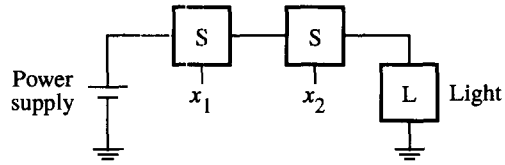
This simple *logic expression* describes the output as a function of the input. We say that $L(x) = x$ is a *logic function* and that x is an *input variable*.

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, perhaps 5 volts. One side of the power supply is connected to ground, as shown in Figure 2.2*b*. The ground connection may also be used as the return path for the current, to close the loop, which is achieved by connecting one side of the light to ground as indicated in the figure. Of course, the light can also be connected by a wire directly to the grounded side of the power supply, as in Figure 2.2*a*.

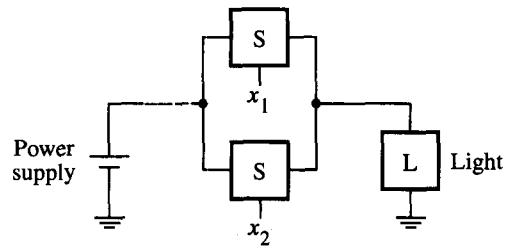
Consider now the possibility of using two switches to control the state of the light. Let x_1 and x_2 be the control inputs for these switches. The switches can be connected either in series or in parallel as shown in Figure 2.3. Using a series connection, the light will be turned on only if both switches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

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

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



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3 Two basic functions.

The parallel connection of two switches is given in Figure 2.3*b*. In this case the light will be on if either 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*.

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

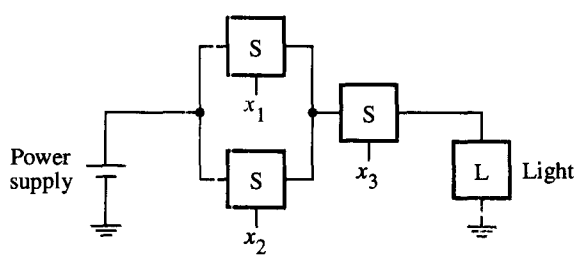


Figure 2.4 A series-parallel connection.

for the implementation of all logic circuits. Figure 2.4 illustrates how three switches can be used to control the light in a more complex way. This series-parallel connection of switches realizes the logic function

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

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

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

$$L(x) = \bar{x}$$

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

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

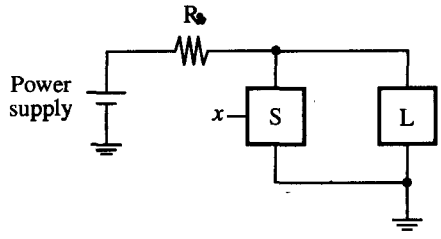


Figure 2.5 An inverting circuit.

placed after the variable, or the exclamation mark (!) or the tilde character (~) is placed in front of the variable to denote the complementation. Thus the following are equivalent:

$$\bar{x} = x' = !x = \sim x$$

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

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

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

$$\overline{x_1 + x_2} = (x_1 + x_2)' = !(x_1 + x_2) = \sim(x_1 + x_2)$$

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

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

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

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

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

Figure 2.7 Three-input AND and OR operations.

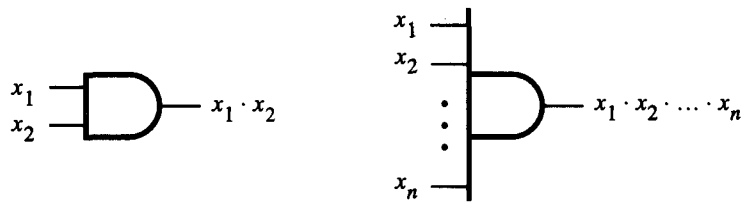
The AND and OR operations can be extended to n variables. An AND function of variables $x_1, x_2, \ldots, x_n$ has the value 1 only if all n variables are equal to 1. An OR function of variables $x_1, x_2, \ldots, x_n$ has the value 1 if at least one, or more, of the variables is equal to 1.

2.4 LOGIC GATES AND NETWORKS

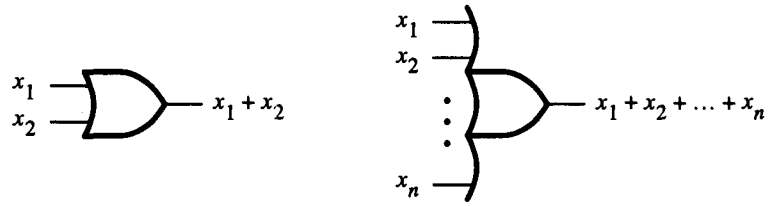
The three basic logic operations introduced in the previous sections can be used to implement logic functions of any complexity. A complex function may require many of these basic operations for its implementation. Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*. A logic gate has one or more inputs and one output that is a function of its inputs. It is often convenient to describe a logic circuit by drawing a circuit diagram, or *schematic*, consisting of graphical symbols representing the logic gates. The graphical symbols for the AND, OR, and NOT gates are shown in Figure 2.8. The figure indicates on the left side how the AND and OR gates are drawn when there are only a few inputs. On the right side it shows how the symbols are augmented to accommodate a greater number of inputs. We will show how logic gates are built using transistors in Chapter 3.

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

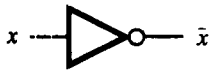
In technical jargon a network of gates is often called a *logic network* or simply a *logic circuit*. We will use these terms interchangeably.



(a) AND gates



(b) OR gates



(c) NOT gate

Figure 2.8 The basic gates.

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 determine its functional behavior, we can consider what happens if we apply all possible input signals to it. Suppose that we start by making $x_1 = x_2 = 0$. This forces the output of the NOT gate

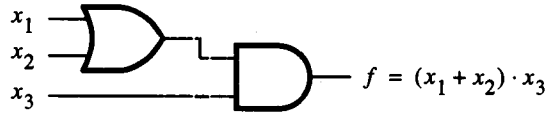
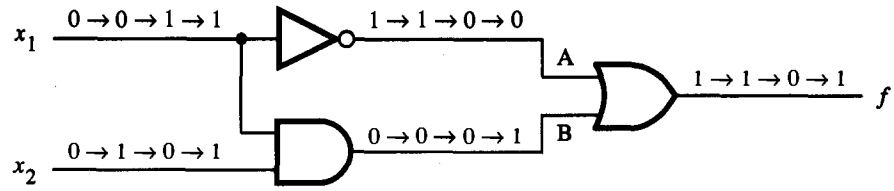


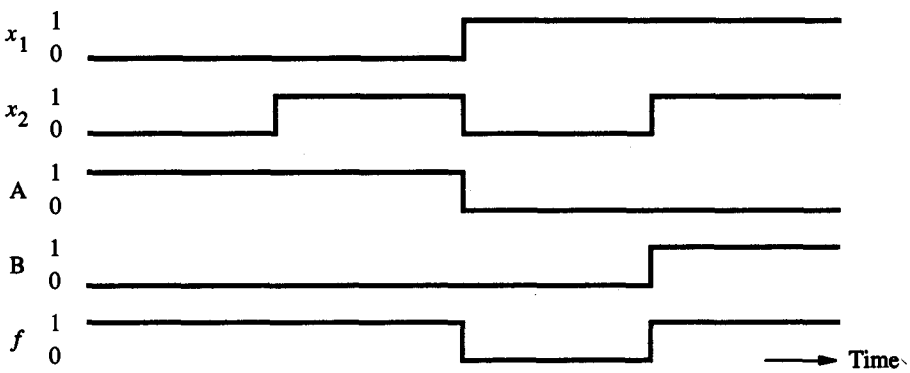
Figure 2.9 The function from Figure 2.4.



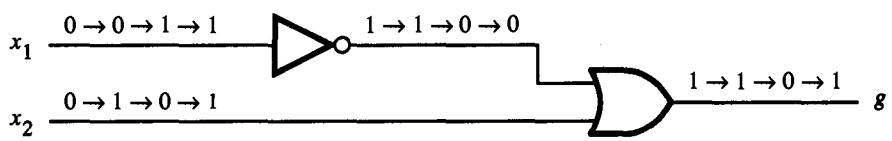
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

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

(b) Truth table for f



(c) Timing diagram



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

Figure 2.10 An example of logic networks.

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 let $x_1 = 0$ and $x_2 = 1$, then 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 period. 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.

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? Many techniques exist for synthesizing logic functions. We will discuss the main approaches in Chapter 4. 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 $f(x_1, x_2) = \bar{x}_1 + x_1 \cdot x_2$ and $g(x_1, x_2) = \bar{x}_1 + x_2$, there must exist some rules that can be used to show the equivalence

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

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 the next section we will discuss a mathematical approach for dealing with logic functions, which provides the basis for modern design techniques.

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 B involves elements that take on one of two values, 0 and 1. Assume that the following axioms are true:

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

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

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

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

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

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

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

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

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 variable in B, then the following theorems hold:

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

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

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

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

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

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

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

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

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

It is easy to prove the validity of these theorems by perfect induction, that is, by substituting the values $x = 0$ and $x = 1$ into the expressions and using the axioms given above. For example, in theorem 5*a*, if $x = 0$, then the theorem states that $0 \cdot 0 = 0$, which is true

according to axiom 1*a*. Similarly, if $x = 1$, then theorem 5*a* states that $1 \cdot 0 = 0$, which is also true according to axiom 3*a*. The reader should verify that theorems 5*a* to 9 can be proven in this way.

Duality

Notice that we have listed the axioms and the single-variable theorems in pairs. This is done to reflect the important *principle of duality*. Given a logic expression, its *dual* is obtained by replacing all + operators with · 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 will 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 the variables in B, then the following properties hold:

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

Again, we can prove the validity of these properties either by perfect induction or by performing algebraic manipulation. Figure 2.11 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.

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

Figure 2.11 Proof of DeMorgan's theorem in 15*a*.

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

Example 2.1

Let us prove the validity of the logic equation

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

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

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

Applying the distributive property again yields

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

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

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

From 6*b* it follows that

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

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

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

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

Example 2.2

Consider the logic equation

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

The left-hand side can be manipulated as follows

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

$$= x_1 \cdot 1 + \bar{x}_2 \cdot 1 \quad \text{using } 8b$$
$$= x_1 + \bar{x}_2 \quad \text{using } 6a$$

The right-hand side can be manipulated as

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

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

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

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

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

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 2.1 and 2.2 illustrate the purpose of the axioms, theorems, and properties as a mechanism for algebraic manipulation. Even these simple examples suggest that it is impractical to deal with highly complex expressions in this way. However, these theorems and properties provide the basis for automating the synthesis of logic functions in CAD tools. To understand what can be achieved using these tools, the designer needs to be aware of the fundamental concepts.

2.5.1 THE VENN DIAGRAM

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

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

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 2.12. The universe B is represented by a square. Then the constants 1 and 0 are represented as shown in parts (a) and (b) of the figure. A variable, say, x, is represented by a circle, such that the area inside the circle corresponds to $x = 1$, while the area outside the circle corresponds to $x = 0$. This is illustrated in part (c). An expression involving one or more variables is depicted by

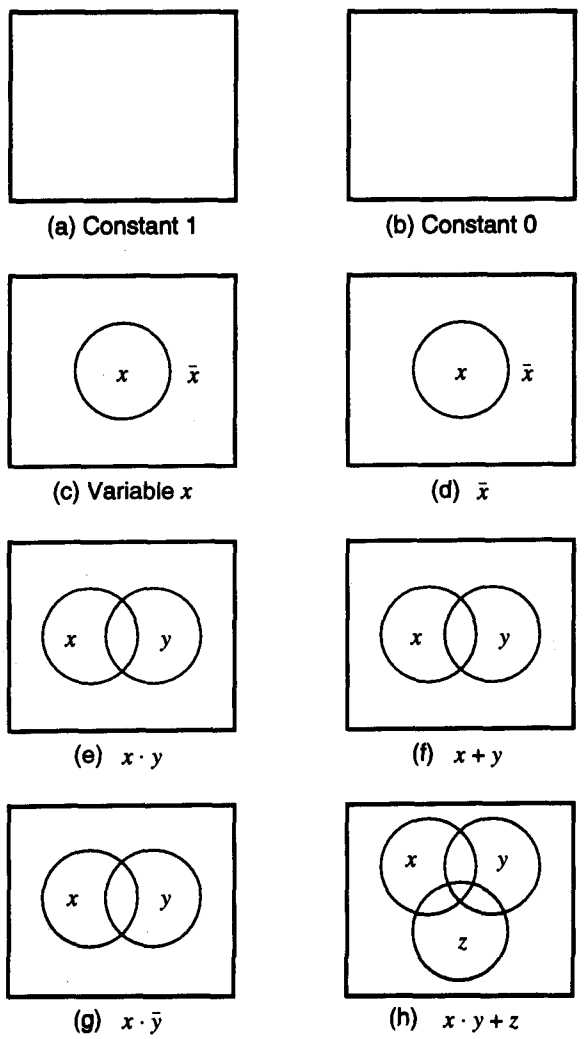


Figure 2.12 The Venn diagram representation.

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

To see how we can use Venn diagrams to verify the equivalence of two expressions, let us demonstrate the validity of the distributive property, 12*a*, in section 2.5. Figure 2.13 gives the construction of the left and right sides of the identity that defines the property

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

Part (*a*) shows the area where $x = 1$. Part (*b*) indicates the area for $y + z$. Part (*c*) gives the diagram for $x \cdot (y + z)$, the intersection of shaded areas in parts (*a*) and (*b*). The right-hand

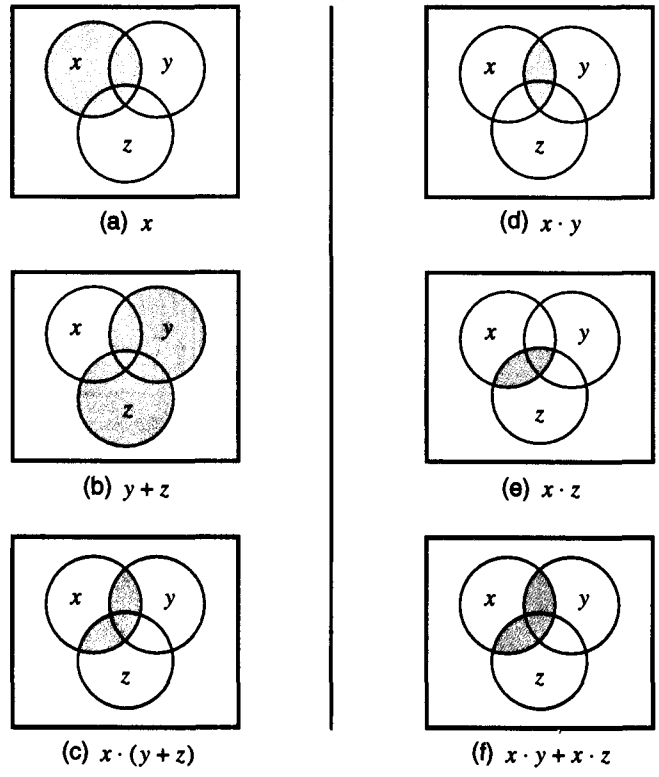


Figure 2.13 Verification of the distributive property $x \cdot (y + z) = x \cdot y + x \cdot z$.

side is constructed in parts (*d*), (*e*), and (*f*). Parts (*d*) and (*e*) describe the terms $x \cdot y$ and $x \cdot z$, respectively. The union of the shaded areas in these two diagrams then corresponds to the expression $x \cdot y + x \cdot z$, as seen in part (*f*). Since the shaded areas in parts (*c*) and (*f*) are identical, it follows that the distributive property is valid.

As another example, consider the identity

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

which is illustrated in Figure 2.14. Notice that this identity states that the term $y \cdot z$ is fully covered by the terms $x \cdot y$ and $\bar{x} \cdot z$; therefore, this term can be omitted.

The reader should use the Venn diagram to prove some other identities. It is particularly instructive to prove the validity of DeMorgan's theorem in this way.

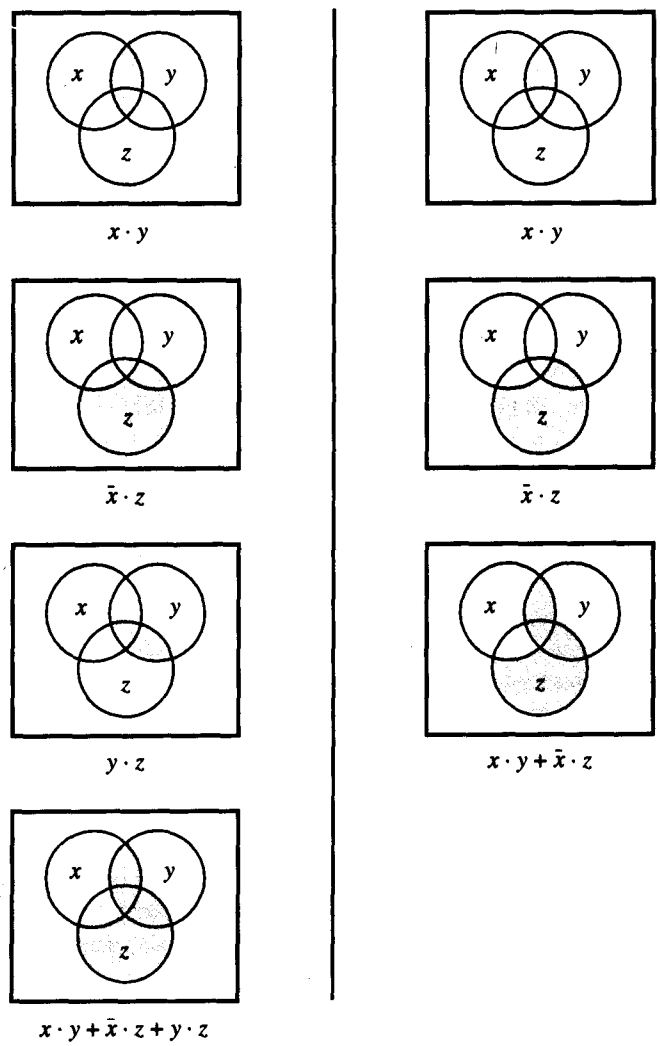


Figure 2.14 Verification of $x \cdot y + \bar{x} \cdot z + y \cdot z = x \cdot y + \bar{x} \cdot z$.

2.5.2 NOTATION AND TERMINOLOGY

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

When dealing with digital circuits, most of the time the $+$ symbol obviously represents the OR operation. However, when the task involves the design of logic circuits that perform arithmetic operations, some confusion may develop about the use of the $+$ symbol. To avoid such confusion, an alternative set of symbols exists for the AND and OR operations. It is quite common to use the $\wedge$ symbol to denote the AND operation, and the $\vee$ symbol for the OR operation. Thus, instead of $x_1 \cdot x_2$, we can write $x_1 \wedge x_2$, and instead of $x_1 + x_2$, we can write $x_1 \vee x_2$.

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

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

is a sum of three product terms, whereas the expression

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

is a product of three sum terms.

2.5.3 PRECEDENCE OF OPERATIONS

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

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

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

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

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

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

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 be open (0) or closed (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. Another way of stating the required functional behavior of this circuit is that the output must be equal to 0 if the switch x_1 is closed and x_2 is open; otherwise, the output must be 1. We can express the required behavior using a truth table, as shown in Figure 2.15.

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

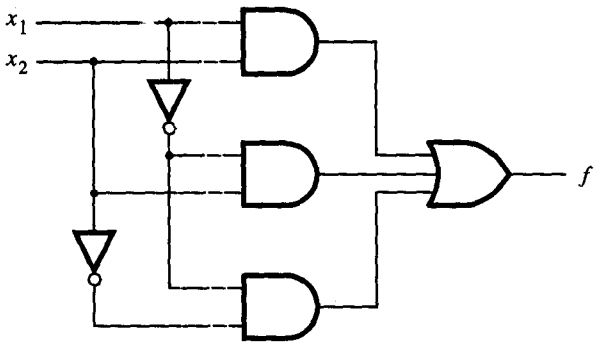
$$f(x_1, x_2) = x_1x_2 + \bar{x}_1\bar{x}_2 + \bar{x}_1x_2$$

The logic network that corresponds to this expression is shown in Figure 2.16*a*.

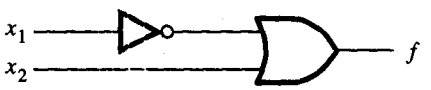
Although this network implements f correctly, it is not the simplest such network. To find a simpler network, we can manipulate the obtained expression using the theorems and

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

Figure 2.15 A function to be synthesized.



(a) Canonical sum-of-products



(b) Minimal-cost realization

Figure 2.16 Two implementations of the function in Figure 2.15.

properties from section 2.5. According to theorem 7*b*, we can replicate any term in a logical sum expression. Replicating the third product term, the above expression becomes

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

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

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

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

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

Applying theorem 8*b* we get

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

Finally, theorem 6*a* leads to

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

The network described by this expression is given in Figure 2.16*b*. Obviously, the cost of this network is much less than the cost of the network in part (*a*) of the figure.

This simple example illustrates two things. First, a straightforward implementation of a function can be obtained by using a product term (AND gate) for each row of the truth table for which the function is equal to 1. Each product term contains all input variables,

and it is formed such that if the input variable x_i is equal to 1 in the given row, then x_i is entered in the term; if $x_i = 0$, then $\bar{x}_i$ 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 2.16 from the truth table in Figure 2.15. 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.

2.6.1 SUM-OF-PRODUCTS AND PRODUCT-OF-SUMS FORMS

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

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

Minterms

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

To illustrate this concept, consider the truth table in Figure 2.17. We have numbered the rows of the table from 0 to 7, so that we can refer to them easily. (The reader who is already familiar with the binary number representation will realize that the row numbers chosen are just the numbers represented by the bit patterns of variables x_1, x_2, and x_3; we will discuss number representation in Chapter 5.) The figure shows all minterms for the three-variable table. For example, in the first row the variables have the values $x_1 = x_2 = x_3 = 0$, which leads to the minterm $\bar{x}_1\bar{x}_2\bar{x}_3$. In the second row $x_1 = x_2 = 0$ and $x_3 = 1$, which gives the minterm $\bar{x}_1\bar{x}_2x_3$, and so on. To be able to refer to the individual minterms easily, it is convenient to identify each minterm by an index that corresponds to the row numbers shown in the figure. We will use the notation m_i to denote the minterm for row number i. Thus $m_0 = \bar{x}_1\bar{x}_2\bar{x}_3$, $m_1 = \bar{x}_1\bar{x}_2x_3$, and so on.

Sum-of-Products Form

A function f can be represented by an expression that is a sum of minterms, where each minterm is ANDed with the value of f for the corresponding valuation of input variables. For example, the two-variable minterms are $m_0 = \bar{x}_1\bar{x}_2$, $m_1 = \bar{x}_1x_2$, $m_2 = x_1\bar{x}_2$, and $m_3 = x_1x_2$. The function in Figure 2.15 can be represented as

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

Figure 2.17 Three-variable minterms and maxterms.

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

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

Any function f can be represented by a sum of minterms that correspond to the rows in the truth table for which $f = 1$. The resulting implementation is functionally correct and unique, but it is not necessarily the lowest-cost implementation of f. A logic expression consisting of product (AND) terms that are summed (ORed) is said to be of the *sum-of-products* 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 2.16, 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 2.18. To synthesize this function, we have to include the minterms m_1, m_4, m_5, and m_6. Copying these minterms from Figure 2.17 leads to the following canonical sum-of-products expression for f

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

This expression can be manipulated as follows

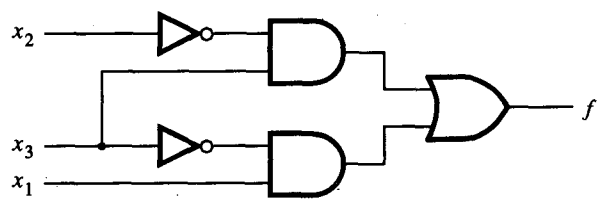
$$\begin{aligned} f &= (\bar{x}_1 + x_1)\bar{x}_2x_3 + x_1(\bar{x}_2 + x_2)\bar{x}_3 \\ &= 1 \cdot \bar{x}_2x_3 + x_1 \cdot 1 \cdot \bar{x}_3 \\ &= \bar{x}_2x_3 + x_1\bar{x}_3 \end{aligned}$$

This is the minimum-cost sum-of-products expression for f. It describes the circuit shown in Figure 2.19*a*. A good indication of the *cost* of a logic circuit is the total number of gates

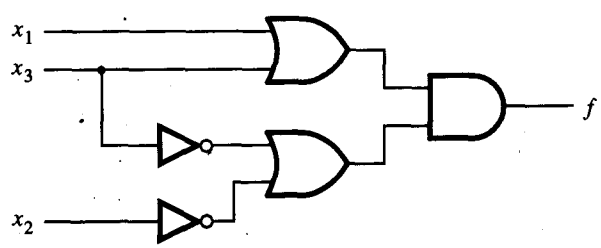
Row number	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

Figure 2.18 A three-variable function.

plus the total number of inputs to all gates in the circuit. Using this measure, the cost of the network in Figure 2.19*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.



(a) A minimal sum-of-products realization



(b) A minimal product-of-sums realization

Figure 2.19 Two realizations of the function in Figure 2.18.

Minterms, with their row-number subscripts, can also be used to specify a given function in a more concise form. For example, the function in Figure 2.18 can be specified as

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

or even more simply as

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

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

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 2.17. We will refer to a maxterm M_j by the same row number as its corresponding minterm m_j as shown in the figure.

Product-of-Sums Form

If a given function f is specified by a truth table, then its complement $\bar{f}$ can be represented by a sum of minterms for which $\bar{f} = 1$, which are the rows where $f = 0$. For example, for the function in Figure 2.15

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

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

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

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

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

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

As another example, consider again the function in Figure 2.18. The complement of this function can be represented as

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

Then f can be expressed as

$$\begin{aligned}f &= \overline{m_0 + m_2 + m_3 + m_7} \\ &= \bar{m}_0 \cdot \bar{m}_2 \cdot \bar{m}_3 \cdot \bar{m}_7\end{aligned}$$

$$= M_0 \cdot M_2 \cdot M_3 \cdot M_7$$
$$= (x_1 + x_2 + x_3)(x_1 + \bar{x}_2 + x_3)(x_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)$$

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* form. If each sum term is a maxterm, then the expression is called a *canonical product-of-sums* for the given function. Any function f can be synthesized by finding its canonical product-of-sums. This involves taking the maxterm for each row in the truth table for which $f = 0$ and forming a product of these maxterms.

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

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

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

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

The corresponding network is given in Figure 2.19*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 2.19*a*, the reader should not assume that the cost of a network derived in the sum-of-products form will in general be equal to the cost of a corresponding circuit derived in the product-of-sums form.

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

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

or more simply

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

The Π sign denotes the logical product operation.

The preceding discussion has shown how logic functions can be realized in the form of logic circuits, consisting of networks of gates that implement basic functions. A given function may be realized with circuits of a different structure, 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 the most important techniques for finding minimum-cost implementations in Chapter 4.

2.7 NAND AND NOR LOGIC NETWORKS

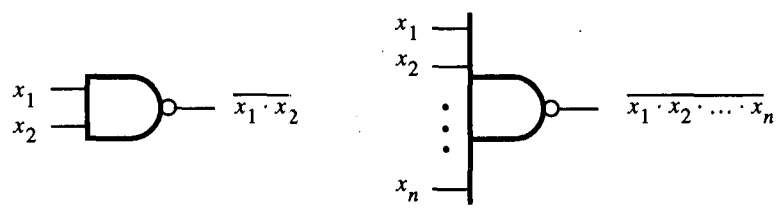
We have discussed the use of AND, OR, and NOT gates in the synthesis of logic circuits. There are other basic logic functions that are also used for this purpose. Particularly useful are the NAND and NOR functions which are obtained by complementing the output generated by AND and OR operations, respectively. These functions are attractive because they are implemented with simpler electronic circuits than the AND and OR functions, as

we will see in Chapter 3. Figure 2.20 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.

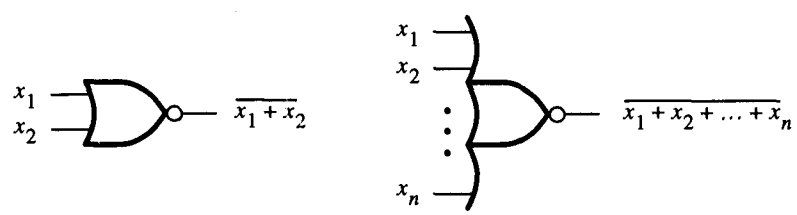
If NAND and NOR gates are realized with simpler circuits than AND and OR gates, then we should ask whether these gates can be used directly in the synthesis of logic circuits. In section 2.5 we introduced DeMorgan's theorem. Its logic gate interpretation is shown in Figure 2.21. Identity 15*a* is interpreted 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.

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 22 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 2.21*a*, the

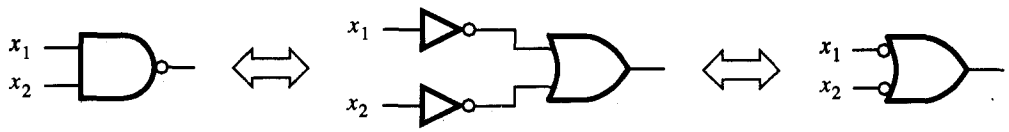


(a) NAND gates

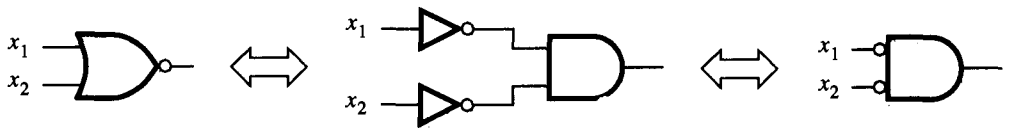


(b) NOR gates

Figure 2.20 NAND and NOR gates.



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



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

Figure 2.21 DeMorgan's theorem in terms of logic gates.

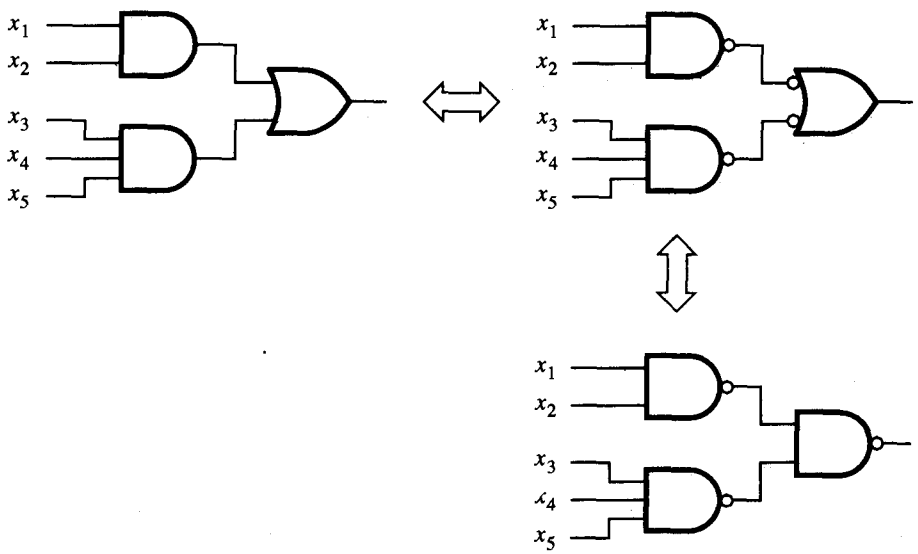


Figure 2.22 Using NAND gates to implement a sum-of-products.

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 2.22. This example shows that any AND-OR network can be implemented as a NAND-NAND network having the same topology.

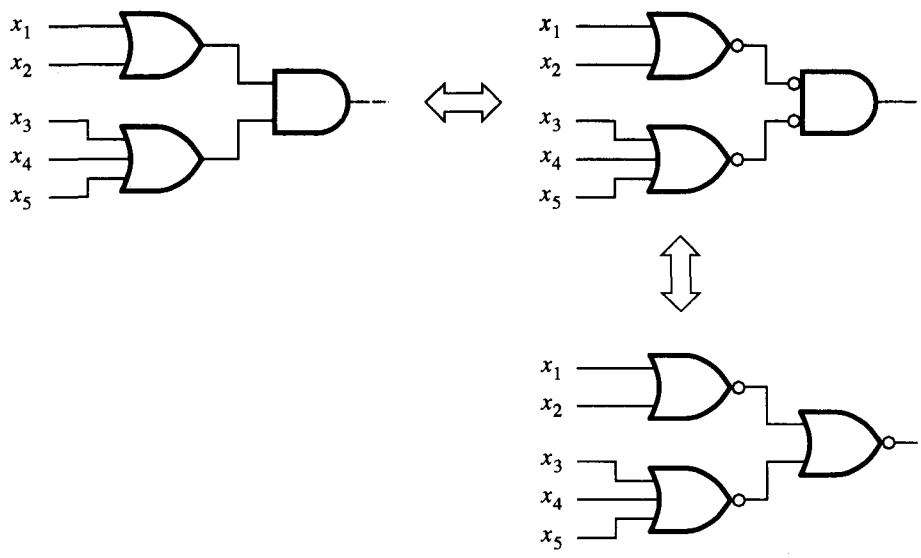


Figure 2.23 Using NOR gates to implement a product-of-sums.

Figure 2.23 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 2.22 except that now the identity in Figure 2.21*b* is applied. The conclusion is that any OR-AND network can be implemented as a NOR-NOR network having the same topology.

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 two 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 2.24. The canonical sum-of-products expression for the specified function is

$$\begin{aligned} f &= m_1 + m_2 + m_4 + m_7 \\ &= \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + x_1\bar{x}_2\bar{x}_3 + x_1x_2x_3 \end{aligned}$$

This expression cannot be simplified into a lower-cost sum-of-products expression. The resulting circuit is shown in Figure 2.25*a*.

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

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

The resulting circuit is depicted in Figure 2.25*b*. It has the same cost as the circuit in part (*a*) of the figure.

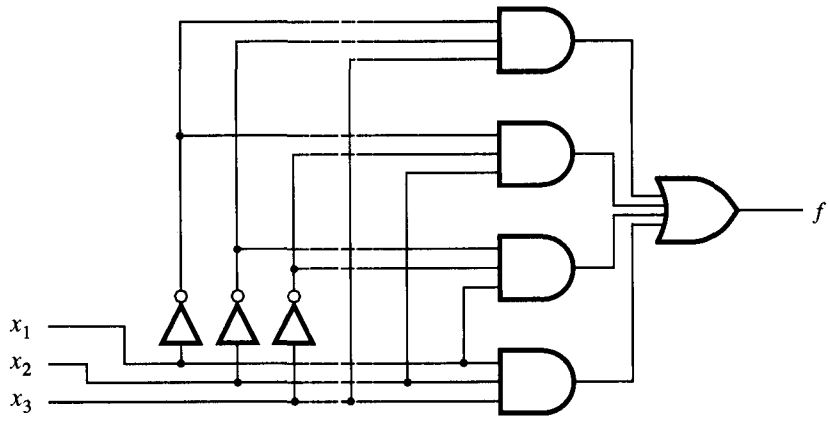
When the designed circuit is implemented, it can be tested by applying the various input valuations to the circuit and checking whether the output corresponds to the values specified in the truth table. A straightforward approach is to check that the correct output is produced for all eight possible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

2.8.2 MULTIPLEXER CIRCUIT

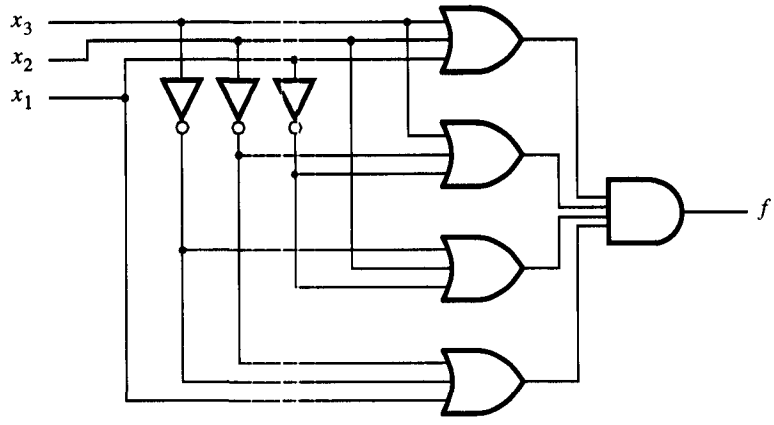
In computer systems it is often necessary to choose data from exactly one of a number of possible sources. Suppose that there are two sources of data, provided as input signals x_1 and x_2. The values of these signals change in time, perhaps at regular intervals. Thus

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.24 Truth table for the three-way light control.



(a) Sum-of-products realization



(b) Product-of-sums realization

Figure 2.25 Implementation of the function in Figure 2.24.

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 2.26*a*. From the truth table, we derive the canonical sum of products

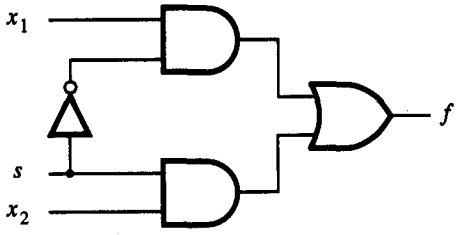
$$f(s, x_1, x_2) = \bar{s}x_1\bar{x}_2 + \bar{s}x_1x_2 + s\bar{x}_1x_2 + sx_1x_2$$

Using the distributive property, this expression can be written as

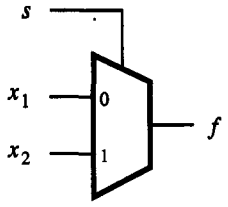
$$f = \bar{s}x_1(\bar{x}_2 + x_2) + s(\bar{x}_1 + x_1)x_2$$

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

(a) Truth table



(b) Circuit



(c) Graphical symbol

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

(d) More compact truth-table representation

Figure 2.26 Implementation of a multiplexer.

Applying theorem 8*b* yields

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

Finally, theorem 6*a* gives

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

A circuit that implements this function is shown in Figure 2.26*b*. Circuits of this type are used so extensively that they are given a special name. A circuit that generates an output that exactly reflects the state of one of a number of data inputs, based on the value of one or more selection control inputs, is called a *multiplexer*. We say that a multiplexer circuit "multiplexes" input signals onto a single output.

In this example we derived a multiplexer with two data inputs, which is referred to as a "2-to-1 multiplexer." A commonly used graphical symbol for the 2-to-1 multiplexer is shown in Figure 2.26*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 2.26*d*. In later chapters we will have occasion to use such representation.

We showed how a multiplexer can be built using AND, OR, and NOT gates. The same circuit structure can be used to implement the multiplexer using NAND gates, as explained in section 2.7. In Chapter 3 we will show other possibilities for constructing multiplexers. In Chapter 6 we will discuss the use of multiplexers in considerable detail.

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

2.9 INTRODUCTION TO CAD TOOLS

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

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

2.9.1 DESIGN ENTRY

The starting point in the process of designing a logic circuit is the conception of what the circuit is supposed to do and the formulation of its general structure. This step is done manually by the designer because it requires design experience and intuition. The rest of the design process is done with the aid of CAD tools. The first stage of this process

involves entering into the CAD system a description of the circuit being designed. This stage is called *design entry*. We will describe three design entry methods: using truth tables, using schematic capture, and writing source code in a hardware description language.

Design Entry with Truth Tables

We have already seen that any logic function of a few variables can be described conveniently by a truth table. Many CAD systems allow design entry using truth tables, where the table is specified as a plain text file. Alternatively, it may also be possible to specify a truth table as a set of waveforms in a timing diagram. We illustrated the equivalence of these two ways of representing truth tables in the discussion of Figure 2.10. The CAD system provided with this book supports both methods of using truth tables for design entry. Figure 2.27 shows an example in which the *Waveform Editor* is used to draw the timing diagram in Figure 2.10. The CAD system is capable of transforming this timing diagram automatically into a network of logic gates equivalent to that shown in Figure 2.10*d*.

Because truth tables are practical only for functions with a small number of variables, this design entry method is not appropriate for large circuits. It can, however, be applied for a small logic function that is part of a larger circuit. In this case the truth table becomes a subcircuit that can be interconnected to other subcircuits and logic gates. The most commonly used type of CAD tool for interconnecting such circuit elements is called a *schematic capture* tool. The word *schematic* refers to a diagram of a circuit in which circuit elements, such as logic gates, are depicted as graphical symbols and connections between circuit elements are drawn as lines.

Schematic Capture

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

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

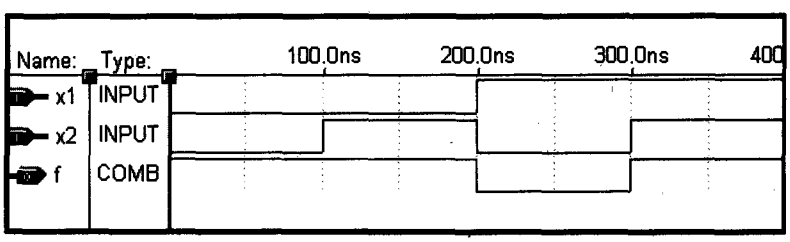


Figure 2.27 Screen capture of the Waveform Editor.

Figure 2.28 gives an example of a hierarchical design created with the schematic capture tool, provided with the CAD system, called the *Graphic Editor*. The circuit includes a subcircuit represented as a rectangular graphical symbol. This subcircuit represents the logic function entered by way of the timing diagram in Figure 2.27. Note that the complete circuit implements the function $f = \bar{x}_1 + x_2\bar{x}_3$.

In comparison to design entry with truth tables, the schematic-capture facility is more amenable for dealing with larger circuits. A disadvantage of using schematic capture is that every commercial tool of this type has a unique user interface and functionality. Therefore, extensive training is often required for a designer to learn how to use such a tool, and this training must be repeated if the designer switches to another tool at a later date. Another drawback is that the graphical user interface for schematic capture becomes awkward to use when the circuit being designed is large. A useful method for dealing with large circuits is to write source code using a hardware description language to represent the circuit.

Hardware Description Languages

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

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

In comparison to performing schematic capture, using Verilog offers a number of advantages. Because it is supported by most companies that offer digital hardware technology,

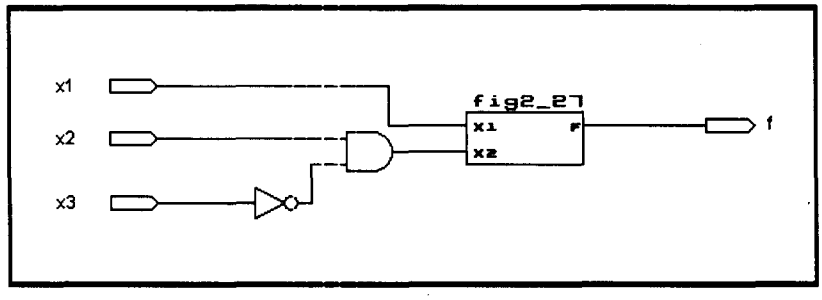


Figure 2.28 Screen capture of the Graphic Editor.

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

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

Similar to the way in which large circuits are handled in schematic capture, Verilog code can be written in a modular way that facilitates hierarchical design. Both small and large logic circuit designs can be efficiently represented in Verilog code. Verilog has been used to define circuits such as microprocessors with millions of transistors.

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

2.9.2 SYNTHESIS

In section 2.4.1 we said that synthesis is the process of generating a logic circuit from a truth table. Synthesis CAD tools perform this process automatically. However, the synthesis tools also handle many other tasks. The process of *translating*, or *compiling*, Verilog code into a network of logic gates is part of synthesis.

When the Verilog code representing a circuit is passed through initial synthesis tools, the output is a lower-level description of the circuit. For simplicity we will assume that this process produces a set of logic expressions that describe the logic functions needed to realize the circuit. These expressions are then manipulated further by the synthesis tools. If the design entry is performed using schematic capture, then the synthesis tools produce a set of logic equations representing the circuit from the schematic diagram. Similarly, if truth tables are used for design entry, then the synthesis tools generate expressions for the logic functions represented by the truth tables.

Regardless of what type of design entry is used, the initial logic expressions produced by the synthesis tools are not likely to be in an optimal form. Because these expressions reflect the designer's input to the CAD tools, it is difficult for a designer to manually produce optimal results, especially for large circuits. One of the most important tasks of the synthesis tools is to manipulate the user's design to automatically produce an equivalent but better circuit. This step of synthesis is called *logic synthesis*, or *logic optimization*.

The measure of what makes one circuit better than another depends on the particular needs of a design project and the technology chosen for implementation. In section 2.6 we suggested that a good circuit might be one that has the lowest cost. There are other possible optimization goals, which are motivated by the type of hardware technology

used for implementation of the circuit. We will discuss implementation technologies in Chapter 3 and return to the issue of optimization goals in Chapter 4.

After logic synthesis the optimized circuit is still represented in the form of logic equations. The final task in the synthesis process is to determine exactly how the circuit will be realized in a specific hardware technology. This task involves deciding how each logic function, represented by an expression, should be implemented using whatever physical resources are available in the technology. The task involves two steps called *technology mapping*, followed by *layout synthesis*, or *physical design*. We will discuss these steps in detail in Chapter 4.

2.9.3 FUNCTIONAL SIMULATION

Once the design entry and synthesis are complete, it is useful to verify that the designed circuit functions as expected. The tool that performs this task is called a *functional simulator*, and it uses two types of information. First, the user's initial design is represented by the logic equations generated during synthesis. Second, the user specifies valuations of the circuit's inputs that should be applied to these equations during simulation. For each valuation, the simulator evaluates the outputs produced by the equations. The output of the simulation is provided either in truth-table form or as a timing diagram. The user examines this output to verify that the circuit operates as required.

The logic equations used by the simulator are those produced by the synthesis tools before any optimizations are applied during logic synthesis. There would be no advantage in using the optimized form of the equations, because the intent is to evaluate the basic functionality of the design, which does not change as a result of optimization. The functional simulator assumes that the time needed for signals to propagate through the logic gates is negligible. In real logic gates this assumption is not realistic, regardless of the hardware technology chosen for implementation of the circuit. However, the functional simulation provides a first step in validating the basic operation of a design without concern for the effects of implementation technology. Accurate simulations that account for the timing details related to technology can be obtained by using a *timing simulator*. We will discuss timing simulation in Chapter 4.

2.9.4 SUMMARY

The CAD tools discussed in this section form a part of a CAD system. A typical design flow that the user follows is illustrated in Figure 2.29. After the design entry, initial synthesis tools perform various steps. For a function described by a truth table, the synthesis approach discussed in section 2.6 is applied to produce a logic expression for the function. For Verilog the translation process turns the Verilog source code into logic functions, which can be represented as logic expressions. As mentioned earlier, the designer can use a mixture of design entry methods. In Figure 2.29 this flexibility is reflected by the step labeled Merge, in which the components produced using any of the design entry methods are automatically merged into a single design. At this point the circuit is represented in the CAD system as a set of logic equations.

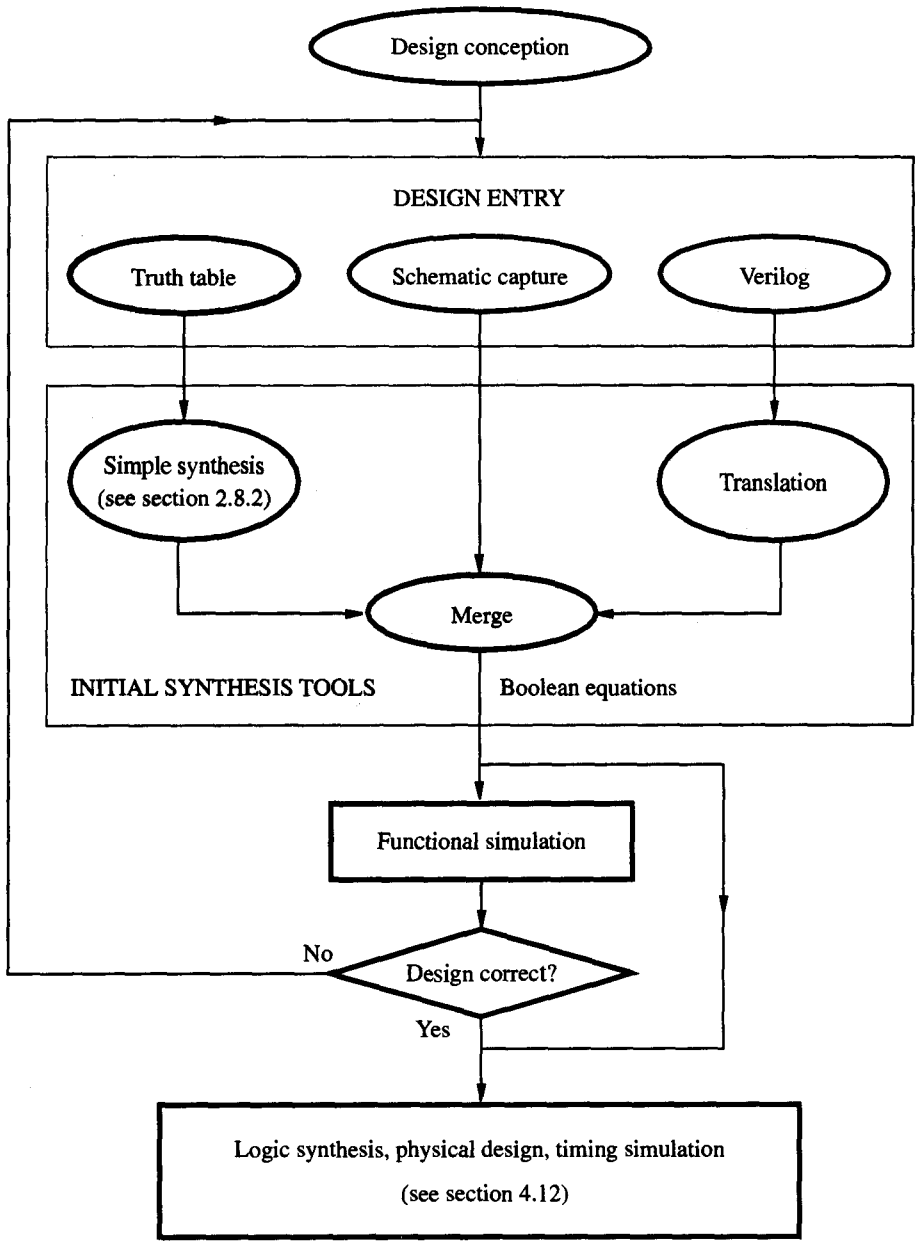


Figure 2.29 The first stages of a typical CAD system.

After the initial synthesis the correct operation of the designed circuit can be verified by using functional simulation. As shown in Figure 2.29, this step is not a requirement in the CAD flow and can be skipped at the designer's discretion. In practice, however, it is wise to verify that the designed circuit works as expected as early in the design process as possible.

Any problems discovered during the simulation are fixed by returning to the design entry stage. Once errors are no longer apparent, the designer proceeds with the remaining tools in the CAD flow. These include logic synthesis, layout synthesis, timing simulation, and others. We have mentioned these tools only briefly thus far. The remaining CAD steps will be described in Chapter 4.

At this point the reader should have some appreciation for what is involved when using CAD tools. However, the tools can be fully appreciated only when they are used first-hand. In Appendexes B to D, we provide step-by-step tutorials that illustrate how to use the MAX+plusII CAD system, which is included with this book. The tutorial in Appendix B covers design entry with both schematic capture and Verilog, as well as functional simulation. We strongly encourage the reader to work through the hands-on material. Because the tutorial uses Verilog for design entry, we provide an introduction to Verilog in the following section.

2.10 INTRODUCTION TO VERILOG

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design System. In 1990 Verilog was put into the public domain, and it has since become the most popular language for describing digital circuits. In 1995 Verilog was adopted as an official IEEE Standard, called 1364-1995. An enhanced version of Verilog, called Verilog 2001, was adopted as IEEE Standard 1364-2001 in 2001. The vast majority of the examples presented in this book are compatible with the original Verilog standard, and we also introduce some of the key features of Verilog 2001.

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

Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. To simplify the presentation, we will focus the discussion on the features of Verilog that are actually used in the examples in the book. The material presented is sufficient to allow the reader to design a wide range of circuits. The reader who wishes to learn the complete Verilog language can refer to one of the specialized texts [4–10]. Verilog has a number of constructs similar to the C programming language. A reader who knows C will find Verilog easy to learn.

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

Representation of Digital Circuits in Verilog

When using CAD tools to synthesize a logic circuit, the designer can provide the initial description of the circuit in several different ways, as we explained in section 2.9.1. One efficient way is to write this description in the form of Verilog source code. The Verilog compiler translates this code into a logic circuit.

Verilog allows the designer to represent circuits in two fundamentally different ways. One possibility is to use Verilog constructs that represent simple circuit elements such as logic gates or even transistors. A larger circuit is defined by writing code that connects such elements together. This is referred to as the *structural* representation of logic circuits. The second possibility is to describe a circuit by using logic expressions and programming constructs that define the behavior of the circuit but not its actual structure in terms of gates. This is called the *behavioral* representation.

2.10.1 STRUCTURAL SPECIFICATION OF LOGIC CIRCUITS

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

and ($y, x1, x2$);

A four-input OR gate is specified as

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

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

not (y, x);

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

A logic circuit is specified in the form of a *module* that contains the statements that define the circuit. The module may have inputs and outputs, which are referred to as its *ports*. The name port is a commonly used term that refers to an input or output connection to an electronic circuit. Consider the circuit in Figure 2.30. This circuit can be represented by the Verilog code in Figure 2.31. The first statement gives the module a name, *example1*, and indicates that there are four port signals. The next two statements declare that x_1, x_2, and x_3 are to be treated as **input** signals, while f is the **output**. The actual structure of the circuit is specified in the four statements that follow. The NOT gate gives $k = \bar{x}_2$. The AND gates produce $g = x_1x_2$ and $h = \bar{x}_2x_3$. The outputs of AND gates are combined in the OR gate to form

$$\begin{aligned} f &= g + h \\ &= x_1x_2 + \bar{x}_2x_3 \end{aligned}$$

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

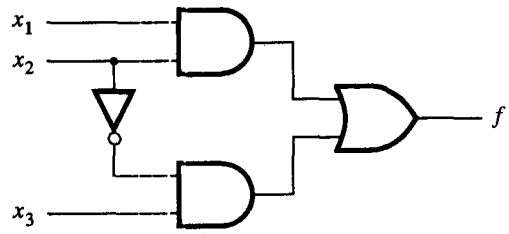


Figure 2.30 A simple logic function.

```
module example1 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;

  and (g, x1, x2);
  not (k, x2);
  and (h, k, x3);
  or (f, g, h);

endmodule
```

Figure 2.31 Verilog code for the circuit in Figure 2.30.

A second example of Verilog code is given in Figure 2.32. It defines a circuit that has four input signals, x_1, x_2, x_3, and x_4, and three output signals, f, g, and h. It implements the logic functions

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

Instead of using explicit NOT gates to define $\bar{x}_2$ and $\bar{x}_3$, we have used the Verilog operator "~" (tilde character on the keyboard) to denote complementation. Thus, $\bar{x}_2$ is indicated as ~x2 in the code. The circuit produced by the Verilog compiler for this example is shown in Figure 2.33.

Verilog Syntax

The names of modules and signals in Verilog code follow two simple rules: The name must start with a letter and it can contain any letter or number plus the "_" underscore and $ characters. Verilog is case sensitive. Thus, the name *k* is not the same as *K* and *Example1* is not the same as *example1*. The Verilog syntax does not enforce a particular style of code. For example, multiple statements can appear on a single line. White space characters, such as SPACE and TAB, and blank lines are ignored. As a matter of good style, code should be

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

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

endmodule
```

Figure 2.32 Verilog code for a four-input circuit.

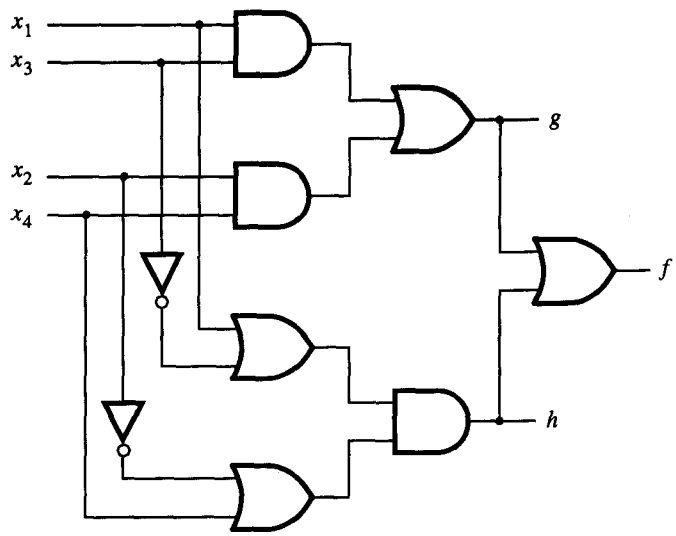


Figure 2.33 Logic circuit for the code in Figure 2.32.

formatted in such a way that it is easy to read. Indentation and blank lines can be used to make separate parts of the code easily recognizable, as we have done in Figures 2.31 and 2.32. Comments may be included in the code to improve its readability. A comment begins with the double slash "//" and continues to the end of the line.

2.10.2 BEHAVIORAL SPECIFICATION OF LOGIC CIRCUITS

Using gate level primitives can be tedious when large circuits have to be designed. An alternative is to use more abstract expressions and programming constructs to describe the behavior of a digital circuit. One possibility is to define the circuit using logic expressions. Figure 2.34 shows how the circuit in Figure 2.30 can be defined with the expression

$$f = x_1x_2 + \bar{x}_2x_3$$

The AND and OR operations are indicated by the "&" and "|" signs, respectively. The *assign* keyword provides a *continuous assignment* for the signal f. The word continuous stems from the use of Verilog for simulation; whenever any signal on the right-hand side changes its state, the value of f will be re-evaluated. The effect is equivalent to using the gate level primitives in Figure 2.31. Following this approach, the circuit in Figure 2.33 can be specified as shown in Figure 2.35.

Using logic expressions makes it easier to write Verilog code. But even higher levels of abstraction can often be used to advantage. Consider again the circuit in Figure 2.30. This circuit is similar to the 2-to-1 multiplexer circuit discussed in section 2.8.2, with x_2 being the selection control input and x_1 and x_3 being the data inputs. The circuit can be described in words by saying that $f = x_1$ if $x_2 = 1$ and $f = x_3$ if $x_2 = 0$. In Verilog, this

```
module example3 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;

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

endmodule
```

Figure 2.34 Using the continuous assignment to specify the circuit in Figure 2.30.

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

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

endmodule
```

Figure 2.35 Using the continuous assignment to specify the circuit in Figure 2.33.

behavior can be defined with the **if-else** statement

```
if (x2 == 1)
   f = x1;
else
   f = x3;
```

The complete code is given in Figure 2.36. The first line illustrates how a comment can be inserted. The **if-else** statement is an example of a Verilog *procedural* statement. We will introduce other procedural statements, such as loop statements, in Chapters 5 and 6.

Verilog syntax requires that procedural statements be contained inside a construct called an **always** block, as shown in Figure 2.36. An **always** block can contain a single statement, as in this example, or it can contain many statements. A typical Verilog design module may include several **always** blocks, each representing a part of the circuit being modeled. An important property of the **always** block is that the statements it contains are evaluated in the order given in the code. This is in contrast to the continuous assignment statements, which are evaluated concurrently and hence have no meaningful order.

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

If a signal is assigned a value using procedural statements, then Verilog syntax requires that it be declared as a *variable*; this is accomplished by using the keyword **reg** in Figure 2.36. This term also derives from the simulation jargon: It means that, once a variable's

```
// Behavioral specification
module example5 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;
   reg f;

   always @(x1 or x2 or x3)
      if (x2 == 1)
         f = x1;
      else
         f = x3;

endmodule
```

Figure 2.36 Behavioral specification of the circuit in Figure 2.30.

value is assigned with a procedural statement, the simulator "registers" this value and it will not change until the **always** block is executed again. We will discuss this issue in detail in Chapter 5.

All Verilog statements end with a semicolon. Appendix A provides a full definition of Verilog statements.

Behavioral specification of a logic circuit defines only its behavior. CAD synthesis tools use this specification to construct the actual circuit. The detailed structure of the synthesized circuit will depend on the technology used. As we will see in the chapters that follow, it is possible to achieve the same functional behavior using differently structured circuits.

2.10.3 HOW *NOT* TO WRITE VERILOG CODE

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

Once complete Verilog code is written for a particular design, the reader is encouraged to analyze the resulting circuit produced by the synthesis tools. Much can be learned about Verilog, logic circuits, and logic synthesis through this process.

2.11 CONCLUDING REMARKS

In this chapter we introduced the concept of logic circuits. We showed that such circuits can be implemented using logic gates and that they can be described using a mathematical model called Boolean algebra. Because practical logic circuits are often large, it is important to have good CAD tools to help the designer. This book is accompanied by the MAX+PlusII software, which is a CAD tool provided by Altera Corporation. We introduced a few basic features of this tool and urge the reader to start using this software as soon as possible.

Our discussion so far has been quite elementary. We will deal with both the logic circuits and the CAD tools in much more depth in the chapters that follow. But first, in Chapter 3 we will examine the most important electronic technologies used to construct logic circuits. This material will give the reader an appreciation of practical constraints that a designer of logic circuits must face.

PROBLEMS

2.1 Use algebraic manipulation to prove that $x + yz = (x + y) \cdot (x + z)$. Note that this is the distributive ruxe, as stated in identity 12*b* in section 2.5.

2.2 Use algebraic manipulation to prove that $(x + y) \cdot (x + \bar{y}) = x$.

2.3 Use the Venn diagram to prove the identity in problem 1.

2.4 Use the Venn diagram to prove DeMorgan's theorem, as given in expressions 15*a* and 15*b* in section 2.5.

2.5 Use the Venn diagram to prove

$$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3) = x_1 + x_2$$

2.6 Determine whether or not the following expressions are valid, i.e., whether the left- and right-hand sides represent same function.
(a) $\bar{x}_1x_3 + x_1x_2\bar{x}_3 + \bar{x}_1x_2 + x_1\bar{x}_2 = \bar{x}_2x_3 + x_1\bar{x}_3 + x_2\bar{x}_3 + \bar{x}_1x_2x_3$
(b) $x_1\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3 = (x_1 + \bar{x}_2 + x_3)(x_1 + x_2 + \bar{x}_3)(\bar{x}_1 + x_2 + \bar{x}_3)$
(c) $(x_1 + x_3)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)(\bar{x}_1 + x_2) = (x_1 + x_2)(x_2 + x_3)(\bar{x}_1 + \bar{x}_3)$

2.7 Draw a timing diagram for the circuit in Figure 2.19*a*. Show the waveforms that can be observed on all wires in the circuit.

2.8 Repeat problem 2.7 for the circuit in Figure 2.19*b*.

2.9 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\sum m(1, 2, 3, 4, 5, 6, 7) = x_1 + x_2 + x_3$$

2.10 Use algebraic manipulation to show that for three input variables x_1, x_2, and x_3

$$\prod M(0, 1, 2, 3, 4, 5, 6) = x_1x_2x_3$$

2.11 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1x_3 + x_1\bar{x}_2 + \bar{x}_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3$.

2.12 Use algebraic manipulation to find the minimum sum-of-products expression for the function $f = x_1\bar{x}_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3\bar{x}_4$.

2.13 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_3 + x_4) \cdot (x_1 + \bar{x}_2 + x_3) \cdot (x_1 + \bar{x}_2 + \bar{x}_3 + x_4)$.

2.14 Use algebraic manipulation to find the minimum product-of-sums expression for the function $f = (x_1 + x_2 + x_3) \cdot (x_1 + \bar{x}_2 + x_3) \cdot (\bar{x}_1 + \bar{x}_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3)$.

2.15 (a) Show the location of all minterms in a three-variable Venn diagram.
(b) Show a separate Venn diagram for each product term in the function $f = x_1\bar{x}_2x_3 + x_1x_2 + \bar{x}_1x_3$. Use the Venn diagram to find the minimal sum-of-products form of f.

2.16 Represent the function in Figure 2.18 in the form of a Venn diagram and find its minimal sum-of-products form.

2.17 Figure P2.1 shows two attempts to draw a Venn diagram for four variables. For parts (*a*) and (*b*) of the figure, explain why the Venn diagram is not correct. (Hint: the Venn diagram must be able to represent all 16 minterms of the four variables.)

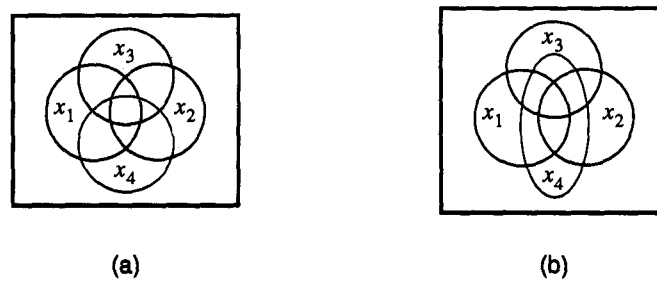


Figure P2.1 Two attempts to draw a four-variable Venn diagram.

2.18 Figure P2.2 gives a representation of a four-variable Venn diagram and shows the location of minterms m_0, m_1, and m_2. Show the location of the other minterms in the diagram. Represent the function $f = \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1x_2x_3x_4 + \bar{x}_1x_2$ on this diagram.

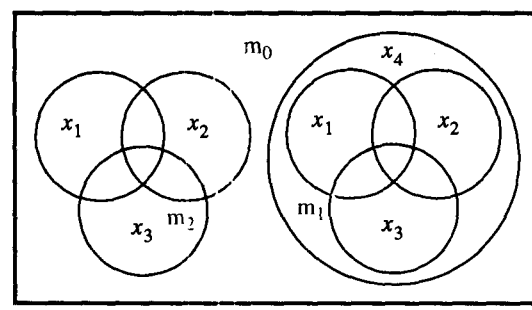


Figure P2.2 A four-variable Venn diagram.

2.19 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$.

2.20 Design the simplest sum-of-products circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$.

2.21 Design the simplest product-of-sums circuit that implements the function $f(x_1, x_2, x_3) = \prod M\ (0, 2, 5)$.

2.22 Design the simplest product-of-sums expression for the function $f(x_1, x_2, x_3) = \Pi M\ (0, 1, 5, 7)$.

2.23 Design the simplest circuit that has three inputs, x_1, x_2, and x_3, which produces an output value of 1 whenever two or more of the input variables have the value 1; otherwise, the output has to be 0.

2.24 For the timing diagram in Figure P2.3, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

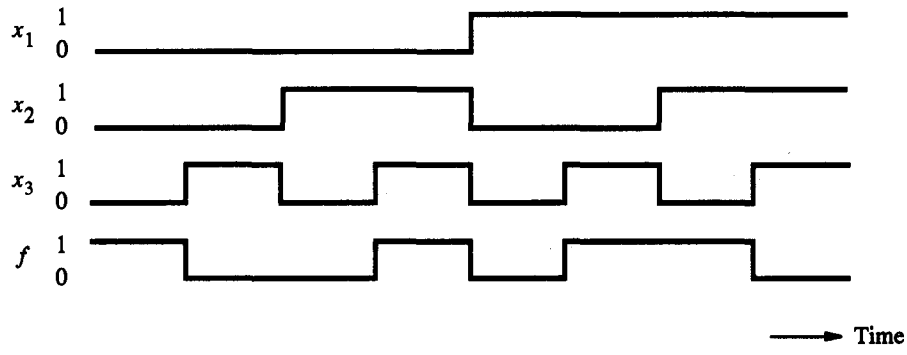


Figure P2.3 A timing diagram representing a logic function.

2.25 For the timing diagram in Figure P2.4, synthesize the function $f(x_1, x_2, x_3)$ in the simplest sum-of-products form.

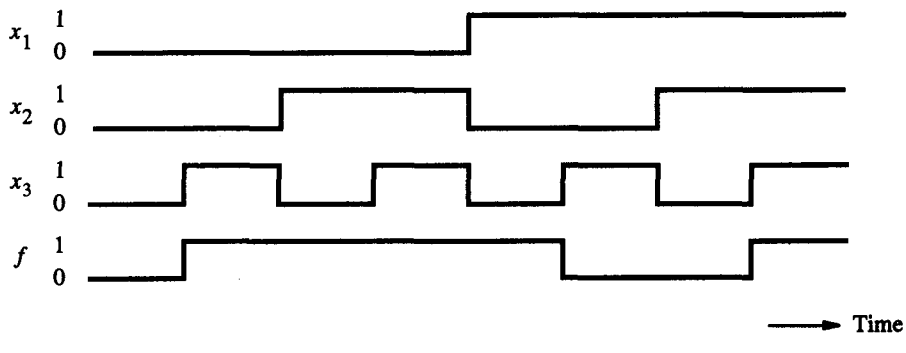


Figure P2.4 A timing diagram representing a logic function.

2.26 Design a circuit with output f and inputs x_1, x_0, y_1, and y_0. Let $X = x_1x_0$ be a number, where the four possible values of X, namely, 00, 01, 10, and 11, represent the four numbers

0, 1, 2, and 3, respectively. (We discuss representation of numbers in Chapter 5.) Similarly, let $Y = y_1y_0$ represent another number with the same four possible values. The output f should be 1 if the numbers represented by X and Y are equal. Otherwise, f should be 0.
(a) Show the truth table for f.
(b) Synthesize the simplest possible product-of-sums expression for f.

2.27 Repeat problem 2.26 for the case where f should be 1 only if $X \geq Y$.
(a) Show the truth table for f.
(b) Show the canonical sum-of-products expression for f.
(c) Show the simplest possible sum-of-products expression for f.

2.28 Implement the function in Figure 2.24 using only NAND gates.

2.29 Implement the function in Figure 2.24 using only NOR gates.

2.30 Implement the circuit in Figure 2.33 using NAND and NOR gates.

2.31 Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ using NAND gates.

2.32 Design the simplest circuit that implements the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ using NAND gates.

2.33 Repeat problem 2.31 using NOR gates.

2.34 Repeat problem 2.32 using NOR gates.

2.35 (a) Use a schematic capture tool to draw schematics for the following functions

$$f_1 = x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_4 + \bar{x}_1x_2x_3 + x_1x_2x_3$$
$$f_2 = x_2\bar{x}_4 + \bar{x}_1x_2 + x_2x_3$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.36 (a) Use a schematic capture tool to draw schematics for the following functions

$$f_1 = (x_1 + x_2 + \bar{x}_4) \cdot (\bar{x}_2 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_3 + \bar{x}_4)$$
$$f_2 = (x_2 + \bar{x}_4) \cdot (x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.37 Write Verilog code to implement the circuit in Figure 2.25*a* using the gate level primitives.

2.38 Repeat problem 2.37 for the circuit in Figure 2.25*b*.

2.39 Write Verilog code to implement the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 4, 5, 6)$ using the gate level primitives. Ensure that the resulting circuit is as simple as possible.

2.40 Write Verilog code to implement the function $f(x_1, x_2, x_3) = \sum m(0, 1, 3, 4, 5, 6)$ using the continuous assignment.

2.41 (a) Write Verilog code to describe the following functions

$$f_1 = x_1\bar{x}_3 + x_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2 + x_1\bar{x}_4$$
$$f_2 = (x_1 + \bar{x}_3) \cdot (x_1 + x_2 + \bar{x}_4) \cdot (x_2 + \bar{x}_3 + \bar{x}_4)$$

(b) Use functional simulation to prove that $f_1 = f_2$.

2.42 Consider the following Verilog statements

$$f1 = (x1 \,\&\, x3) \mid (\sim x1 \,\&\, \sim x3) \mid (x2 \,\&\, x4) \mid (\sim x2 \,\&\, \sim x4);$$
$$f2 = (x1 \,\&\, x2 \,\&\, \sim x3 \,\&\, \sim x4) \mid (\sim x1 \,\&\, \sim x2 \,\&\, x3 \,\&\, x4) \mid$$
$$(x1 \,\&\, \sim x2 \,\&\, \sim x3 \,\&\, x4) \mid (\sim x1 \,\&\, x2 \,\&\, x3 \,\&\, \sim x4);$$

(a) Write complete Verilog code to implement $f1$ and $f2$.
(b) Use functional simulation to prove that $f1 = \overline{f2}$.

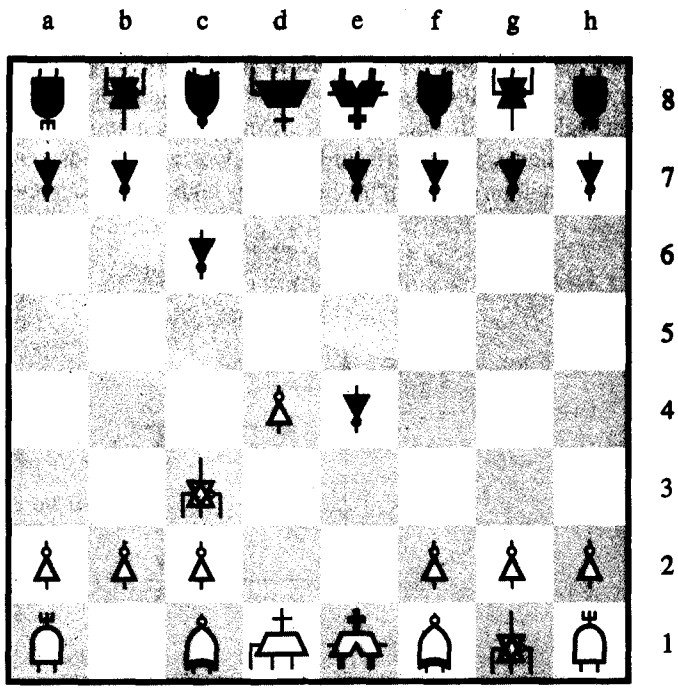
REFERENCES

1. G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by Dover Publications, New York, 1954.
2. C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits," *Transactions of AIEE* 57 (1938), pp. 713–723.
3. E. V. Huntington, "Sets of Independent Postulates for the Algebra of Logic," *Transactions of the American Mathematical Society* 5 (1904), pp. 288–309.
4. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
5. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice Hall: Upper Saddle River, NJ, 1996).
6. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice Hall: Upper Saddle River, NJ, 2000).
7. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
8. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
9. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
10. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

chapter

3

IMPLEMENTATION TECHNOLOGY



3. Nb1–c3, d5xe4

In section 1.2 we said that logic circuits are implemented using transistors and that a number of different technologies exist. We now explore technology issues in more detail.

Let us first consider how logic variables can be physically represented as signals in electronic circuits. Our discussion will be restricted to binary variables, which can take on only the values 0 and 1. In a circuit these values can be represented either as levels of voltage or current. Both alternatives are used in different technologies. We will focus on the simplest and most popular representation, using voltage levels.

The most obvious way of representing two logic values as voltage levels is to define a *threshold* voltage; any voltage below the threshold represents one logic value, and voltages above the threshold correspond to the other logic value. It is an arbitrary choice as to which logic value is associated with the low and high voltage levels. Usually, logic 0 is represented by the low voltage levels and logic 1 by the high voltages. This is known as a *positive logic* system. The opposite choice, in which the low voltage levels are used to represent logic 1 and the higher voltages are used for logic 0 is known as a *negative logic* system. In this book we use only the positive logic system, but negative logic is discussed briefly in section 3.4.

Using the positive logic system, the logic values 0 and 1 are referred to simply as "low" and "high." To implement the threshold-voltage concept, a range of low and high voltage levels is defined, as shown in Figure 3.1. The figure gives the minimum voltage, called V_{SS}, and the maximum voltage, called V_{DD}, that can exist in the circuit. We will assume that V_{SS} is 0 volts, corresponding to electrical ground, denoted *Gnd*. The voltage V_{DD} represents the power supply voltage. The most common levels for V_{DD} are between 5 volts and 1.5 volts. In this chapter we will usually assume that $V_{DD} = 5$ V. Figure 3.1 indicates that voltages in the range *Gnd* to $V_{0,max}$ represent logic value 0. The name $V_{0,max}$ means the maximum voltage level that a logic circuit must recognize as low. Similarly, the range from $V_{1,min}$ to V_{DD} corresponds to logic value 1, and $V_{1,min}$ is the minimum voltage level that a logic circuit must interpret as high. The exact levels of $V_{0,max}$ and $V_{1,min}$

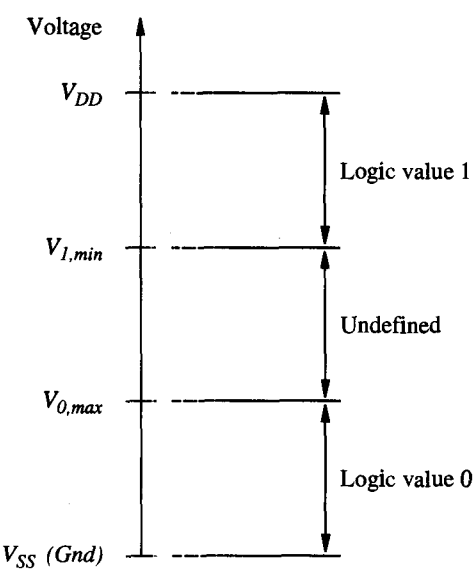


Figure 3.1 Representation of logic values by voltage levels.

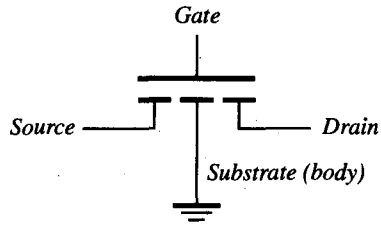
depend on the particular technology used; a typical example might set $V_{0,max}$ to 40 percent of V_{DD} and $V_{1,min}$ to 60 percent of V_{DD}. The range of voltages between $V_{0,max}$ and $V_{1,min}$ is undefined. Logic signals do not normally assume voltages in this range except in transition from one logic value to the other. We will discuss the voltage levels used in logic circuits in more depth in section 3.8.3.

3.1 TRANSISTOR SWITCHES

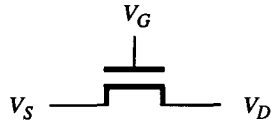
Logic circuits are built with transistors. A full treatment of transistor behavior is beyond the scope of this text; it can be found in electronics textbooks, such as [1] and [2]. For the purpose of understanding how logic circuits are built, we can assume that a transistor operates as a simple switch. Figure 3.2*a* shows a switch controlled by a logic signal, x. When x is low, the switch is open, and when x is high, the switch is closed. The most popular type of transistor for implementing a simple switch is the *metal oxide semiconductor field-effect transistor (MOSFET)*. There are two different types of MOSFETs, known as *n-channel*, abbreviated *NMOS*, and *p-channel*, denoted *PMOS*.

Figure 3.2*b* gives a graphical symbol for an NMOS transistor. It has four electrical terminals, called the *source*, *drain*, *gate*, and *substrate*. In logic circuits the substrate (also

(a) A simple switch controlled by the input x



(b) NMOS transistor



(c) Simplified symbol for an NMOS transistor

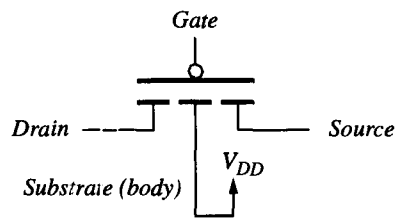
Figure 3.2 NMOS transistor as a switch.

called *body*) terminal is connected to *Gnd*. We will use the simplified graphical symbol in Figure 3.2*c*, which omits the substrate node. There is no physical difference between the source and drain terminals. They are distinguished in practice by the voltage levels applied to the transistor; by convention, the terminal with the lower voltage level is deemed to be the source.

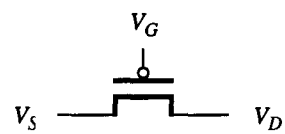
A detailed explanation of how the transistor operates will be presented in section 3.8.1. For now it is sufficient to know that it is controlled by the voltage V_G at the gate terminal. If V_G is low, then there is no connection between the source and drain, and we say that the transistor is *turned off*. If V_G is high, then the transistor is *turned on* and acts as a closed switch that connects the source and drain terminals. In section 3.8.2 we show how to calculate the resistance between the source and drain terminals when the transistor is turned on, but for now assume that the resistance is 0 Ω.

PMOS transistors have the opposite behavior of NMOS transistors. The former are used to realize the type of switch illustrated in Figure 3.3*a*, where the switch is open when the control input x is high and closed when x is low. A symbol is shown in Figure 3.3*b*. In logic circuits the substrate of the PMOS transistor is always connected to V_{DD}, leading to the simplified symbol in Figure 3.3*c*. If V_G is high, then the PMOS transistor is turned off and acts like an open switch. When V_G is low, the transistor is turned on and acts as a closed switch that connects the source and drain. In the PMOS transistor the source is the node with the higher voltage.

(a) A switch with the opposite behavior of Figure 3.2(*a*)



(b) PMOS transistor



(c) Simplified symbol for an PMOS transistor

Figure 3.3 PMOS transistor as a switch.

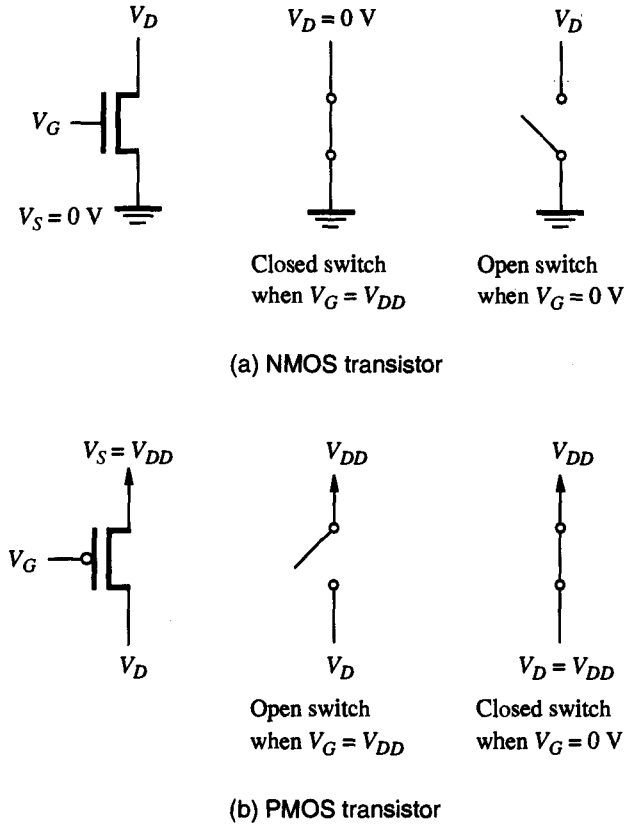


Figure 3.4 NMOS and PMOS transistors in logic circuits.

Figure 3.4 summarizes the typical use of NMOS and PMOS transistors in logic circuits. An NMOS transistor is turned on when its gate terminal is high, while a PMOS transistor is turned on when its gate is low. When the NMOS transistor is turned on, its drain is *pulled down* to *Gnd*, and when the PMOS transistor is turned on, its drain is *pulled up* to V_{DD}. Because of the way the transistors operate, an NMOS transistor cannot be used to pull its drain terminal completely up to V_{DD}. Similarly, a PMOS transistor cannot be used to pull its drain terminal completely down to *Gnd*. We discuss the operation of MOSFETs in considerable detail in section 3.8.

3.2 NMOS LOGIC GATES

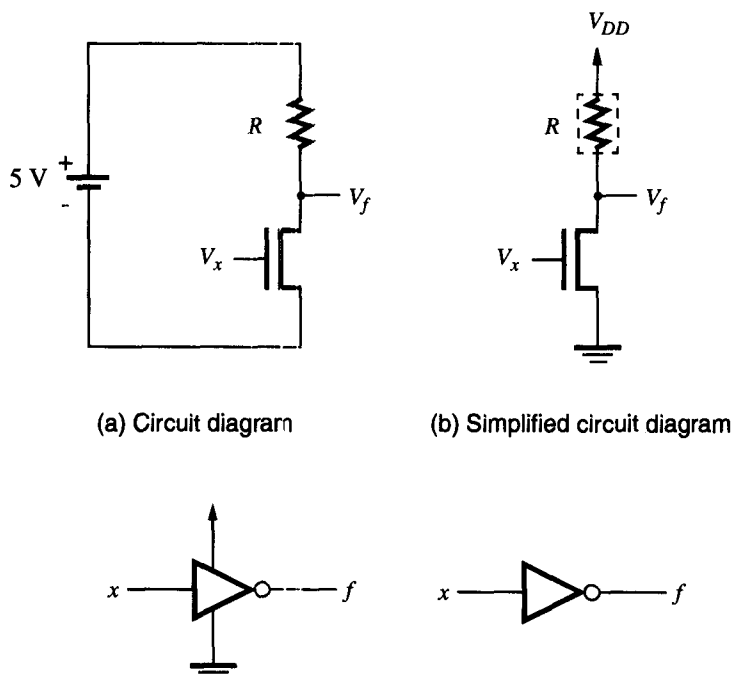
The first schemes for building logic gates with MOSFETs became popular in the 1970s and relied on either PMOS or NMOS transistors, but not both. Since the early 1980s, a combination of both NMOS and PMOS transistors has been used. We will first describe how logic circuits can be built using NMOS transistors because these circuits are easier

to understand. Such circuits are known as NMOS circuits. Then we will show how NMOS and PMOS transistors are combined in the presently popular technology known as *complementary MOS*, or *CMOS*.

In the circuit in Figure 3.5*a*, when $V_x = 0$ V, the NMOS transistor is turned off. No current flows through the resistor R, and $V_f = 5$ V. On the other hand, when $V_x = 5$ V, the transistor is turned on and pulls V_f to a low voltage level. The exact voltage level of V_f in this case depends on the amount of current that flows through the resistor and transistor. Typically, V_f is about 0.2 V (see section 3.8.3). If V_f is viewed as a function of V_x, then the circuit is an NMOS implementation of a NOT gate. In logic terms this circuit implements the function $f = \overline{x}$. Figure 3.5*b* gives a simplified circuit diagram in which the connection to the positive terminal on the power supply is indicated by an arrow labeled V_{DD} and the connection to the negative power-supply terminal is indicated by the *Gnd* symbol. We will use this simplified style of circuit diagram throughout this chapter.

The purpose of the resistor in the NOT gate circuit is to limit the amount of current that flows when $V_x = 5$ V. Rather than using a resistor for this purpose, a transistor is normally used. We will discuss this issue in more detail in section 3.8.3. In subsequent diagrams a dashed box is drawn around the resistor R as a reminder that it is implemented using a transistor.

Figure 3.5*c* presents the graphical symbols for a NOT gate. The left symbol shows the input, output, power, and ground terminals, and the right symbol is simplified to show only



(a) Circuit diagram (b) Simplified circuit diagram

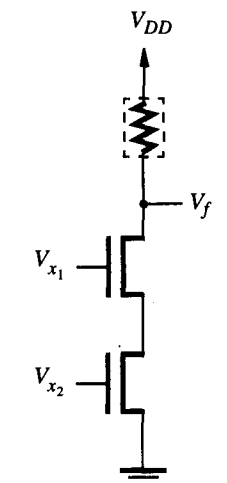
(c) Graphical symbols

Figure 3.5 A NOT gate built using NMOS technology.

the input and output terminals. In practice only the simplified symbol is used. Another name often used for the NOT gate is *inverter*. We use both names interchangeably in this book.

In section 2.1 we saw that a series connection of switches corresponds to the logic AND function, while a parallel connection represents the OR function. Using NMOS transistors, we can implement the series connection as depicted in Figure 3.6*a*. If $V_{x_1} = V_{x_2} = 5$ V, both transistors will be on and V_f will be close to 0 V. But if either V_{x_1} or V_{x_2} is 0, then no current will flow through the series-connected transistors and V_f will be pulled up to 5 V. The resulting truth table for f, provided in terms of logic values, is given in Figure 3.6*b*. The realized function is the complement of the AND function, called the *NAND* function, for NOT-AND. The circuit realizes a NAND gate. Its graphical symbols are shown in Figure 3.6*c*.

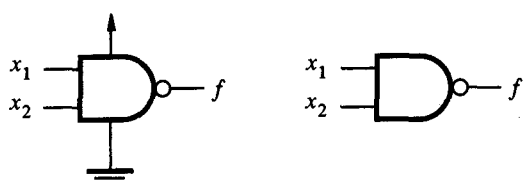
The parallel connection of NMOS transistors is given in Figure 3.7*a*. Here, if either $V_{x_1} = 5$ V or $V_{x_2} = 5$ V, then V_f will be close to 0 V. Only if both V_{x_1} and V_{x_2} are 0 will V_f be pulled up to 5 V. A corresponding truth table is given in Figure 3.7*b*. It shows that the circuit realizes the complement of the OR function, called the *NOR* function, for NOT-OR. The graphical symbols for the NOR gate appear in Figure 3.7*c*.



(a) Circuit

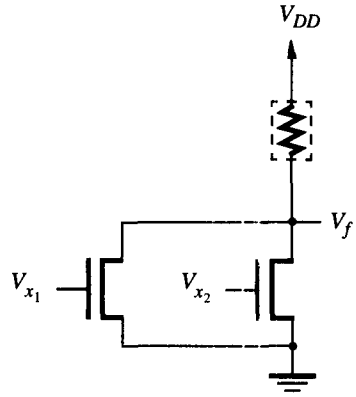
x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Graphical symbols

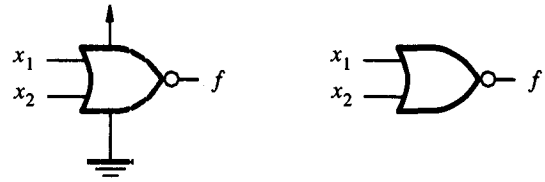
Figure 3.6 NMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

(b) Truth table



(c) Graphical symbols

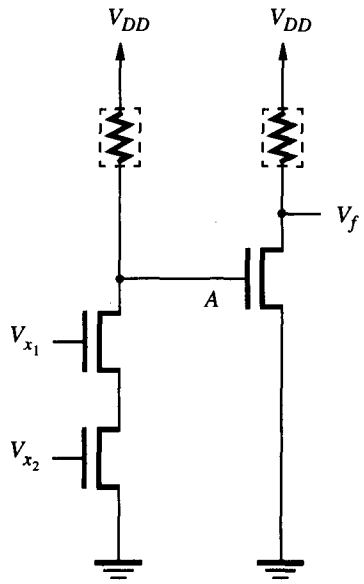
Figure 3.7 NMOS realization of a NOR gate.

In addition to the NAND and NOR gates just described, the reader would naturally be interested in the AND and OR gates that were used extensively in the previous chapter. Figure 3.8 indicates how an AND gate is built in NMOS technology by following a NAND gate with an inverter. Node A realizes the NAND of inputs x_1 and x_2, and f represents the AND function. In a similar fashion an OR gate is realized as a NOR gate followed by an inverter, as depicted in Figure 3.9.

3.3 CMOS LOGIC GATES

So far we have considered how to implement logic gates using NMOS transistors. For each of the circuits that has been presented, it is possible to derive an equivalent circuit that uses PMOS transistors. However, it is more interesting to consider how both NMOS and PMOS transistors can be used together. The most popular such approach is known as CMOS technology. We will see in section 3.8 that CMOS technology offers some attractive practical advantages in comparison to NMOS technology.

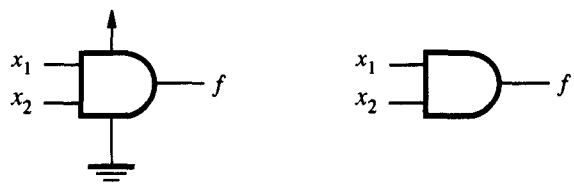
In NMOS circuits the logic functions are realized by arrangements of NMOS transistors, combined with a pull-up device that acts as a resistor. We will refer to the part of the circuit



(a) Circuit

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

(b) Truth table

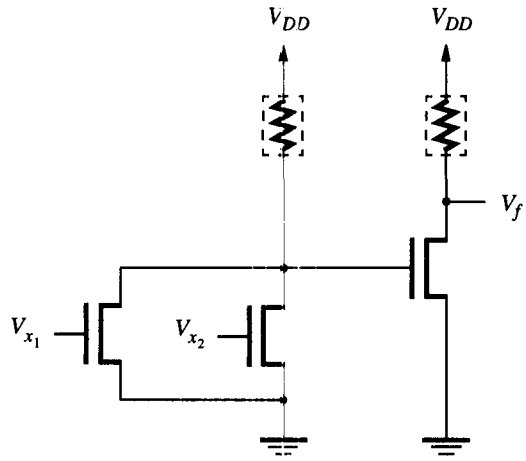


(c) Graphical symbols

Figure 3.8 NMOS realization of an AND gate.

that involves NMOS transistors as the *pull-down network (PDN)*. Then the structure of the circuits in Figures 3.5 through 3.9 can be characterized by the block diagram in Figure 3.10. The concept of CMOS circuits is based on replacing the pull-up device with a *pull-up network (PUN)* that is built using PMOS transistors, such that the functions realized by the PDN and PUN networks are complements of each other. Then a logic circuit, such as a typical logic gate, is implemented as indicated in Figure 3.11. For any given valuation of the input signals, either the PDN pulls V_f down to *Gnd* or the PUN pulls V_f up to V_{DD}. The PDN and the PUN have equal numbers of transistors, which are arranged so that the two networks are *duals* of one another. Wherever the PDN has NMOS transistors in series, the PUN has PMOS transistors in parallel, and vice versa.

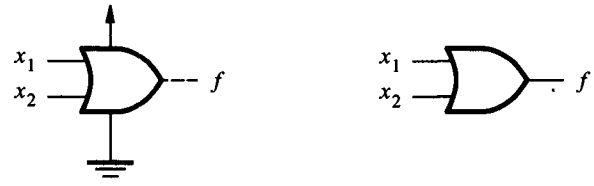
The simplest example of a CMOS circuit, a NOT gate, is shown in Figure 3.12. When $V_x = 0$ V, transistor T_2 is off and transistor T_1 is on. This makes $V_f = 5$ V, and since T_2 is



(a) Circuit

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

(b) Truth table



(c) Graphical symbols

Figure 3.9 NMOS realization of an OR gate.

off, no current flows through the transistors. When $V_x = 5$ V, T_2 is on and T_1 is off. Thus $V_f = 0$ V, and no current flows because T_1 is off.

A key point is that no current flows in a CMOS inverter when the input is either low or high. This is true for all CMOS circuits; no current flows, and hence no power is dissipated under steady state conditions. This property has led to CMOS becoming the most popular technology in use today for building logic circuits. We will discuss current flow and power dissipation in detail in section 3.8.

Figure 3.13 provides a circuit diagram of a CMOS NAND gate. It is similar to the NMOS circuit presented in Figure 3.6 except that the pull-up device has been replaced by the PUN with two PMOS transistors connected in parallel. The truth table in the figure specifies the state of each of the four transistors for each logic valuation of inputs x_1 and x_2. The reader can verify that the circuit properly implements the NAND function. Under static conditions no path exists for current flow from V_{DD} to *Gnd*.

The circuit in Figure 3.13 can be derived from the logic expression that defines the NAND operation, $f = \overline{x_1 x_2}$. This expression specifies the conditions for which $f = 1$; hence it defines the PUN. Since the PUN consists of PMOS transistors, which are turned on when their control (gate) inputs are set to 0, an input variable x_i turns on a transistor if

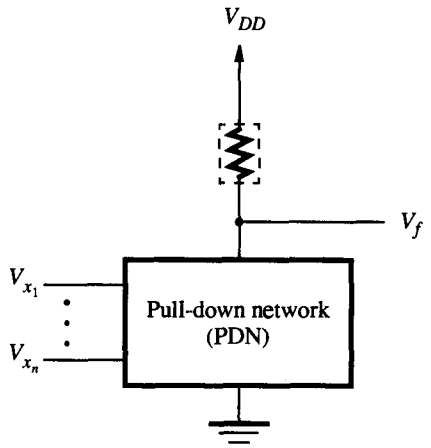


Figure 3.10 Structure of an NMOS circuit.

$x_i = 0$. From DeMorgan's law, we have

$$f = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$$

Thus $f = 1$ when *either* input x_1 or x_2 has the value 0, which means that the PUN must have two PMOS transistors connected in parallel. The PDN must implement the complement of f, which is

$$\bar{f} = x_1 x_2$$

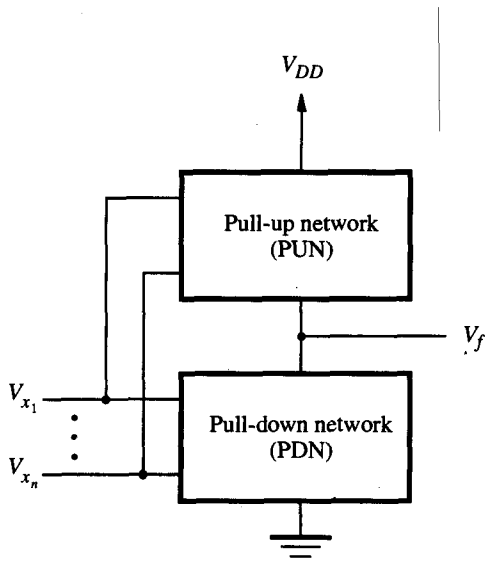
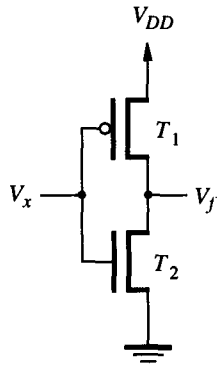


Figure 3.11 Structure of a CMOS circuit.



x	T_1	T_2	f
0	on	off	1
1	off	on	0

(a) Circuit (b) Truth table and transistor states

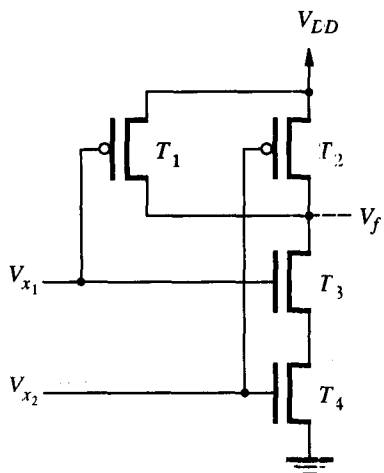
Figure 3.12 CMOS realization of a NOT gate.

Since $\overline{f} = 1$ when *both* x_1 and x_2 are 1, it follows that the PDN must have two NMOS transistors connected in series.

The circuit for a CMOS NOR gate is derived from the logic expression that defines the NOR operation

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

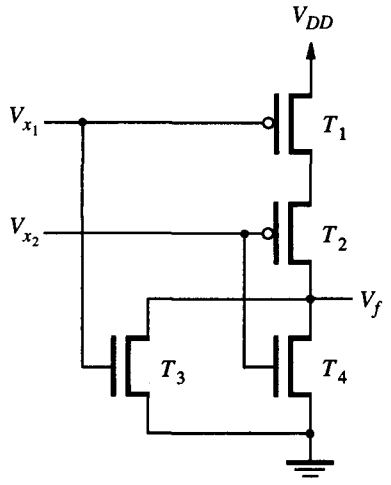
Since $f = 1$ only if both x_1 and x_2 have the value 0, then the PUN consists of two PMOS transistors connected in series. The PDN, which realizes $\overline{f} = x_1 + x_2$, has two NMOS transistors in parallel, leading to the circuit shown in Figure 3.14.



x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(a) Circuit (b) Truth table and transistor states

Figure 3.13 CMOS realization of a NAND gate.



(a) Circuit

x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	0
1	0	off	on	on	off	0
1	1	off	off	on	on	0

(b) Truth table and transistor states

Figure 3.14 CMOS realization of a NOR gate.

A CMOS AND gate is built by connecting a NAND gate to an inverter, as illustrated in Figure 3.15. Similarly, an OR gate is constructed with a NOR gate followed by a NOT gate.

The above procedure for deriving a CMOS circuit can be applied to more general logic functions to create *complex gates*. This process is illustrated in the following two examples.

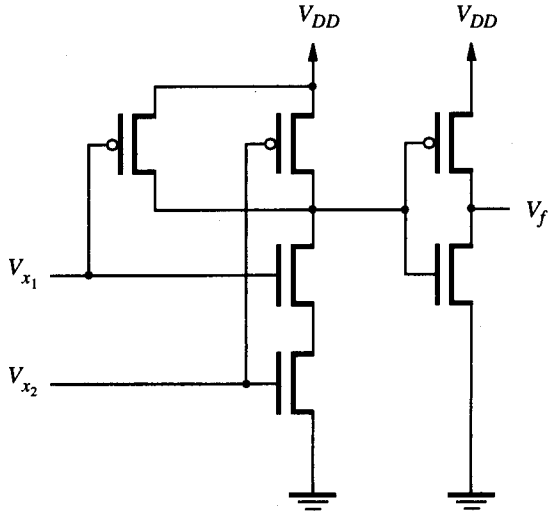


Figure 3.15 CMOS realization of an AND gate.

Example 3.1 Consider the function

$$f = \bar{x}_1 + \bar{x}_2\bar{x}_3$$

Since all variables appear in their complemented form, we can directly derive the PUN. It consists of a PMOS transistor controlled by x_1 in parallel with a series combination of PMOS transistors controlled by x_2 and x_3. For the PDN we have

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

This expression gives the PDN that has an NMOS transistor controlled by x_1 in series with a parallel combination of NMOS transistors controlled by x_2 and x_3. The circuit is shown in Figure 3.16.

Example 3.2 Consider the function

$$f = \bar{x}_1 + (\bar{x}_2 + \bar{x}_3)\bar{x}_4$$

Then

$$\bar{f} = x_1(x_2x_3 + x_4)$$

These expressions lead directly to the circuit in Figure 3.17.

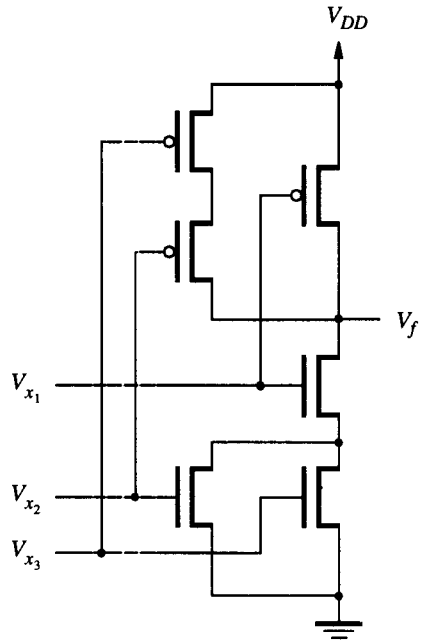


Figure 3.16 The circuit for Example 3.1.

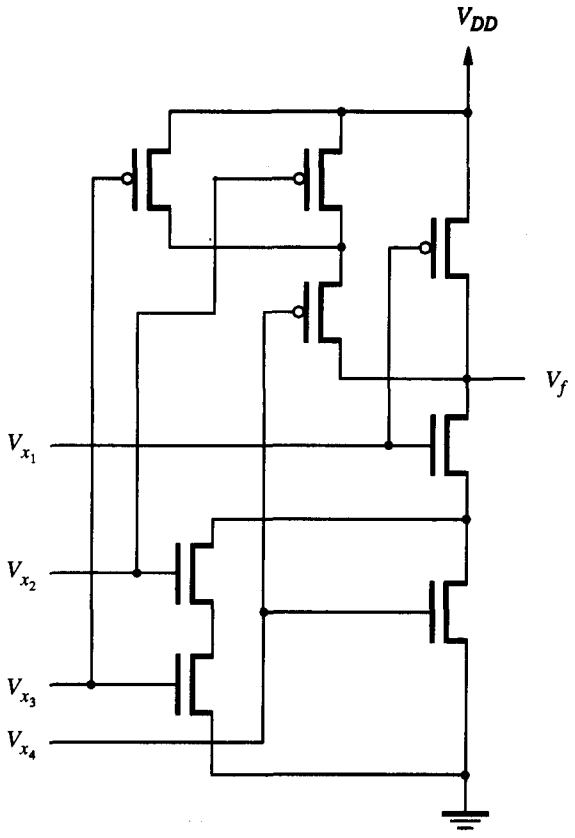


Figure 3.17 The circuit for Example 3.2.

The circuits in Figures 3.16 and 3.17 show that it is possible to implement fairly complex logic functions using combinations of series and parallel connections of transistors (acting as switches), without implementing each series or parallel connection as a complete AND (using the structure introduced in Figure 3.15) or OR gate.

3.3.1 SPEED OF LOGIC GATE CIRCUITS

In the preceding sections we have assumed that transistors operate as ideal switches that present no resistance to current flow. Hence, while we have derived circuits that realize the functionality needed in logic gates, we have ignored the important issue of the speed of operation of the circuits. In reality transistor switches have a significant resistance when turned on. Also, transistor circuits include capacitors, which are created as a side effect of the manufacturing process. These factors affect the amount of time required for signal values to propagate through logic gates. We provide a detailed discussion of the speed of logic circuits, as well as a number of other practical issues, in section 3.8.

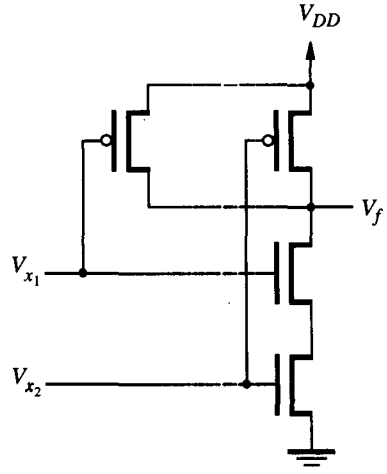
3.4 NEGATIVE LOGIC SYSTEM

At the beginning of this chapter, we said that logic values are represented as two distinct ranges of voltage levels. We are using the convention that the higher voltage levels represent logic value 1 and the lower voltages represent logic value 0. This convention is known as the positive logic system, and it is the one used in most practical applications. In this section we briefly consider the negative logic system in which the association between voltage levels and logic values is reversed.

Let us reconsider the CMOS circuit in Figure 3.13, which is reproduced in Figure 3.18*a*. Part (*b*) of the figure gives a truth table for the circuit, but the table shows voltage levels instead of logic values. In this table, L refers to the low voltage level in the circuit, which is 0 V, and H represents the high voltage level, which is V_{DD}. This is the style of truth table that manufacturers of integrated circuits often use in data sheets to describe the functionality of the chips. It is entirely up to the user of the chip as to whether L and H are interpreted in terms of logic values such that $L = 0$ and $H = 1$, or $L = 1$ and $H = 0$.

Figure 3.19*a* illustrates the positive logic interpretation in which $L = 0$ and $H = 1$. As we already know from the discussions of Figure 3.13, the circuit represents a NAND gate under this interpretation. The opposite interpretation is shown in Figure 3.19*b*. Here negative logic is used so that $L = 1$ and $H = 0$. The truth table specifies that the circuit represents a NOR gate in this case. Note that the truth table rows are listed in the opposite order from what we normally use, to be consistent with the L and H values in Figure 3.18*b*. Figure 3.19*b* also gives the logic gate symbol for the NOR gate, which includes small triangles on the gate's terminals to indicate that the negative logic system is used.

As another example, consider again the circuit in Figure 3.15. Its truth table, in terms of voltage levels, is given in Figure 3.20*a*. Using the positive logic system, this circuit



V_{x_1}	V_{x_2}	V_f
L	L	H
L	H	H
H	L	H
H	H	L

(a) Circuit (b) Voltage levels

Figure 3.18 Voltage levels in the circuit in Figure 3.13.

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

(a) Positive logic truth table and gate symbol

x_1	x_2	f
1	1	0
1	0	0
0	1	0
0	0	1

(b) Negative logic truth table and gate symbol

Figure 3.19 Interpretation of the circuit in Figure 3.18.

represents an AND gate, as indicated in Figure 3.20*b*. But using the negative logic system, the circuit represents an OR gate, as depicted in Figure 3.20*c*.

It is possible to use a mixture of positive and negative logic in a single circuit, which is known as a *mixed logic system.* In practice, the positive logic system is used in most applications. We will not consider the negative logic system further in this book.

3.5 STANDARD CHIPS

In Chapter 1 we mentioned that several different types of integrated circuit chips are available for implementation of logic circuits. We now discuss the available choices in some detail.

3.5.1 7400-SERIES STANDARD CHIPS

An approach used widely until the mid-1980s was to connect together multiple chips, each containing only a few logic gates. A wide assortment of chips, with different types of logic gates, is available for this purpose. They are known as 7400-series parts because the chip part numbers always begin with the digits 74. An example of a 7400-series part is given in Figure 3.21. Part (*a*) of the figure shows a type of package that the chip is provided in, called a *dual-inline package (DIP).* Part (*b*) illustrates the 7404 chip, which comprises six NOT gates. The chip's external connections are called *pins* or *leads.* Two pins are used to connect to V_{DD} and *Gnd*, and other pins provide connections to the NOT gates. Many

V_{x_1}	V_{x_2}	V_f
L	L	L
L	H	L
H	L	L
H	H	H

(a) Voltage levels

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

(b) Positive logic

x_1	x_2	f
1	1	1
1	0	1
0	1	1
0	0	0

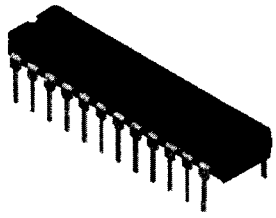
(c) Negative logic

Figure 3.20 Interpretation of the circuit in Figure 3.15.

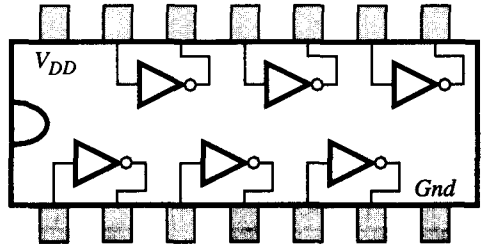
7400-series chips exist, and they are described in the data books produced by manufacturers of these chips [3–7]. Diagrams of some of the chips are also included in several textbooks, such as [8–12].

The 7400-series chips are produced in standard forms by a number of integrated circuit manufacturers, using agreed-upon specifications. Competition among various manufacturers works to the designer's advantage because it tends to lower the price of chips and ensures that parts are always readily available. For each specific 7400-series chip, several variants are built with different technologies. For instance, the part called 74LS00 is built with a technology called transistor-transistor logic (TTL), which is described in Appendix E, whereas the 74HC00 is fabricated using CMOS technology. In general, the most popular chips used today are the CMOS variants.

As an example of how a logic circuit can be implemented using 7400-series chips, consider the function $f = x_1x_2 + \bar{x}_2x_3$, which is shown in the form of a logic diagram



(a) Dual-inline package



(b) Structure of 7404 chip

Figure 3.21 A 7400-series chip.

in Figure 2.30. A NOT gate is required to produce $\bar{x}_2$, as well as 2 two-input AND gates and a two-input OR gate. Figure 3.22 shows three 7400-series chips that can be used to implement the function. We assume that the three input signals x_1, x_2, and x_3 are produced as outputs by some other circuitry that can be connected by wires to the three chips. Notice that power and ground connections are included for all three chips. This example makes use of only a portion of the gates available on the three chips, hence the remaining gates can be used to realize other functions.

Because of their low logic capacity, the standard chips are seldom used in practice today, with one exception. Many modern products include standard chips that contain buffers. Buffers are logic gates that are usually used to improve the speed of circuits. An example of a buffer chip is depicted in Figure 3.23. It is the 74244 chip, which comprises eight *tri-state buffers*. We describe how tri-state buffers work in section 3.8.8. Rather than showing how the buffers are arranged inside the chip package, as we did for the NOT gates in Figure 3.21, we show only the pin numbers of the package pins that are connected to the buffers. The package has 20 pins, and they are numbered in the same manner as shown for Figure 3.21; *Gnd* and V_{DD} connections are provided on pins 10 and 20, respectively. Many other buffer chips also exist. For example, the 162244 chip has 16 tri-state buffers. It is part of a family of devices that are similar to the 7400-series chips but with twice as many gates in each chip. These chips are available in multiple types of packages, with the most popular being a *small-outline integrated circuit (SOIC)* package. An SOIC package has a similar shape to a DIP, but the SOIC is considerably smaller in physical size.

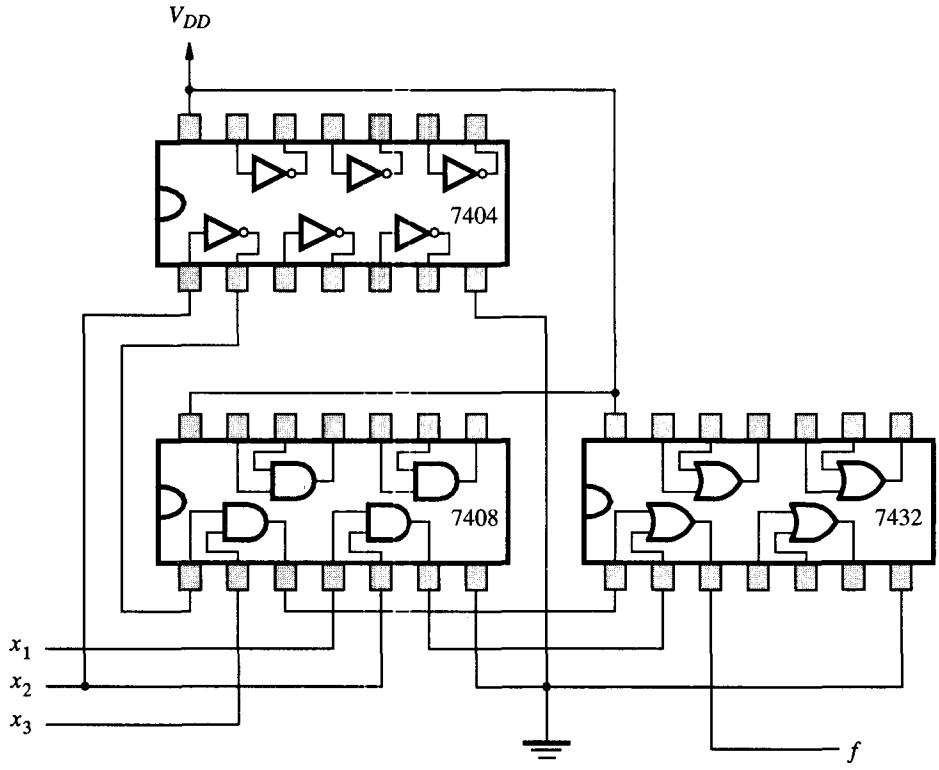


Figure 3.22 An implementation of $f = x_1x_2 + \bar{x}_2x_3$.

As integrated circuit technology has improved over time, a system of classifying chips according to their size has evolved. The earliest chips produced, such as the 7400-series chips, comprise only a few logic gates. The technology used to produce these chips is referred to as *small-scale integration (SSI)*. Chips that include slightly more logic circuitry, typically about 10 to 100 gates, represent *medium-scale integration (MSI)*. Until the mid-1980s chips that were too large to qualify as MSI were classified as *large-scale integration*

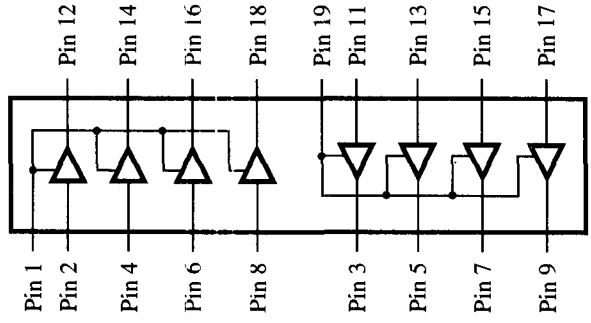


Figure 3.23 The 74244 buffer chip.

(LSI). In recent years the concept of classifying circuits according to their size has become of little practical use. Most integrated circuits today contain many thousands or millions of transistors. Regardless of their exact size, these large chips are said to be made with *very large scale integration (VLSI)* technology. The trend in digital hardware products is to integrate as much circuitry as possible onto a single chip. Thus most of the chips used today are built with VLSI technology, and the older types of chips are used rarely.

3.6 PROGRAMMABLE LOGIC DEVICES

The function provided by each of the 7400-series parts is fixed and cannot be tailored to suit a particular design situation. This fact, coupled with the limitation that each chip contains only a few logic gates, makes these chips inefficient for building large logic circuits. It is possible to manufacture chips that contain relatively large amounts of logic circuitry with a structure that is not fixed. Such chips were first introduced in the 1970s and are called *programmable logic devices (PLDs).*

A PLD is a general-purpose chip for implementing logic circuitry. It contains a collection of logic circuit elements that can be customized in different ways. A PLD can be viewed as a "black box" that contains logic gates and programmable switches, as illustrated in Figure 3.24. The programmable switches allow the logic gates inside the PLD to be connected together to implement whatever logic circuit is needed.

3.6.1 PROGRAMMABLE LOGIC ARRAY (PLA)

Several types of PLDs are commercially available. The first developed was the *programmable logic array (PLA).* The general structure of a PLA is depicted in Figure 3.25. Based on the idea that logic functions can be realized in sum-of-products form, a PLA

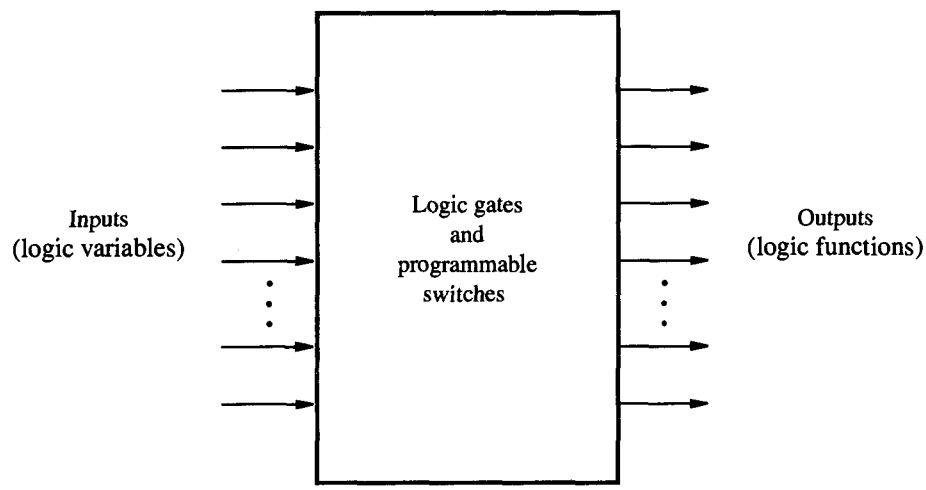


Figure 3.24 Programmable logic device as a black box.

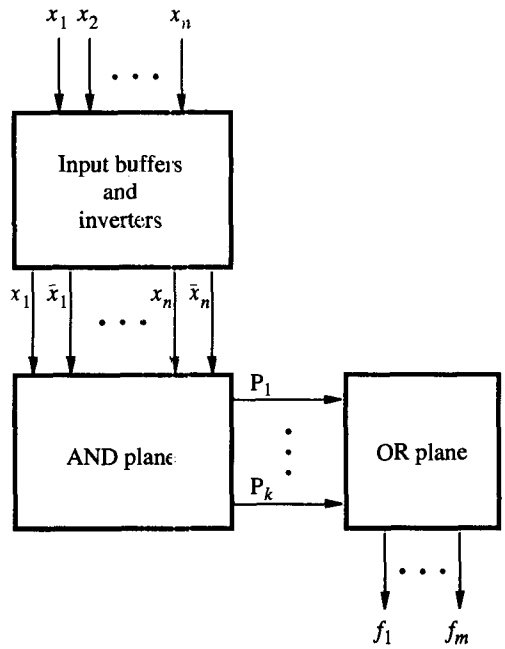


Figure 3.25 General structure of a PLA.

comprises a collection of AND gates that feeds a set of OR gates. As shown in the figure, the PLA's inputs $x_1, \ldots, x_n$ pass through a set of buffers (which provide both the true value and complement of each input) into a circuit block called an *AND plane*, or *AND array*. The AND plane produces a set of product terms $P_1, \ldots, P_k$. Each of these terms can be configured to implement any AND function of $x_1, \ldots, x_n$. The product terms serve as the inputs to an *OR plane*, which produces the outputs $f_1, \ldots, f_m$. Each output can be configured to realize any sum of $P_1, \ldots, P_k$ and hence any sum-of-products function of the PLA inputs.

A more detailed diagram of a small PLA is given in Figure 3.26, which shows a PLA with three inputs, four product terms, and two outputs. Each AND gate in the AND plane has six inputs, corresponding to the true and complemented versions of the three input signals. Each connection to an AND gate is programmable; a signal that is connected to an AND gate is indicated with a wavy line, and a signal that is not connected to the gate is shown with a broken line. The circuitry is designed such that any unconnected AND-gate inputs do not affect the output of the AND gate. In commercially available PLAs, several methods of realizing the programmable connections exist. Detailed explanation of how a PLA can be built using transistors is given in section 3.10.

In Figure 3.26 the AND gate that produces P_1 is shown connected to the inputs x_1 and x_2. Hence $P_1 = x_1x_2$. Similarly, $P_2 = x_1\bar{x}_3$, $P_3 = \bar{x}_1\bar{x}_2x_3$, and $P_4 = x_1x_3$. Programmable connections also exist for the OR plane. Output f_1 is connected to product terms P_1, P_2, and P_3. It therefore realizes the function $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$. Similarly, output

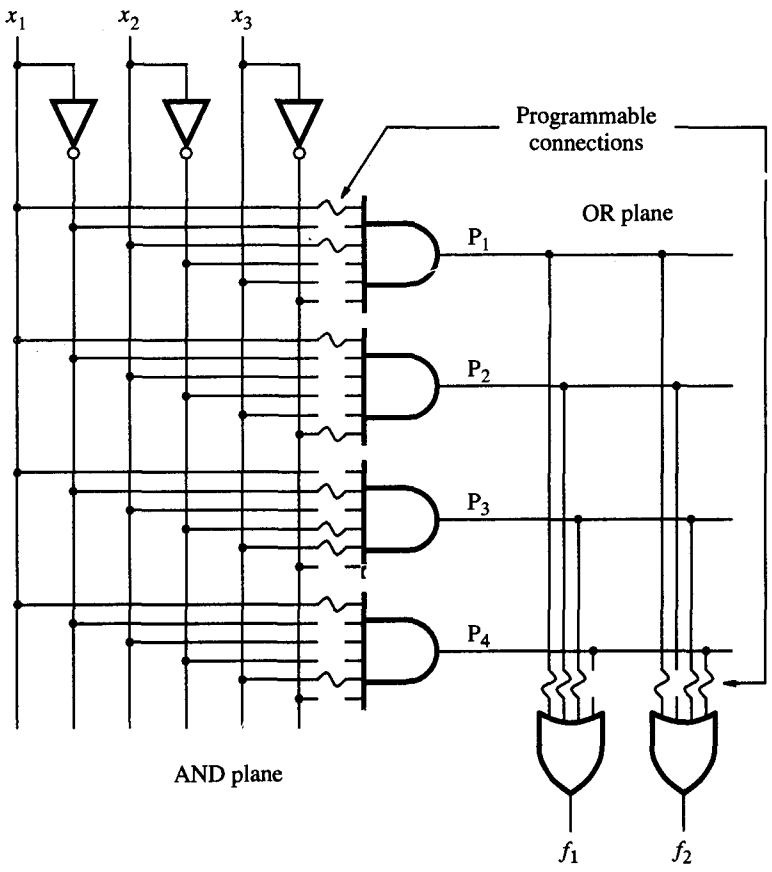


Figure 3.26 Gate-level diagram of a PLA.

$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Although Figure 3.26 depicts the PLA programmed to implement the functions described above, by programming the AND and OR planes differently, each of the outputs f_1 and f_2 could implement various functions of x_1, x_2, and x_3. The only constraint on the functions that can be implemented is the size of the AND plane because it produces only four product terms. Commercially available PLAs come in larger sizes than we have shown here. Typical parameters are 16 inputs, 32 product terms, and eight outputs.

Although Figure 3.26 illustrates clearly the functional structure of a PLA, this style of drawing is awkward for larger chips. Instead, it has become customary in technical literature to use the style shown in Figure 3.27. Each AND gate is depicted as a single horizontal line attached to an AND-gate symbol. The possible inputs to the AND gate are drawn as vertical lines that cross the horizontal line. At any crossing of a vertical and horizontal line, a programmable connection, indicated by an X, can be made. Figure 3.27 shows the programmable connections needed to implement the product terms in Figure 3.26. Each OR gate is drawn in a similar manner, with a vertical line attached to an OR-gate symbol.

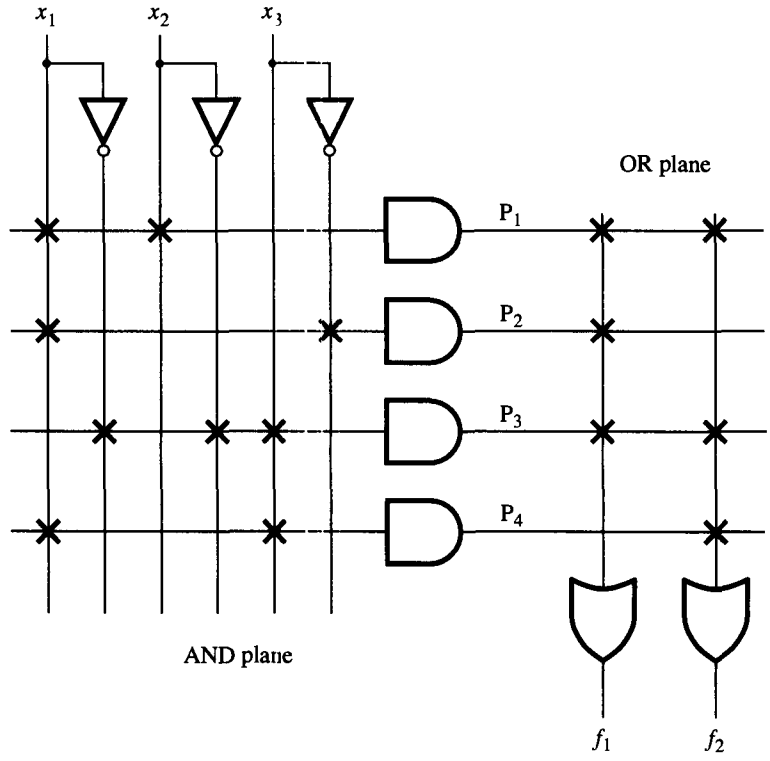


Figure 3.27 Customary schematic for the PLA in Figure 3.26.

The AND-gate outputs cross these lines, and corresponding programmable connections can be formed. The figure illustrates the programmable connections that produce the functions f_1 and f_2 from Figure 3.26.

The PLA is efficient in terms of the area needed for its implementation on an integrated circuit chip. For this reason, PLAs are often included as part of larger chips, such as microprocessors. In this case a PLA is created so that the connections to the AND and OR gates are fixed, rather than programmable. In section 3.10 we will show that both fixed and programmable PLAs can be created with similar structures.

3.6.2 PROGRAMMABLE ARRAY LOGIC (PAL)

In a PLA both the AND and OR planes are programmable. Historically, the programmable switches presented two difficulties for manufacturers of these devices: they were hard to fabricate correctly, and they reduced the speed-performance of circuits implemented in the PLAs. These drawbacks led to the development of a similar device in which the AND plane is programmable, but the OR plane is fixed. Such a chip is known as a *programmable array logic (PAL)* device. Because they are simpler to manufacture, and thus less expensive than PLAs, and offer better performance, PALs have become popular in practical applications.

An example of a PAL with three inputs, four product terms, and two outputs is given in Figure 3.28. The product terms P_1 and P_2 are hardwired to one OR gate, and P_3 and P_4 are hardwired to the other OR gate. The PAL is shown programmed to realize the two logic functions $f_1 = x_1x_2\bar{x}_3 + \bar{x}_1x_2x_3$ and $f_2 = \bar{x}_1\bar{x}_2 + x_1x_2x_3$. In comparison to the PLA in Figure 3.27, the PAL offers less flexibility; the PLA allows up to four product terms per OR gate, whereas the OR gates in the PAL have only two inputs. To compensate for the reduced flexibility, PALs are manufactured in a range of sizes, with various numbers of inputs and outputs, and different numbers of inputs to the OR gates. An example of a commercial PAL is given in Appendix E.

So far we have assumed that the OR gates in a PAL, as in a PLA, connect directly to the output pins of the chip. In many PALs extra circuitry is added at the output of each OR gate to provide additional flexibility. It is customary to use the term *macrocell* to refer to the OR gate combined with the extra circuitry. An example of the flexibility that may be provided in a macrocell is given in Figure 3.29. The symbol labeled *flip-flop* represents a memory element. It stores the value produced by the OR gate output at a particular point in time and can hold that value indefinitely. The flip-flop is controlled by the signal called *clock*. When *clock* makes a transition from logic value 0 to 1, the flip-flop stores the value at its *D* input at that time and this value appears at the flip-flop's Q output. Flip-flops are used for implementing many types of logic circuits, as we will show in Chapter 7.

In section 2.8.2 we discussed a 2-to-1 multiplexer circuit. It has two data inputs, a select input, and one output. The select input is used to choose one of the data inputs as

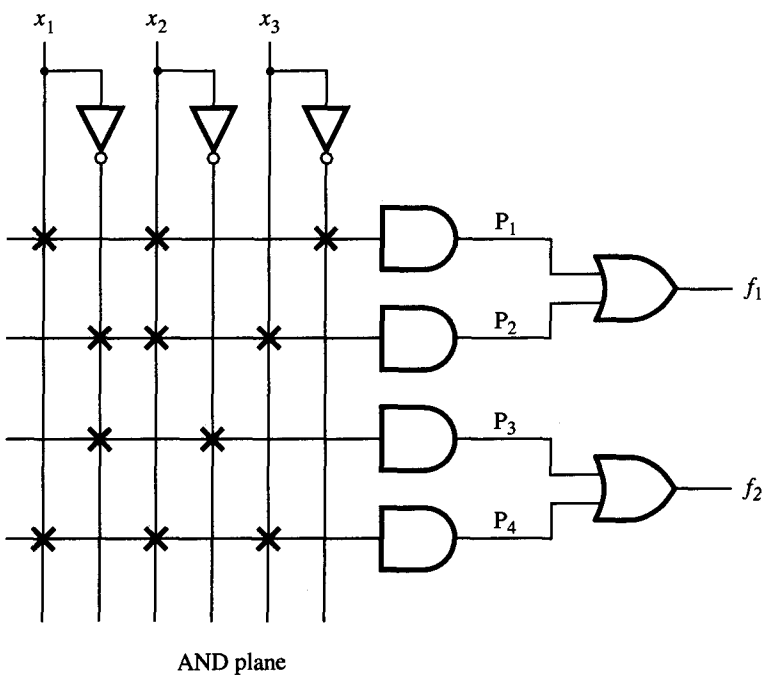


Figure 3.28 An example of a PAL.

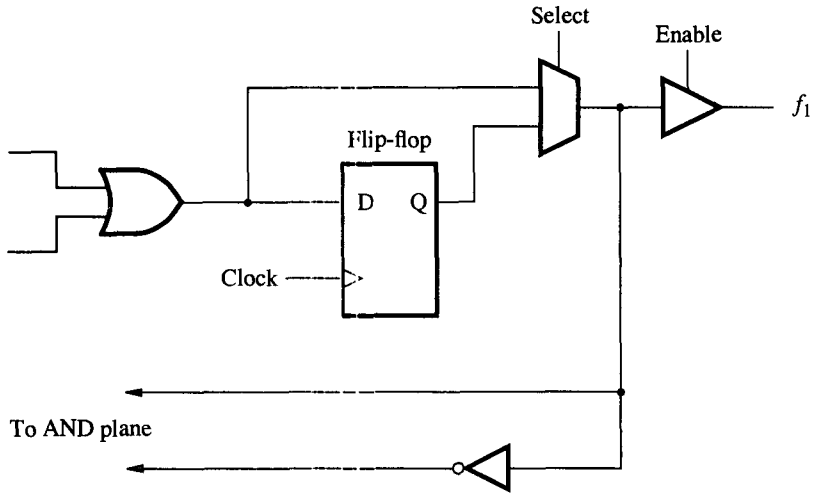


Figure 3.29 Extra circuitry added to OR-gate outputs from Figure 3.28.

the multiplexer's output. In Figure 3.29 a 2-to-1 multiplexer selects as an output from the PAL either the OR-gate output or the flip-flop output. The multiplexer's select line can be programmed to be either 0 or 1. Figure 3.29 shows another logic gate, called a tri-state buffer, connected between the multiplexer and the PAL output. We discuss tri-state buffers in section 3.8.8. Finally, the multiplexer's output is "fed back" to the AND plane in the PAL. This feedback connection allows the logic function produced by the multiplexer to be used internally in the PAL, which allows the implementation of circuits that have multiple stages, or levels, of logic gates.

A number of companies manufacture PLAs or PALs, or other, similar types of *simple PLDs (SPLDs)*. A partial list of companies, and the types of SPLDs that they manufacture, is given in Appendix E. An interested reader can examine the information that these companies provide on their products, which is available on the World Wide Web (WWW). The WWW locator for each company is given in Table E.1 in Appendix E.

3.6.3 PROGRAMMING OF PLAS AND PALS

In Figures 3.27 and 3.28, each connection between a logic signal in a PLA or PAL and the AND/OR gates is shown as an X. We describe how these switches are implemented using transistors in section 3.10. Users' circuits are implemented in the devices by *configuring*, or *programming*, these switches. Commercial chips contain a few thousand programmable switches; hence it is not feasible for a user of these chips to specify manually the desired programming state of each switch. Instead, CAD systems are employed for this purpose. We introduced CAD tools in Chapter 2 and described methods for design entry and simulation of circuits. For CAD systems that support targeting of circuits to PLDs, the tools have the capability to automatically produce the necessary information for programming each of the

switches in the device. A computer system that runs the CAD tools is connected by a cable to a dedicated *programming unit*. Once the user has completed the design of a circuit, the CAD tools generate a file, often called a *programming file* or *fuse map*, that specifies the state that each switch in the PLD should have, to realize correctly the designed circuit. The PLD is placed into the programming unit, and the programming file is transferred from the computer system. The programming unit then places the chip into a special *programming mode* and configures each switch individually. A photograph of a programming unit is shown in Figure 3.30. Several adaptors are shown beside the main unit; each adaptor is used for a specific type of chip package.

The programming procedure may take a few minutes to complete. Usually, the programming unit can automatically "read back" the state of each switch after programming, to verify that the chip has been programmed correctly. A detailed discussion of the process involved in using CAD tools to target designed circuits to programmable chips is given in Appendices B, C, and D.

PLAs or PALs used as part of a logic circuit usually reside with other chips on a printed circuit board (PCB). The procedure described above assumes that the chip can be removed from the circuit board for programming in the programming unit. Removal is made possible by using a socket on the PCB, as illustrated in Figure 3.31. Although PLAs and PALs are available in the DIP packages shown in Figure 3.21*a*, they are also available in another popular type of package, called a *plastic-leaded chip carrier (PLCC)*, which is depicted in Figure 3.31. On all four of its sides, the PLCC package has pins that "wrap around" the edges of the chip, rather than extending straight down as in the case of a DIP. The socket that houses the PLCC is attached by solder to the circuit board, and the PLCC is held in the socket by friction.

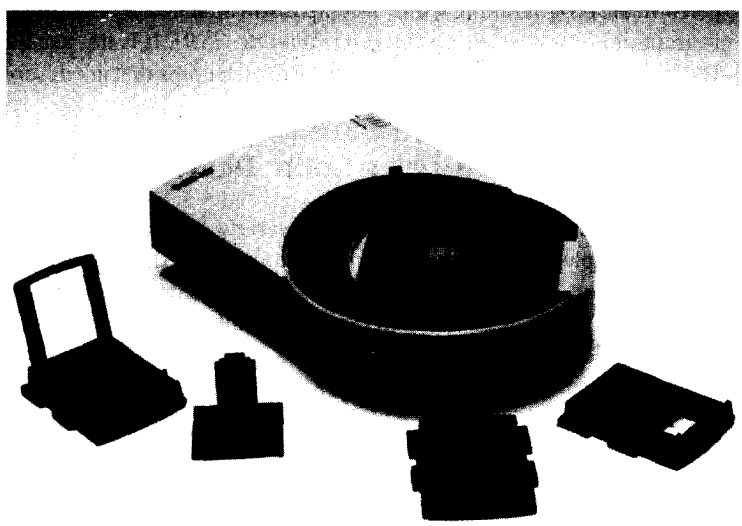


Figure 3.30 A PLD programming unit (courtesy of Data IO Corp.).

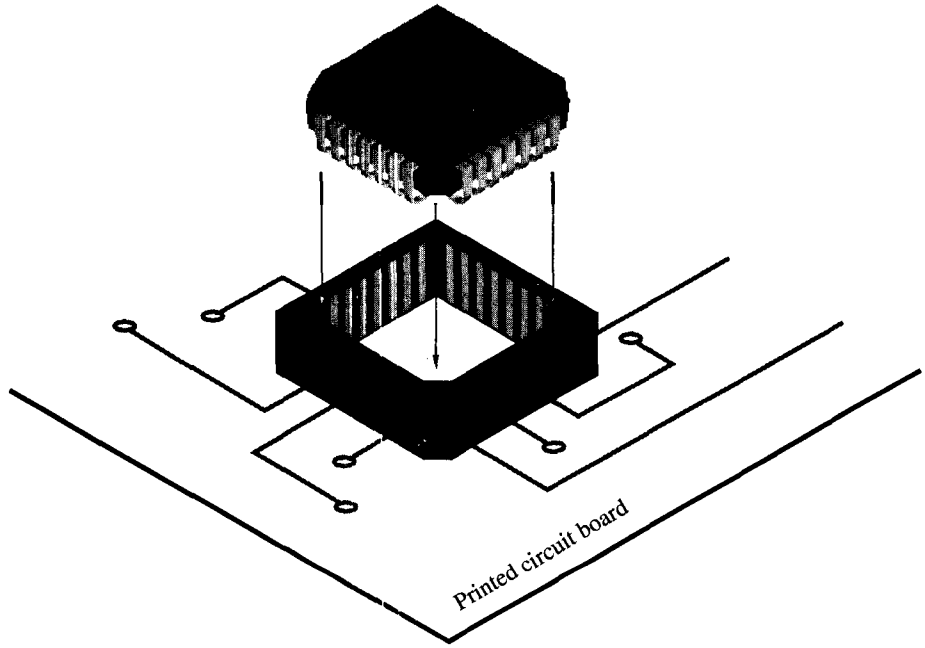


Figure 3.31 A PLCC package with socket.

Instead of relying on a programming unit to configure a chip, it would be advantageous to be able to perform the programming while the chip is still attached to its circuit board. This method of programming is called *in-system programming (ISP)*. It is not usually provided for PLAs or PALs, but is available for the more sophisticated chips that are described below.

3.6.4 COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDS)

PLAs and PALs are useful for implementing a wide variety of small digital circuits. Each device can be used to implement circuits that do not require more than the number of inputs, product terms, and outputs that are provided in the particular chip. These chips are limited to fairly modest sizes, typically supporting a combined number of inputs plus outputs of not more than 32. For implementation of circuits that require more inputs and outputs, either multiple PLAs or PALs can be employed or else a more sophisticated type of chip, called a *complex programmable logic device (CPLD)*, can be used.

A CPLD comprises multiple circuit blocks on a single chip, with internal wiring resources to connect the circuit blocks. Each circuit block is similar to a PLA or a PAL; we will refer to the circuit blocks as *PAL-like blocks*. An example of a CPLD is given in Figure 3.32. It includes four PAL-like blocks that are connected to a set of *interconnection wires*. Each PAL-like block is also connected to a subcircuit labeled *I/O block*, which is attached to a number of the chip's input and output pins.

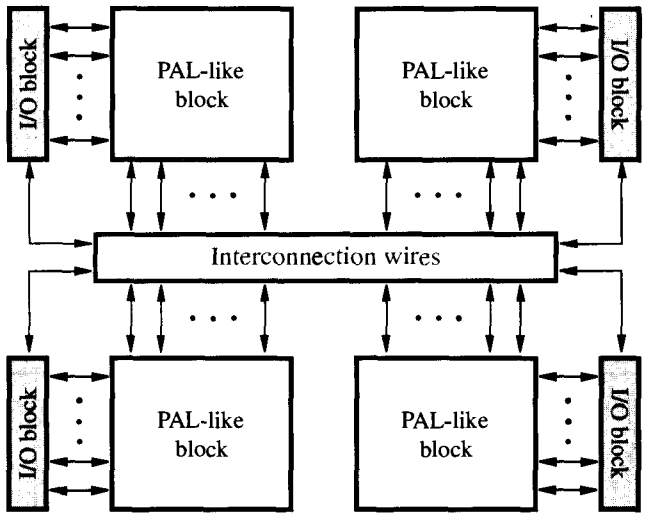


Figure 3.32 Structure of a complex programmable logic device (CPLD).

Figure 3.33 shows an example of the wiring structure and the connections to a PAL-like block in a CPLD. The PAL-like block includes 3 macrocells (real CPLDs typically have about 16 macrocells in a PAL-like block), each consisting of a four-input OR gate (real CPLDs usually provide between 5 and 20 inputs to each OR gate). The OR-gate output is connected to another type of logic gate that we have not yet introduced. It is called an Exclusive-OR (XOR) gate. We discuss XOR gates in section 3.9.1. The behavior of an XOR gate is the same as for an OR gate except that if both of the inputs are 1, the XOR gate produces a 0. One input to the XOR gate in Figure 3.33 can be programmably connected to 1 or 0; if 1, then the XOR gate complements the OR-gate output, and if 0, then the XOR gate has no effect. In many CPLDs the XOR gates can be used in other ways also, which we will see in Example 4.21, in Chapter 4. The macrocell also includes a flip-flop, a multiplexer, and a tri-state buffer. As we mentioned in the discussion for Figure 3.29, the flip-flop is used to store the output value produced by the OR gate. Each tri-state buffer (see section 3.8.8) is connected to a pin on the CPLD package. The tri-state buffer acts as a switch that allows each pin to be used either as an output from the CPLD or as an input. To use a pin as an output, the corresponding tri-state buffer is enabled, acting as a switch that is turned on. If the pin is to be used as an input, then the tri-state buffer is disabled, acting as a switch that is turned off. In this case an external source can drive a signal onto the pin, which can be connected to other macrocells using the interconnection wiring.

The interconnection wiring contains programmable switches that are used to connect the PAL-like blocks. Each of the horizontal wires can be connected to some of the vertical wires that it crosses, but not to all of them. Extensive research has been done to decide how many switches should be provided for connections between the wires. The number of switches is chosen to provide sufficient flexibility for typical circuits without wasting

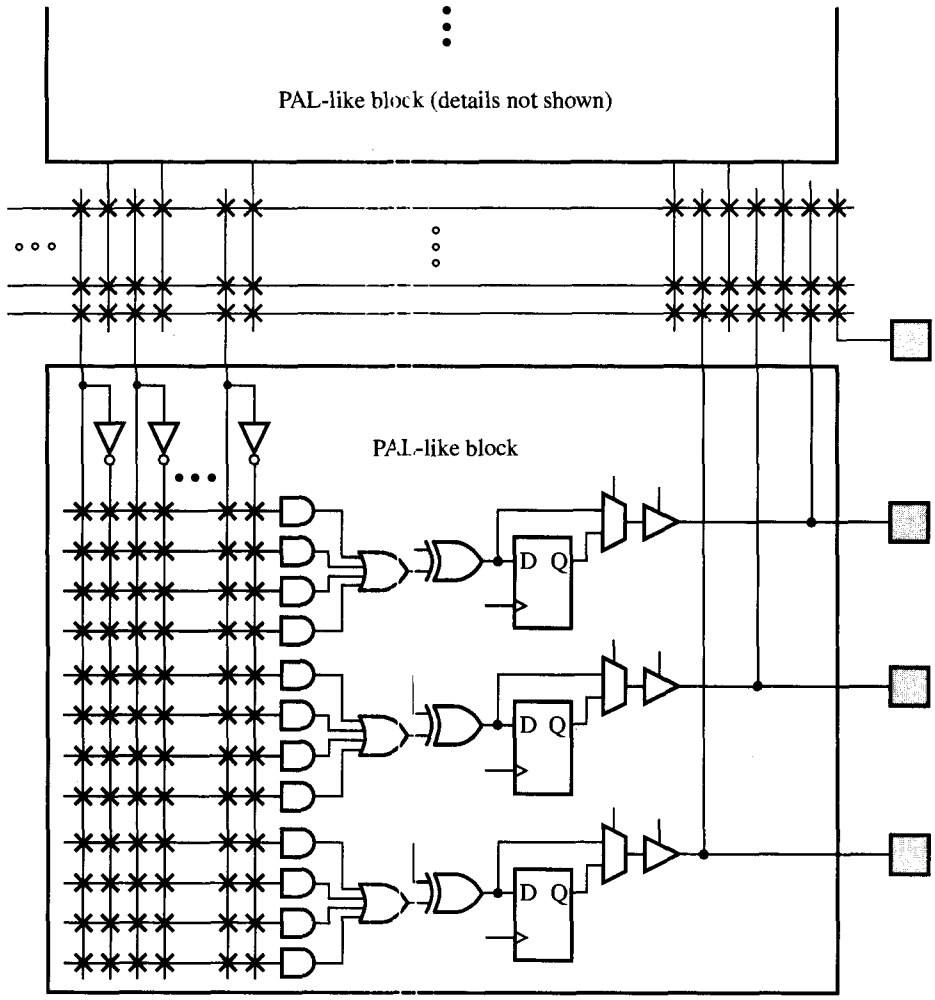
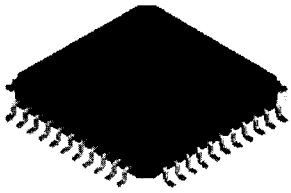


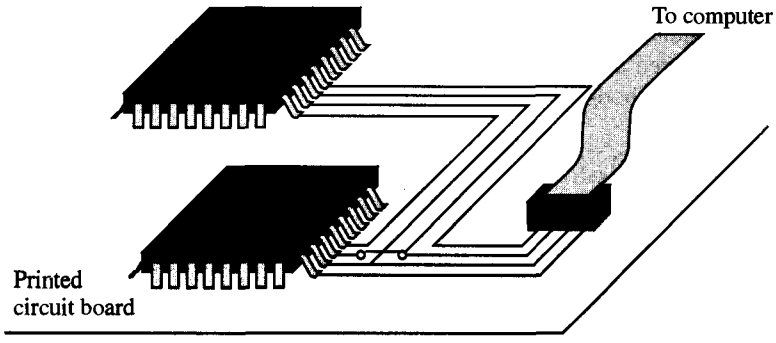
Figure 3.33 A section of the CPLD in Figure 3.32.

many switches in practice. One detail to note is that when a pin is used as an input, the macrocell associated with that pin cannot be used and is therefore wasted. Some CPLDs include additional connections between the macrocells and the interconnection wiring that avoids wasting macrocells in such situations.

Commercial CPLDs range in size from only 2 PAL-like blocks to more than 100 PAL-like blocks. They are available in a variety of packages, including the PLCC package that is shown in Figure 3.31. Figure 3.34*a* shows another type of package used to house CPLD chips, called a *quad flat pack* (QFP). Like a PLCC package, the QFP package has pins on all four sides, but whereas the PLCC's pins wrap around the edges of the package, the QFP's pins extend outward from the package, with a downward-curving shape. The QFP's pins



(a) CPLD in a quad flat pack (QFP) package



(b) JTAG programming

Figure 3.34 CPLD packaging and programming.

are much thinner than those on a PLCC, which means that the package can support a larger number of pins; QFPs are available with more than 200 pins, whereas PLCCs are limited to fewer than 100 pins.

Most CPLDs contain the same type of programmable switches that are used in SPLDs, which are described in section 3.10. Programming of the switches may be accomplished using the same technique described in section 3.6.3, in which the chip is placed into a special-purpose programming unit. However, this programming method is rather inconvenient for large CPLDs for two reasons. First, large CPLDs may have more than 200 pins on the chip package, and these pins are often fragile and easily bent. Second, to be programmed in a programming unit, a socket is required to hold the chip. Sockets for large QFP packages are very expensive; they sometimes cost more than the CPLD device itself. For these reasons, CPLD devices usually support the ISP technique. A small connector is included on the PCB that houses the CPLD, and a cable is connected between that connector and a computer system. The CPLD is programmed by transferring the programming information generated by a CAD system through the cable, from the computer into the CPLD. The circuitry on the CPLD that allows this type of programming has been standardized by the IEEE and is usually called a *JTAG port*. It uses four wires to transfer information between the computer and the device being programmed. The term *JTAG* stands for Joint Test Action

Group. Figure 3.34*b* illustrates the use of a JTAG port for programming two CPLDs on a circuit board. The CPLDs are connected together so that both can be programmed using the same connection to the computer system. Once a CPLD is programmed, it retains the programmed state permanently, even when the power supply for the chip is turned off. This property is called *nonvolatile* programming.

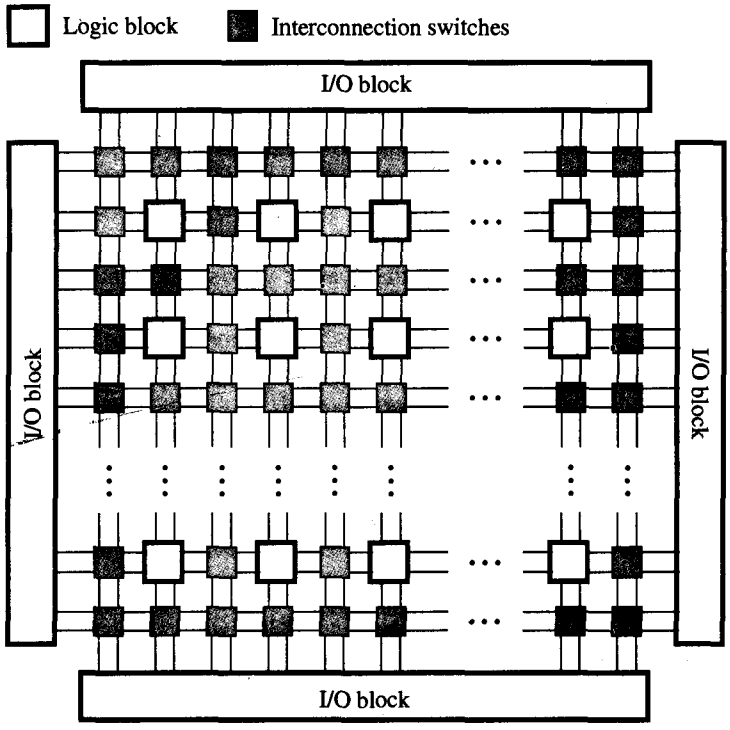
CPLDs are used for the implementation of many types of digital circuits. In industrial designs that employ some type of PLD device, CPLDs are used in about half the cases (SPLDs are used in only a small fraction of recently produced designs). A number of companies offer competing CPLDs. Appendix E lists, in Table E.2, the names of the major companies involved and shows the company's WWW locator. The reader is encouraged to examine the product information that each company provides on its Web pages. One example of a commercially available CPLD is described in detail in Appendix E. This CPLD family, manufactured by Altera and called the MAX 7000, is used in several examples presented later in the book.

3.6.5 FIELD-PROGRAMMABLE GATE ARRAYS

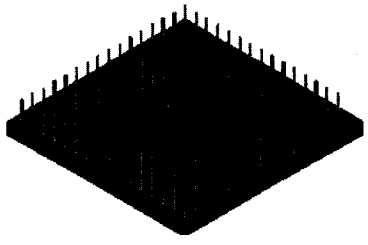
The types of chips described above, 7400 series, SPLDs, and CPLDs, are useful for implementation of a wide range of logic circuits. Except for CPLDs, these devices are rather small and are suitable only for relatively simple applications. Even for CPLDs, only moderately large logic circuits can be accommodated in a single chip. For cost and performance reasons, it is prudent to implement a desired logic circuit using as few chips as possible, so the amount of circuitry on a given chip and its functional capability are important. One way to quantify a circuit's *size* is to assume that the circuit is to be built using only simple logic gates and then estimate how many of these gates are needed. A commonly used measure is the total number of two-input NAND gates that would be needed to build the circuit; this measure is often called the number of *equivalent gates*.

Using the equivalent-gates metric, the size of a 7400-series chip is simple to measure because each chip contains only simple gates. For SPLDs and CPLDs the typical measure used is that each macrocell represents about 20 equivalent gates. Thus a typical PAL that has eight macrocells can accommodate a circuit that needs up to about 160 gates, and a large CPLD that has 1000 macrocells can implement circuits of up to about 20,000 equivalent gates.

By modern standards, a logic circuit with 20,000 gates is not large. To implement larger circuits, it is convenient to use a different type of chip that has a larger logic capacity. A *field-programmable gate array (FPGA)* is a programmable logic device that supports implementation of relatively large logic circuits. FPGAs are quite different from SPLDs and CPLDs because FPGAs do not contain AND or OR planes. Instead, FPGAs provide *logic blocks* for implementation of the required functions. The general structure of an FPGA is illustrated in Figure 3.35*a*. It contains three main types of resources: logic blocks, I/O blocks for connecting to the pins of the package, and interconnection wires and switches. The logic blocks are arranged in a two-dimensional array, and the interconnection wires are organized as horizontal and vertical *routing channels* between rows and columns of logic blocks. The routing channels contain wires and programmable switches that allow the logic blocks to be interconnected in many ways. Figure 3.35*a* shows two locations for



(a) General structure of an FPGA



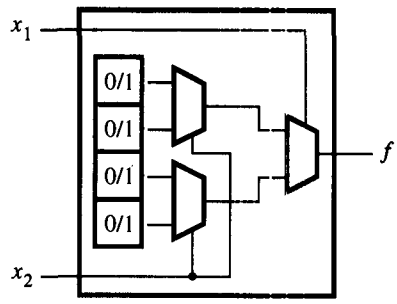
(b) Pin grid array (PGA) package (bottom view)

Figure 3.35 A field-programmable gate array (FPGA).

programmable switches; the blue boxes adjacent to logic blocks hold switches that connect the logic block input and output terminals to the interconnection wires, and the blue boxes that are diagonally between logic blocks connect one interconnection wire to another (such as a vertical wire to a horizontal wire). Programmable connections also exist between the I/O blocks and the interconnection wires. The actual number of programmable switches and wires in an FPGA varies in commercially available chips.

FPGAs can be used to implement logic circuits of more than a few hundred thousand equivalent gates in size. Two examples of FPGAs, called the Altera FLEX 10K and the Xilinx XC4000, are described in Appendix E. FPGAs are available in a variety of packages, including the PLCC and QFP packages described earlier. Figure 3.35*b* depicts another type of package, called a *pin grid array (PGA)*. A PGA package may have up to a few hundred pins in total, which extend straight outward from the bottom of the package, in a grid pattern. Yet another packaging technology that has emerged is known as the *ball grid array (BGA)*. The BGA is similar to the PGA except that the pins are small round balls, instead of posts. The advantage of BGA packages is that the pins are very small; hence more pins can be provided on the package.

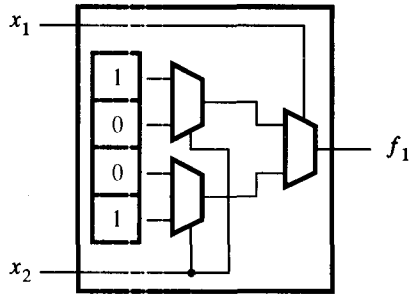
Each logic block in an FPGA typically has a small number of inputs and outputs. A number of FPGA products are on the market, featuring different types of logic blocks. The most commonly used logic block is a *lookup table (LUT)*, which contains *storage cells* that are used to implement a small logic function. Each cell is capable of holding a single logic value, either 0 or 1. The stored value is produced as the output of the storage cell. LUTs of various *sizes* may be created, where the size is defined by the number of inputs. Figure 3.36*a* shows the structure of a small LUT. It has two inputs, x_1 and x_2, and one output, f.



(a) Circuit for a two-input LUT

x_1	x_2	f_1
0	0	1
0	1	0
1	0	0
1	1	1

(b) $f_1 = \bar{x}_1\bar{x}_2 + x_1x_2$



(c) Storage cell contents in the LUT

Figure 3.36 A two-input lookup table (LUT).

It is capable of implementing any logic function of two variables. Because a two-variable truth table has four rows, this LUT has four storage cells. One cell corresponds to the output value in each row of the truth table. The input variables x_1 and x_2 are used as the select inputs of three multiplexers, which, depending on the valuation of x_1 and x_2, select the content of one of the four storage cells as the output of the LUT. We introduced multiplexers in section 2.8.2 and will discuss storage cells in Chapter 10.

To see how a logic function can be realized in the two-input LUT, consider the truth table in Figure 3.36*b*. The function f_1 from this table can be stored in the LUT as illustrated in Figure 3.36*c*. The arrangement of multiplexers in the LUT correctly realizes the function f_1. When $x_1 = x_2 = 0$, the output of the LUT is driven by the top storage cell, which represents the entry in the truth table for $x_1x_2 = 00$. Similarly, for all valuations of x_1 and x_2, the logic value stored in the storage cell corresponding to the entry in the truth table chosen by the particular valuation appears on the LUT output. Providing access to the contents of storage cells is only one way in which multiplexers can be used to implement logic functions. A detailed presentation of the applications of multiplexers is given in Chapter 6.

Figure 3.37 shows a three-input LUT. It has eight storage cells because a three-variable truth table has eight rows. In commercial FPGA chips, LUTs usually have either four or five inputs, which require 16 and 32 storage cells, respectively. In Figure 3.29 we showed that PALs usually have extra circuitry included with their AND-OR gates. The same is true for FPGAs, which usually have extra circuitry, besides a LUT, in each logic block. Figure 3.38 shows how a flip-flop may be included in an FPGA logic block. As discussed for Figure 3.29, the flip-flop is used to store the value of its *D* input under control of its *clock* input. Examples of logic blocks in commercial FPGAs are presented in Appendix E.

For a logic circuit to be realized in an FPGA, each logic function in the circuit must be small enough to fit within a single logic block. In practice, a user's circuit is automatically

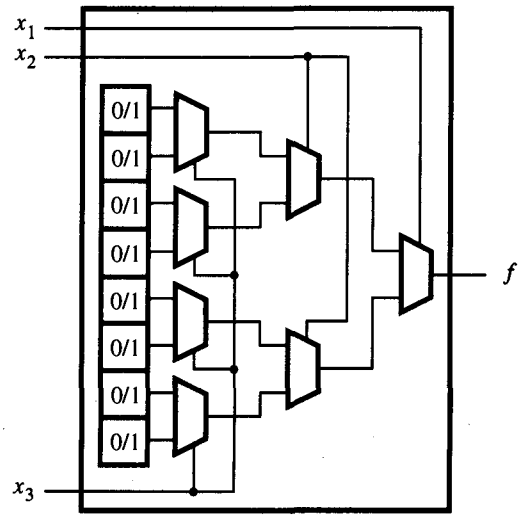


Figure 3.37 A three-input LUT.

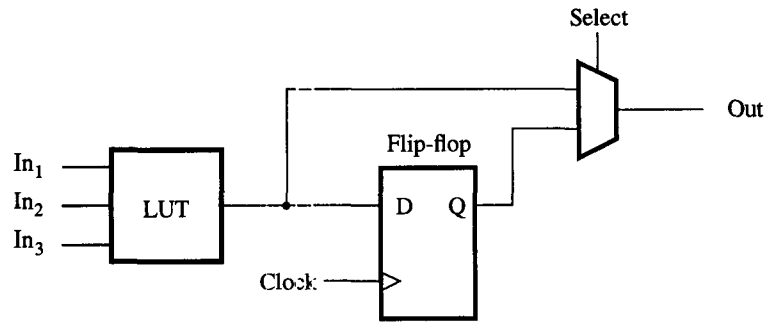


Figure 3.38 Inclusion of a flip-flop in an FPGA logic block.

translated into the required form by using CAD tools (see section 4.12). When a circuit is implemented in an FPGA, the logic blocks are programmed to realize the necessary functions and the routing channels are programmed to make the required interconnections between logic blocks. FPGAs are configured by using the ISP method, which we explained in section 3.6.4. The storage cells in the LUTs in an FPGA are *volatile*, which means that they lose their stored contents whenever the power supply for the chip is turned off. Hence the FPGA has to be programmed every time power is applied. Often a small memory chip that holds its data permanently, called a *programmable read-only memory (PROM)*, is included on the circuit board that houses the FPGA. The storage cells in the FPGA are loaded automatically from the PROM when power is applied to the chips.

A small FPGA that has been programmed to implement a circuit is depicted in Figure 3.39. The FPGA has two-input LUTs, and there are four wires in each routing channel. The figure shows the programmed states of both the logic blocks and wiring switches in a section of the FPGA. Programmable wiring switches are indicated by an X. Each switch shown in blue is turned on and makes a connection between a horizontal and vertical wire. The switches shown in black are turned off. We describe how the switches are implemented by using transistors in section 3.10.1. The truth tables programmed into the logic blocks in the top row of the FPGA correspond to the functions $f_1 = x_1x_2$ and $f_2 = \bar{x}_2x_3$. The logic block in the bottom right of the figure is programmed to produce $f = f_1 + f_2 = x_1x_2 + \bar{x}_2x_3$.

3.6.6 USING CAD TOOLS TO IMPLEMENT CIRCUITS IN CPLDS AND FPGAS

In section 2.9 we suggested that the reader should work through Tutorial 1, in Appendix B, to gain some experience using real CAD tools. Tutorial 1 covers the steps of design entry and functional simulation. Now that we have discussed some of the details of the implementation of circuits in chips, the reader may wish to experiment further with the CAD tools. In Tutorial 2, section C.3, we illustrate how to download a circuit from a computer into a CPLD or FPGA.

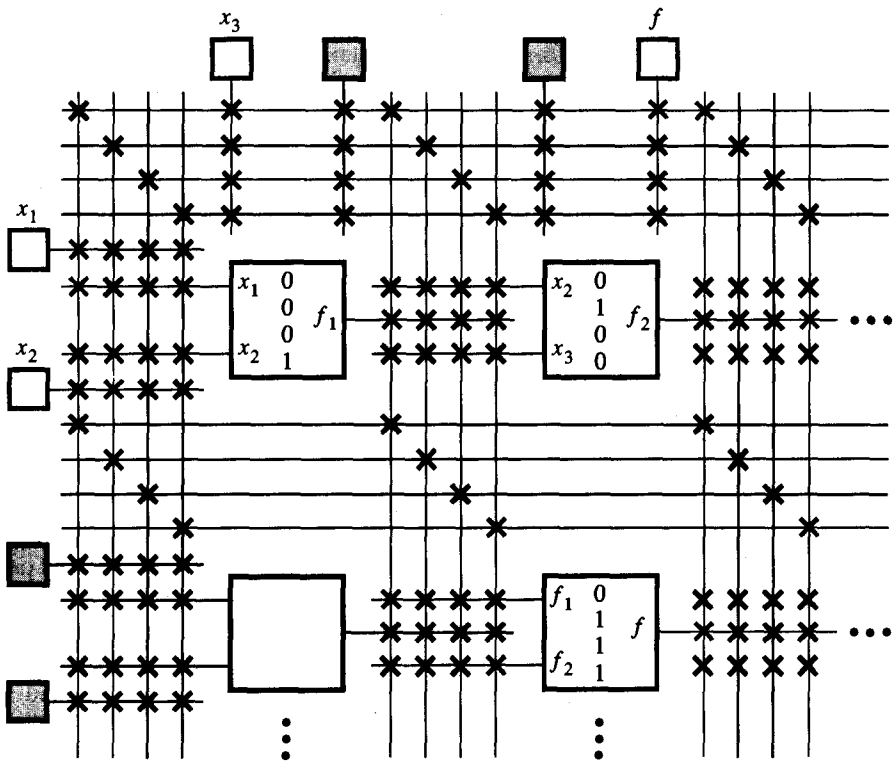


Figure 3.39 A section of a programmed FPGA.

3.7 CUSTOM CHIPS, STANDARD CELLS, AND GATE ARRAYS

The key factor that limits the size of a circuit that can be accommodated in a PLD is the existence of programmable switches. Although these switches provide the benefit of user programmability, they consume a significant amount of space on the chip. They also result in a reduction in the speed of operation of circuits. In this section we will introduce some integrated circuit technologies that do not contain programmable switches.

Chips that provide the largest number of logic gates and the highest speed are so-called *custom chips*. Whereas a PLD is prefabricated, containing logic gates and programmable switches that are programmed to realize a user's circuit, a custom chip is created from scratch. The designer of a custom chip has complete flexibility to decide the size of the chip, the number of transistors the chip contains, the placement of each transistor on the chip, and the way the transistors are connected together. The process of defining exactly where on the chip each transistor and wire is situated is called *chip layout*. For a custom chip the designer may create any layout that is desired. Because it may contain millions of transistors, a custom chip requires a large amount of design effort and therefore is

expensive. Consequently, custom chips are used only when a very large number of transistors is needed and high-speed performance is important. Also, the product being designed must be expected to sell in sufficient quantities to recoup the expense. Two examples of products that are usually realized with custom chips are microprocessors and memory chips.

Some of the design effort incurred for a custom chip can be avoided by using a technology known as *standard cells*. Chips made using this technology are often called *application-specific integrated circuits (ASICs)*. This technology is illustrated in Figure 3.40, which depicts a small portion of a chip. The rows of logic gates may be connected by wires that are created in the *routing channels* between the rows of gates. In general, many types of logic gates may be used in such a chip. The available gates are prebuilt and are stored in a library that can be accessed by the designer. In Figure 3.40 the wires are drawn in two colors. This scheme is used because metal wires can be created on integrated circuits in multiple *layers*, which makes it possible for two wires to cross one another without creating a short circuit. The blue wires represent one layer of metal wires, and the black wires are a different layer. Each blue square represents a hard-wired connection (called a *via*) between a wire on one layer and a wire on the other layer. In current technology it is possible to have eight or more layers of metal wiring. Some of the metal layers can be placed on top of the transistors in the logic gates, resulting in a more efficient chip layout.

Like a custom chip, a standard-cell chip is created from scratch according to a user's specifications. The circuitry shown in Figure 3.40 implements the two logic functions that we realized in a PLA in Figure 3.26, namely, $f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$ and $f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$. Because of the expense involved, a standard-cell chip would never be created for a small circuit such as this one, and thus the figure shows only a portion of a much larger chip. The layout of individual gates (standard cells) is predesigned and fixed. The chip layout can be created automatically by CAD tools because of the regular arrangement of the logic gates (cells) in rows. A typical chip has many long rows of logic gates with a large number of wires between each pair of rows. The I/O blocks around the periphery connect to the pins of the chip package, which is usually a QFP, PGA, or BGA package.

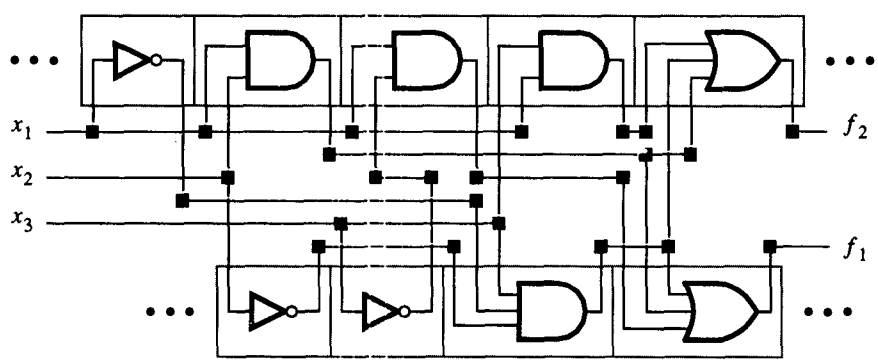


Figure 3.40 A section of two rows in a standard-cell chip.

Another technology, similar to standard cells, is the *gate-array* technology. In a gate array parts of the chip are prefabricated, and other parts are custom fabricated for a particular user's circuit. This concept exploits the fact that integrated circuits are fabricated in a sequence of steps, some steps to create transistors and other steps to create wires to connect the transistors together. In gate-array technology, the manufacturer performs most of the fabrication steps, typically those involved in the creation of the transistors, without considering the requirements of a user's circuit. This process results in a silicon wafer (see Figure 1.1) of partially finished chips, called the gate-array *template*. Later the template is modified, usually by fabricating wires that connect the transistors together, to create a user's circuit in each finished chip. The gate-array approach provides cost savings in comparison to the custom-chip approach because the gate-array manufacturer can amortize the cost of chip fabrication over a large number of template wafers, all of which are identical. Many variants of gate-array technology exist. Some have relatively large logic cells, while others are configurable at the level of a single transistor.

An example of a gate-array template is given in Figure 3.41. The gate array contains a two-dimensional array of logic cells. The chip's general structure is similar to a standard-cell chip except that in the gate array all logic cells are identical. Although the types of logic cells used in gate arrays vary, one common example is a two- or three-input NAND gate. In some gate arrays empty spaces exist between the rows of logic cells to accommodate the wires that will be added later to connect the logic cells together. However, most gate arrays do not have spaces between rows of logic cells, and the interconnection wires are fabricated on top of the logic cells. This design is possible because, as discussed for Figure 3.40, metal wires can be created on a chip in multiple layers. This technology is known

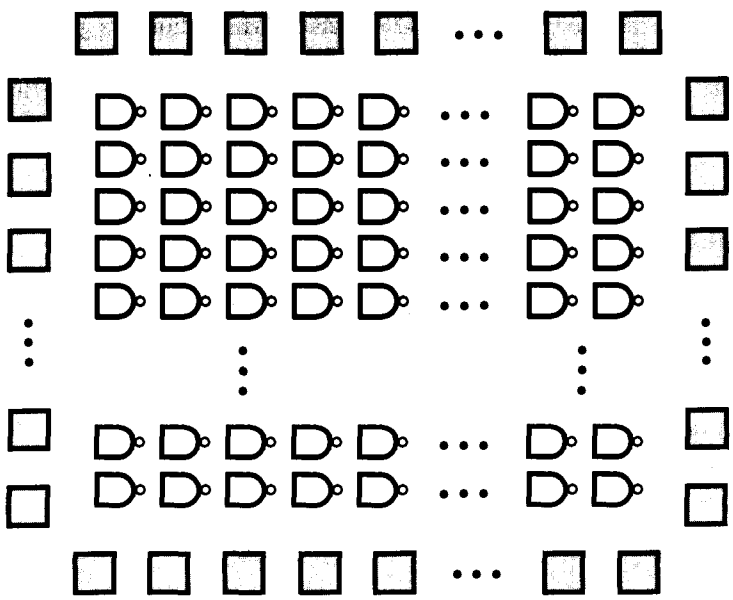


Figure 3.41 A sea-of-gates gate array.

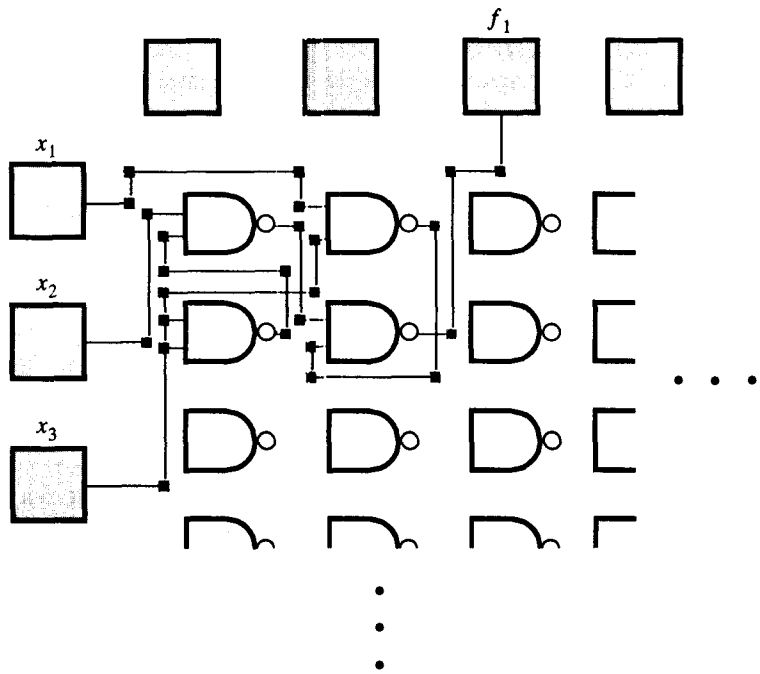


Figure 3.42 The logic function $f_1 = x_2\bar{x}_3 + x_1x_3$ in the gate array of Figure 3.41.

as the *sea-of-gates* technology. Figure 3.42 depicts a small section of a gate array that has been customized to implement the logic function $f = x_2\bar{x}_3 + x_1x_3$. As we showed in section 2.7, it is easy to verify that this circuit with only NAND gates is equivalent to the AND-OR form of the circuit.

3.8 PRACTICAL ASPECTS

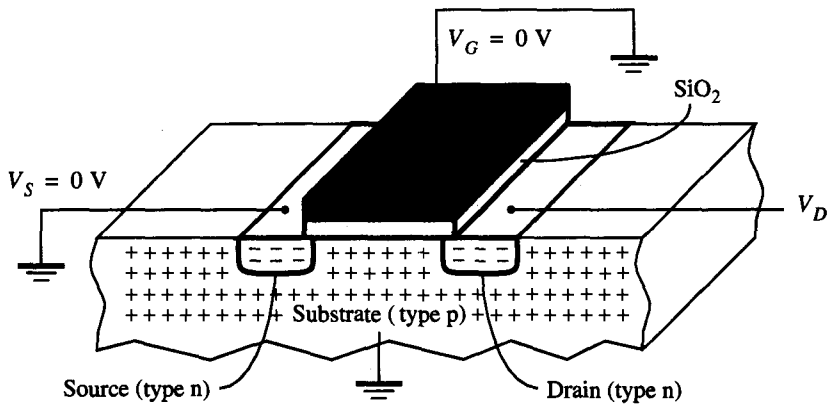
So far in this chapter, we have described the basic aspects of logic gate circuits and given examples of commercial chips. In this section we provide more detailed information on several aspects of digital circuits. We describe how transistors are fabricated in silicon and give a detailed explanation of how transistors operate. We discuss the robustness of logic circuits and discuss the important issues of signal propagation delays and power dissipation in logic gates.

3.8.1 MOSFET FABRICATION AND BEHAVIOR

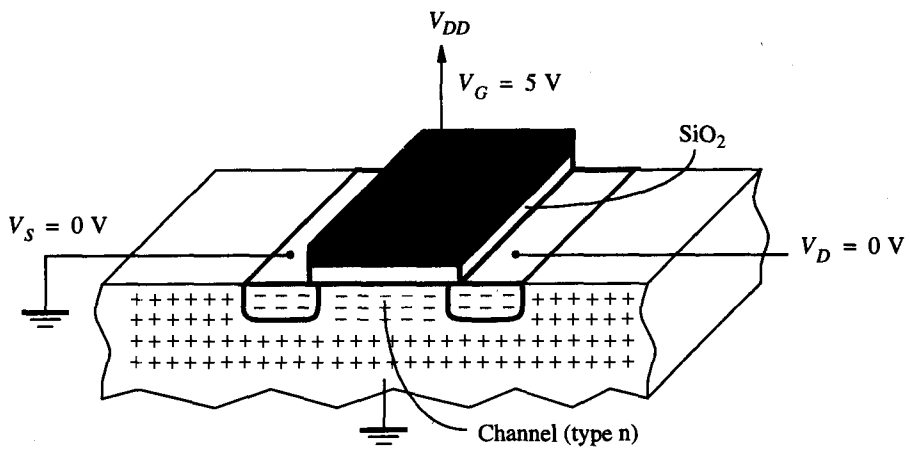
To understand the operation of NMOS and PMOS transistors, we need to consider how they are built in an integrated circuit. Integrated circuits are fabricated on silicon wafers.

A silicon wafer (see Figure 1.1) is usually about 6 or 8 inches in diameter and is somewhat similar in appearance to an audio compact disc (CD). Many integrated circuit chips are fabricated on one wafer, and the wafer is then cut to provide the individual chips.

Silicon is an *electrical semiconductor*, which means that it can be manipulated such that it sometimes conducts electrical current and at other times does not. A transistor is fabricated by creating areas in the silicon substrate that have an excess of either positive or negative electrical charge. Negatively charged areas are called *type n*, and positively charged areas are *type p*. Figure 3.43 illustrates the structure of an NMOS transistor. It has



(a) When V_{GS} = 0 V, the transistor is off



(b) When V_{GS} = 5 V, the transistor is on

Figure 3.43 Physical structure of an NMOS transistor.

type n silicon for both the source and drain terminals, and type p for the substrate terminal. Metal wiring is used to make electrical connections to the source and drain terminals.

When MOSFETs were invented, the gate terminal was made of metal. Now a material known as *polysilicon* is used. Like metal, polysilicon is a conductor, but polysilicon is preferable to metal because the former has properties that allow MOSFETs to be fabricated with extremely small dimensions. The gate is electrically isolated from the rest of the transistor by a layer of silicon dioxide (SiO_2), which is a type of glass that acts as an electrical insulator between the gate terminal and the substrate of the transistor. The transistor's operation is governed by electrical fields caused by voltages applied to its terminals, as discussed below.

In Figure 3.43 the voltage levels applied at the source, gate, and drain terminals are labeled V_S, V_G, and V_D, respectively. Consider first the situation depicted in Figure 3.43*a* in which both the source and gate are connected to *Gnd* ($V_S = V_G = 0$ V). The type n source and type n drain are isolated from one another by the type p substrate. In electrical terms two diodes exist between the source and drain. One diode is formed by the p–n junction between the substrate and source, and the other diode is formed by the p–n junction between the substrate and drain. These back-to-back diodes represent a very high resistance (about 10^{12} Ω [1]) between the drain and source that prevents current flow. We say that the transistor is *turned off*, or *cut off*, in this state.

Next consider the effect of increasing the voltage at the gate terminal with respect to the voltage at the source. Let V_{GS} represent the gate-to-source voltage. If V_{GS} is greater than a certain minimum positive voltage, called the *threshold voltage* V_T, then the transistor changes from an open switch to a closed switch, as explained below. The exact level of V_T depends on many factors, but it is typically about 0.2 V_{DD}.

The transistor's state when $V_{GS} > V_T$ is illustrated in Figure 3.43*b*. The gate terminal is connected to V_{DD}, resulting in $V_{GS} = 5$ V. The positive voltage on the gate attracts free electrons that exist in the type n source terminal, as well as in other areas of the transistor, toward the gate. Because the electrons cannot pass through the layer of glass under the gate, they gather in the region of the substrate between the source and drain, which is called the *channel*. This concentration of electrons *inverts* the silicon in the area of the channel from type p to type n, which effectively connects the source and the drain. The size of the channel is determined by the length and width of the gate. The channel length L is the dimension of the gate between the source and drain, and the channel width W is the other dimension. The channel can also be thought of as having a depth, which is dependent on the applied voltages at the source, gate, and drain.

No current can flow through the gate node of the transistor, because of the layer of glass that insulates the gate from the substrate. A current I_D may flow from the drain node to the source. For a fixed value of $V_{GS} > V_T$, the value of I_D depends on the voltage applied across the channel V_{DS}. If $V_{DS} = 0$ V, then no current flows. As V_{DS} is increased, I_D increases approximately linearly with the applied V_{DS}, as long as V_D is sufficiently small to provide at least V_T volts across the drain end of the channel, that is $V_{GD} > V_T$. In this range of voltages, namely, $0 < V_{DS} < (V_{GS} - V_T)$, the transistor is said to operate in the *triode region*, also called the *linear region*. The relationship between voltage and current is approximated by the equation

$$I_D = k_n' \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \quad \textbf{[3.1]}$$

The symbol k_n' is called the *process transconductance parameter.* It is a constant that depends on the technology being used and has the units A/V^2.

As V_D is increased, the current flow through the transistor increases, as given by equation 3.1, but only to a certain point. When $V_{DS} = V_{GS} - V_T$, the current reaches its maximum value. For larger values of V_{DS}, the transistor is no longer operating in the triode region. Since the current is at its saturated (maximum) value, we say that the transistor is in the *saturation region.* The current is now independent of V_{DS} and is given by the expression

$$I_D = \frac{1}{2}k_n' \frac{W}{L}(V_{GS} - V_T)^2 \quad \textbf{[3.2]}$$

Figure 3.44 shows the shape of the current-voltage relationship in the NMOS transistor for a fixed value of $V_{GS} > V_T$. The figure indicates the point at which the transistor leaves the triode region and enters the saturation region, which occurs at $V_{DS} = V_{GS} - V_T$.

Example 3.3

Assume the values $k_n' = 60\ \mu A/V^2$, $W/L = 2.0\ \mu m/0.5\ \mu m$, $V_S = 0$ V, $V_G = 5$ V, and $V_T = 1$ V. If $V_D = 2.5$ V, the current in the transistor is given by equation 3.1 as $I_D \approx 1.7$ mA. If $V_D = 5$ V, the saturation current is calculated using equation 3.2 as $I_D \approx 2$ mA.

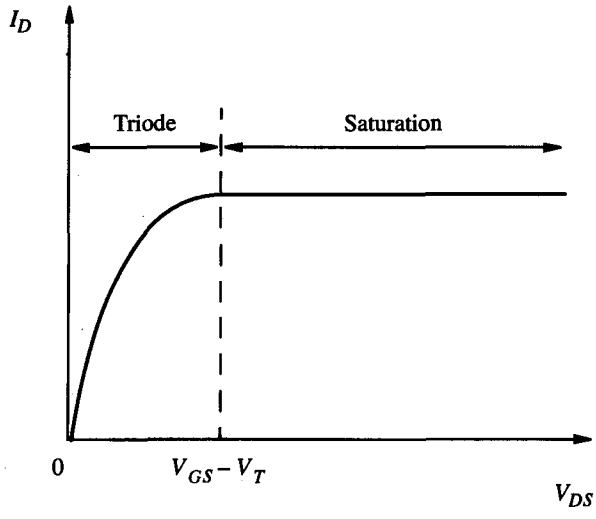


Figure 3.44 The current-voltage relationship in the NMOS transistor.

The PMOS Transistor

The behavior of PMOS transistors is the same as for NMOS except that all voltages and currents are reversed. The source terminal of the PMOS transistor is the terminal with the higher voltage level (recall that for an NMOS transistor the source terminal is the one with the lower voltage level), and the threshold voltage required to turn the transistor on has a negative value. PMOS transistors have the same physical construction as NMOS transistors except that wherever the NMOS transistor has type *n* silicon, the PMOS transistor has type *p*, and vice versa. For a PMOS transistor the equivalent of Figure 3.43*a* is to connect both the source and gate nodes to V_{DD}, in which case the transistor is turned off. To turn the PMOS transistor on, equivalent to Figure 3.43*b*, we would set the gate node to *Gnd*, resulting in $V_{GS} = -5$ V.

Because the channel is type *p* silicon, instead of type *n*, the physical mechanism for current conduction in PMOS transistors is different from that in NMOS transistors. A detailed discussion of this issue is beyond the scope of this book, but one implication has to be mentioned. Equations 3.1 and 3.2 use the parameter k'_n. The corresponding parameter for a PMOS transistor is k'_p, but current flows more readily in type *n* silicon than in type *p*, with the result that in a typical technology $k'_p \approx 0.4 \times k'_n$. For a PMOS transistor to have current capacity equal to that of an NMOS transistor, we must use W/L of about two to three times larger in the PMOS transistor. In logic gates the sizes of NMOS and PMOS transistors are usually chosen to account for this factor.

3.8.2 MOSFET ON-RESISTANCE

In section 3.1 we considered MOSFETs as ideal switches that have infinite resistance when turned off and zero resistance when on. The actual resistance in the channel when the transistor is turned on, referred to as the *on-resistance*, is given by V_{DS}/I_D. Using equation 3.1 we can calculate the on-resistance in the triode region, as shown in Example 3.4.

Example 3.4 Consider a CMOS inverter in which the input voltage V_x is equal to 5 V. The NMOS transistor is turned on, and the output voltage V_f is close to 0 V. Hence V_{DS} for the NMOS transistor is close to zero and the transistor is operating in the triode region. In the curve in Figure 3.44, the transistor is operating at a point very close to the origin. Although the value of V_{DS} is small, it is not exactly zero. In the next section we explain that V_{DS} would typically be about 0.1 mV. Hence the current I_D is not exactly zero; it is defined by equation 3.1. In this equation we can ignore the term involving V_{DS}^2 because V_{DS} is small. In this case the on-resistance is approximated by

$$R_{DS} = V_{DS}/I_D = 1/\left[k'_n \frac{W}{L}(V_{GS} - V_T)\right] \qquad \textbf{[3.3]}$$

Assuming the values $k'_n = 60\ \mu A/V^2$, $W/L = 2.0\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V, and $V_T = 1$ V, we get $R_{DS} \approx 1\ k\Omega$.

3.8.3 VOLTAGE LEVELS IN LOGIC GATES

In Figure 3.1 we showed that the logic values are represented by a range of voltage levels. We should now consider the issue of voltage levels more carefully.

The high and low voltage levels in a logic family are characterized by the operation of its basic inverter. Figure 3.45*a* reproduces the circuit in Figure 3.5 for an inverter built with NMOS technology. When $V_x = 0$ V, the NMOS transistor is turned off. No current flows; hence $V_f = 5$ V. When $V_x = V_{DD}$, the NMOS transistor is turned on. To calculate the value of V_f, we can represent the NMOS transistor by a resistor with the value R_{DS}, as illustrated in Figure 3.45*b*. Then V_f is given by the voltage divider

$$V_f = V_{DD}\frac{R_{DS}}{R_{DS} + R}$$

Example 3.5

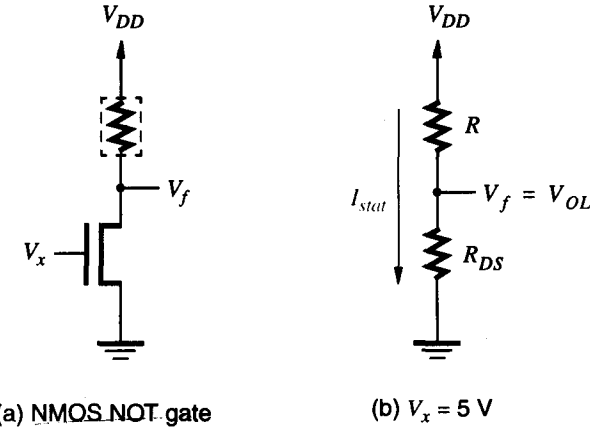
Assume that $R = 25$ kΩ. Using the result from Example 3.4, $R_{DS} = 1$ kΩ, which gives $V_f \approx 0.2$ V.

As indicated in Figure 3.45*b*, a current I_{stat} flows through the NMOS inverter under the static condition $V_x = V_{DD}$. This current is given by

$$I_{stat} = V_f/R_{DS} = 0.2 \text{ V}/1 \text{ k}\Omega = 0.2 \text{ mA}$$

This static current has important implications, which we discuss in section 3.8.6.

In modern NMOS circuits, the pull-up device R is implemented using a PMOS transistor. Such circuits are referred to as *pseudo-NMOS circuits*. They are fully compatible with CMOS circuits; hence a single chip may contain both CMOS and pseudo-NMOS gates. Problem 3.20 shows the circuit for a pseudo-NMOS inverter and discusses how to calculate its output voltage levels.



(a) NMOS NOT gate (b) $V_x = 5$ V

Figure 3.45 Voltage levels in the NMOS inverter.

The CMOS Inverter

It is customary to use the symbols V_{OH} and V_{OL} to characterize the voltage levels in a logic circuit. The meaning of V_{OH} is the voltage produced when the output is high. Similarly, V_{OL} refers to the voltage produced when the output is low. As discussed above, in the NMOS inverter $V_{OH} = V_{DD}$ and V_{OL} is about 0.2 V.

Consider again the CMOS inverter in Figure 3.12*a*. Its output-input voltage relationship is summarized by the *voltage transfer characteristic* shown in Figure 3.46. The curve gives the steady-state value of V_f for each value of V_x. When $V_x = 0$ V, the NMOS transistor is off. No current flows; hence $V_f = V_{OH} = V_{DD}$. When $V_x = V_{DD}$, the PMOS transistor is off, no current flows, and $V_f = V_{OL} = 0$ V. For completeness we should mention that even when a transistor is turned off, a small current, called the *leakage* current, may flow through it. This current has a slight effect on V_{OH} and V_{OL}. For example, a typical value of V_{OL} is 0.1 mV, rather than 0 V [1].

Figure 3.46 includes labels at the points where the output voltage begins to change from high to low, and vice versa. The voltage V_{IL} represents the point where the output voltage is high and the slope of the curve equals -1. This voltage level is defined as the maximum input voltage level that the inverter will interpret as low, hence producing a high output. Similarly, the voltage V_{IH}, which is the other point on the curve where the slope equals -1,

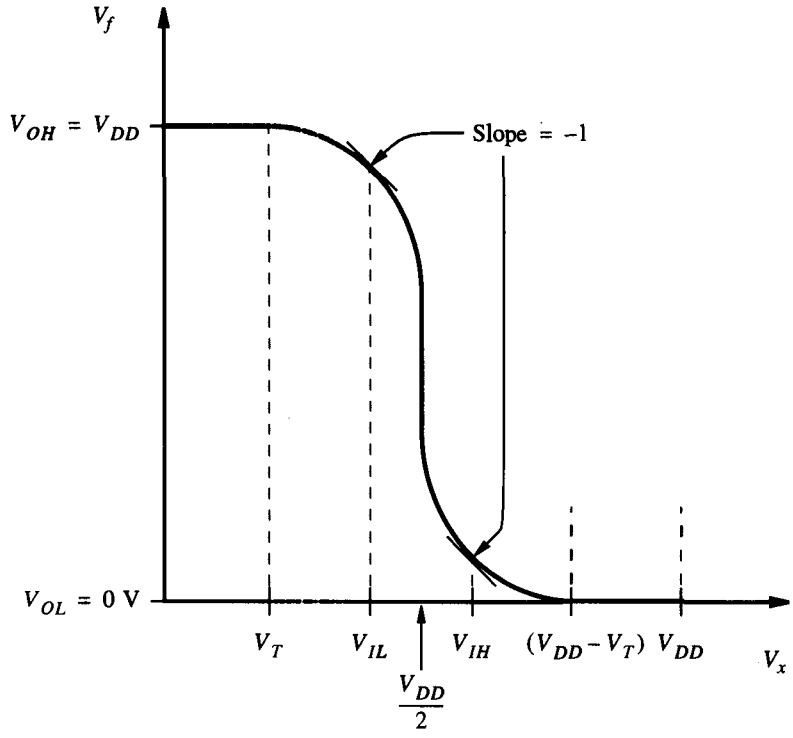


Figure 3.46 The voltage transfer characteristic for the CMOS inverter.

is the minimum input voltage level that the inverter will interpret as high, hence producing a low output. The parameters V_{OH}, V_{OL}, V_{IL}, and V_{IH} are important for quantifying the robustness of a logic family, as discussed below.

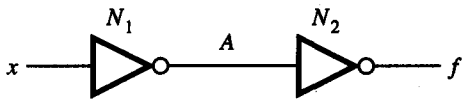
3.8.4 NOISE MARGIN

Consider the two NOT gates shown in Figure 3.47*a*. Let us refer to the gates on the left and right as N_1 and N_2, respectively. Electronic circuits are constantly subjected to random perturbations, called *noise*, which can alter the output voltage levels produced by the gate N_1. It is essential that this noise not cause the gate N_2 to misinterpret a low logic value as a high one, or vice versa. Consider the case where N_1 produces its low voltage level V_{OL}. The presence of noise may alter the voltage level, but as long as it remains less than V_{IL}, it will be interpreted correctly by N_2. The ability to tolerate noise without affecting the correct operation of the circuit is known as *noise margin*. For the low output voltage, we define the low *noise margin* as

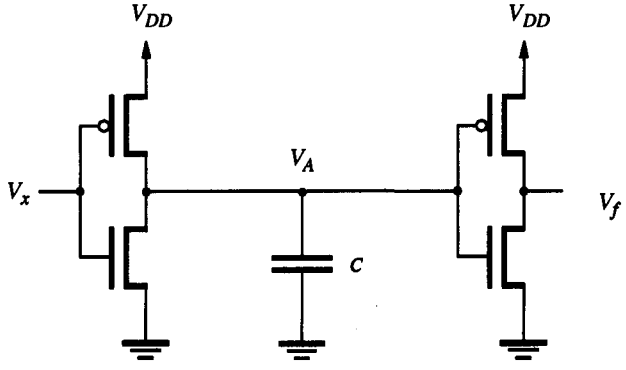
$$NM_L = V_{IL} - V_{OL}$$

A similar situation exists when N_1 produces its high output voltage V_{OH}. Any existing noise in the circuit may alter the voltage level, but it will be interpreted correctly by N_2 as long as the voltage is greater than V_{IH}. The high noise margin is defined as

$$NM_H = V_{OH} - V_{IH}$$



(a) A NOT gate driving another NOT gate



(b) The capacitive load at node A

Figure 3.47 Parasitic capacitance in integrated circuits.

Example 3.6 For a given technology the voltage transfer characteristic of the basic inverter determines the levels V_{OH}, V_{OL}, V_{IL}, and V_{IH}. For CMOS we showed in Figure 3.46 that $V_{OH} = V_{DD}$ and $V_{OL} = 0$ V. By finding the two points where the slope of the voltage transfer characteristic is equal to -1, it can be shown [1] that $V_{IL} \cong \frac{1}{8}(3V_{DD} + 2V_T)$ and $V_{IH} \cong \frac{1}{8}(5V_{DD} - 2V_T)$. For the typical value $V_T = 0.2\ V_{DD}$, this gives

$$NM_L = NM_H = 0.425 \times V_{DD}$$

Hence the available noise margin depends on the power supply voltage level. For $V_{DD} = 5$ V, the noise margin is 2.1 V, and for $V_{DD} = 3.3$ V, the noise margin is 1.4 V.

3.8.5 DYNAMIC OPERATION OF LOGIC GATES

In Figure 3.47*a* the node between the two gates is labeled *A*. Because of the way in which transistors are constructed in silicon, N_2 has the effect of contributing to a capacitive load at node *A*. Figure 3.43 shows that transistors are constructed by using several layers of different materials. Wherever two types of material meet or overlap inside the transistor, a capacitor may be effectively created. This capacitance is called *parasitic*, or *stray*, capacitance because it results as an undesired side effect of transistor fabrication. In Figure 3.47 we are interested in the capacitance that exists at node *A*. A number of parasitic capacitors are attached to this node, some caused by N_1 and others caused by N_2. One significant parasitic capacitor exists between the input of inverter N_2 and ground. The value of this capacitor depends on the sizes of the transistors in N_2. Each transistor contributes a *gate capacitance*, $C_g = W \times L \times C_{ox}$. The parameter C_{ox}, called the *oxide capacitance*, is a constant for the technology being used and has the units fF/μm^2. Additional capacitance is caused by the transistors in N_1 and by the metal wiring that is attached to node *A*. It is possible to represent all of the parasitic capacitance by a single equivalent capacitance between node *A* and ground [2]. In Figure 3.47*b* this equivalent capacitance is labeled *C*.

The existence of stray capacitance has a negative effect on the speed of operation of logic circuits. Voltage across a capacitor cannot change instantaneously. The time needed to charge or discharge a capacitor depends on the size of the capacitance *C* and on the amount of current through the capacitor. In the circuit of Figure 3.47*b*, when the PMOS transistor in N_1 is turned on, the capacitor is charged to V_{DD}; it is discharged when the NMOS transistor is turned on. In each case the current flow I_D through the involved transistor and the value of *C* determine the rate of charging and discharging the capacitor.

Chapter 2 introduced the concept of a timing diagram, and Figure 2.10 shows a timing diagram in which waveforms have perfectly vertical edges in transition from one logic level to the other. In real circuits, waveforms do not have this "ideal" shape, but instead have the appearance of those in Figure 3.48. The figure gives a waveform for the input V_x in Figure 3.47*b* and shows the resulting waveform at node *A*. We assume that V_x is initially at the voltage level V_{DD} and then makes a transition to 0. Once V_x reaches a sufficiently low voltage, N_1 begins to drive voltage V_A toward V_{DD}. Because of the parasitic capacitance, V_A cannot change instantaneously and a waveform with the shape indicated in the figure results. The time needed for V_A to change from low to high is called the *rise time*, t_r, which

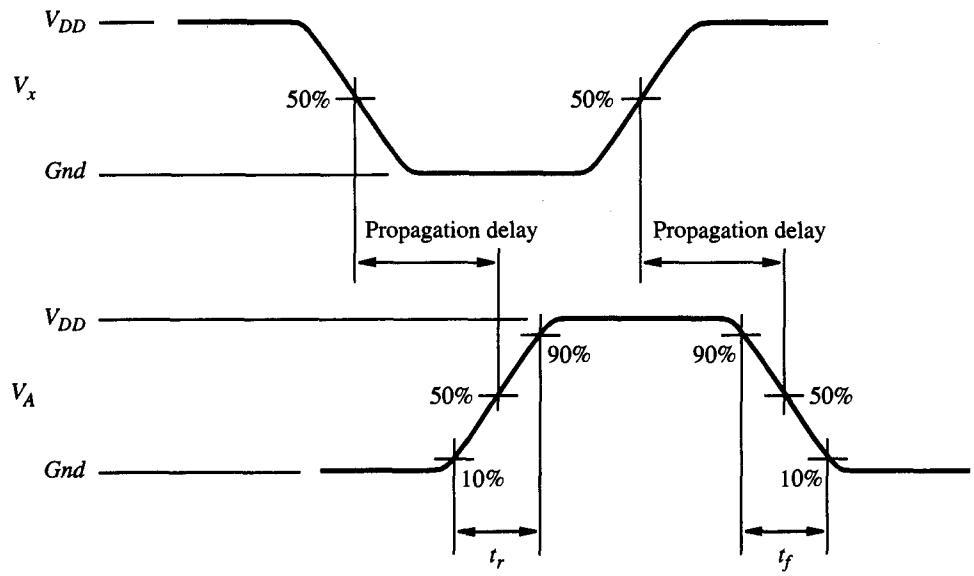


Figure 3.48 Voltage waveforms for logic gates.

is defined as the time elapsed from when V_A is at 10 percent of V_{DD} until it reaches 90 percent of V_{DD}. Figure 3.48 also defines the total amount of time needed for the change at V_x to cause a change in V_A. This interval is called the *propagation delay*, often written t_p, of the inverter. It is the time from when V_x reaches 50 percent of V_{DD} until V_A reaches the same level.

After remaining at 0 V for some time, V_x then changes back to V_{DD}, causing N_1 to discharge C to *Gnd*. In this case the transition time at node A pertains to a change from high to low, which is referred to as the *fall time*, t_f, from 90 percent of V_{DD} to 10 percent of V_{DD}. As indicated in the figure, there is a corresponding propagation delay for the new change in V_x to affect V_A. In a given logic gate, the relative sizes of the PMOS and NMOS transistors are usually chosen such that t_r and t_f have about the same value.

Equations 3.1 and 3.2 specify the amount of current flow through an NMOS transistor. Given the value of C in Figure 3.47, it is possible to calculate the propagation delay for a change in V_A from high to low. For simplicity, assume that V_x is initially 0 V; hence the PMOS transistor is turned on, and $V_A = 5$ V. Then V_x changes to V_{DD} at time 0, causing the PMOS transistor to turn off and the NMOS to turn on. The propagation delay is then the time required to discharge C through the NMOS transistor to the voltage $V_{DD}/2$. When V_x first changes to V_{DD}, $V_A = 5$ V; hence the NMOS transistor will have $V_{DS} = V_{DD}$ and will be in the saturation region. The current I_D is given by equation 3.2. Once V_A drops below $V_{DD} - V_T$, the NMOS transistor will enter the triode region where I_D is given by equation 3.1. For our purposes, we can approximate the current flow as V_A changes from V_{DD} to $V_{DD}/2$ by finding the average of the values given by equation 3.2 with $V_{DS} = V_{DD}$ and equation 3.1 with $V_{DS} = V_{DD}/2$. Using the basic expression for the time needed to

charge a capacitor, we have

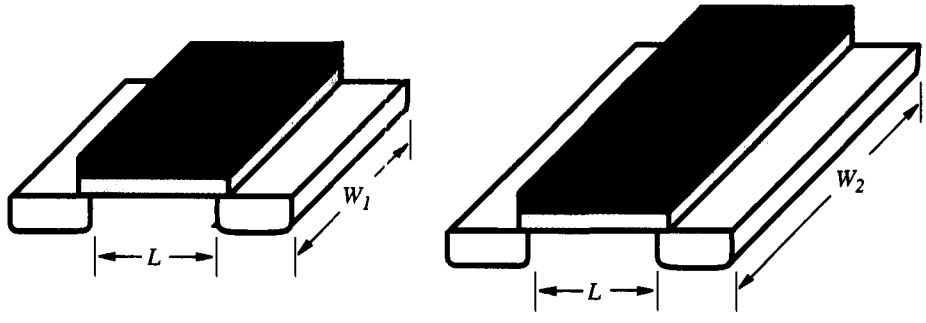
$$t_p = \frac{C\Delta V}{I_D} = \frac{CV_{DD}/2}{I_D}$$

Substituting for the average value of I_D as discussed above, yields [1]

$$t_p \cong \frac{1.7\,C}{k_n' \frac{W}{L} V_{DD}} \qquad \textbf{[3.4]}$$

This expression specifies that the speed of the circuit depends both on the value of C and on the dimensions of the transistor. The delay can be reduced by making C smaller or by making the ratio W/L larger. The expression shows the propagation time when the output changes from a high level to a low level. The low-to-high propagation time is given by the same expression but using k_p' and W/L of the PMOS transistor.

In logic circuits, L is usually set to the minimum value that is permitted according to the specifications of the fabrication technology used. The value of W is chosen depending on the amount of current flow, hence propagation delay, that is desired. Figure 3.49 illustrates two sizes of transistors. Part (*a*) depicts a minimum-size transistor, which would be used in a circuit wherever capacitive loading is small or where speed of operation is not critical. Figure 3.49*b* shows a larger transistor, which has the same length as the transistor in part (*a*) but a larger width. There is a trade-off involved in choosing transistor sizes, because a larger transistor takes more space on a chip than a smaller one. Also, increasing W not only increases the amount of current flow in the transistor but also results in an increase in the parasitic capacitance (recall that the capacitance C_g between the gate terminal and ground is proportional to $W \times L$), which tends to offset some of the expected improvement in performance. In logic circuits large transistors are used where high capacitive loads must be driven and where signal propagation delays must be minimized.



(a) Small transistor (b) Larger transistor

Figure 3.49 Transistor sizes.

Example 3.7

In the circuit in Figure 3.47, assume that $C = 70$ fF and that $W/L = 2.0\ \mu m/0.5\ \mu m$. Also, $k_n' = 60\ \mu A/V^2$ and $V_{DD} = 5$ V. Using equation 3.4, the high-to-low propagation delay of the inverter is $t_p \approx 0.1$ ns.

3.8.6 POWER DISSIPATION IN LOGIC GATES

In an electronic circuit it is important to consider the amount of electrical power consumed by the transistors. Integrated circuit technology allows fabrication of millions of transistors on a single chip; hence the amount of power used by an individual transistor must be small. Power dissipation is an important consideration in all applications of logic circuits, but it is crucial in situations that involve battery-operated equipment, such as portable computers and the like.

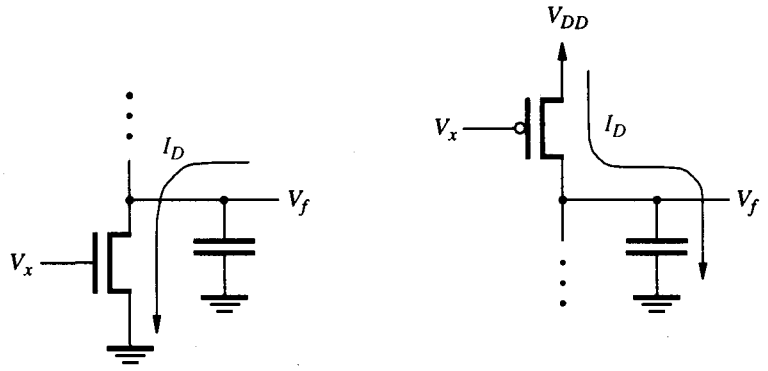
Consider again the NMOS inverter in Figure 3.45. When $V_x = 0$, no current flows and hence no power is used. But when $V_x = 5$ V, power is consumed because of the current I_{stat}. The power consumed in the steady state is given by $P_S = I_{stat} V_{DD}$. In Example 3.5 we calculated $I_{stat} = 0.2$ mA. The power consumed is then $P_S = 0.2\ \text{mA} \times 5\ \text{V} = 1.0$ mW. If we assume that a chip contains, say, the equivalent of 10,000 inverters, then the total power consumption is 10 W! Because of this large power consumption, NMOS-style gates are used only in special-purpose applications, which we discuss in section 3.8.8.

To distinguish between power consumed during steady-state conditions and power consumed when signals are changing, it is customary to define two types of power. *Static power* is dissipated by the current that flows in the steady state, and *dynamic power* is consumed when the current flows because of changes in signal levels. NMOS circuits consume static power as well as dynamic power, while CMOS circuits consume only dynamic power.

Consider the CMOS inverter presented in Figure 3.12*a*. When the input V_x is low, no current flows because the NMOS transistor is off. When V_x is high, the PMOS transistor is off and again no current flows. Hence no current flows in a CMOS circuit under steady-state conditions. Current does flow in CMOS circuits, however, for a short time when signals change from one voltage level to another.

Figure 3.50*a* depicts the following situation. Assume that V_x has been at 0 V for some time; hence $V_f = 5$ V. Now let V_x change to 5 V. The NMOS transistor turns on, and it pulls V_f toward *Gnd*. Because of the parasitic capacitance C at node f, voltage V_f does not change instantaneously, and current I_D flows through the NMOS transistor for a short time while the capacitor is being discharged. A similar situation occurs when V_x changes from 5 V to 0, as illustrated in Figure 3.50*b*. Here the capacitor C initially has 0 volts across it and is then charged to 5 V by the PMOS transistor. Current flows from the power supply through the PMOS transistor while the capacitor is being charged.

The voltage transfer characteristic for the CMOS inverter, shown in Figure 3.46, indicates that a range of input voltage V_x exists for which both transistors in the inverter are turned on. Within this voltage range, specifically $V_T < V_x < (V_{DD} - V_T)$, current flows from V_{DD} to *Gnd* through both transistors. This current is often referred to as the *short-circuit current* in the gate. In comparison to the amount of current used to (dis)charge the capacitor C, the short-circuit current is negligible.



(a) Current flow when input V_x changes from 0 V to 5 V

(b) Current flow when input V_x changes from 5 V to 0 V

Figure 3.50 Dynamic current flow in CMOS circuits.

The power used by a single CMOS inverter is extremely small. Consider again the situation in Figure 3.50*a* when $V_f = V_{DD}$. The amount of energy stored in the capacitor is equal to $CV_{DD}^2/2$. When the capacitor is discharged to 0 V, this stored energy is dissipated in the NMOS transistor. Similarly, for the situation in Figure 3.50*b*, the energy $CV_{DD}^2/2$ is dissipated in the PMOS transistor when C is charged up to V_{DD}. Thus for each cycle in which the inverter charges and discharges C, the amount of energy dissipated is equal to CV_{DD}^2. Since power is defined as energy used per unit time, the power dissipated in the inverter is the product of the energy used in one discharge/charge cycle times the number of such cycles per second, f. Hence the dynamic power consumed is

$$P_D = fCV_{DD}^2$$

In practice, the total amount of dynamic power used in CMOS circuits is significantly lower than the total power needed in other technologies, such as NMOS. For this reason, virtually all large integrated circuits fabricated today are based on CMOS technology.

Example 3.8 For a CMOS inverter, assume that $C = 70$ fF and $f = 100$ MHz. The dynamic power consumed by the gate is $P_D = 175\ \mu$W. If we assume that a chip contains the equivalent of 10,000 inverters and that, on average, 20 percent of the gates change values at any given time, then the total amount of dynamic power used in the chip is $P_D = 0.2 \times 10,000 \times 0.175\ \mu\text{W} = 0.35$ mW.

3.8.7 PASSING 1S AND 0S THROUGH TRANSISTOR SWITCHES

In Figure 3.4 we showed that NMOS transistors are used as pull-down devices and PMOS transistors are used as pull-up devices. We now consider using the transistors in the opposite

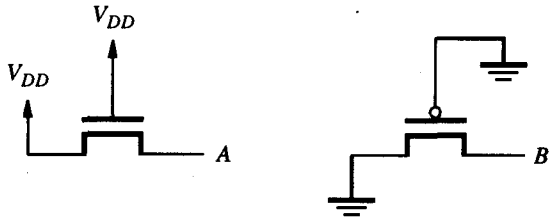
way, that is, using an NMOS transistor to drive an output high and using a PMOS transistor to drive an output low.

Figure 3.51*a* illustrates the case of an NMOS transistor for which both the gate terminal and one side of the switch are driven to V_{DD}. Let us assume initially that both V_G and node A are at 0 V, and we then change V_G to 5 V. Node A is the transistor's source terminal because it has the lowest voltage. Since $V_{GS} = V_{DD}$, the transistor is turned on and drives node A toward V_{DD}. When the voltage at node A rises, V_{GS} decreases until the point when V_{GS} is no longer greater than V_T. At this point the transistor turns off. Thus in the steady state $V_A = V_{DD} - V_T$, which means that an NMOS transistor can only partially pass a high voltage signal.

A similar situation occurs when a PMOS transistor is used to pass a low voltage level, as depicted in Figure 3.51*b*. Here assume that initially both V_G and node B are at 5 V. Then we change V_G to 0 V so that the transistor turns on and drives the source node (node B) toward 0 V. When node B is decreased to V_T, the transistor turns off; hence the steady-state voltage is equal to V_T.

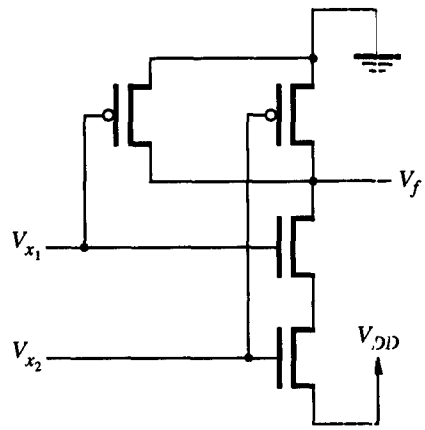
In section 3.1 we said that for an NMOS transistor the substrate (body) terminal is connected to *Gnd* and for a PMOS transistor the substrate is connected to V_{DD}. The voltage between the source and substrate terminals, V_{SB}, which is called the *substrate bias voltage*, is normally equal to 0 V in a logic circuit. But in Figure 3.51 both the NMOS and PMOS transistors have $V_{SB} = V_{DD}$. The bias voltage has the effect of increasing the threshold voltage in the transistor V_T by a factor of about 1.5 or higher [2, 1]. This issue is known as the *body effect*.

Consider the logic gate shown in Figure 3.52. In this circuit the V_{DD} and *Gnd* connections are reversed from the way in which they were used in previously discussed circuits. When both V_{x_1} and V_{x_2} are high, then V_f is pulled up to the high output voltage, $V_{OH} = V_{DD} - 1.5\,V_T$. If $V_{DD} = 5$ V and $V_T = 1$ V, then $V_{OH} = 3.5$ V. When either V_{x_1} or V_{x_2} is low, then V_f is pulled down to the low output voltage, $V_{OL} = 1.5\,V_T$, or about 1.5 V. As shown by the truth table in the figure, the circuit represents an AND gate. In comparison to the normal AND gate shown in Figure 3.15, the circuit in Figure 3.52 appears to be better because it requires fewer transistors. But a drawback of this circuit is that it offers a lower noise margin because of the poor levels of V_{OH} and V_{OL}.



(a) NMOS transistor (b) PMOS transistor

Figure 3.51 NMOS and PMOS transistors used in the opposite way from Figure 3.4.



(a) An AND gate circuit

Logic value		Voltage	Logic value
x_1	x_2	V_f	f
0	0	1.5 V	0
0	1	1.5 V	0
1	0	1.5 V	0
1	1	3.5 V	1

(b) Truth table and voltage levels

Figure 3.52 A poor implementation of a CMOS AND gate.

Another important weakness of the circuit in Figure 3.52 is that it causes static power dissipation, unlike a normal CMOS AND gate. Assume that the output of such an AND gate drives the input of a CMOS inverter. When $V_f = 3.5$ V, the NMOS transistor in the inverter is turned on and the inverter output has a low voltage level. But the PMOS transistor in the inverter is not turned off, because its gate-to-source voltage is -1.5 V, which is larger than V_T. Static current flows from V_{DD} to *Gnd* through the inverter. A similar situation occurs when the AND gate produces the low output $V_f = 1.5$ V. Here the PMOS transistor in the inverter is turned on, but the NMOS transistor is not turned off. The AND gate implementation in Figure 3.52 is not used in practice.

3.8.8 FAN-IN AND FAN-OUT IN LOGIC GATES

The *fan-in* of a logic gate is defined as the number of inputs to the gate. Depending on how a logic gate is constructed, it may be impractical to increase the number of inputs beyond a small number. For example, consider the NMOS NAND gate in Figure 3.53, which has k inputs. We wish to consider the effect of k on the propagation delay t_p through the gate. Assume that all k NMOS transistors have the same width W and length L. Because the transistors are connected in series, we can consider them to be equivalent to one long transistor with length $k \times L$ and width W. Using equation 3.4 (which can be applied to both CMOS and NMOS gates), the propagation delay is given by

$$t_p \cong \frac{1.7\,C}{k'_n \frac{W}{L} V_{DD}} \times k$$

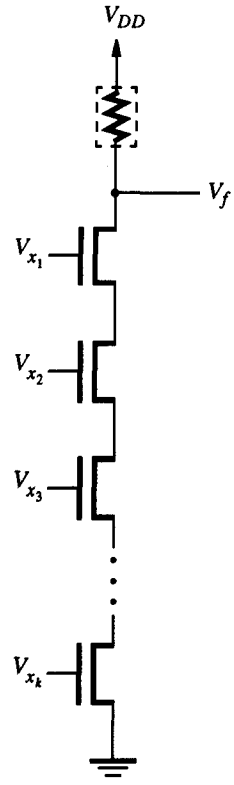


Figure 3.53 High fan-in NMOS NAND gate.

Here C is the equivalent capacitance at the output of the gate, including the parasitic capacitance contributed by each of the k transistors. The performance of the gate can be improved somewhat by increasing W for each NMOS transistor. But this change further increases C and comes at the expense of chip area. Another drawback of the circuit is that each NMOS transistor has the effect of increasing V_{OL}, hence reducing the noise margin. It is practical to build NAND gates in this manner only if the fan-in is small.

As another example of fan-in, Figure 3.54 shows an NMOS k-input NOR gate. In this case the k NMOS transistors connected in parallel can be viewed as one large transistor with width $k \times W$ and length L. According to equation 3.4, the propagation delay should be decreased by the factor k. However, the parallel-connected transistors increase the load capacitance C at the gate's output and, more importantly, it is extremely unlikely that all of the transistors would be turned on when V_f is changing from a high to low level. It is thus practical to build high fan-in NOR gates in NMOS technology. We should note, however, that in an NMOS gate the low-to-high propagation delay may be slower than the high-to-low delay as a result of the current-limiting effect of the pull-up device (see problems 3.20 and 3.21).

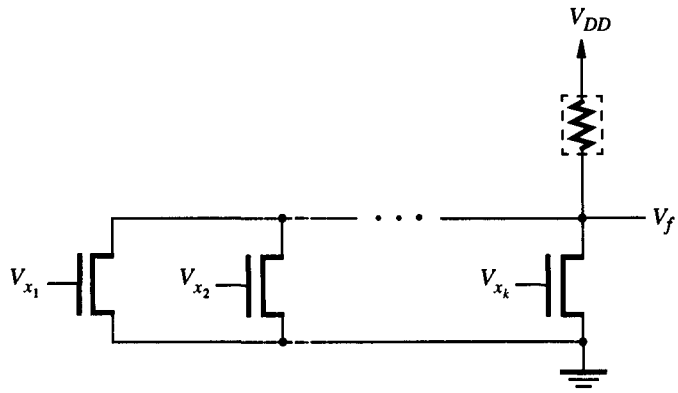


Figure 3.54 High fan-in NMOS NOR gate.

High fan-in CMOS logic gates always require either k NMOS or k PMOS transistors in series and are therefore never practical. In CMOS the only reasonable way to construct a high fan-in gate is to use two or more lower fan-in gates. For example, one way to realize a six-input AND gate is as 2 three-input AND gates that connect to a two-input AND gate. It is possible to build a six-input CMOS AND gate using fewer transistors than needed with this approach, but we leave this as an exercise for the reader (see problem 3.4).

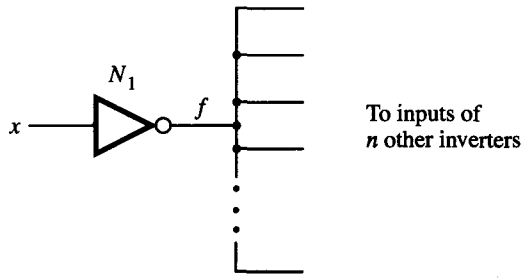
Fan-out

Figure 3.48 illustrated timing delays for one NOT gate driving another. In real circuits each logic gate may be required to drive several others. The number of other gates that a specific gate drives is called its *fan-out*. An example of fan-out is depicted in Figure 3.55*a*, which shows an inverter N_1 that drives the inputs of n other inverters. Each of the other inverters contributes to the total capacitive loading on node f. In part (*b*) of the figure, the n inverters are represented by one large capacitor C_n. For simplicity, assume that each inverter contributes a capacitance C and that $C_n = n \times C$. Equation 3.4 shows that the propagation delay increases in direct proportion to n.

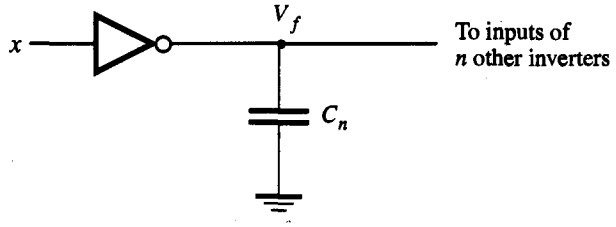
Figure 3.55*c* illustrates how n affects the propagation delay. It assumes that a change from logic value 1 to 0 on signal x occurs at time 0. One curve represents the case where $n = 1$, and the other curve corresponds to $n = 4$. Using the parameters from Example 3.7, when $n = 1$, we have $t_p = 0.1$ ns. Then for $n = 4$, $t_p \approx 0.4$ ns. It is possible to reduce t_p by increasing the W/L ratios of the transistors in N_1.

Buffers

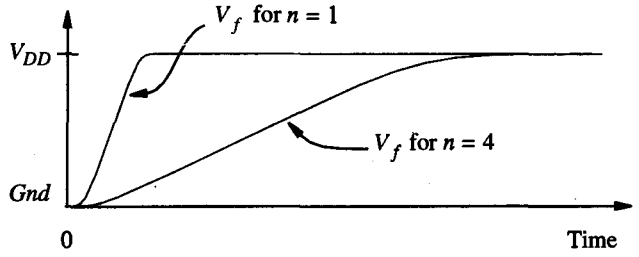
In circuits in which a logic gate has to drive a large capacitive load, buffers are often used to improve performance. A *buffer* is a logic gate with one input, x, and one output, f, which produces $f = x$. The simplest implementation of a buffer uses two inverters, as shown in Figure 3.56*a*. Buffers can be created with different amounts of *drive capability*, depending on the sizes of the transistors (see Figure 3.49). In general, because they are used for driving higher-than-normal capacitive loads, buffers have transistors that are larger than those in typical logic gates. The graphical symbol for a noninverting buffer is given in Figure 3.56*b*.



(a) Inverter that drives n other inverters



(b) Equivalent circuit for timing purposes

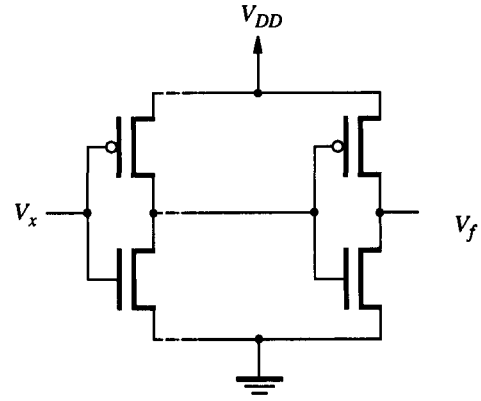


(c) Propagation times for different values of n

Figure 3.55 The effect of fan-out on propagation delay.

Another type of buffer is the *inverting buffer*. It produces the same output as an inverter, $f = \bar{x}$, but is built with relatively large transistors. The graphical symbol for the inverting buffer is the same as for the NOT gate; an inverting buffer is just a NOT gate that is capable of driving large capacitive loads. In Figure 3.55 for large values of n an inverting buffer could be used for the inverter labeled N_1.

In addition to their use for improving the speed performance of circuits, buffers are also used when high current flow is needed to drive external devices. Buffers can handle relatively large amounts of current flow because they are built with large transistors. A common example of this use of buffers is to control a light-emitting diode (LED). We describe an example of this application of buffers in section 7.14.3.



(a) Imp ementation of a buffer

(b) Graphical symbol

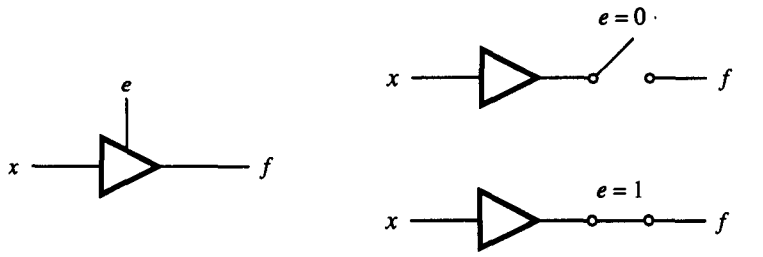
Figure 3.56 A noninverting buffer.

In general, fan-out, capacitive loading, and current flow are important issues that the designer of a digital circuit must consider carefully. In practice, the decision as to whether or not buffers are needed in a circuit is made with the aid of CAD tools.

Tri-state Buffers

In section 3.6.2 we mentioned that a type of buffer called a tri-state buffer is included in some standard chips and in PLDs. A tri-state buffer has one input, x, one output, f, and a control input, called *enable*, e. The graphical symbol for a tri-state buffer is given in Figure 3.57*a*. The enable input is used to determine whether or not the tri-state buffer produces an output signal, as illustrated in Figure 3.57*b*. When $e = 0$, the buffer is completely disconnected from the output f. When $e = 1$, the buffer drives the value of x onto f, causing $f = x$. This behavior is described in truth-table form in part (*c*) of the figure. For the two rows of the table where $e = 0$, the output is denoted by the logic value Z, which is called the *high-impedance state*. The name *tri-state* derives from the fact that there are two normal states for a logic signal, 0 and 1, and Z represents a third state that produces no output signal. Figure 3.57*d* shows a possible implementation of the tri-state buffer.

Figure 3.58 shows several types of tri-state buffers. The buffer in part (*b*) has the same behavior as the buffer in part (*a*), except that when $e = 1$, it produces $f = \bar{x}$. Part (*c*) of the figure gives a tri-state buffer for which the enable signal has the opposite behavior; that is, when $e = 0, f = x$, and when $e = 1, f = Z$. The term often used to describe this type of behavior is to say that the enable is *active low*. The buffer in Figure 3.58*d* also features an active-low enable, and it produces $f = \bar{x}$ when $e = 0$.

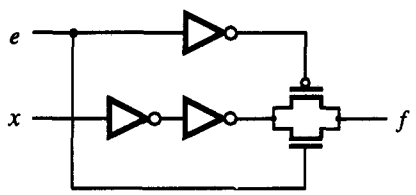


(a) A tri-state buffer

(b) Equivalent circuit

e	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

(c) Truth table



(d) Implementation

Figure 3.57 Tri-state buffer.

As a small example of how tri-state buffers can be used, consider the circuit in Figure 3.59. In this circuit the output f is equal to either x_1 or x_2, depending on the value of s. When $s = 0, f = x_1$, and when $s = 1, f = x_2$. Circuits of this kind, which choose one of the inputs and reproduce the signal on this input at the output terminal, are called *multiplexer* circuits. A circuit that implements the multiplexer using AND and OR gates is shown in

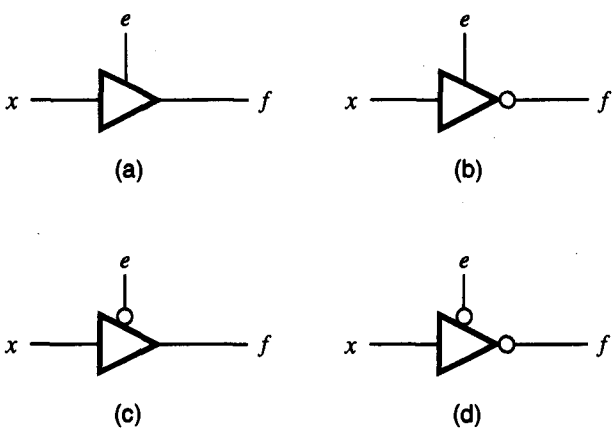


Figure 3.58 Four types of tri-state buffers.

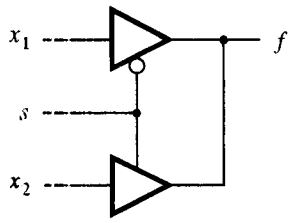


Figure 3.59 An application of tri-state buffers.

Figure 2.26. We will present another way of building multiplexer circuits in section 3.9.2 and will discuss them in detail in Chapter 6.

In the circuit of Figure 3.59, the outputs of the tri-state buffers are wired together. This connection is possible because the control input s is connected so that one of the two buffers is guaranteed to be in the high-impedance state. The x_1 buffer is active only when $s = 0$, and the x_2 buffer is active only when $s = 1$. It would be disastrous to allow both buffers to be active at the same time. Doing so would create a short circuit between V_{DD} and *Gnd* as soon as the two buffers produce different values. For example, assume that $x_1 = 1$ and $x_2 = 0$. The x_1 buffer produces the output V_{DD}, and the x_2 buffer produces *Gnd*. A short circuit is formed between V_{DD} and *Gnd*, through the transistors in the tri-state buffers. The amount of current that flows through such a short circuit is usually sufficient to destroy the circuit.

The kind of *wired* connection used for the tri-state buffers is not possible with ordinary logic gates, because their outputs are always active; hence a short circuit would occur. As we already know, for normal logic circuits the equivalent result of the wired connection is achieved by using an OR gate to combine signals, as is done in the sum-of-products form.

3.9 TRANSMISSION GATES

In section 3.8.7 we showed that an NMOS transistor passes 0 well and 1 poorly, while a PMOS transistor passes 1 well and 0 poorly. It is possible to combine an NMOS and a PMOS transistor into a single switch that is capable of driving its output terminal either to a low or high voltage equally well. Figure 3.60*a* gives the circuit for a *transmission gate*. As indicated in parts (*b*) and (*c*) of the figure, it acts as a switch that connects x to f. Switch control is provided by the *select* input s and its complement $\bar{s}$. The switch is turned on by setting $V_s = 5$ V and $V_{\bar{s}} = 0$. When V_x is 0, the NMOS transistor will be turned on (because $V_{GS} = V_s - V_x = 5$ V) and V_f will be 0. On the other hand, when V_x is 5 V, then the PMOS transistor will be on ($V_{GS} = V_{\bar{s}} - V_x = -5$ V) and V_f will be 5 V. A graphical symbol for the transmission gate is given in Figure 3.60*d*.

Transmission gates can be used in a variety of applications. We will show next how they lead to efficient implementations of *Exclusive OR (XOR)* logic gates and multiplexer circuits.

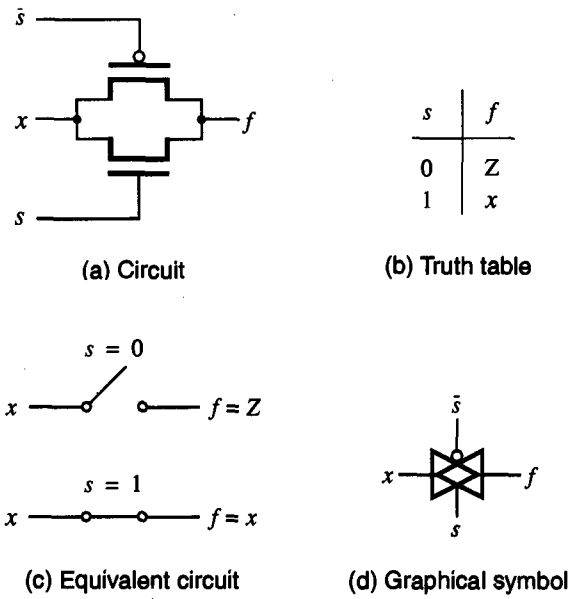


Figure 3.60 A transmission gate.

3.9.1 EXCLUSIVE-OR GATES

So far we have encountered AND, OR, NOT, NAND, and NOR gates as the basic elements from which logic circuits can be constructed. There is another basic element that is very useful in practice, particularly for building circuits that perform arithmetic operations, as we will see in Chapter 5. This element realizes the Exclusive-OR function defined in Figure 3.61*a*. The truth table for this function is similar to the OR function except that $f = 0$ when both inputs are 1. Because of this similarity, the function is called Exclusive-OR, which is commonly abbreviated as XOR. The graphical symbol for a gate that implements XOR is given in part (*b*) of the figure.

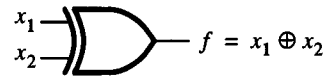
The XOR operation is usually denoted with the $\oplus$ symbol. It can be realized in the sum-of-products form as

$$x_1 \oplus x_2 = \bar{x}_1 x_2 + x_1 \bar{x}_2$$

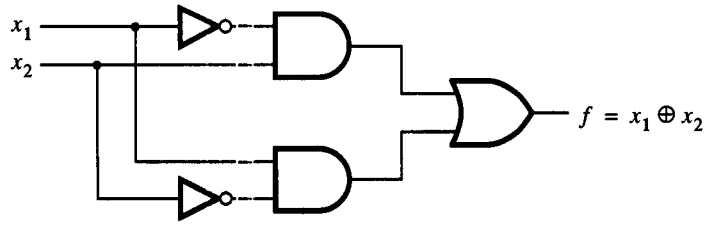
which leads to the circuit in Figure 3.61*c*. We know from section 3.3 that each AND and OR gate requires six transistors, while a NOT gate needs two transistors. Hence 22 transistors are required to implement this circuit in CMOS technology. It is possible to greatly reduce the number of transistors needed by making use of transmission gates. Figure 3.61*d* gives a circuit for an XOR gate that uses two transmission gates and two inverters. The output f is set to the value of x_2 when $x_1 = 0$ by the top transmission gate. The bottom transmission gate sets f to $\bar{x}_2$ when $x_1 = 1$. The reader can verify that this circuit properly implements the XOR function. We show how such circuits are derived in Chapter 6.

x_1	x_2	$f = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

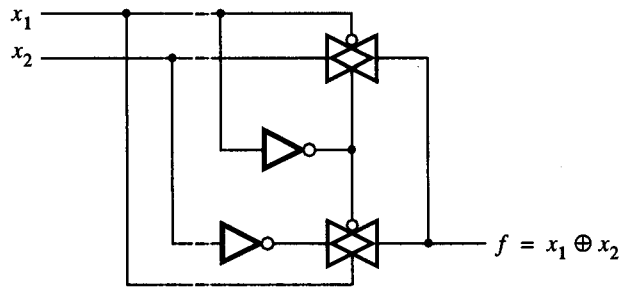
(a) Truth table



(b) Graphical symbol



(c) Sum-of-products implementation



(d) CMOS implementation

Figure 3.61 Exclusive-OR gate.

3.9.2 MULTIPLEXER CIRCUIT

In Figure 3.59 we showed how a multiplexer can be constructed with tri-state buffers. A similar structure can be used to realize a multiplexer with transmission gates, as indicated in Figure 3.62. The select input s is used to choose whether the output f should have the value of input x_1 or x_2. If $s = 0$, then $f = x_1$; if $s = 1$, then $f = x_2$.

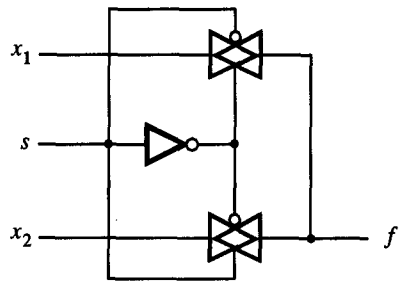


Figure 3.62 A 2-to-1 multiplexer built using transmission gates.

3.10 IMPLEMENTATION DETAILS FOR SPLDs, CPLDs, AND FPGAs

We introduced PLDs in section 3.6. In the chip diagrams shown in that section, the programmable switches are represented using the symbol X. We now show how these switches are implemented using transistors.

In commercial SPLDs two main technologies are used to manufacture the programmable switches. The oldest technology is based on using metal-alloy fuses as programmable links. In this technology the PLAs and PALs are manufactured so that each pair of horizontal and vertical wires that cross is connected by a small metal fuse. When the chip is programmed, for every connection that is not wanted in the circuit being implemented, the associated fuse is melted. The programming process is not reversible, because the melted fuses are destroyed. We will not elaborate on this technology, because it has mostly been replaced by a newer, better method.

In currently produced PLAs and PALs, programmable switches are implemented using a special type of *programmable transistor*. Because CPLDs comprise PAL-like blocks, the technology used in SPLDs is also applicable to CPLDs. We will illustrate the main ideas by first describing PLAs. For a PLA to be useful for implementing a wide range of logic functions, it should support both functions of only a few variables and functions of many variables. In section 3.8.8 we discussed the issue of fan-in of logic gates. We showed that when the fan-in is high, the best type of gate to use is the NMOS NOR gate. Hence PLAs are usually based on this type of gate.

As a small example of PLA implementation, consider the circuit in Figure 3.63. The horizontal wire labeled S_1 is the output of an NMOS NOR gate with the inputs x_2 and $\bar{x}_3$. Thus $S_1 = \overline{x_2 + \bar{x}_3}$. Similarly, S_2 and S_3 are the outputs of NOR gates that produce $S_2 = \overline{x_1 + x_3}$ and $S_3 = \overline{x_1 + \bar{x}_2 + x_3}$. The three NOR gates that produce S_1, S_2, and S_3 are arranged in a regular structure that is efficient to create on an integrated circuit. This structure is called a *NOR plane*. The NOR plane is extended to larger sizes by adding columns for additional inputs and adding rows for more NOR gates.

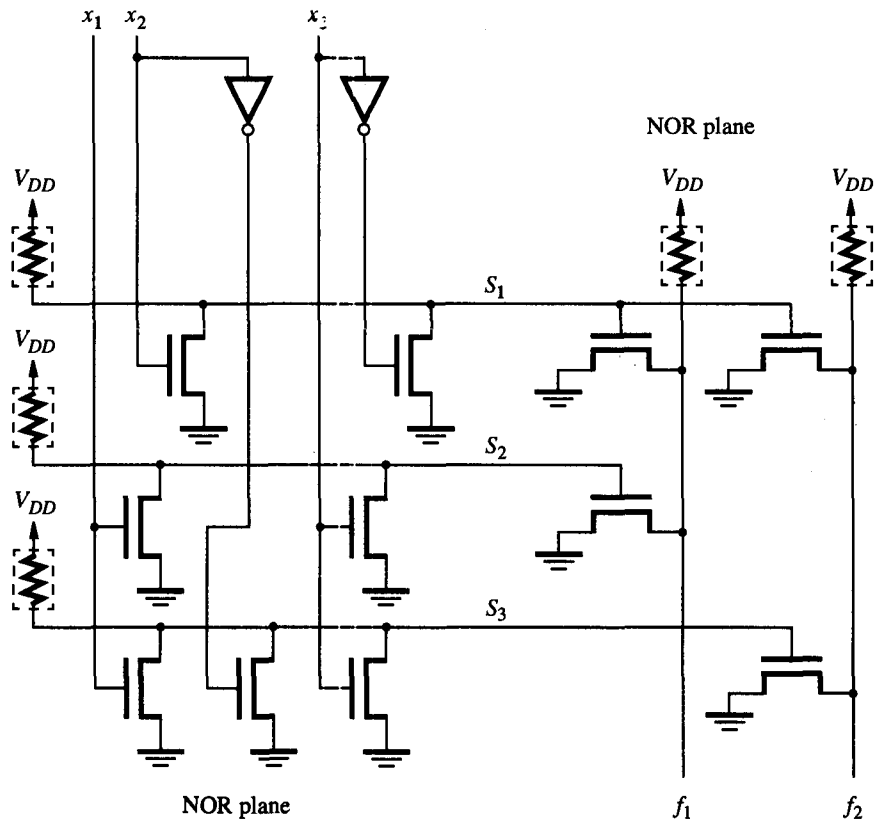


Figure 3.63 An example of a NOR-NOR PLA.

The signals S_1, S_2, and S_3 serve as inputs to a second NOR plane. This NOR plane is turned 90 degrees clockwise with respect to the first NOR plane to make the diagram easier to draw. The NOR gate that produces the output f_1 has the inputs S_1 and S_2. Thus

$$f_1 = \overline{S_1 + S_2} = \overline{(x_2 + \bar{x}_3) + (x_1 + x_3)}$$

Using DeMorgan's theorem, this expression is equivalent to the product-of-sums expression

$$f_1 = \bar{S}_1 \bar{S}_2 = (x_2 + \bar{x}_3)(x_1 + x_3)$$

Similarly, the NOR gate with output f_2 has inputs S_1 and S_3. Therefore,

$$f_2 = \overline{S_1 + S_3} = \overline{(x_2 + \bar{x}_3) + (x_1 + \bar{x}_2 + x_3)}$$

which is equivalent to

$$f_2 = \bar{S}_1 \bar{S}_3 = (x_2 + \bar{x}_3)(x_1 + \bar{x}_2 + x_3)$$

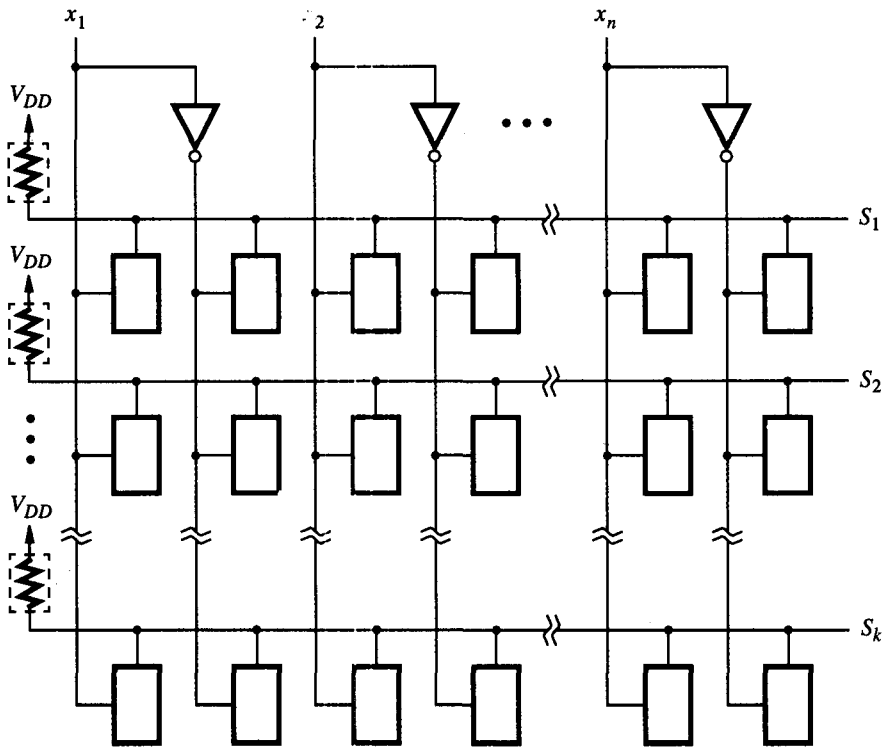
The style of PLA illustrated in Figure 3.63 is called a *NOR-NOR PLA*. Alternative implementations also exist, but because of its simplicity, the NOR-NOR style is the most popular choice. The reader should note that the PLA in Figure 3.63 is not programmable—with the transistors connected as shown, it realizes only the two specific logic functions f_1 and f_2. But the NOR-NOR structure can be used in a programmable version of the PLA, as explained below.

Strictly speaking, the term PLA should be used only for the fixed type of PLA depicted in Figure 3.63. The proper technical term for a programmable type of PLA is *field-programmable logic array (FPLA)*. However, it is common usage to omit the *F*. Figure 3.64*a* shows a programmable version of a NOR plane. It has n inputs, $x_1, \ldots, x_n$, and k outputs, $S_1, \ldots, S_k$. At each crossing point of a horizontal and vertical wire there exists a programmable switch. The switch comprises two transistors connected in series, an NMOS transistor, and an *electrically erasable programmable read-only memory (EEPROM)* transistor.

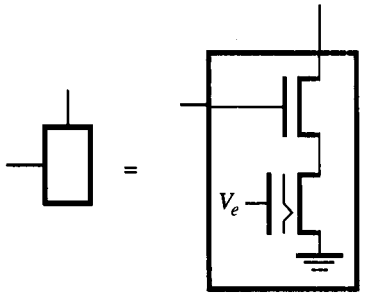
The programmable switch is based on the behavior of the EEPROM transistor. Electronics textbooks, such as [1, 2], give detailed explanations of how EEPROM transistors operate. Here we will provide only a brief description. A programmable switch is depicted in Figure 3.64*b*, and the structure of the EEPROM transistor is given in Figure 3.64*c*. The EEPROM transistor has the same general appearance as the NMOS transistor (see Figure 3.43) with one major difference. The EEPROM transistor has two gates: the normal gate that an NMOS transistor has and a second *floating gate*. The floating gate is so named because it is surrounded by insulating glass and is not connected to any part of the transistor. When the transistor is in the original unprogrammed state, the floating gate has no effect on the transistor's operation and it works as a normal NMOS transistor. During normal use of the PLA, the voltage on the floating gate V_e is set to V_{DD} by circuitry not shown in the figure, and the EEPROM transistor is turned on.

Programming of the EEPROM transistor is accomplished by turning on the transistor with a higher-than-normal voltage level (typically, $V_e = 12$ V), which causes a large amount of current to flow through the transistor's channel. Figure 3.64*c* shows that a part of the floating gate extends downward so that it is very close to the top surface of the channel. A high current flowing through the channel causes an effect, known as *Fowler-Nordheim tunneling*, in which some of the electrons in the channel "tunnel" through the insulating glass at its thinnest point and become trapped under the floating gate. After the programming process is completed, the trapped electrons repel other electrons from entering the channel. When the voltage $V_e = 5$ V is applied to the EEPROM transistor, which would normally cause it to turn on, the trapped electrons keep the transistor turned off. Hence in the NOR plane in Figure 3.64*a*, programming is used to "disconnect" inputs from the NOR gates. For the inputs that should be connected to each NOR gate, the corresponding EEPROM transistors are left in the unprogrammed state.

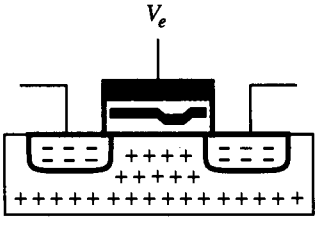
Once an EEPROM transistor is programmed, it retains the programmed state permanently. However, the programming process can be reversed. This step is called *erasing*, and it is done using voltages that are of the opposite polarity to those used for programming. In this case, the applied voltage causes the electrons that are trapped under the floating gate to tunnel back to the channel. The EEPROM transistor returns to its original state and again acts like a normal NMOS transistor.



(a) Programmable NOR-plane



(b) A programmable switch



(c) EEPROM transistor

Figure 3.64 Using EEPROM transistors to create a programmable NOR plane.

For completeness, we should also mention another technology that is similar to EEPROM, called *erasable PROM (EPROM)*. This type of transistor, which was actually created as the predecessor of EEPROM, is programmed in a similar fashion to EEPROM. However, erasing is done differently: to erase an EPROM transistor, it must be exposed to light energy of specific wavelengths. To facilitate this process, chips based on EPROM technology are

housed in packages with a clear glass window through which the chip is visible. To erase a chip, it is placed under an *ultraviolet light* source for several minutes. Because erasure of EPROM transistors is more awkward than the electrical process used to erase EEPROM transistors, EPROM technology has essentially been replaced by EEPROM technology in practice.

A complete NOR-NOR PLA using EEPROM technology, with four inputs, six sum terms in the first NOR plane, and two outputs, is depicted in Figure 3.65. Each programmable switch that is programmed to the off state is shown as X in black, and each switch that is left unprogrammed is shown in blue. With the programming states shown in the figure, the PLA realizes the logic functions $f_1 = (x_1 + x_3)(x_1 + \bar{x}_2)(\bar{x}_1 + x_2 + \bar{x}_3)$ and $f_2 = (x_1 + \bar{x}_3)(\bar{x}_1 + x_2)(x_1 + \bar{x}_2)$.

Rather than implementing logic functions in product-of-sums form, a PLA can also be used to realize the sum-of-products form. For sum-of-products we need to implement AND gates in the first NOR plane of the PLA. If we first complement the inputs to the NOR plane, then according to DeMorgan's theorem, this is equivalent to creating an AND plane. We can generate the complements at no cost in the PLA because each input is already provided in both true and complemented forms. An example that illustrates implementation of the sum-of-products form is given in Figure 3.66. The outputs from the first NOR plane

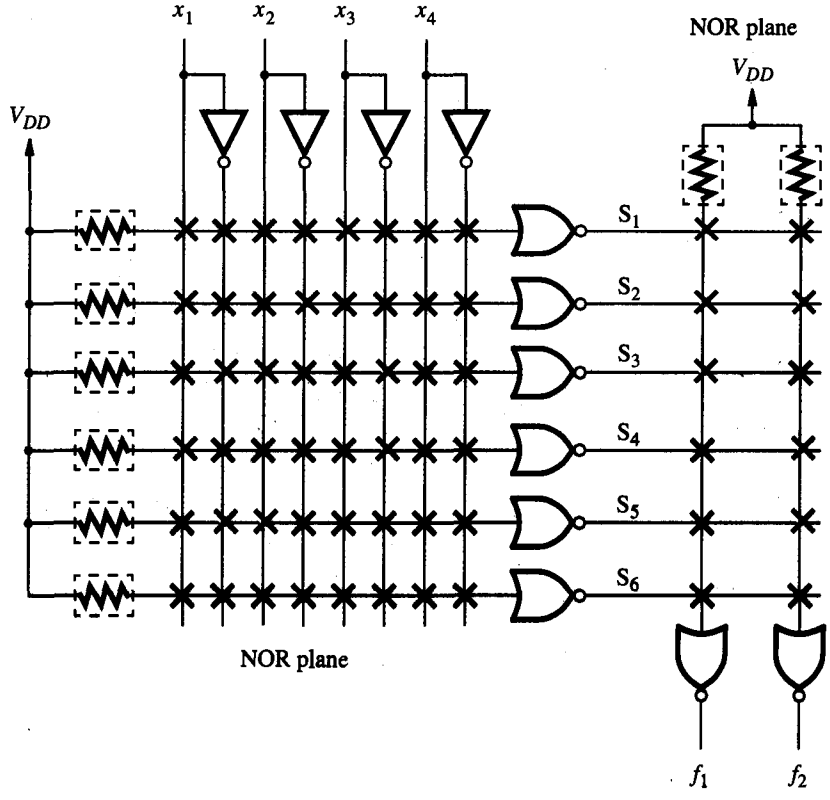


Figure 3.65 Programmable version of the NOR-NOR PLA.

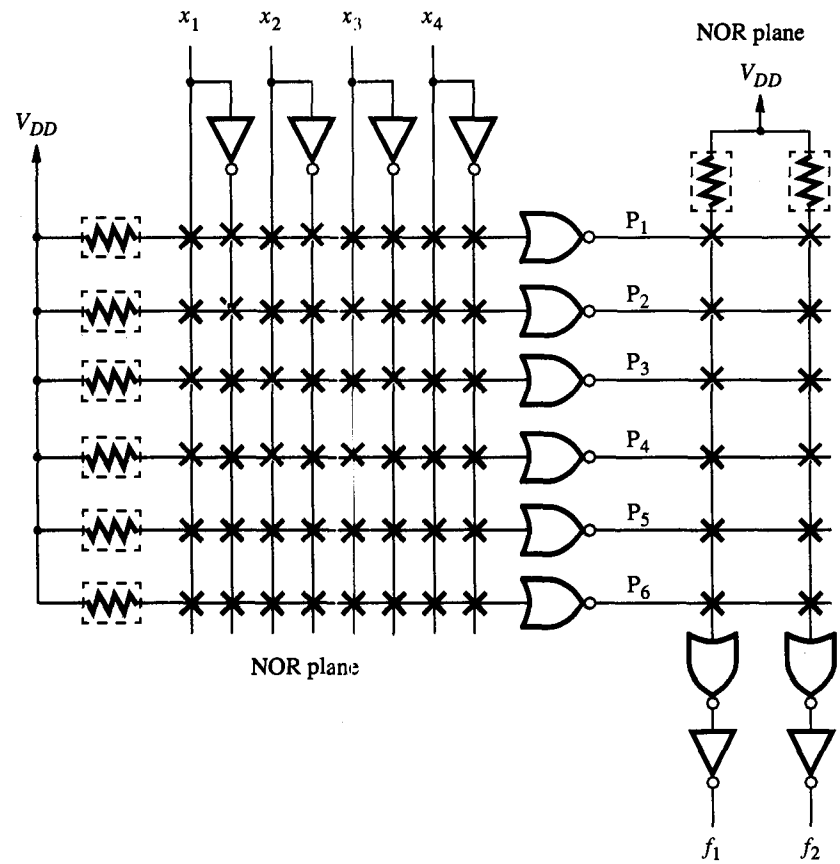


Figure 3.66 A NOR-NOR PLA used for sum-of-products.

are labeled $P_1, \ldots, P_6$ to reflect our interpretation of them as product terms. The signal P_1 is programmed to realize $\overline{\bar{x}_1 + \bar{x}_2} = x_1x_2$. Similarly, $P_2 = x_1\bar{x}_3$, $P_3 = \bar{x}_1\bar{x}_2x_3$, and $P_4 = \bar{x}_1\bar{x}_2\bar{x}_3$. Having generated the desired product terms, we now need to OR them. This operation can be accomplished by complementing the outputs of the second NOR plane. Figure 3.66 includes NOT gates for this purpose. The states indicated for the programmable switches in the OR plane (the second NOR plane) in the figure yield the following outputs: $f_1 = P_1 + P_2 + P_3 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$, and $f_2 = P_1 + P_4 = x_1x_2 + \bar{x}_1\bar{x}_2\bar{x}_3$.

The concepts described above for PLAs can also be used in PALs. Figure 3.67 shows a PAL with four inputs and two outputs. Let us assume that the first NOR plane is programmed to realize product terms in the manner described above. Notice in the figure that the product terms are hardwired in groups of three to OR gates that produce the outputs of the PAL. As we illustrated in Figure 3.29, the PAL may also contain extra circuitry between the OR gates and the output pins, which is not shown in Figure 3.67. The PAL is programmed to realize the same logic functions, f_1 and f_2, that were generated in the PLA in Figure 3.66. Observe that the product term x_1x_2 is implemented twice in the PAL, on both P_1 and P_4. Duplication is necessary because in a PAL product terms cannot be shared by multiple

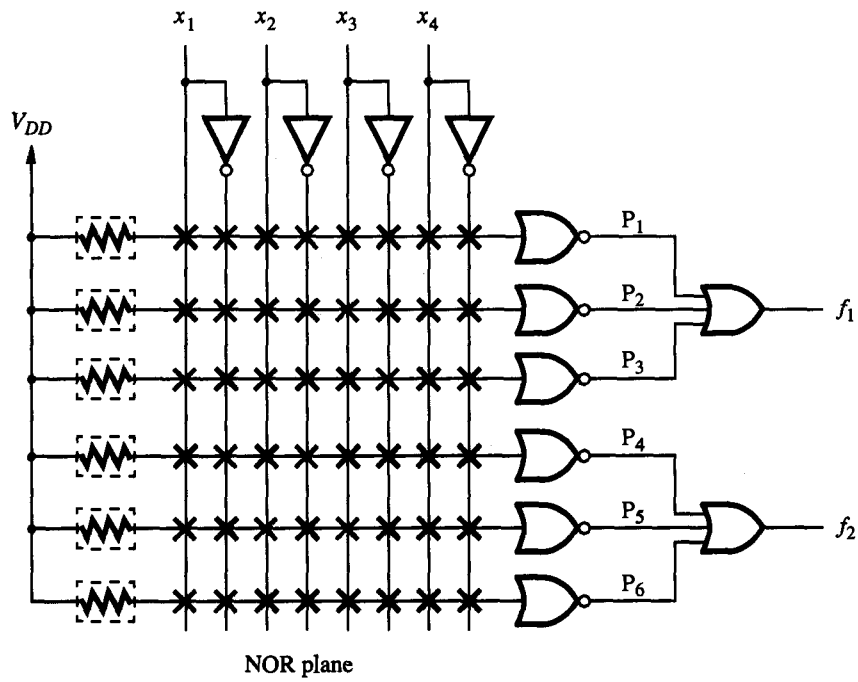


Figure 3.67 PAL programmed to implement the functions in Figure 3.66.

outputs, as they can be in a PLA. Another detail to observe in Figure 3.67 is that although the function f_2 requires only two product terms, each OR gate is hardwired to three product terms. The extra product term P_6 must be set to logic value 0, so that it has no effect. This is accomplished by programming P_6 so that it produces the product of an input and that input's complement, which always results in 0. In the figure, $P_6 = x_1\bar{x}_1 = 0$, but any other input could also be used for this purpose.

The PAL-like blocks contained in CPLDs are usually implemented using the techniques discussed in this section. In a typical CPLD, the AND plane is built using NMOS NOR gates, with appropriate complementing of the inputs. The OR plane is hardwired as it is in a PAL, rather than being fully programmable as in a PLA. However, some flexibility exists in the number of product terms that feed each OR gate. This flexibility is accomplished by using a programmable circuit that can allocate the product terms to whichever OR gates the user desires. An example of this type of flexibility, provided in a commercial CPLD, is given in Appendix E.

3.10.1 IMPLEMENTATION IN FPGAS

FPGAs do not use EEPROM technology to implement the programmable switches. Instead, the programming information is stored in memory cells, called *static random access memory (SRAM)* cells. The operation of this type of storage cell is described in detail in section 10.1.3. For now it is sufficient to know that each cell can store either a logic 0 or 1, and it

provides this stored value as an output. An SRAM cell is used for each truth-table value stored in a LUT. SRAM cells are also used to configure the interconnection wires in an FPGA.

Figure 3.68 depicts a small section of the FPGA from Figure 3.39. The logic block shown produces the output f_1, which is driven onto the horizontal wire drawn in blue. This wire can be connected to some of the vertical wires that it crosses, using programmable switches. Each switch is implemented using an NMOS transistor, with its gate terminal controlled by an SRAM cell. Such a switch is known as a *pass-transistor switch*. If a 0 is stored in an SRAM cell, then the associated NMOS transistor is turned off. But if a 1 is stored in the SRAM cell, as shown for the switch drawn in blue, then the NMOS transistor is turned on. This switch forms a connection between the two wires attached to its source and drain terminals. The number of switches that are provided in the FPGA depends on the specific chip architecture. In some FPGAs some of the switches are implemented using tri-state buffers, instead of pass transistors. Examples of commercial FPGA chips are presented in Appendix E.

In section 3.8.7 we showed that an NMOS transistor can only partially pass a high logic value. Hence in Figure 3.68 if V_{f_1} is a high voltage level, then V_A is only partially high. Using the values from section 3.8.7, if $V_{f_1} = 5$ V, then $V_A = 3.5$ V. As we explained in section 3.8.7, this degraded voltage level has the result of causing static power to be consumed (see Problem 3.35). One solution to this problem [1] is illustrated in Figure 3.69. We assume that the signal V_A passes through another pass-transistor switch before reaching its destination at another logic block. The signal V_B has the same value as V_A because the threshold voltage drop occurs only when passing through the first pass-transistor switch. To restore the level of V_B, it is buffered with an inverter. A PMOS transistor is connected between the input of the inverter and V_{DD}, and that transistor is controlled by the inverter's output. The PMOS transistor has no effect on the inverter's output voltage level when $V_B = 0$ V. But when $V_B = 3.5$ V, then the inverter output is low, which turns on the PMOS transistor. This transistor quickly restores V_B to the proper level of V_{DD}, thus preventing current from flowing in the steady state. Instead of using this pull-up transistor solution,

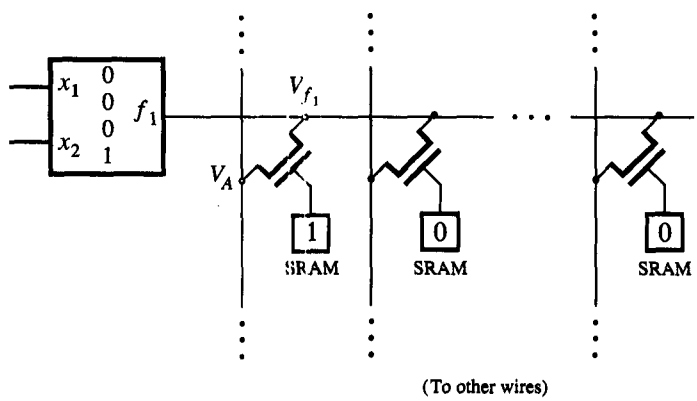


Figure 3.68 Pass-transistor switches in FPGAs.

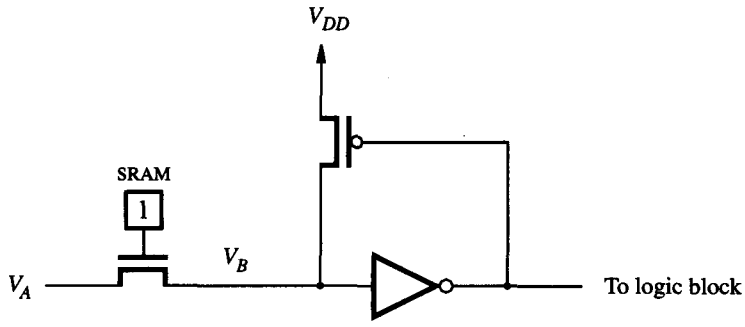


Figure 3.69 Restoring a high voltage level.

another possible approach is to alter the threshold voltage of the PMOS transistor (during the integrated circuit manufacturing process) in the inverter in Figure 3.69, such that the magnitude of its threshold voltage is large enough to keep the transistor turned off when $V_B = 3.5$ V. In commercial FPGAs both of these solutions are used in different chips.

An alternative to using a single NMOS transistor is to use a transmission gate, described in section 3.9, for each switch. While this solves the voltage-level problem, it has two drawbacks. First, having both an NMOS and PMOS transistor in the switch increases the capacitive loading on the interconnection wires, which increases the propagation delays. Second, the transmission gate takes more chip area than does a single NMOS transistor. For these reasons, commercial FPGA chips do not currently use transmission-gate switches.

3.11 CONCLUDING REMARKS

We have described the most important concepts that are needed to understand how logic gates are built using transistors. Our discussions of transistor fabrication, voltage levels, propagation delays, power dissipation, and the like are meant to give the reader an appreciation of the practical issues that have to be considered when designing and using logic circuits.

We have introduced several types of integrated circuit chips. Each type of chip is appropriate for specific types of applications. The standard chips, such as the 7400 series, contain only a few simple gates and are rarely used today. Exceptions to this are the buffer chips, which are employed in digital circuits that must drive large capacitive loads at high speeds. The various types of PLDs are widely used in many types of applications. Simple PLDs, like PLAs and PALs, are appropriate for implementation of small logic circuits. The SPLDs offer low cost and high speed. CPLDs can be used for the same applications as SPLDs, but CPLDs are also well suited for implementation of larger circuits of more than 20,000 gates. Many of the applications that can be targeted to CPLDs can alternatively be realized with FPGAs. Which of these two types of chips are used in a specific design situation depends on many factors. For some types of circuits, CPLDs provide slightly

faster speeds than FPGAs do, but FPGAs can support larger circuits. Following the trend of putting as much circuitry as possible into a single chip, CPLDs and FPGAs are much more widely used than SPLDs. Most digital designs created in the industry today contain some type of PLD.

The gate-array, standard-cell, and custom-chip technologies are used in cases where PLDs are not appropriate. Typical applications are those that entail very large circuits, where the designed product is expected to sell in large volume.

The next chapter examines the issue of optimization of logic functions. Some of the techniques discussed are appropriate for use in the synthesis of logic circuits regardless of what type of technology is used for implementation. Other techniques are suitable for synthesizing circuits so that they can be implemented in chips with specific types of resources. We will show that when synthesizing a logic function to create a circuit, the optimization methods used depend, at least in part, on which type of chip is being used.

PROBLEMS

3.1 Consider the circuit shown in Figure P3.1.

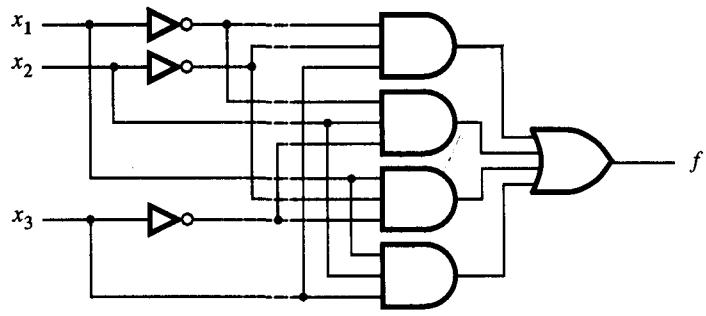


Figure P3.1 A sum-of-products CMOS circuit.

(a) Show the truth table for the logic function f.
(b) If each gate in the circuit is implemented as a CMOS gate, how many transistors are needed?

3.2 (a) Show that the circuit in Figure P3.2 is functionally equivalent to the circuit in Figure P3.1.
(b) How many transistors are needed to build this CMOS circuit?

3.3 (a) Show that the circuit in Figure P3.3 is functionally equivalent to the circuit in Figure P3.2.
(b) How many transistors are needed to build this CMOS circuit if each XOR gate is implemented using the circuit in Figure 3.61*d*?

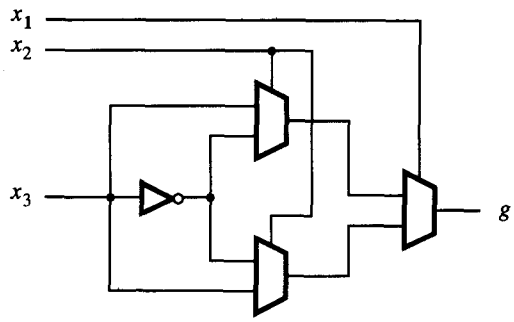


Figure P3.2 A CMOS circuit built with multiplexers.

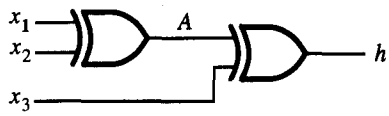


Figure P3.3 Circuit for problem 3.3.

3.4 In section 3.8.8 we said that a six-input CMOS AND gate can be constructed using 2 three-input AND gates and a two-input AND gate. This approach requires 22 transistors. Show how you can use only CMOS NAND and NOR gates to build the six-input AND gate and then calculate the number of transistors needed. (Hint: use DeMorgan's theorem.)

3.5 Repeat problem 3.4 for an eight-input CMOS OR gate.

3.6 (a) Give the truth table for the CMOS circuit in Figure P3.4.
(b) Derive a canonical sum-of-products expression for the truth table from part (*a*). How many transistors are needed to build a circuit representing the canonical form if only AND, OR, and NOT gates are used?

3.7 (a) Give the truth table for the CMOS circuit in Figure P3.5.
(b) Derive the simplest sum-of-products expression for the truth table in part (*a*). How many transistors are needed to build the sum-of-products circuit using CMOS AND, OR, and NOT gates?

3.8 Figure P3.6 shows half of a CMOS circuit. Derive the other half that contains the PMOS transistors.

3.9 Figure P3.7 shows half of a CMOS circuit. Derive the other half that contains the NMOS transistors.

3.10 Derive a CMOS complex gate for the logic function $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 5, 6, 8, 9, 10)$.

3.11 Derive a CMOS complex gate for the logic function $f(x_1, x_2, x_3, x_4) = \sum m(0, 1, 2, 4, 6, 8, 10, 12, 14)$.

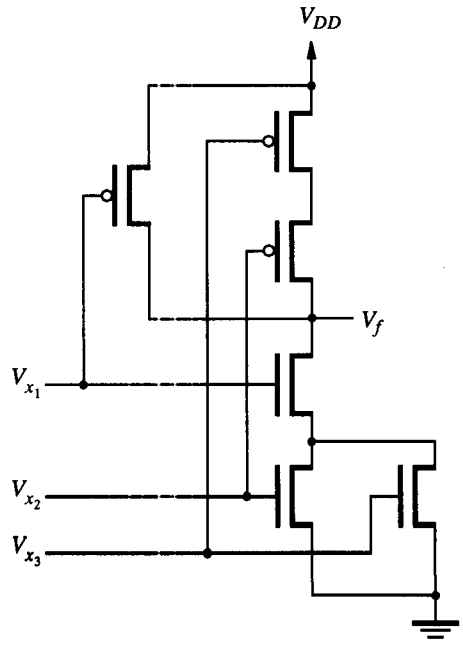


Figure P3.4 A three-input CMOS circuit.

3.12 Derive a CMOS complex gate for the logic function $f = xy + xz$. Use as few transistors as possible (Hint: consider $\bar{f}$).

3.13 Derive a CMOS complex gate for the logic function $f = xy + xz + yz$. Use as few transistors as possible (Hint: consider $\bar{f}$).

3.14 For an NMOS transistor, assume that $k_n' = 20\ \mu A/V^2$, $W/L = 2.5\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V, and $V_T = 1$ V. Calculate
(a) I_D when $V_{DS} = 5$ V.
(b) I_D when $V_{DS} = 0.2$ V.

3.15 For a PMOS transistor, assume that $k_p' = 10\ \mu A/V^2$, $W/L = 2.5\ \mu m/0.5\ \mu m$, $V_{GS} = -5$ V, and $V_T = -1$ V. Calculate
(a) I_D when $V_{DS} = -5$ V.
(b) I_D when $V_{DS} = -0.2$ V.

3.16 For an NMOS transistor, assume that $k_n' = 20\ \mu A/V^2$, $W/L = 5.0\ \mu m/0.5\ \mu m$, $V_{GS} = 5$ V, and $V_T = 1$ V. For small V_{DS}, calculate R_{DS}.

3.17 For an NMOS transistor, assume that $k_n' = 40\ \mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = 3.3$ V, and $V_T = 0.66$ V. For small V_{DS}, calculate R_{DS}.

3.18 For a PMOS transistor, assume that $k_p' = 10\ \mu A/V^2$, $W/L = 5.0\ \mu m/0.5\ \mu m$, $V_{GS} = -5$ V, and $V_T = -1$ V. For $V_{DS} = -4.8$ V, calculate R_{DS}.

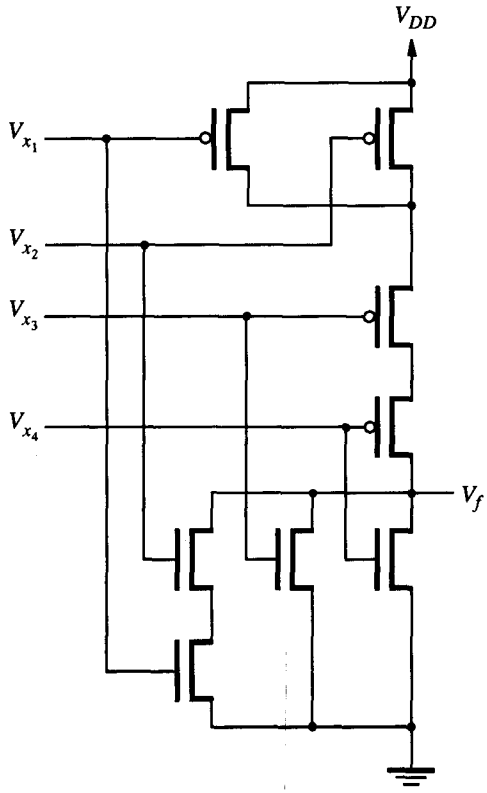


Figure P3.5 A four-input CMOS circuit.

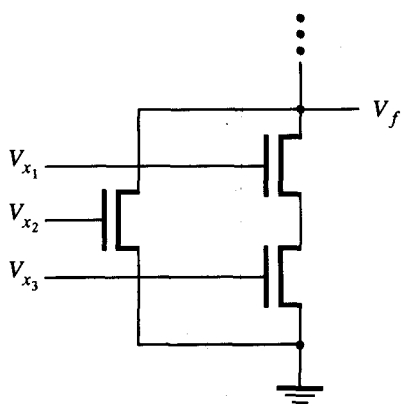


Figure P3.6 The PDN in a CMOS circuit.

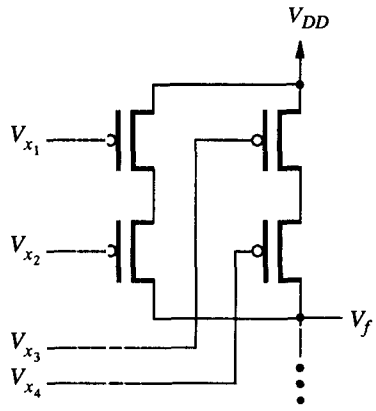


Figure P3.7 The PUN in a CMOS circuit.

3.19 For a PMOS transistor, assume that $k_p' = 16\,\mu A/V^2$, $W/L = 3.5\ \mu m/0.35\ \mu m$, $V_{GS} = -3.3$ V, and $V_T = -0.66$ V. For $V_{DS} = -3.2$ V, calculate R_{DS}.

3.20 In the original NMOS technology, the pull-up device was an *n*-channel MOSFET. But most integrated circuits fabricated today use CMOS technology. Hence it is convenient to implement the pull-up resistor using a PMOS transistor, as shown in Figure P3.8. Such a circuit is referred to as a *pseudo-NMOS* circuit. The pull-up device is called a "weak" PMOS transistor because it has a small W/L ratio.

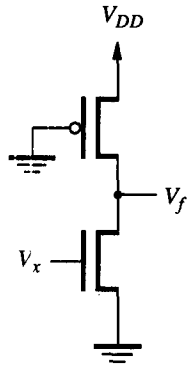


Figure P3.8 The pseudo-NMOS inverter.

When $V_x = V_{DD}$, V_f has a low value. The NMOS transistor is operating in the triode region, while the PMOS transistor limits the current flow because it is operating in the saturation

region. The current through the NMOS and PMOS transistors has to be equal and is given by equations 3.1 and 3.2. Show that the low-output voltage, $V_f = V_{OL}$ is given by

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

where k_p and k_n, called the *gain factors*, depend on the sizes of the PMOS and NMOS transistors, respectively. They are defined by $k_p = k'_p W_p/L_p$ and $k_n = k'_n W_n/L_n$.

3.21 For the circuit in Figure P3.8, assume the values $k'_n = 60\ \mu A/V^2$, $k'_p = 0.4k'_n$, $W_n/L_n = 2.0\ \mu m/0.5\ \mu m$, $W_p/L_p = 0.5\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V, and $V_T = 1$ V. When $V_x = V_{DD}$, calculate (*a*) through (*e*).
(a) The static current I_{stat}.
(b) The on-resistance of the NMOS transistor.
(c) V_{OL}.
(d) The static power dissipated in the inverter.
(e) The on-resistance of the PMOS transistor.
(f) Assume that the inverter is used to drive a capacitive load of 70 fF. Using equation 3.4, calculate the low-to-high and high-to-low propagation delays.

3.22 Repeat problem 3.21 assuming that the size of the PMOS transistor is changed to $W_p/L_p = 2.0\ \mu m/0.5\ \mu m$.

3.23 Figure P3.8 shows that in the pseudo-NMOS technology, the pull-up device is implemented using a PMOS transistor. Repeat problem 3.21 for a NAND gate built with pseudo-NMOS technology. Assume that both of the NMOS transistors in the gate have the same parameters, as given in problem 3.21.

3.24 Repeat problem 3.23 for a pseudo-NMOS NOR gate.

3.25 (a) For $V_{IH} = 4$ V, $V_{OH} = 4.5$ V, $V_{IL} = 1$ V, $V_{OL} = 0.3$ V, and $V_{DD} = 5$ V, calculate the noise margins NM_H and NM_L.
(b) Consider an eight-input NAND gate built using NMOS technology. If the voltage drop across each transistor is 0.1 V, what is V_{OL}? What is the corresponding NM_L using the other parameters from part (*a*)?

3.26 Under steady-state conditions for an *n*-input CMOS NAND gate, what are the voltage levels of V_{OL} and V_{OH}? Explain.

3.27 For a CMOS inverter, assume that the load capacitance is $C = 150$ fF and $V_{DD} = 5$ V. The inverter is cycled through the low and high voltage levels at an average rate of $f = 75$ MHz.
(a) Calculate the dynamic power dissipated in the inverter.
(b) For a chip that contains the equivalent of 250,000 inverters, calculate the total dynamic power dissipated if 20 percent of the gates change values at any given time.

3.28 Repeat problem 3.27 for $C = 120$ fF, $V_{DD} = 3.3$ V, and $f = 125$ MHz.

3.29 In a CMOS inverter, assume that $k'_n = 20\ \mu A/V^2$, $k'_p = 0.4 \times k'_n$, $W_n/L_n = 5.0\ \mu m/0.5\ \mu m$, $W_p/L_p = 5.0\ \mu m/0.5\ \mu m$, and $V_{DD} = 5$ V. The inverter drives a load capacitance of 150 fF.

(a) Find the high-to-low propagation delay.
(b) Find the low-to-high propagation delay.
(c) What should be the dimensions of the PMOS transistor such that the low-to-high and high-to-low propagation delays are equal? Ignore the effect of the PMOS transistor's size on the load capacitance of the inverter.

3.30 Repeat problem 3.29 for the parameters $k'_n = 40\ \mu A/V^2$, $k'_p = 0.4 \times k'_n$, $W_n/L_n = W_p/L_p = 3.5\ \mu m/0.35\ \mu m$, and $V_{DD} = 3.3$ V.

3.31 In a CMOS inverter, assume that $W_n/L_n = 2$ and $W_p/L_p = 4$. For a CMOS NAND gate, calculate the required W/L ratios of the NMOS and PMOS transistors such that the available current in the gate to drive the output both low and high is equal to that in the inverter.

3.32 Repeat problem 3.31 for a CMOS NOR gate.

3.33 Repeat problem 3.31 for the CMOS complex gate in Figure 3.16. The transistor sizes should be chosen so that in the worst case the available current is at least as large as in the inverter.

3.34 Repeat problem 3.31 for the CMOS complex gate in Figure 3.17.

3.35 In Figure 3.69 we showed a solution to the static power dissipation problem when NMOS pass transistors are used. Assume that the PMOS pull-up transistor is removed from this circuit. Assume the parameters $k'_n = 60\ \mu A/V^2$, $k'_p = 0.5 \times k'_n$, $W_n/L_n = 2.0\ \mu m/0.5\ \mu m$, $W_p/L_p = 4.0\ \mu m/0.5\ \mu m$, $V_{DD} = 5$ V, and $V_T = 1$ V. For $V_B = 3.5$ V, calculate (*a*) through (*d*).
(a) The static current I_{stat}.
(b) The voltage V_f at the output of the inverter.
(c) The static power dissipation in the inverter.
(d) If a chip contains 250,000 inverters used in this manner, find the total static power dissipation.

3.36 Using the style of drawing in Figure 3.66, draw a picture of a PLA programmed to implement $f_1(x_1, x_2, x_3) = \sum m(1, 2, 4, 7)$. The PLA should have the inputs $x_1, \ldots, x_3$; the product terms $P_1, \ldots, P_4$; and the outputs f_1 and f_2.

3.37 Using the style of drawing in Figure 3.66, draw a picture of a PLA programmed to implement $f_1(x_1, x_2, x_3) = \sum m(0, 3, 5, 6)$. The PLA should have the inputs $x_1, \ldots, x_3$; the product terms $P_1, \ldots, P_4$; and the outputs f_1 and f_2.

3.38 Show how function f_1 from problem 3.36 can be realized in a PLA of the type shown in Figure 3.65. Draw a picture of such a PLA programmed to implement f_1. The PLA should have the inputs $x_1, \ldots, x_3$; the sum terms $S_1, \ldots, S_4$; and the outputs f_1 and f_2.

3.39 Show how function f_1 from problem 3.37 can be realized in a PLA of the type shown in Figure 3.65. Draw a picture of such a PLA programmed to implement f_1. The PLA should have the inputs $x_1, \ldots, x_3$; the sum terms $S_1, \ldots, S_4$; and the outputs f_1 and f_2.

3.40 Repeat problem 3.38 using the style of PLA drawing shown in Figure 3.63.

3.41 Repeat problem 3.39 using the style of PLA drawing shown in Figure 3.63.

3.42 Given that f_1 is implemented as described in problem 3.36, list all of the other possible logic functions that can be realized using output f_2 in the PLA.

3.43 Given that f_1 is implemented as described in problem 3.37, list all of the other possible logic functions that can be realized using output f_2 in the PLA.

3.44 Consider the function $f(x_1, x_2, x_3) = x_1\bar{x}_2 + x_1x_3 + x_2\bar{x}_3$. Show a circuit using 5 two-input lookup-tables (LUTs) to implement this expression. As shown in Figure 3.39, give the truth table implemented in each LUT. You do not need to show the wires in the FPGA.

3.45 Consider the function $f(x_1, x_2, x_3) = \sum m(2, 3, 4, 6, 7)$. Show how it can be realized using 2 two-input LUTs. As shown in Figure 3.39, give the truth table implemented in each LUT. You do not need to show the wires in the FPGA.

3.46 Given the function $f = x_1x_2x_4 + x_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_3$, a straightforward implementation in an FPGA with three-input LUTs requires four LUTs. Show how it can be done using only 3 three-input LUTs. Label the output of each LUT with an expression representing the logic function that it implements.

3.47 For f in problem 3.46, show a circuit of two-input LUTs that realizes the function. You are to use exactly 7 two-input LUTs. Label the output of each LUT with an expression representing the logic function that it implements.

3.48 Figure 3.39 shows an FPGA programmed to implement a function. The figure shows one pin used for function f and several pins that are unused. Without changing the programming of any switch that is turned *on* in the FPGA in the figure, list four other logic functions, in addition to f, that can be implemented on the unused pins.

3.49 Assume that a gate array contains the type of logic cell depicted in Figure P3.9. The inputs $in_1, \ldots, in_7$ can be connected to either 1 or 0, or to any logic signal.

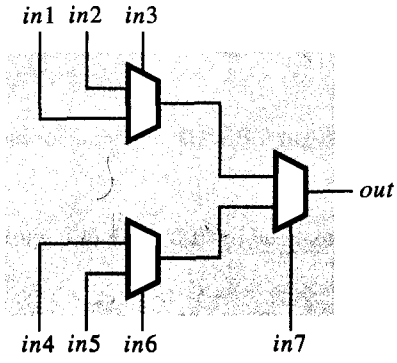


Figure P3.9 A gate-array logic cell.

(a) Show how the logic cell can be used to realize $f = x_1x_2 + x_3$.
(b) Show how the logic cell can be used to realize $f = x_1x_3 + x_2x_3$.

3.50 Assume that a gate array exists in which the logic cell used is a three-input NAND gate. The inputs to each NAND gate can be connected to either 1 or 0, or to any logic signal. Show

how the following logic functions can be realized in the gate array. (Hint: use DeMorgan's theorem.)
(a) $f = x_1x_2 + x_3$.
(b) $f = x_1x_2x_4 + x_2x_3\bar{x}_4 + \bar{x}_1$.

3.51 Write Verilog code to represent the function $f = x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2x_4 + \bar{x}_1x_2x_3 + x_1x_2x_3$.
(a) Use your CAD tools to implement f in some type of chip, such as a CPLD. Show the logic expression generated for f by the tools. Use timing simulation to determine the time needed for a change in inputs x_1, x_2, or x_3 to propagate to the output f.
(b) Repeat part (*a*) using a different chip, such as an FPGA for implementation of the circuit.

3.52 Repeat problem 3.51 for the function $f = (x_1 + x_2 + \bar{x}_4) \cdot (\bar{x}_2 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + x_3 + \bar{x}_4) \cdot (\bar{x}_1 + \bar{x}_3 + \bar{x}_4)$.

3.53 Repeat problem 3.51 for the function $f(x_1, ..., x_7) = x_1x_3\bar{x}_6 + x_1x_4x_5\bar{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$.

3.54 What logic gate is realized by the circuit in Figure P3.10? Does this circuit suffer from any major drawbacks?

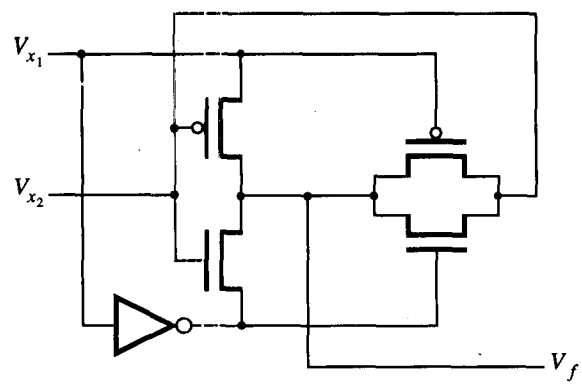


Figure P3.10 Circuit for problem 3.54.

3.55 What logic gate is realized by the circuit in Figure P3.11? Does this circuit suffer from any major drawbacks?

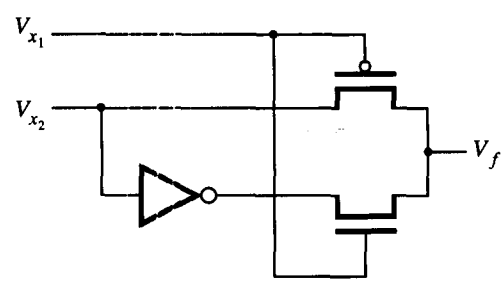


Figure P3.11 Circuit for problem 3.55.

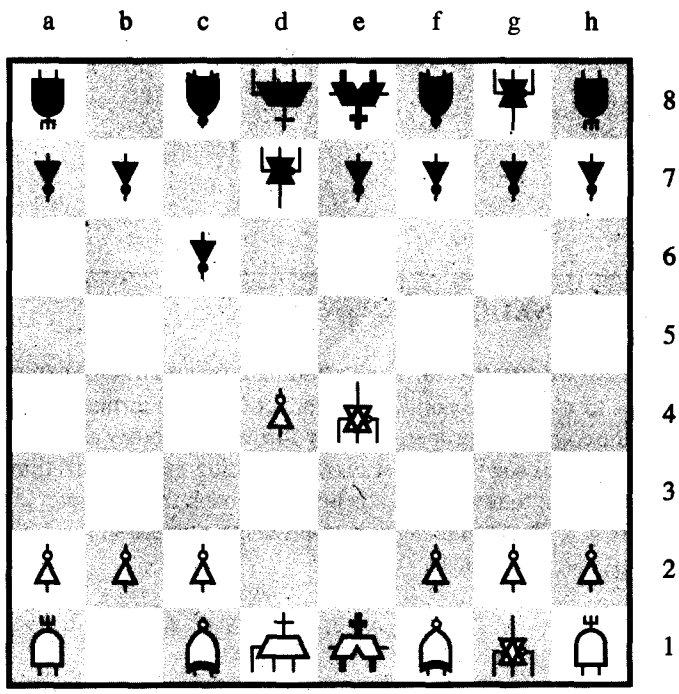
REFERENCES

1. A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 4th ed. (Oxford University Press: New York, 1998).
2. J. M. Rabaey, *Digital Integrated Circuits*, (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. Texas Instruments, *Logic Products Selection Guide and Databook CD-ROM*, 1997.
4. National Semiconductor, *VHC/VHCT Advanced CMOS Logic Databook*, 1993.
5. Motorola, *CMOS Logic Databook*, 1996.
6. Toshiba America Electronic Components, *TC74VHC/VHCT Series CMOS Logic Databook*, 1994.
7. Integrated Devices Technology, *High Performance Logic Databook*, 1994.
8. J. F. Wakerly, *Digital Design Principles and Practices* (Prentice-Hall: Englewood Cliffs, NJ, 1990).
9. M. M. Mano, *Digital Design* (Prentice-Hall: Englewood Cliffs, NJ, 1991).
10. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
11. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
12. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).

chapter

4

Optimized Implementation of Logic Functions



4. Nc3xe4, Nb8–d7

In Chapter 2 we showed that algebraic manipulation can be used to find the lowest-cost implementations of logic functions. The purpose of that chapter was to introduce the basic concepts in the synthesis process. The reader is probably convinced that it is easy to derive a straightforward realization of a logic function in a canonical form, but it is not at all obvious how to choose and apply the theorems and properties of section 2.5 to find a minimum-cost circuit. Indeed, the algebraic manipulation is rather tedious and quite impractical for functions of many variables.

If CAD tools are used to design logic circuits, the task of minimizing the cost of implementation does not fall to the designer; the tools perform the necessary optimizations automatically. Even so, it is essential to know something about this process. Most CAD tools have many features and options that are under control of the user. To know when and how to apply these options, the user must have an understanding of what the tools do.

In this chapter we will introduce some of the optimization techniques implemented in CAD tools and show how these techniques can be automated. As a first step we will discuss a graphical approach, known as the Karnaugh map, which provides a neat way to manually derive minimum-cost implementations of simple logic functions. Although it is not suitable for implementation in CAD tools, it illustrates a number of key concepts. We will show how both two-level and multilevel circuits can be designed. Then we will describe a cubical representation for logic functions, which is suitable for use in CAD tools. We will also continue our discussion of the Verilog language and CAD tools.

4.1 KARNAUGH MAP

In section 2.6 we saw that the key to finding a minimum-cost expression for a given logic function is to reduce the number of product (or sum) terms needed in the expression, by applying the combining property 14*a* (or 14*b*) as judiciously as possible. The Karnaugh map approach provides a systematic way of performing this optimization. To understand how it works, it is useful to review the algebraic approach from Chapter 2. Consider the function f in Figure 4.1. The canonical sum-of-products expression for f consists of minterms m_0, m_2, m_4, m_5, and m_6, so that

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

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

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

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

Figure 4.1 The function $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

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

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

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

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

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

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

Recall that theorem 7*b* in section 2.5 indicates that

$$m_4 = m_4 + m_4$$

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

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

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

The expression has the product term $\bar{x}_3$ because $f = 1$ when $x_3 = 0$ regardless of the values of x_1 and x_2. The four minterms m_0, m_2, m_4, and m_6 represent all possible minterms for which $x_3 = 0$; they include all four valuations, 00, 01, 10, and 11, of variables x_1 and x_2. Thus if $x_3 = 0$, then it is guaranteed that $f = 1$. This may not be easy to see directly from the truth table in Figure 4.1, but it is obvious if we write the corresponding valuations grouped together:

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

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

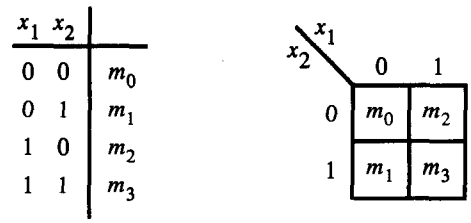
	x_1	x_2	x_3
m_4	1	0	0
m_5	1	0	1

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

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

The *Karnaugh map* [1] is an alternative to the truth-table form for representing a function. The map consists of *cells* that correspond to the rows of the truth table. Consider the two-variable example in Figure 4.2. Part (*a*) depicts the truth-table form, where each of the four rows is identified by a minterm. Part (*b*) shows the Karnaugh map, which has four cells. The columns of the map are labeled by the value of x_1, and the rows are labeled by x_2. This labeling leads to the locations of minterms as shown in the figure. Compared to the truth table, the advantage of the Karnaugh map is that it allows easy recognition of minterms that can be combined using property 14*a* from section 2.5. Minterms in any two cells that are adjacent, either in the same row or the same column, can be combined. For example, the minterms m_2 and m_3 can be combined as

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



(a) Truth table (b) Karnaugh map

Figure 4.2 Location of two-variable minterms.

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

Two-Variable Map

A Karnaugh map for a two-variable function is given in Figure 4.3. It corresponds to the function f of Figure 2.15. The value of f for each valuation of the variables x_1 and x_2 is indicated in the corresponding cell of the map. Because a 1 appears in both cells of the bottom row and these cells are adjacent, there exists a single product term that can cause f to be equal to 1 when the input variables have the values that correspond to either of these cells. To indicate this fact, we have circled the cell entries in the map. Rather than using the combining property formally, we can derive the product term intuitively. Both of the cells are identified by $x_2 = 1$, but $x_1 = 0$ for the left cell and $x_1 = 1$ for the right cell. Thus if $x_2 = 1$, then $f = 1$ regardless of whether x_1 is equal to 0 or 1. The product term representing the two cells is simply x_2.

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

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

Evidently, to find a minimum-cost implementation of a given function, it is necessary to find the smallest number of product terms that produce a value of 1 for all cases where

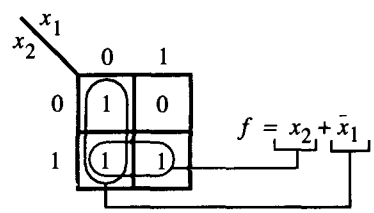


Figure 4.3 The function of Figure 2.15.

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

Three-Variable Map

A three-variable Karnaugh map is constructed by placing 2 two-variable maps side by side. Figure 4.4 shows the map and indicates the locations of minterms in it. In this case each valuation of x_1 and x_2 identifies a column in the map, while the value of x_3 distinguishes the two rows. To ensure that minterms in the adjacent cells in the map can always be combined into a single product term, the adjacent cells must differ in the value of only one variable. Thus the columns are identified by the sequence of (x_1, x_2) values of 00, 01, 11, and 10, rather than the more obvious 00, 01, 10, and 11. This makes the second and third columns different only in variable x_1. Also, the first and the fourth columns differ only in variable x_1, which means that these columns can be considered as being adjacent. The reader may find it useful to visualize the map as a rectangle folded into a cylinder where the left and the right edges in Figure 4.4*b* are made to touch. (A sequence of codes, or valuations, where consecutive codes differ in one variable only is known as the *Gray code*. This code is used for a variety of purposes, some of which will be encountered later in the book.)

Figure 4.5*a* represents the function of Figure 2.18 in Karnaugh-map form. To synthesize this function, it is necessary to cover the four 1s in the map as efficiently as possible. It is not difficult to see that two product terms suffice. The first covers the 1s in the top row, which are represented by the term $x_1\bar{x}_3$. The second term is $\bar{x}_2x_3$, which covers the 1s in the bottom row. Hence the function is implemented as

$$f = x_1\bar{x}_3 + \bar{x}_2x_3$$

which describes the circuit obtained in Figure 2.19*a*.

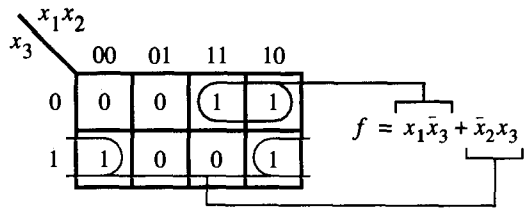
x_1	x_2	x_3	
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

(a) Truth table

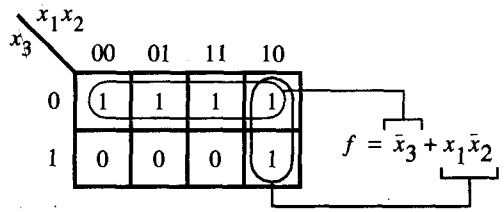
x_3 \ x_1x_2	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

(b) Karnaugh map

Figure 4.4 Location of three-variable minterms.



(a) The function of Figure 2.18



(b) The function of Figure 4.1

Figure 4.5 Examples of three-variable Karnaugh maps.

In a three-variable map it is possible to combine cells to produce product terms that correspond to a single cell, two adjacent cells, or a group of four adjacent cells. Realization of a group of four adjacent cells using a single product term is illustrated in Figure 4.5*b*, using the function from Figure 4.1. The four cells in the top row correspond to the (x_1, x_2, x_3) valuations 000, 010, 110, and 100. As we discussed before, this indicates that if $x_3 = 0$, then $f = 1$ for all four possible valuations of x_1 and x_2, which means that the only requirement is that $x_3 = 0$. Therefore, the product term $\bar{x}_3$ represents these four cells. The remaining 1, corresponding to minterm m_5, is best covered by the term $x_1\bar{x}_2$, obtained by combining the two cells in the right-most column. The complete realization of f is

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

It is also possible to have a group of eight 1s in a three-variable map. This is the trivial case where $f = 1$ for all valuations of input variables; in other words, f is equal to the constant 1.

The Karnaugh map provides a simple mechanism for generating the product terms that should be used to implement a given function. A product term must include only those variables that have the same value for all cells in the group represented by this term. If the variable is equal to 1 in the group, it appears uncomplemented in the product term; if it is equal to 0, it appears complemented. Each variable that is sometimes 1 and sometimes 0 in the group does not appear in the product term.

Four-Variable Map

A four-variable map is constructed by placing 2 three-variable maps together to create four rows in the same fashion as we used 2 two-variable maps to form the four columns in a

three-variable map. Figure 4.6 shows the structure of the four-variable map and the location of minterms. We have included in this figure another frequently used way of designating the rows and columns. As shown in blue, it is sufficient to indicate the rows and columns for which a given variable is equal to 1. Thus $x_1 = 1$ for the two right-most columns, $x_2 = 1$ for the two middle columns, $x_3 = 1$ for the bottom two rows, and $x_4 = 1$ for the two middle rows.

Figure 4.7 gives four examples of four-variable functions. The function f_1 has a group of four 1s in adjacent cells in the bottom two rows, for which $x_2 = 0$ and $x_3 = 1$—they are represented by the product term $\bar{x}_2 x_3$. This leaves the two 1s in the second row to be covered, which can be accomplished with the term $x_1\bar{x}_3x_4$. Hence the minimum-cost implementation of the function is

$$f_1 = \bar{x}_2x_3 + x_1\bar{x}_3x_4$$

The function f_2 includes a group of eight 1s that can be implemented by a single term, x_3. Again, the reader should note that if the remaining two 1s were implemented separately, the result would be the product term $x_1\bar{x}_3x_4$. Implementing these 1s as a part of a group of four 1s, as shown in the figure, gives the less expensive product term x_1x_4.

Just as the left and the right edges of the map are adjacent in terms of the assignment of the variables, so are the top and the bottom edges. Indeed, the four corners of the map are adjacent to each other and thus can form a group of four 1s, which may be implemented by the product term $\bar{x}_2\bar{x}_4$. This case is depicted by the function f_3. In addition to this group of 1s, there are four other 1s that must be covered to implement f_3. This can be done as shown in the figure.

In all examples that we have considered so far, a unique solution exists that leads to a minimum-cost circuit. The function f_4 provides an example where there is some choice. The groups of four 1s in the top-left and bottom-right corners of the map are realized by the terms $\bar{x}_1\bar{x}_3$ and x_1x_3, respectively. This leaves the two 1s that correspond to the term $x_1x_2\bar{x}_3$. But these two 1s can be realized more economically by treating them as a part of a group of four 1s. They can be included in two different groups of four, as shown in the figure.

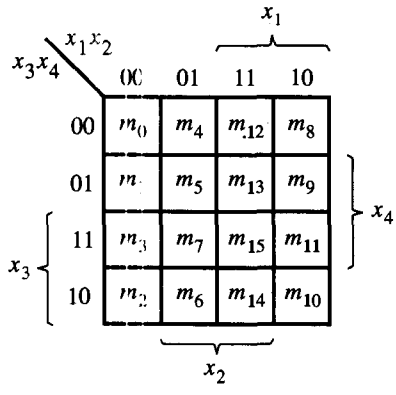


Figure 4.6 A four-variable Karnaugh map.

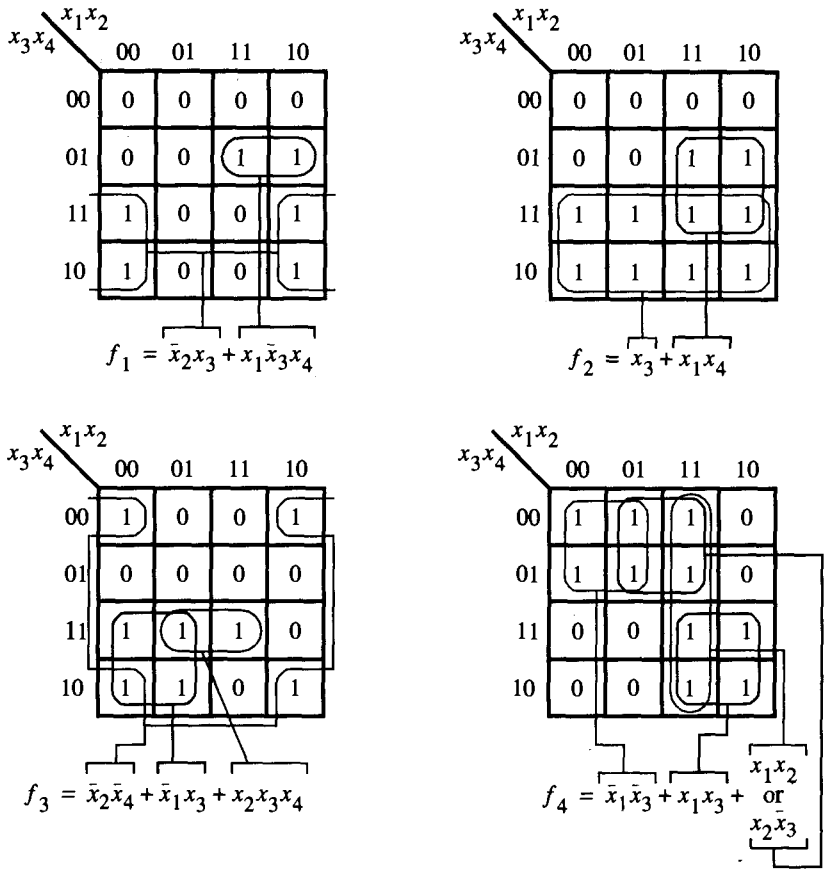


Figure 4.7 Examples of four-variable Karnaugh maps.

One choice leads to the product term x_1x_2, and the other leads to $x_2\bar{x}_3$. Both of these terms have the same cost; hence it does not matter which one is chosen in the final circuit. Note that the complement of x_3 in the term $x_2\bar{x}_3$ does not imply an increased cost in comparison with x_1x_2, because this complement must be generated anyway to produce the term $\bar{x}_1\bar{x}_3$, which is included in the implementation.

Five-Variable Map

We can use 2 four-variable maps to construct a five-variable map. It is easy to imagine a structure where one map is directly behind the other, and they are distinguished by $x_5 = 0$ for one map and $x_5 = 1$ for the other map. Since such a structure is awkward to draw, we can simply place the two maps side by side as shown in Figure 4.8. For the logic function given in this example, two groups of four 1s appear in the same place in both four-variable maps; hence their realization does not depend on the value of x_5. The same is true for the two groups of two 1s in the second row. The 1 in the top-right corner appears only in the

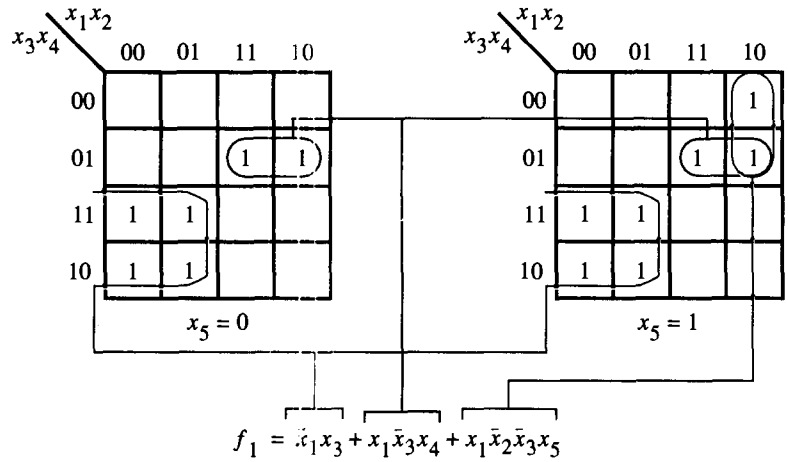


Figure 4.8 A five-variable Karnaugh map.

right map, where $x_5 = 1$; it is a part of the group of two 1s realized by the term $x_1\bar{x}_2\bar{x}_3x_5$. Note that in this map we left blank those cells for which $f = 0$, to make the figure more readable. We will do likewise in a number of maps that follow.

Using a five-variable map is obviously more awkward than using maps with fewer variables. Extending the Karnaugh map concept to more variables is not useful from the practical point of view. This is not troublesome, because practical synthesis of logic functions is done with CAD tools that perform the necessary minimization automatically. Although Karnaugh maps are occasionally useful for designing small logic circuits, our main reason for introducing the Karnaugh maps is to provide a simple vehicle for illustrating the ideas involved in the minimization process.

4.2 STRATEGY FOR MINIMIZATION

For the examples in the preceding section, we used an intuitive approach to decide how the 1s in a Karnaugh map should be grouped together to obtain the minimum-cost implementation of a given function. Our intuitive strategy was to find as few as possible and as large as possible groups of 1s that cover all cases where the function has a value of 1. Each group of 1s has to comprise cells that can be represented by a single product term. The larger the group of 1s, the fewer the number of variables in the corresponding product term. This approach worked well because the Karnaugh maps in our examples were small. For larger logic functions, which have many variables, such intuitive approach is unsuitable. Instead, we must have an organized method for deriving a minimum-cost implementation. In this section we will introduce a possible method, which is similar to the techniques that are automated in CAD tools. To illustrate the main ideas, we will use Karnaugh maps. Later,

in section 4.8, we will describe a different way of representing logic functions, which is used in CAD tools.

4.2.1 TERMINOLOGY

A huge amount of research work has gone into the development of techniques for synthesis of logic functions. The results of this research have been published in numerous papers. To facilitate the presentation of the results, certain terminology has evolved that avoids the need for using highly descriptive phrases. We define some of this terminology in the following paragraphs because it is useful for describing the minimization process.

Literal

A given product term consists of some number of variables, each of which may appear either in uncomplemented or complemented form. Each appearance of a variable, either uncomplemented or complemented, is called a *literal*. For example, the product term $x_1\bar{x}_2x_3$ has three literals, and the term $\bar{x}_1x_3\bar{x}_4x_6$ has four literals.

Implicant

A product term that indicates the input valuation(s) for which a given function is equal to 1 is called an *implicant* of the function. The most basic implicants are the minterms, which we introduced in section 2.6.1. For an n-variable function, a minterm is an implicant that consists of n literals.

Consider the three-variable function in Figure 4.9. There are 11 possible implicants for this function. This includes the five minterms: $\bar{x}_1\bar{x}_2\bar{x}_3$, $\bar{x}_1\bar{x}_2x_3$, $\bar{x}_1x_2\bar{x}_3$, $\bar{x}_1x_2x_3$, and $x_1x_2x_3$. Then there are the implicants that correspond to all possible pairs of minterms that can be combined, namely, $\bar{x}_1\bar{x}_2$ (m_0 and m_1), $\bar{x}_1\bar{x}_3$ (m_0 and m_2), $\bar{x}_1x_3$ (m_1 and m_3), $\bar{x}_1x_2$ (m_2 and m_3), and x_2x_3 (m_3 and m_7). Finally, there is one implicant that covers a group of four minterms, which consists of a single literal $\bar{x}_1$.

Prime Implicant

An implicant is called a *prime implicant* if it cannot be combined into another implicant that has fewer literals. Another way of stating this definition is to say that it is impossible to delete any literal in a prime implicant and still have a valid implicant.

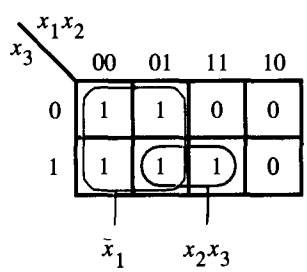


Figure 4.9 Three-variable function $f(x_1, x_2, x_3) = \sum m(0, 1, 2, 3, 7)$.

In Figure 4.9 there are two prime implicants: $\bar{x}_1$ and x_2x_3. It is not possible to delete a literal in either of them. Doing so for $\bar{x}_1$ would make it disappear. For x_2x_3, deleting a literal would leave either x_2 or x_3. But x_2 is not an implicant because it includes the valuation $(x_1, x_2, x_3) = 110$ for which $f = 0$, and x_3 is not an implicant because it includes $(x_1, x_2, x_3) = 101$ for which $f = 0$.

Cover

A collection of implicants that account for all valuations for which a given function is equal to 1 is called a *cover* of that function. A number of different covers exist for most functions. Obviously, a set of all minterms for which $f = 1$ is a cover. It is also apparent that a set of all prime implicants is a cover.

A cover defines a particular implementation of the function. In Figure 4.9 a cover consisting of minterms leads to the expression

$$f = \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_2x_3 + x_1x_2x_3$$

Another valid cover is given by the expression

$$f = \bar{x}_1\bar{x}_2 + \bar{x}_1x_2 + x_2x_3$$

The cover comprising the prime implicants is

$$f = \bar{x}_1 + x_2x_3$$

While all of these expressions represent the function f correctly, the cover consisting of prime implicants leads to the lowest-cost implementation.

Cost

In Chapter 2 we suggested that a good indication of the cost of a logic circuit is the number of gates plus the total number of inputs to all gates in the circuit. We will use this definition of cost throughout the book. But we will assume that primary inputs, namely, the input variables, are available in both true and complemented forms at zero cost. Thus the expression

$$f = x_1\bar{x}_2 + x_3\bar{x}_4$$

has a cost of nine because it can be implemented using two AND gates and one OR gate, with six inputs to the AND and OR gates.

If an inversion is needed inside a circuit, then the corresponding NOT gate and its input are included in the cost. For example, the expression

$$g = \overline{x_1\bar{x}_2 + x_3}(\bar{x}_4 + x_5)$$

is implemented using two AND gates, two OR gates, and one NOT gate to complement $(x_1\bar{x}_2 + x_3)$, with nine inputs. Hence the total cost is 14.

4.2.2 MINIMIZATION PROCEDURE

We have seen that it is possible to implement a given logic function with various circuits. These circuits may have different structures and different costs. When designing a logic

circuit, there are usually certain criteria that must be met. One such criterion is likely to be the cost of the circuit, which we considered in the previous discussion. In general, the larger the circuit, the more important the cost issue becomes. In this section we will assume that the main objective is to obtain a minimum-cost circuit.

Having said that cost is the primary concern, we should note that other optimization criteria may be more appropriate in some cases. For instance, in Chapter 3 we described several types of programmable-logic devices (PLDs) that have a predefined basic structure and can be programmed to realize a variety of different circuits. For such devices the main objective is to design a particular circuit so that it will fit into the target device. Whether or not this circuit has the minimum cost is not important if it can be realized successfully on the device. A CAD tool intended for design with a specific device in mind will automatically perform optimizations that are suitable for that device. We will show in section 4.6 that the way in which a circuit should be optimized may be different for different types of devices.

In the previous subsection we concluded that the lowest-cost implementation is achieved when the cover of a given function consists of prime implicants. The question then is how to determine the minimum-cost subset of prime implicants that will cover the function. Some prime implicants may have to be included in the cover, while for others there may be a choice. If a prime implicant includes a minterm for which $f = 1$ that is not included in any other prime implicant, then it must be included in the cover and is called an *essential prime implicant.* In the example in Figure 4.9, both prime implicants are essential. The term x_2x_3 is the only prime implicant that covers the minterm m_7, and $\bar{x}_1$ is the only one that covers the minterms m_0, m_1, and m_2. Notice that the minterm m_3 is covered by both of these prime implicants. The minimum-cost realization of the function is

$$f = \bar{x}_1 + x_2x_3$$

We will now present several examples in which there is a choice as to which prime implicants to include in the final cover. Consider the four-variable function in Figure 4.10. There are five prime implicants: $\bar{x}_1x_3$, $\bar{x}_2x_3$, $x_3\bar{x}_4$, $\bar{x}_1x_2x_4$, and $x_2\bar{x}_3x_4$. The essential ones

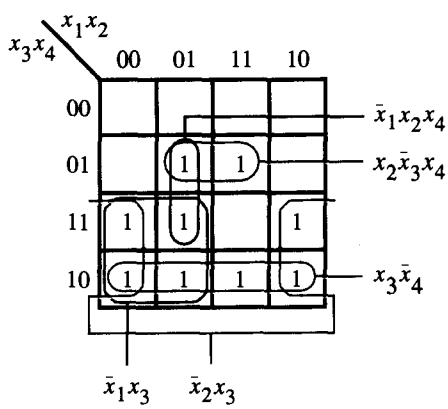


Figure 4.10 Four-variable function $f(x_1, \ldots, x_4) = \sum m(2, 3, 5, 6, 7, 10, 11, 13, 14)$.

(highlighted in blue) are $\bar{x}_2x_3$ (because of m_{11}), $x_3\bar{x}_4$ (because of m_{14}), and $x_2\bar{x}_3x_4$ (because of m_{13}). They must be included in the cover. These three prime implicants cover all minterms for which $f = 1$ except m_7. It is clear that m_7 can be covered by either $\bar{x}_1x_3$ or $\bar{x}_1x_2x_4$. Because $\bar{x}_1x_3$ has a lower cost, it is chosen for the cover. Therefore, the minimum-cost realization is

$$f = \bar{x}_2x_3 + x_3\bar{x}_4 + x_2\bar{x}_3x_4 + \bar{x}_1x_3$$

From the preceding discussion, the process of finding a minimum-cost circuit involves the following steps:

1. Generate all prime implicants for the given function f.
2. Find the set of essential prime implicants.
3. If the set of essential prime implicants covers all valuations for which $f = 1$, then this set is the desired cover of f. Otherwise, determine the nonessential prime implicants that should be added to form a complete minimum-cost cover.

The choice of nonessential prime implicants to be included in the cover is governed by the cost considerations. This choice is often not obvious. Indeed, for large functions there may exist many possibilities, and some *heuristic* approach (i.e., an approach that considers only a subset of possibilities but gives good results most of the time) has to be used. One such approach is to arbitrarily select one nonessential prime implicant and include it in the cover and then determine the rest of the cover. Next, another cover is determined assuming that this prime implicant is not in the cover. The costs of the resulting covers are compared, and the less-expensive cover is chosen for implementation.

We can illustrate the process by using the function in Figure 4.11. Of the six prime implicants, only $\bar{x}_3\bar{x}_4$ is essential. Consider next $x_1x_2\bar{x}_3$ and assume first that it will be included in the cover. Then the remaining three minterms, m_{10}, m_{11}, and m_{15}, will require two more prime implicants to be included in the cover. A possible implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_3x_4 + x_1\bar{x}_2x_3$$

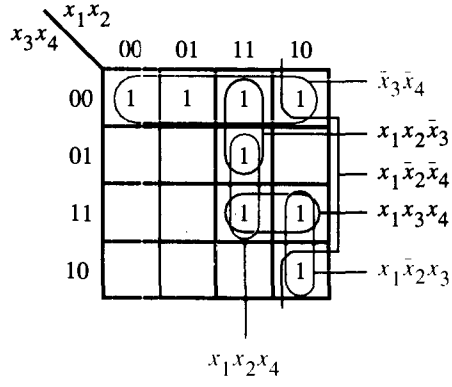


Figure 4.11 The function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$.

The second possibility is that $x_1x_2\bar{x}_3$ is not included in the cover. Then $x_1x_2x_4$ becomes essential because there is no other way of covering m_{13}. Because $x_1x_2x_4$ also covers m_{15}, only m_{10} and m_{11} remain to be covered, which can be achieved with $x_1\bar{x}_2x_3$. Therefore, the alternative implementation is

$$f = \bar{x}_3\bar{x}_4 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

Clearly, this implementation is a better choice.

Sometimes there may not be any essential prime implicants at all. An example is given in Figure 4.12. Choosing any of the prime implicants and first including it, then excluding it from the cover leads to two alternatives of equal cost. One includes the prime implicants indicated in black, which yields

$$f = \bar{x}_1\bar{x}_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_1x_3x_4 + \bar{x}_2x_3\bar{x}_4$$

The other includes the prime implicants indicated in blue, which yields

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + \bar{x}_1x_2\bar{x}_3 + x_1x_2x_4 + x_1\bar{x}_2x_3$$

This procedure can be used to find minimum-cost implementations of both small and large logic functions. For our small examples it was convenient to use Karnaugh maps to determine the prime implicants of a function and then choose the final cover. Other techniques based on the same principles are much more suitable for use in CAD tools; we will introduce such techniques in sections 4.9 and 4.10.

The previous examples have been based on the sum-of-products form. We will next illustrate that the same concepts apply for the product-of-sums form.

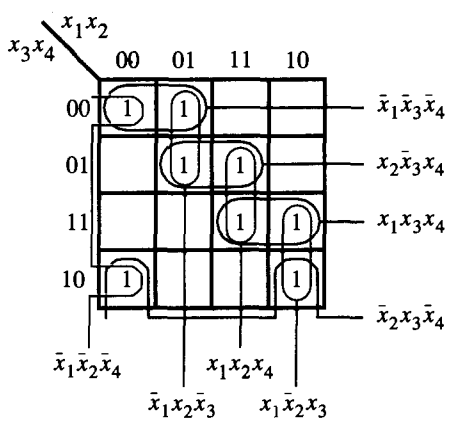


Figure 4.12 The function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$.

4.3 MINIMIZATION OF PRODUCT-OF-SUMS FORMS

Now that we know how to find the minimum-cost sum-of-products (SOP) implementations of functions, we can use the same techniques and the principle of duality to obtain minimum-cost product-of-sums (POS) implementations. In this case it is the maxterms for which $f = 0$ that have to be combined into sum terms that are as large as possible. Again, a sum term is considered larger if it covers more maxterms, and the larger the term, the less costly it is to implement.

Figure 4.13 depicts the same function as Figure 4.9 depicts. There are three maxterms that must be covered: M_4, M_5, and M_6. They can be covered by two sum terms shown in the figure, leading to the following implementation:

$$f = (\bar{x}_1 + x_2)(\bar{x}_1 + x_3)$$

A circuit corresponding to this expression has two OR gates and one AND gate, with two inputs for each gate. Its cost is greater than the cost of the equivalent SOP implementation derived in Figure 4.9, which requires only one OR gate and one AND gate.

The function from Figure 4.10 is reproduced in Figure 4.14. The maxterms for which $f = 0$ can be covered as shown, leading to the expression

$$f = (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)$$

This expression represents a circuit with three OR gates and one AND gate. Two of the OR gates have two inputs, and the third has four inputs; the AND gate has three inputs. Assuming that both the complemented and uncomplemented versions of the input variables x_1 to x_4 are available at no extra cost, the cost of this circuit is 15. This compares favorably with the SOP implementation derived from Figure 4.10, which requires five gates and 13 inputs at a total cost of 18.

In general, as we already know from section 2.6.1, the SOP and POS implementations of a given function may or may not entail the same cost. The reader is encouraged to find the POS implementations for the functions in Figures 4.11 and 4.12 and compare the costs with the SOP forms.

We have shown how to obtain minimum-cost POS implementations by finding the largest sum terms that cover all maxterms for which $f = 0$. Another way of obtaining

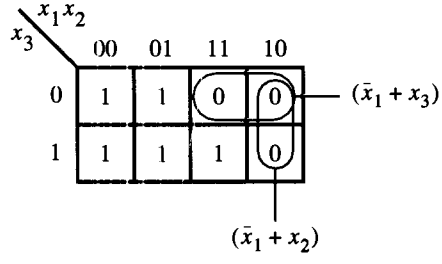


Figure 4.13 POS minimization of $f(x_1, x_2, x_3) = \Pi M(4, 5, 6)$.

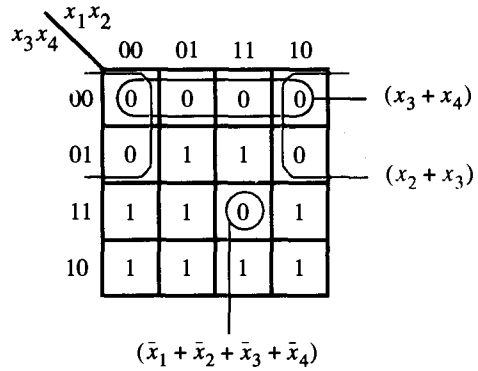


Figure 4.14 POS minimization of $f(x_1, \ldots, x_4) = \Pi M(0, 1, 4, 8, 9, 12, 15)$.

the same result is by finding a minimum-cost SOP implementation of the complement of f. Then we can apply DeMorgan's theorem to this expression to obtain the simplest POS realization because $f = \bar{\bar{f}}$. For example, the simplest SOP implementation of $\bar{f}$ in Figure 4.13 is

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

Complementing this expression using DeMorgan's theorem yields

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

which is the same result as obtained above.

Using this approach for the function in Figure 4.14 gives

$$\bar{f} = \bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4$$

Complementing this expression produces

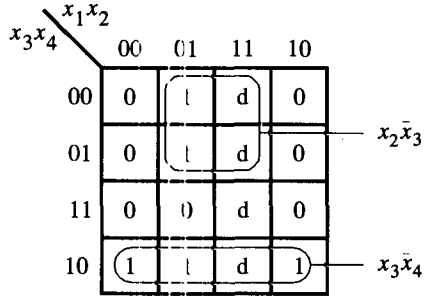
$$\begin{aligned} f = \bar{\bar{f}} &= \overline{\bar{x}_2\bar{x}_3 + \bar{x}_3\bar{x}_4 + x_1x_2x_3x_4} \\ &= \overline{\bar{x}_2\bar{x}_3} \cdot \overline{\bar{x}_3\bar{x}_4} \cdot \overline{x_1x_2x_3x_4} \\ &= (x_2 + x_3)(x_3 + x_4)(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \end{aligned}$$

which matches the previously derived implementation.

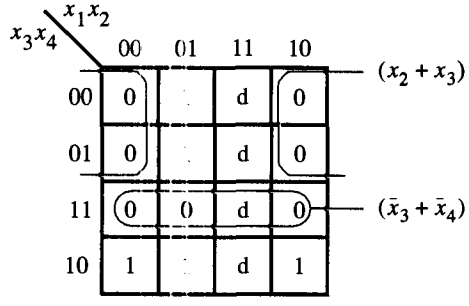
4.4 INCOMPLETELY SPECIFIED FUNCTIONS

In digital systems it often happens that certain input conditions can never occur. For example, suppose that x_1 and x_2 control two interlocked switches such that both switches cannot be closed at the same time. Thus the only three possible states of the switches are that both switches are open or that one switch is open and the other switch is closed. Namely, the input valuations $(x_1, x_2) = 00, 01$, and 10 are possible, but 11 is guaranteed not to occur. Then we say that $(x_1, x_2) = 11$ is a *don't-care condition*, meaning that a circuit with x_1 and x_2 as inputs can be designed by ignoring this condition. A function that has don't-care condition(s) is said to be *incompletely specified.*

Don't-care conditions, or *don't cares* for short, can be used to advantage in the design of logic circuits. Since these input valuations will never occur, the designer may assume that the function value for these valuations is either 1 or 0, whichever is more useful in trying to find a minimum-cost implementation. Figure 4.15 illustrates this idea. The required function has a value of 1 for minterms m_2, m_4, m_5, m_6, and m_{10}. Assuming the above-



(a) SOP implementation



(b) POS implementation

Figure 4.15 Two implementations of the function $f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$.

mentioned interlocked switches, the x_1 and x_2 inputs will never be equal to 1 at the same time; hence the minterms m_{12}, m_{13}, m_{14}, and m_{15} can all be used as don't cares. The don't cares are denoted by the letter d in the map. Using the shorthand notation, the function f is specified as

$$f(x_1, \ldots, x_4) = \sum m(2, 4, 5, 6, 10) + D(12, 13, 14, 15)$$

where D is the set of don't cares.

Part (a) of the figure indicates the best sum-of-products implementation. To form the largest possible groups of 1s, thus generating the lowest-cost prime implicants, it is necessary to assume that the don't cares D_{12}, D_{13}, and D_{14} (corresponding to minterms m_{12}, m_{13}, and m_{14}) have the value of 1 while D_{15} has the value of 0. Then there are only two prime implicants, which provide a complete cover of f. The resulting implementation is

$$f = x_2\bar{x}_3 + x_3\bar{x}_4$$

Part (b) shows how the best product-of-sums implementation can be obtained. The same values are assumed for the don't cares. The result is

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)$$

The freedom in choosing the value of don't cares leads to greatly simplified realizations. If we were to naively exclude the don't cares from the synthesis of the function, by assuming that they always have a value of 0, the resulting SOP expression would be

$$f = \bar{x}_1x_2\bar{x}_3 + \bar{x}_1x_3\bar{x}_4 + \bar{x}_2x_3\bar{x}_4$$

and the POS expression would be

$$f = (x_2 + x_3)(\bar{x}_3 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2)$$

Both of these expressions have higher costs than the expressions obtained with a more appropriate assignment of values to don't cares.

Although don't-care values can be assigned arbitrarily, an arbitrary assignment may not lead to a minimum-cost implementation of a given function. If there are k don't cares, then there are 2^k possible ways of assigning 0 or 1 values to them. In the Karnaugh map we can usually see how best to do this assignment to find the simplest implementation.

Using interlocked switches to illustrate how don't-care conditions can occur in a real system may seem to be somewhat contrived. However, in Chapters 6, 8, and 9 we will encounter many examples of don't cares that occur in the course of practical design of digital circuits.

4.5 MULTIPLE-OUTPUT CIRCUITS

In all previous examples we have considered single functions and their circuit implementations. In practical digital systems it is necessary to implement a number of functions as part of some large logic circuit. Circuits that implement these functions can often be

combined into a less-expensive single circuit with multiple outputs by sharing some of the gates needed in the implementation of individual functions.

Example 4.1 An example of gate sharing is given in Figure 4.16. Two functions, f_1 and f_2, of the same variables are to be implemented. The minimum-cost implementations for these functions are obtained as shown in parts (*a*) and (*b*) of the figure. This results in the expressions

$$f_1 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2\bar{x}_3x_4$$

$$f_2 = x_1\bar{x}_3 + \bar{x}_1x_3 + x_2x_3x_4$$

The cost of f_1 is four gates and 10 inputs, for a total of 14. The cost of f_2 is the same. Thus the total cost is 28 if both functions are implemented by separate circuits. A less-expensive

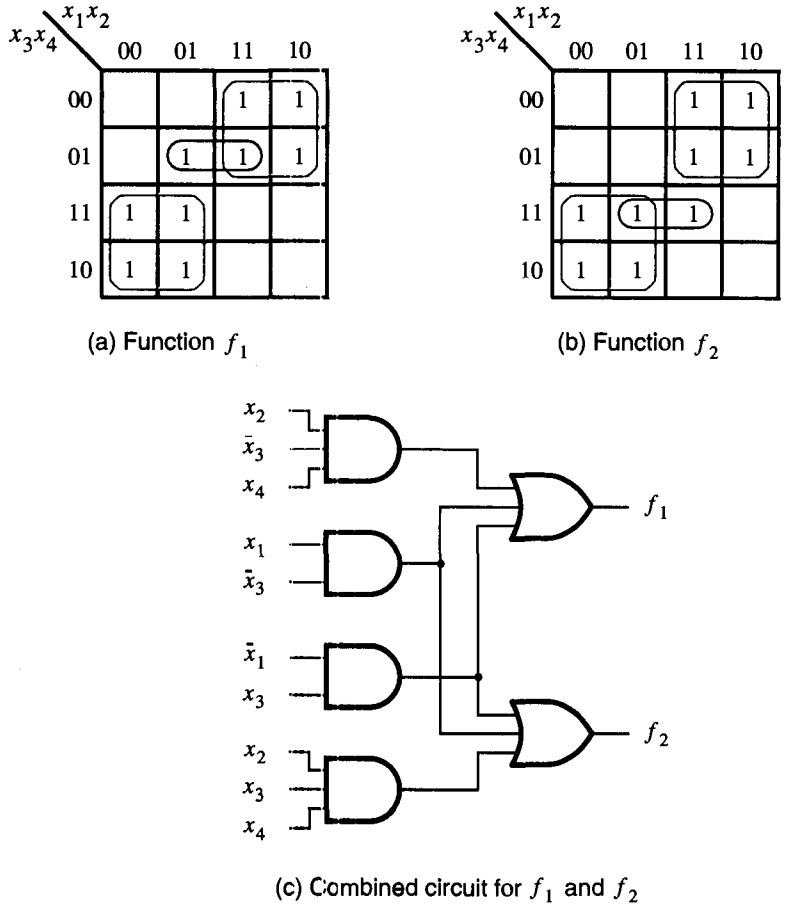


Figure 4.16 An example of multiple-output synthesis.

realization is possible if the two circuits are combined into a single circuit with two outputs. Because the first two product terms are identical in both expressions, the AND gates that implement them need not be duplicated. The combined circuit is shown in Figure 4.16*c*. Its cost is six gates and 16 inputs, for a total of 22.

In this example we reduced the overall cost by finding minimum-cost realizations of f_1 and f_2 and then sharing the gates that implement the common product terms. This strategy does not necessarily always work the best, as the next example shows.

Example 4.2

Figure 4.17 shows two functions to be implemented by a single circuit. Minimum-cost realizations of the individual functions f_3 and f_4 are obtained from parts (*a*) and (*b*) of the figure.

$$f_3 = \bar{x}_1 x_4 + x_2 x_4 + \bar{x}_1 x_2 x_3$$
$$f_4 = x_1 x_4 + \bar{x}_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4$$

None of the AND gates can be shared, which means that the cost of the combined circuit would be six AND gates, two OR gates, and 21 inputs, for a total of 29.

But several alternative realizations are possible. Instead of deriving the expressions for f_3 and f_4 using only prime implicants, we can look for other implicants that may be shared advantageously in the combined realization of the functions. Figure 4.17*c* shows the best choice of implicants, which yields the realization

$$f_3 = x_1 x_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4 + \bar{x}_1 x_4$$
$$f_4 = x_1 x_2 x_4 + \bar{x}_1 x_2 x_3 \bar{x}_4 + \bar{x}_2 x_4$$

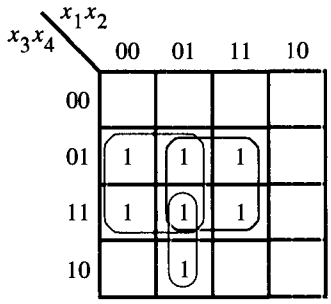
The first two implicants are identical in both expressions. The resulting circuit is given in Figure 4.17*d*. It has the cost of six gates and 17 inputs, for a total of 23.

Example 4.3

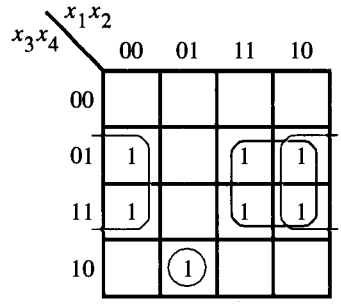
In Example 4.1 we sought the best SOP implementation for the functions f_1 and f_2 in Figure 4.16. We will now consider the POS implementation of the same functions. The minimum-cost POS expressions for f_1 and f_2 are

$$f_1 = (\bar{x}_1 + \bar{x}_3)(x_1 + x_2 + x_3)(x_1 + x_3 + x_4)$$
$$f_2 = (x_1 + x_3)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_3 + x_4)$$

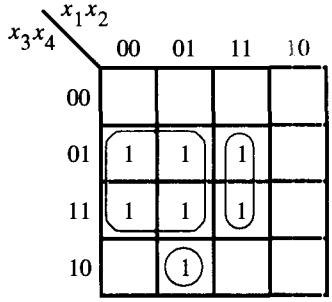
There are no common sum terms in these expressions that could be shared in the implementation. Moreover, from the Karnaugh maps in Figure 4.16, it is apparent that there is no sum term (covering the cells where $f_1 = f_2 = 0$) that can be profitably used in realizing both f_1 and f_2. Thus the best choice is to implement each function separately, according to the preceding expressions. Each function requires three OR gates, one AND gate, and 11 inputs. Therefore, the total cost of the circuit that implements both functions is 30. This realization is costlier than the SOP realization derived in Example 4.1.

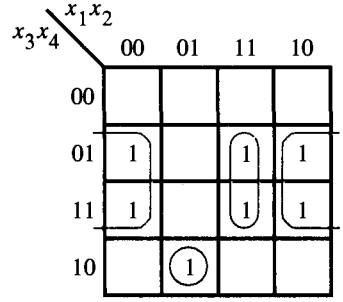


(a) Optimal realization of f_3

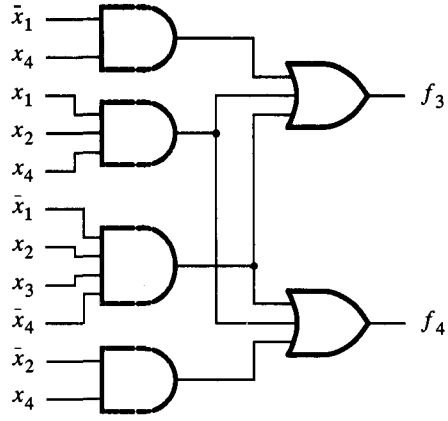


(b) Optimal realization of f_4





(c) Optimal realization of f_3 and f_4 together



(d) Combined circuit for f_3 and f_4

Figure 4.17 Another example of multiple-output synthesis.

Example 4.4

consider now the POS realization of the functions f_3 and f_4 in Figure 4.17. The minimum-cost POS expressions for f_3 and f_4 are

$$f_3 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(\bar{x}_1 + x_2)$$
$$f_4 = (x_3 + x_4)(x_2 + x_4)(\bar{x}_1 + x_4)(x_1 + \bar{x}_2 + \bar{x}_4)$$

The first three sum terms are the same in both f_3 and f_4; they can be shared in a combined circuit. These terms require three OR gates and six inputs. In addition, one 2-input OR gate and one 4-input AND gate are needed for f_3, and one 3-input OR gate and one 4-input AND gate are needed for f_4. Thus the combined circuit comprises five OR gates, two AND gates, and 19 inputs, for a total cost of 26. This cost is slightly higher than the cost of the circuit derived in Example 4.2.

These examples show that the complexities of the best SOP or POS implementations of given functions may be quite different. For the functions in Figures 4.16 and 4.17, the SOP form gives better results. But if we are interested in implementing the complements of the four functions in these figures, then the POS form would be less costly.

Sophisticated CAD tools used to synthesize logic functions will automatically perform the types of optimizations illustrated in the preceding examples.

4.6 MULTILEVEL SYNTHESIS

In the preceding sections our objective was to find a minimum-cost sum-of-products or product-of-sums realization of a given logic function. Logic circuits of this type have *two levels* (stages) of gates. In the sum-of-products form, the first level comprises AND gates that are connected to a second-level OR gate. In the product-of-sums form, the first-level OR gates feed the second-level AND gate. We have assumed that both true and complemented versions of the input variables are available so that NOT gates are not needed to complement the variables.

A two-level realization is usually efficient for functions of a few variables. However, as the number of inputs increases, a two-level circuit may result in fan-in problems. Whether or not this is an issue depends on the type of technology that is used to implement the circuit. For example, consider the following function:

$$f(x_1, \ldots, x_7) = x_1x_3\bar{x}_6 + x_1x_4x_5\bar{x}_6 + x_2x_3x_7 + x_2x_4x_5x_7$$

This is a minimum-cost SOP expression. Now consider implementing f in two types of PLDs: a CPLD and an FPGA. Figure 4.18 shows one of the PAL-like blocks from Figure 3.33. The figure indicates in blue the circuitry used to realize the function f. Clearly, the SOP form of the function is well suited to the chip architecture of the CPLD.

Next, consider implementing f in an FPGA. For this example we will use the FPGA shown in Figure 3.39, which contains two-input LUTs. Since the SOP expression for f requires three- and four-input AND operations and a four-input OR, it cannot be directly

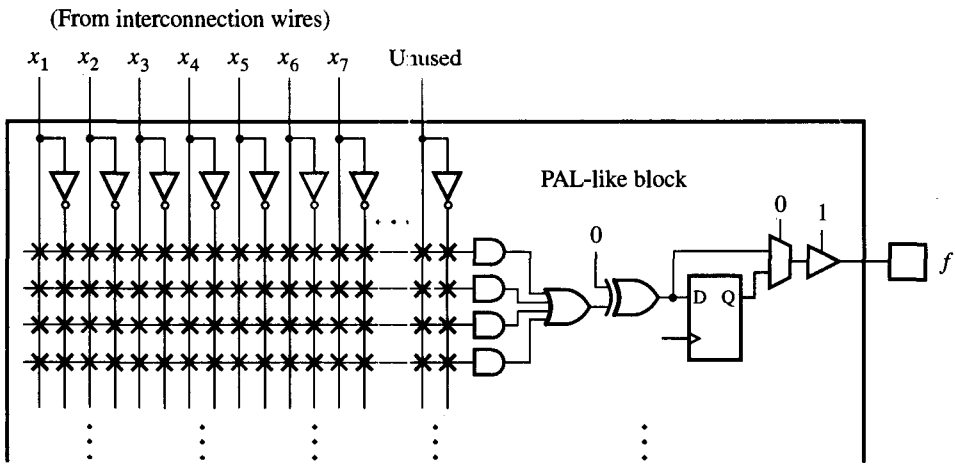


Figure 4.18 Implementation in a CPLD.

implemented in this FPGA. The problem is that the fan-in required to implement the function is too high for our target chip architecture.

To solve the fan-in problem, f must be expressed in a form that has more than two levels of logic operations. Such a form is called a *multilevel* logic expression. There are several different approaches for synthesis of multilevel circuits. We will discuss two important techniques known as *factoring* and *functional decomposition.*

4.6.1 FACTORING

The distributive property in section 2.5 allows us to factor the preceding expression for f as follows

$$\begin{aligned} f &= x_1\bar{x}_6(x_3 + x_4x_5) + x_2x_7(x_3 + x_4x_5) \\ &= (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5) \end{aligned}$$

The corresponding circuit has a maximum fan-in of two; hence it can be realized using two-input LUTs. Figure 4.19 gives a possible implementation using the FPGA from Figure 3.39. Note that a two-variable function that has to be realized by each LUT is indicated in the box that represents the LUT.

Fan-in Problem

In the preceding example, the fan-in restrictions were caused by the fixed structure of the FPGA, where each LUT has only two inputs. However, even when the target chip architecture is not fixed, the fan-in may still be an issue. To illustrate this situation, let us consider the implementation of a circuit in a custom chip. Recall that custom chips usually contain a large number of gates. If the chip is fabricated using CMOS technology, then there will be fan-in limitations as discussed in section 3.8.8. In this technology the number of inputs to a logic gate should be small. For instance, we may wish to limit the number

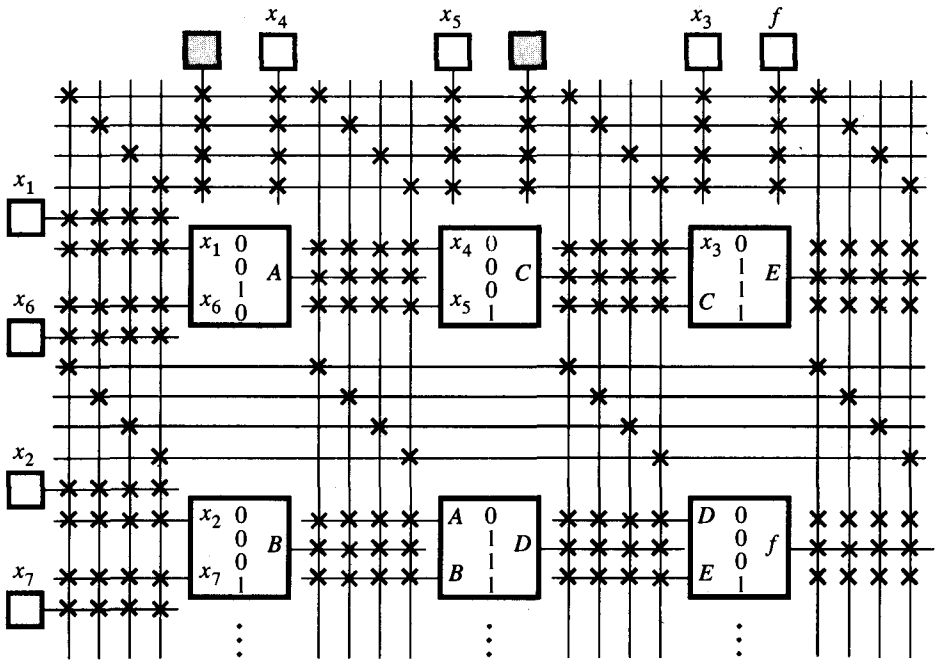


Figure 4.19 Implementation in an FPGA.

of inputs to an AND gate to be less than five. Under this restriction, if a logic expression includes a seven-input product term, we would have to use 2 four-input AND gates, as indicated in Figure 4.20.

Factoring can be used to deal with the fan-in problem. Suppose again that the available gates have a maximum fan-in of four and that we want to realize the function

$$f = x_1\bar{x}_2x_3\bar{x}_4x_5x_6 + x_1x_2\bar{x}_3\bar{x}_4\bar{x}_5x_6$$

This is a minimal sum-of-products expression. Using the approach of Figure 4.20, we will need four AND gates and one OR gate to implement this expression. A better solution is to

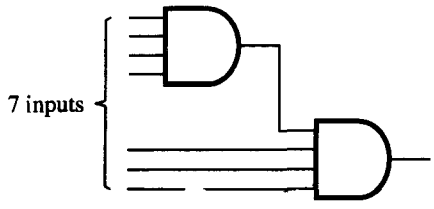


Figure 4.20 Using four-input AND gates to realize a seven-input product term.

factor the expression as follows

$$f = x_1\bar{x}_4x_6(\bar{x}_2x_3x_5 + x_2\bar{x}_3\bar{x}_5)$$

Then three AND gates and one OR gate suffice for realization of the required function, as shown in Figure 4.21.

Example 4.5 In practical situations a designer of logic circuits often encounters specifications that naturally lead to an initial design where the logic expressions are in a factored form. Suppose we need a circuit that meets the following requirements. There are four inputs: x_1, x_2, x_3, and x_4. An output, f_1, must have the value 1 if at least one of the inputs x_1 and x_2 is equal to 1 and both x_3 and x_4 are equal to 1; it must also be 1 if $x_1 = x_2 = 0$ and either x_3 or x_4 is 1. In all other cases $f_1 = 0$. A different output, f_2, is to be equal to 1 in all cases except when both x_1 and x_2 are equal to 0 or when both x_3 and x_4 are equal to 0.

From this specification, the function f_1 can be expressed as

$$f_1 = (x_1 + x_2)x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

This expression can be simplified to

$$f_1 = x_3x_4 + \bar{x}_1\bar{x}_2(x_3 + x_4)$$

which the reader can verify by using a Karnaugh map.

The second function, f_2, is most easily defined in terms of its complement, such that

$$\bar{f}_2 = \bar{x}_1\bar{x}_2 + \bar{x}_3\bar{x}_4$$

Then using DeMorgan's theorem gives

$$f_2 = (x_1 + x_2)(x_3 + x_4)$$

which is the minimum-cost expression for f_2; the cost increases significantly if the SOP form is used.

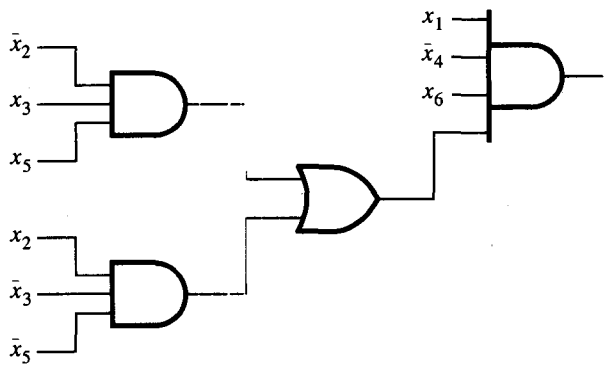


Figure 4.21 A factored circuit.

Because our objective is to design the lowest-cost combined circuit that implements f_1 and f_2, it seems that the best result can be achieved if we use the factored forms for both functions, in which case the sum term $(x_3 + x_4)$ can be shared. Moreover, observing that $\bar{x}_1\bar{x}_2 = \overline{x_1 + x_2}$, the sum term $(x_1 + x_2)$ can also be shared if we express f_1 in the form

$$f_1 = x_3x_4 + \overline{x_1 + x_2}(x_3 + x_4)$$

Then the combined circuit, shown in Figure 4.22, comprises three OR gates, three AND gates, one NOT gate, and 13 inputs, for a total of 20.

Impact on Wiring Complexity

The space on integrated circuit chips is occupied by the circuitry that implements logic gates and by the wires needed to make connections among the gates. The amount of space needed for wiring is a substantial portion of the chip area. Therefore, it is useful to keep the wiring complexity as low as possible.

In a logic expression each literal corresponds to a wire in the circuit that carries the desired logic signal. Since factoring usually reduces the number of literals, it provides a powerful mechanism for reducing the wiring complexity in a logic circuit. In the synthesis process the CAD tools consider many different issues, including the cost of the circuit, the fan-in, and the wiring complexity.

4.6.2 FUNCTIONAL DECOMPOSITION

In the preceding examples, which illustrated the factoring approach, multilevel circuits were used to deal with fan-in limitations. However, such circuits may be preferable to their two-level equivalents even if fan-in is not a problem. In some cases the multilevel circuits may reduce the cost of implementation. On the other hand, they usually imply longer propagation delays, because they use multiple stages of logic gates. We will explore these issues by means of illustrative examples.

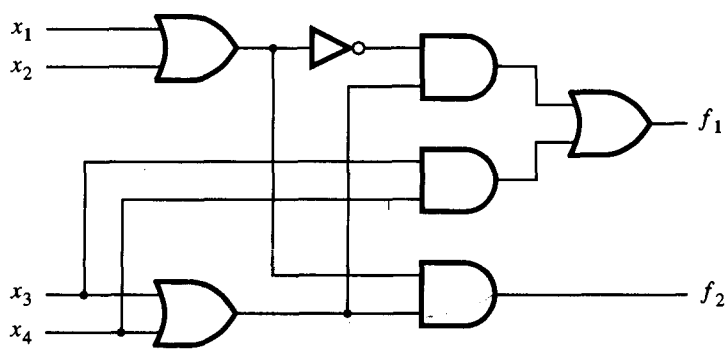


Figure 4.22 Circuit for Example 4.5.

Complexity of a logic circuit, in terms of wiring and logic gates, can often be reduced by *decomposing* a two-level circuit into subcircuits, where one or more subcircuits implement functions that may be used in several places to construct the final circuit. To achieve this objective, a two-level logic expression is replaced by two or more new expressions, which are then combined to define a multilevel circuit. We can illustrate this idea by a simple example.

Example 4.6 Consider the minimum-cost sum-of-products expression

$$f = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 x_4 + \bar{x}_1 \bar{x}_2 x_4$$

and assume that the inputs x_1 to x_4 are available only in their true form. Then the expression defines a circuit that has four AND gates, one OR gate, two NOT gates, and 18 inputs (wires) to all gates. The fan-in is three for the AND gates and four for the OR gate. The reader should observe that in this case we have included the cost of NOT gates needed to complement x_1 and x_2, rather than assume that both true and complemented versions of all input variables are available, as we had done before.

Factoring x_3 from the first two terms and x_4 from the last two terms, this expression becomes

$$f = (\bar{x}_1 x_2 + x_1 \bar{x}_2) x_3 + (x_1 x_2 + \bar{x}_1 \bar{x}_2) x_4$$

Now let $g(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2$ and observe that

$$\begin{aligned}
\bar{g} &= \overline{\bar{x}_1 x_2 + x_1 \bar{x}_2} \\
&= \overline{\bar{x}_1 x_2} \cdot \overline{x_1 \bar{x}_2} \\
&= (x_1 + \bar{x}_2)(\bar{x}_1 + x_2) \\
&= x_1 \bar{x}_1 + x_1 x_2 + \bar{x}_2 \bar{x}_1 + \bar{x}_2 x_2 \\
&= 0 + x_1 x_2 + \bar{x}_1 \bar{x}_2 + 0 \\
&= x_1 x_2 + \bar{x}_1 \bar{x}_2
\end{aligned}$$

Then f can be written as

$$f = g x_3 + \bar{g} x_4$$

which leads to the circuit shown in Figure 4.23. This circuit requires an additional OR gate and a NOT gate to invert the value of g. But it needs only 15 inputs. Moreover, the largest fan-in has been reduced to two. The cost of this circuit is lower than the cost of its two-level equivalent. The trade-off is an increased propagation delay because the circuit has three more levels of logic.

In this example the subfunction g is a function of variables x_1 and x_2. The subfunction is used as an input to the rest of the circuit that completes the realization of the required function f. Let h denote the function of this part of the circuit, which depends on only three inputs: g, x_3, and x_4. Then the decomposed realization of f can be expressed algebraically

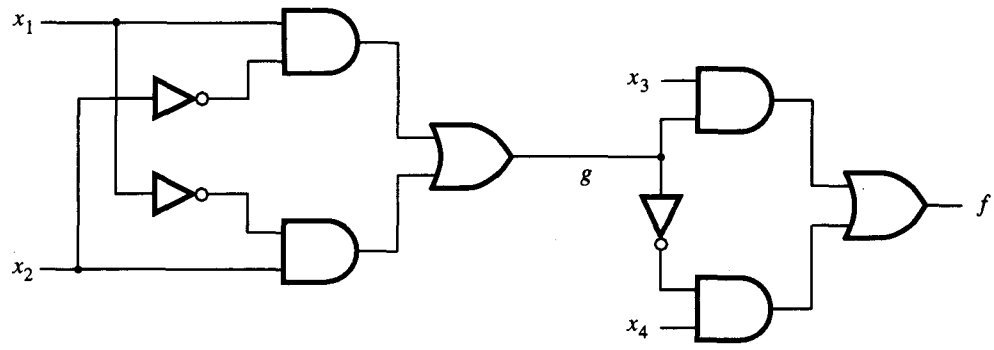


Figure 4.23 Logic circuit for Example 4.6.

as

$$f(x_1, x_2, x_3, x_4) = h[g(x_1, x_2), x_3, x_4]$$

The structure of this decomposition can be described in block-diagram form as shown in Figure 4.24.

While not evident from our first example, functional decomposition can lead to great reductions in the complexity and cost of circuits. The reader will get a good indication of this benefit from the next example.

Example 4.7

Figure 4.25*a* defines a five-variable function f in the form of a Karnaugh map. In searching for a good decomposition for this function, it is necessary to first identify the variables that will be used as inputs to a subfunction. We can get a useful clue from the patterns of 1s in the map. Note that there are only two distinct patterns in the rows of the map. The second

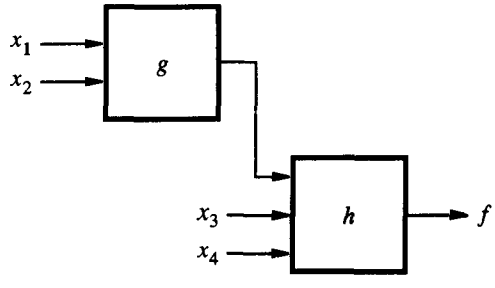
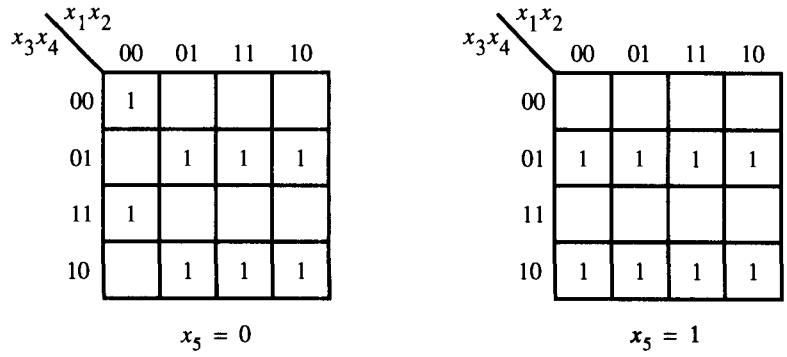
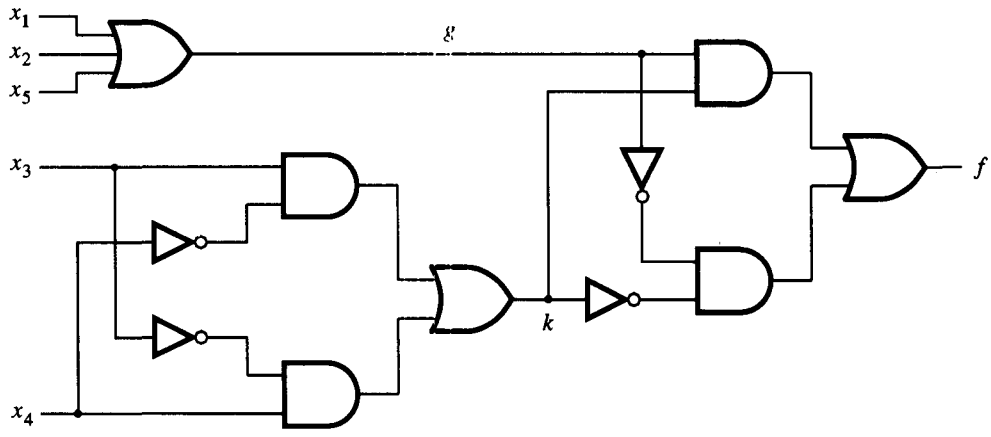


Figure 4.24 The structure of decomposition in Example 4.6.



(a) Karnaugh map for the function f



(b) Circuit obtained using decomposition

Figure 4.25 Decomposition for Example 4.7.

and fourth rows have one pattern, highlighted in blue, while the first and second rows have the other pattern. Once we specify which row each pattern is in, then the pattern itself depends only on the variables that define columns in each row, namely, x_1, x_2, and x_5. Let a subfunction $g(x_1, x_2, x_5)$ represent the pattern in rows 2 and 4. This subfunction is just

$$g = x_1 + x_2 + x_5$$

because the pattern has a 1 wherever any of these variables is equal to 1. To specify the location of rows where the pattern g occurs, we use the variables x_3 and x_4. The terms $\bar{x}_3x_4$ and $x_3\bar{x}_4$ identify the second and fourth rows, respectively. Thus the expression $(\bar{x}_3x_4 + x_3\bar{x}_4) \cdot g$ represents the part of f that is defined in rows 2 and 4.

Next, we have to find a realization for the pattern in rows 1 and 3. This pattern has a 1 only in the cell where $x_1 = x_2 = x_5 = 0$, which corresponds to the term $\bar{x}_1\bar{x}_2\bar{x}_5$. But we can

make a useful observation that this term is just a complement of g. The location of rows 1 and 3 is identified by terms $\bar{x}_3\bar{x}_4$ and x_3x_4, respectively. Thus the expression $(\bar{x}_3\bar{x}_4 + x_3x_4) \cdot \bar{g}$ represents f in rows 1 and 3.

We can make one other useful observation. The expressions $(\bar{x}_3x_4 + x_3\bar{x}_4)$ and $(\bar{x}_3\bar{x}_4 + x_3x_4)$ are complements of each other, as shown in Example 4.6. Therefore, if we let $k(x_3, x_4) = \bar{x}_3x_4 + x_3\bar{x}_4$, the complete decomposition of f can be stated as

$$\begin{aligned} f(x_1, x_2, x_3, x_4, x_5) &= h[g(x_1, x_2, x_5), k(x_3, x_4)] \\ &= kg + \bar{k}\bar{g} \end{aligned}$$

where

$$\begin{aligned} g &= x_1 + x_2 + x_5 \\ k &= \bar{x}_3x_4 + x_3\bar{x}_4 \end{aligned}$$

The resulting circuit is given in Figure 4.25*b*. It requires a total of 11 gates and 19 inputs. The largest fan-in is three.

For comparison, a minimum-cost sum-of-products expression for f is

$$f = x_1\bar{x}_3x_4 + x_1x_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_2x_3\bar{x}_4 + \bar{x}_3x_4x_5 + x_3\bar{x}_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3x_4\bar{x}_5$$

The corresponding circuit requires a total of 14 gates (including the five NOT gates to complement the primary inputs) and 41 inputs. The fan-in for the output OR gate is eight. Obviously, functional decomposition results in a much simpler implementation of this function.

In both of the preceding examples, the decomposition is such that a decomposed subfunction depends on some primary input variables, whereas the remainder of the implementation depends on the rest of the variables. Such decompositions are called *disjoint decompositions* in the technical literature. It is possible to have a *non-disjoint decomposition*, where the variables of the subfunction are also used in realizing the remainder of the circuit. The following example illustrates this possibility.

Example 4.8

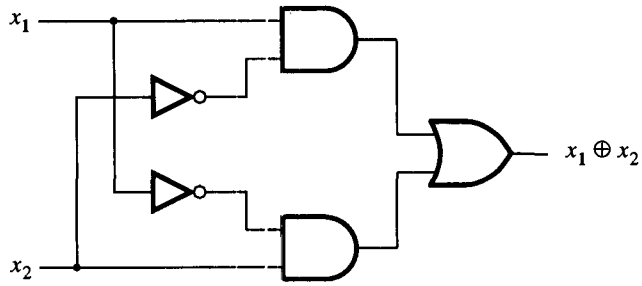
Exclusive-OR (XOR) is a very useful function. In section 3.9.1 we showed how it can be realized using a special circuit. It can also be realized using AND and OR gates as shown in Figure 4.26*a*. In section 2.7 we explained how any AND-OR circuit can be realized as a NAND-NAND circuit that has the same structure.

Let us now try to exploit functional decomposition to find a better implementation of XOR using only NAND gates. Let the symbol $\uparrow$ represent the NAND operation so that $x_1 \uparrow x_2 = \overline{x_1 \cdot x_2}$. A sum-of-products expression for the XOR function is

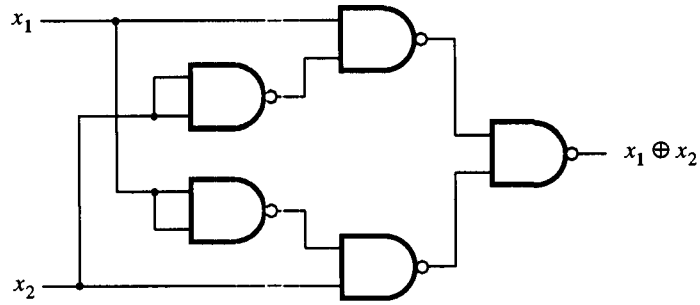
$$x_1 \oplus x_2 = x_1\bar{x}_2 + \bar{x}_1x_2$$

From the discussion in section 2.7, this expression can be written in terms of NAND operations as

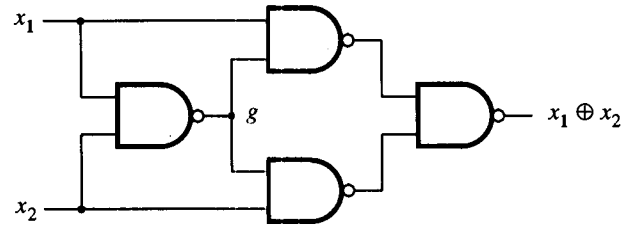
$$x_1 \oplus x_2 = (x_1 \uparrow \bar{x}_2) \uparrow (\bar{x}_1 \uparrow x_2)$$



(a) Sum-of-products implementation



(b) NAND gate implementation



(c) Optimal NAND gate implementation

Figure 4.26 Implementation of XOR.

This expression requires five NAND gates, and it is implemented by the circuit in Figure 4.26*b*. Observe that an inverter is implemented using a two-input NAND gate by tying the two inputs together.

To find a decomposition, we can manipulate the term $(x_1 \uparrow \bar{x}_2)$ as follows:

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

We can perform a similar manipulation for $(\bar{x}_1 \uparrow x_2)$ to generate

$$x_1 \oplus x_2 = (x_1 \uparrow (\bar{x}_1 + \bar{x}_2)) \uparrow ((\bar{x}_1 + \bar{x}_2) \uparrow x_2)$$

DeMorgan's theorem states that $\bar{x}_1 + \bar{x}_2 = x_1 \uparrow x_2$; hence we can write

$$x_1 \oplus x_2 = (x_1 \uparrow (x_1 \uparrow x_2)) \uparrow ((x_1 \uparrow x_2) \uparrow x_2)$$

Now we have a decomposition

$$x_1 \oplus x_2 = (x_1 \uparrow g) \uparrow (g \uparrow x_2)$$

$$g = x_1 \uparrow x_2$$

The corresponding circuit, which requires only four NAND gates, is given in Figure 4.26*c*.

Practical Issues

Functional decomposition is a powerful technique for reducing the complexity of circuits. It can also be used to implement general logic functions in circuits that have built-in constraints. For example, in programmable logic devices (PLDs) that were introduced in Chapter 3 it is necessary to "fit" a desired logic circuit into logic blocks that are available on these devices. The available blocks are a target for decomposed subfunctions that may be used to realize larger functions.

A big problem in functional decomposition is finding the possible subfunctions. For functions of many variables, an enormous number of possibilities should be tried. This situation precludes attempts at finding optimal solutions. Instead, heuristic approaches that lead to acceptable solutions are used.

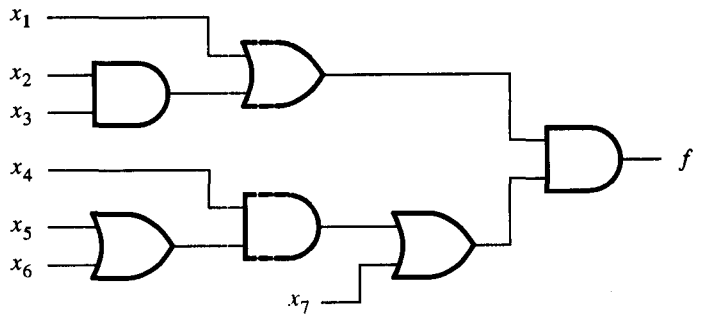
Full discussion of functional decomposition and factoring is beyond the scope of this book. An interested reader may consult other references [2–5]. Modern CAD tools use the concept of decomposition extensively.

4.6.3 MULTILEVEL NAND AND NOR CIRCUITS

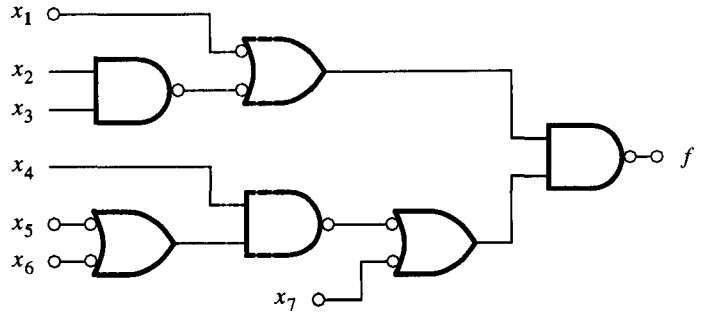
In section 2.7 we showed that two-level circuits consisting of AND and OR gates can be easily converted into circuits that can be realized with NAND and NOR gates, using the same gate arrangement. In particular, an AND-OR (sum-of-products) circuit can be realized as a NAND-NAND circuit, while an OR-AND (product-of-sums) circuit becomes a NOR-NOR circuit. The same conversion approach can be used for multilevel circuits. We will illustrate this approach by an example.

Example 4.9

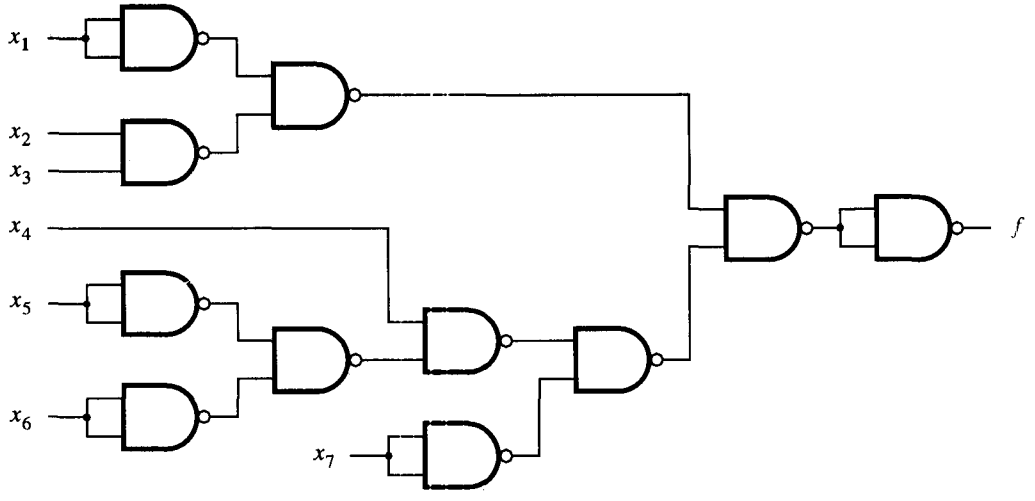
Figure 4.27*a* gives a four-level circuit consisting of AND and OR gates. Let us first derive a functionally equivalent circuit that comprises only NAND gates. Each AND gate is converted to a NAND by inverting its output. Each OR gate is converted to a NAND by inverting its inputs. This is just an application of DeMorgan's theorem, as illustrated in Figure 2.21*a*. Figure 4.27*b* shows the necessary inversions in blue. Note that an inversion is



(a) Circuit with AND and OR gates



(b) Inversions needed to convert to NANDs

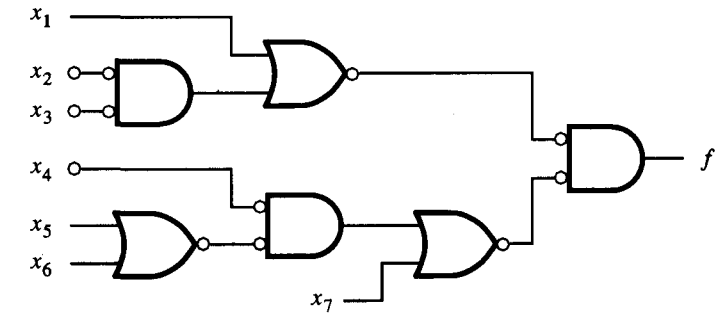


(c) NAND-gate circuit

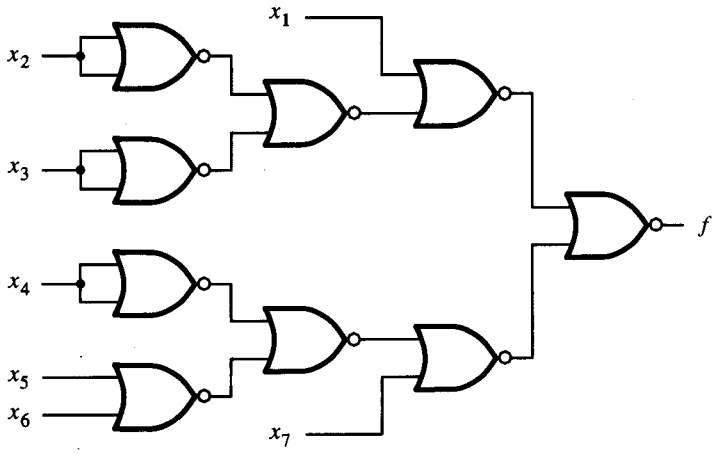
Figure 4.27 Conversion to a NAND-gate circuit.

applied at both ends of a given wire. Now each gate becomes a NAND gate. This accounts for most of the inversions added to the original circuit. But, there are still four inversions that are not a part of any gate; therefore, they must be implemented separately. These inversions are at inputs x_1, x_5, x_6, and x_7 and at the output f. They can be implemented as two-input NAND gates, where the inputs are tied together. The resulting circuit is shown in Figure 4.27*c*.

A similar approach can be used to convert the circuit in Figure 4.27*a* into a circuit that comprises only NOR gates. An OR gate is converted to a NOR gate by inverting its output. An AND becomes a NOR if its inputs are inverted, as indicated in Figure 2.21*b*. Using this approach, the inversions needed for our sample circuit are shown in blue in Figure 4.28*a*.



(a) Inversions needed to convert to NORs



(b) NOR-gate circuit

Figure 4.28 Conversion to a NOR-gate circuit.

Then each gate becomes a NOR gate. The three inversions at inputs x_2, x_3, and x_4 can be realized as two-input NOR gates, where the inputs are tied together. The resulting circuit is presented in Figure 4.28*b*.

It is evident that the basic topology of a circuit does not change substantially when converting from AND and OR gates to either NAND or NOR gates. However, it may be necessary to insert additional gates to serve as NOT gates that implement inversions not absorbed as a part of other gates in the circuit.

4.7 ANALYSIS OF MULTILEVEL CIRCUITS

The preceding section showed that it may be advantageous to implement logic functions using multilevel circuits. It also presented the most commonly used approaches for synthesizing functions in this way. In this section we will consider the task of analyzing an existing circuit to determine the function that it implements.

For two-level circuits the analysis process is simple. If a circuit has an AND-OR (NAND-NAND) structure, then its output function can be written in the SOP form by inspection. Similarly, it is easy to derive a POS expression for an OR-AND (NOR-NOR) circuit. The analysis task is more complicated for multilevel circuits because it is difficult to write an expression for the function by inspection. We have to derive the desired expression by tracing the circuit and determining its functionality. The tracing can be done either starting from the input side and working towards the output, or by starting at the output side and working back towards the inputs. At intermediate points in the circuit, it is necessary to evaluate the subfunctions realized by the logic gates.

ample 4.10 Figure 4.29 replicates the circuit from Figure 4.27*a*. To determine the function f implemented by this circuit, we can consider the functionality at internal points that are the outputs

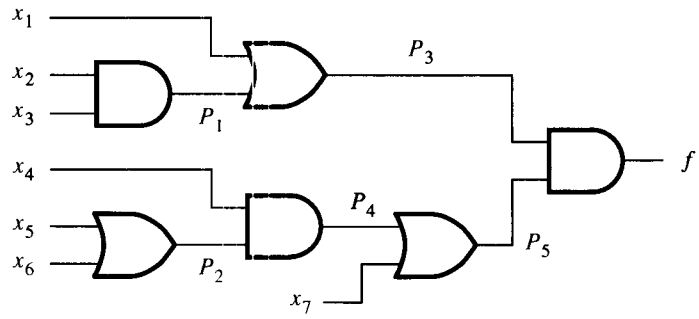


Figure 4.29 Circuit for Example 4.10.

of various gates. These points are labeled P_1 to P_5 in the figure. The functions realized at these points are

$$P_1 = x_2x_3$$

$$P_2 = x_5 + x_6$$

$$P_3 = x_1 + P_1 = x_1 + x_2x_3$$

$$P_4 = x_4P_2 = x_4(x_5 + x_6)$$

$$P_5 = P_4 + x_7 = x_4(x_5 + x_6) + x_7$$

Then f can be evaluated as

$$\begin{aligned} f &= P_3P_5 \\ &= (x_1 + x_2x_3)(x_4(x_5 + x_6) + x_7) \end{aligned}$$

Applying the distributive property to eliminate the parentheses gives

$$f = x_1x_4x_5 + x_1x_4x_6 + x_1x_7 + x_2x_3x_4x_5 + x_2x_3x_4x_6 + x_2x_3x_7$$

Note that the expression represents a circuit comprising six AND gates, one OR gate, and 25 inputs. The cost of this two-level circuit is higher than the cost of the circuit in Figure 4.29, but the circuit has lower propagation delay.

Example 4.11

In Example 4.7 we derived the circuit in Figure 4.25*b*. In addition to AND gates and OR gates, the circuit has some NOT gates. It is reproduced in Figure 4.30, and the internal points are labeled from P_1 to P_{10} as shown. The following subfunctions occur

$$P_1 = x_1 + x_2 + x_5$$

$$P_2 = \overline{x}_4$$

$$P_3 = \overline{x}_3$$

$$P_4 = x_3P_2$$

$$P_5 = x_4P_3$$

$$P_6 = P_4 + P_5$$

$$P_7 = \overline{P}_1$$

$$P_8 = \overline{P}_6$$

$$P_9 = P_1P_6$$

$$P_{10} = P_7P_8$$

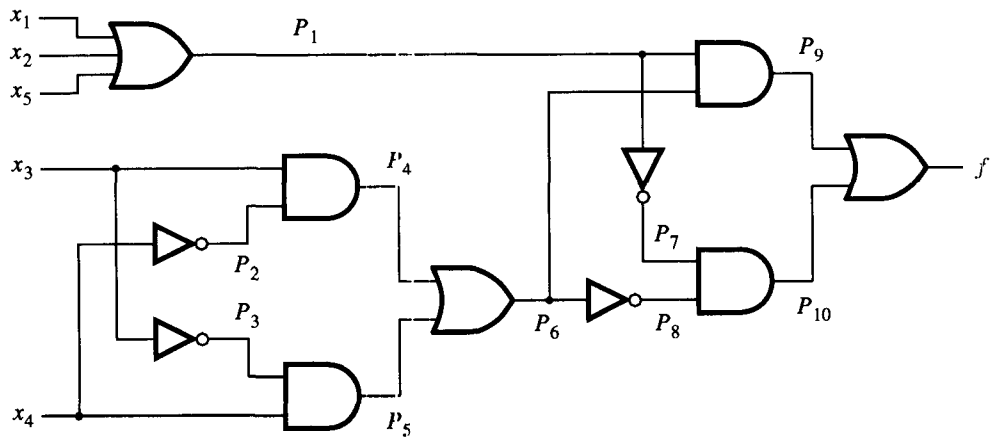


Figure 4.30 Circuit for Example 4.11.

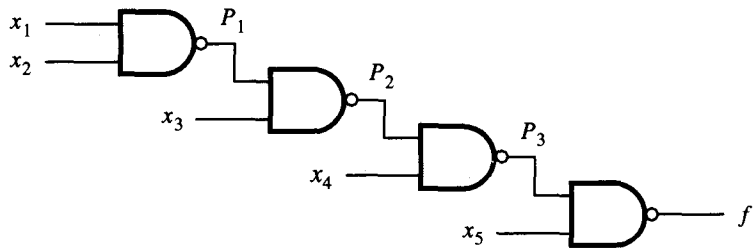
We can derive f by tracing the circuit from the output towards the inputs as follows

$$\begin{aligned}
f &= P_9 + P_{10} \\
&= P_1P_6 + P_7P_8 \\
&= (x_1 + x_2 + x_5)(P_4 + P_5) + \overline{P}_1\overline{P}_6 \\
&= (x_1 + x_2 + x_5)(x_3P_2 + x_4P_3) + \bar{x}_1\bar{x}_2\bar{x}_5\overline{P}_4\overline{P}_5 \\
&= (x_1 + x_2 + x_5)(x_3\bar{x}_4 + x_4\bar{x}_3) + \bar{x}_1\bar{x}_2\bar{x}_5(\bar{x}_3 + \overline{P}_2)(\bar{x}_4 + \overline{P}_3) \\
&= (x_1 + x_2 + x_5)(x_3\bar{x}_4 + \bar{x}_3x_4) + \bar{x}_1\bar{x}_2\bar{x}_5(\bar{x}_3 + x_4)(\bar{x}_4 + x_3) \\
&= x_1x_3\bar{x}_4 + x_1\bar{x}_3x_4 + x_2x_3\bar{x}_4 + x_2\bar{x}_3x_4 + x_5x_3\bar{x}_4 + x_5\bar{x}_3x_4 + \\
&\quad \bar{x}_1\bar{x}_2\bar{x}_5\bar{x}_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_5x_4x_3
\end{aligned}$$

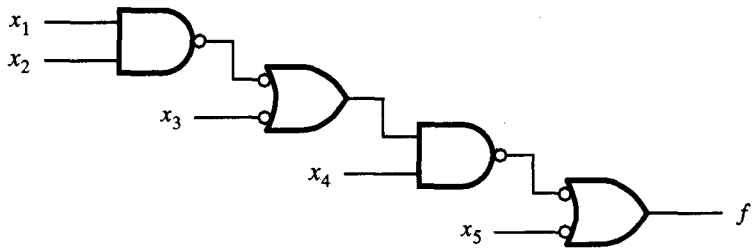
This is the same expression as stated in Example 4.7.

Example 4.12 Circuits based on NAND and NOR gates are slightly more difficult to analyze because each gate involves an inversion. Figure 4.31*a* depicts a simple NAND-gate circuit that illustrates the effect of inversions. We can convert this circuit into a circuit with AND and OR gates using the reverse of the approach described in Example 4.9. Bubbles that denote inversions can be moved, according to DeMorgan's theorem, as indicated in Figure 4.31*b*. Then the circuit can be converted into the circuit in part (*c*) of the figure, which consists of AND and OR gates. Observe that in the converted circuit, the inputs x_3 and x_5 are complemented. From this circuit the function f is determined as

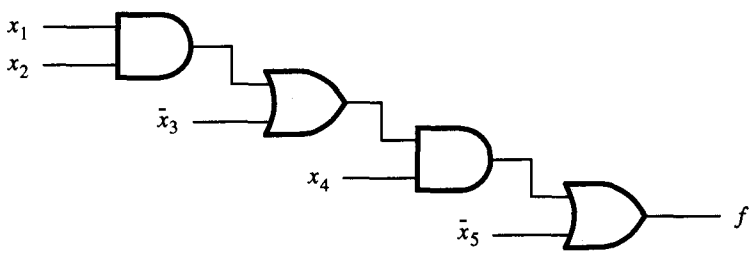
$$\begin{aligned}
f &= (x_1x_2 + \bar{x}_3)x_4 + \bar{x}_5 \\
&= x_1x_2x_4 + \bar{x}_3x_4 + \bar{x}_5
\end{aligned}$$



(a) NAND-gate circuit



(b) Moving bubbles to convert to ANDs and ORs



(c) Circuit with AND and OR gates

Figure 4.31 Circuit for Example 4.12.

It is not necessary to convert a NAND circuit into a circuit with AND and OR gates to determine its functionality. We can use the approach from Examples 4.10 and 4.11 to derive f as follows. Let P_1, P_2, and P_3 label the internal points as shown in Figure 4.31*a*. Then

$$P_1 = \overline{x_1 x_2}$$

$$P_2 = \overline{P_1 x_3}$$

$$
\begin{aligned}
P_3 &= \overline{P_2 x_4} \\
f &= \overline{P_3 x_5} = \overline{P}_3 + \overline{x}_5 \\
&= \overline{\overline{P_2 x_4}} + \overline{x}_5 = P_2 x_4 + \overline{x}_5 \\
&= \overline{P_1 x_3} x_4 + \overline{x}_5 = (\overline{P}_1 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (\overline{\overline{x_1 x_2}} + \overline{x}_3) x_4 + \overline{x}_5 \\
&= (x_1 x_2 + \overline{x}_3) x_4 + \overline{x}_5 \\
&= x_1 x_2 x_4 + \overline{x}_3 x_4 + \overline{x}_5
\end{aligned}
$$

Example 4.13 The circuit in Figure 4.32 consists of NAND and NOR gates. It can be analyzed as follows.

$$
\begin{aligned}
P_1 &= \overline{x_2 x_3} \\
P_2 &= \overline{x_1 P_1} = \overline{x}_1 + \overline{P}_1 \\
P_3 &= \overline{x_3 x_4} = \overline{x}_3 + \overline{x}_4 \\
P_4 &= \overline{P_2 + P_3} \\
f &= \overline{P_4 + x_5} = \overline{P}_4 \overline{x}_5 \\
&= \overline{\overline{P_2 + P_3}} \cdot \overline{x}_5 \\
&= (P_2 + P_3) \overline{x}_5 \\
&= (\overline{x}_1 + \overline{P}_1 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= (\overline{x}_1 + x_2 x_3 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= (\overline{x}_1 + x_2 + \overline{x}_3 + \overline{x}_4) \overline{x}_5 \\
&= \overline{x}_1 \overline{x}_5 + x_2 \overline{x}_5 + \overline{x}_3 \overline{x}_5 + \overline{x}_4 \overline{x}_5
\end{aligned}
$$

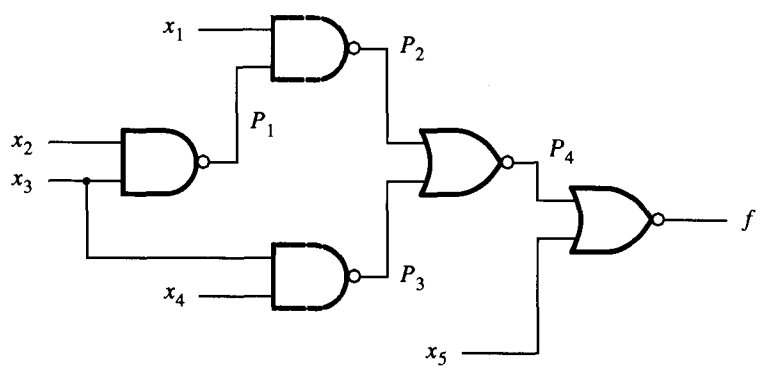


Figure 4.32 Circuit for Example 4.13.

Note that in deriving the second to the last line, we used property 16*a* in section 2.5 to simplify $x_2x_3 + \bar{x}_3$ into $x_2 + \bar{x}_3$.

Analysis of circuits is much simpler than synthesis. With a little practice one can develop an ability to easily analyze even fairly complex circuits.

We have now covered a considerable amount of material on synthesis and analysis of logic functions. We have used the Karnaugh map as a vehicle for illustrating the concepts involved in finding optimal implementations of logic functions. We have also shown that logic functions can be realized in a variety of forms, both with two levels of logic and with multiple levels. In a modern design environment, logic circuits are synthesized using CAD tools, rather than by hand. The concepts that we have discussed in this chapter are quite general; they are representative of the strategies implemented in CAD algorithms. As we have said before, the Karnaugh map scheme for representing logic functions is not appropriate for use in CAD tools. In the next section we discuss an alternative representation of logic functions, which is suitable for use in CAD algorithms.

4.8 CUBICAL REPRESENTATION

The Karnaugh map is an excellent vehicle for illustrating concepts, and it is even useful for manual design if the functions have only a few variables. To deal with larger functions it is necessary to have techniques that are algebraic, rather than graphical, which can be applied to functions of any number of variables.

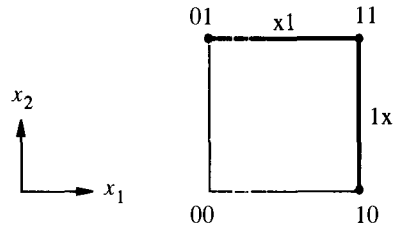
Many algebraic optimization techniques have been developed. We will not pursue these techniques in great detail, but we will attempt to provide the reader with an appreciation of the tasks involved. This helps in gaining an understanding of what the CAD tools can do and what results can be expected from them. The approaches that we will present make use of a cubical representation of logic functions.

4.8.1 CUBES AND HYPERCUBES

So far in this book, we have encountered four different forms for representing logic functions: truth tables, algebraic expressions, Venn diagrams, and Karnaugh maps. Another possibility is to map a function of n variables onto an n-dimensional cube.

Two-Dimensional Cube

A two-dimensional cube is shown in Figure 4.33. The four corners in the cube are called *vertices*, which correspond to the four rows of a truth table. Each vertex is identified by two coordinates. The horizontal coordinate is assumed to correspond to variable x_1, and vertical coordinate to x_2. Thus vertex 00 is the bottom-left corner, which corresponds to row 0 in the truth table. Vertex 01 is the top-left corner, where $x_1 = 0$ and $x_2 = 1$, which corresponds to row 1 in the truth table, and so on for the other two vertices.



x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4.33 Representation of $f(x_1, x_2) = \sum m(1, 2, 3)$.

We will map a function onto the cube by indicating with blue circles those vertices for which $f = 1$. In Figure 4.33 $f = 1$ for vertices 01, 10, and 11. We can express the function as a set of vertices, using the notation $f = \{01, 10, 11\}$. The function f is also shown in the form of a truth table in the figure.

An edge joins two vertices for which the labels differ in the value of only one variable. Therefore, if two vertices for which $f = 1$ are joined by an edge, then this edge represents that portion of the function just as well as the two individual vertices. For example, $f = 1$ for vertices 10 and 11. They are joined by the edge that is labeled 1x. It is customary to use the letter x to denote the fact that the corresponding variable can be either 0 or 1. Hence 1x means that $x_1 = 1$, while x_2 can be either 0 or 1. Similarly, vertices 01 and 11 are joined by the edge labeled x1, indicating that x_1 can be either 0 or 1, but $x_2 = 1$. The reader must not confuse the use of the letter x for this purpose, in contrast to the subscripted use where x_1 and x_2 refer to the variables.

Two vertices being represented by a single edge is the embodiment of the combining property 14*a* from section 2.5. The edge 1x is the logical sum of vertices 10 and 11. It essentially defines the term x_1, which is the sum of minterms $x_1\bar{x}_2$ and x_1x_2. The property 14*a* indicates that

$$x_1\bar{x}_2 + x_1x_2 = x_1$$

Therefore, finding edges for which $f = 1$ is equivalent to applying the combining property. Of course, this is also analogous to finding pairs of adjacent cells in a Karnaugh map for which $f = 1$.

The edges 1x and x1 define fully the function in Figure 4.33; hence we can represent the function as $f = \{1x, x1\}$. This corresponds to the logic expression

$$f = x_1 + x_2$$

which is also obvious from the truth table in the figure.

Three-Dimensional Cube

Figure 4.34 illustrates a three-dimensional cube. The x_1, x_2, and x_3 coordinates are as shown on the left. Each vertex is identified by a specific valuation of the three variables. The function f mapped onto the cube is the function from Figure 4.1, which was used in Figure 4.5*b*. There are five vertices for which $f = 1$, namely, 000, 010, 100, 101, and 110. These vertices are joined by the five edges shown in blue, namely, x00, 0x0, x10, 1x0, and 10x. Because the vertices 000, 010, 100, and 110 include all valuations of x_1 and x_2,

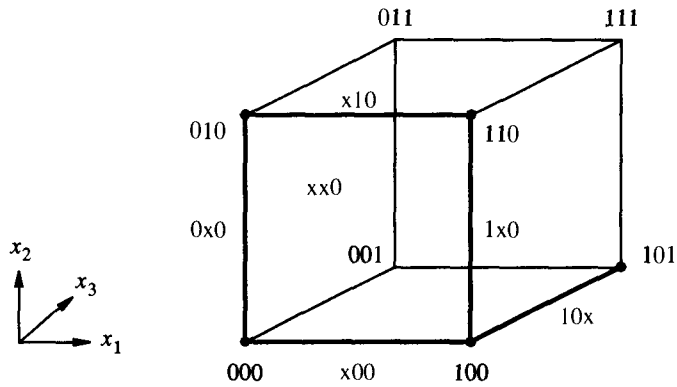


Figure 4.34 Representation of $f(x_1, x_2, x_3) = \sum m(0, 2, 4, 5, 6)$.

when x_3 is 0, they can be specified by the term xx0. This term means that $f = 1$ if $x_3 = 0$, regardless of the values of x_1 and x_2. Notice that xx0 represents the front side of the cube, which is shaded in blue.

From the preceding discussion it is evident that the function f can be represented in several ways. Some of the possibilities are

$$\begin{aligned} f &= \{000, 010, 100, 101, 110\} \\ &= \{0x0, 1x0, 101\} \\ &= \{x00, x10, 101\} \\ &= \{x00, x10, 10x\} \\ &= \{xx0, 10x\} \end{aligned}$$

In a physical realization each of the above terms is a product term implemented by an AND gate. Obviously, the least-expensive circuit is obtained if $f = \{xx0, 10x\}$, which is equivalent to the logic expression

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

This is the expression that we derived using the Karnaugh map in Figure 4.5*b*.

Four-Dimensional Cube

Graphical images of two- and three-dimensional cubes are easy to draw. A four-dimensional cube is more difficult. It consists of 2 three-dimensional cubes with their corners connected. The simplest way to visualize a four-dimensional cube is to have one cube placed inside the other cube, as depicted in Figure 4.35. We have assumed that the x_1, x_2, and x_3 coordinates are the same as in Figure 4.34, while $x_4 = 0$ defines the outer cube and $x_4 = 1$ defines the inner cube. Figure 4.35 indicates how the function f_3 of Figure 4.7 is mapped onto the four-dimensional cube. To avoid cluttering the figure with too many labels, we have labeled only those vertices for which $f_3 = 1$. Again, all edges that connect these vertices are highlighted in blue.

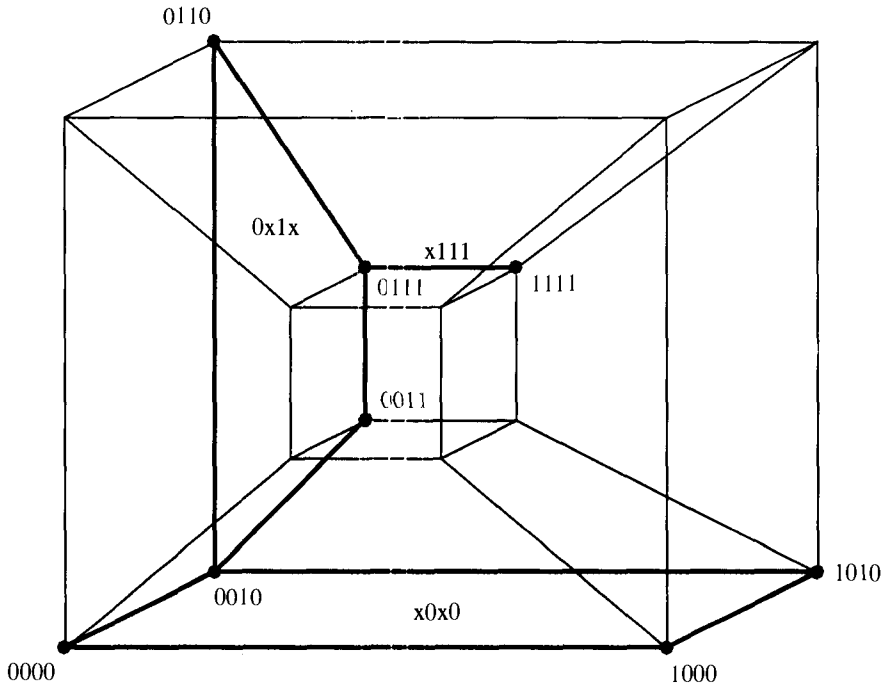


Figure 4.35 Representation of function f_3 from Figure 4.7.

There are two groups of four adjacent vertices for which $f_3 = 1$ that can be represented as planes. The group comprising 0000, 0010, 1000, and 1010 is represented by x0x0. The group 0010, 0011, 0110, and 0111 is represented by 0x1x. These planes are shaded in the figure. The function f_3 can be represented in several ways, for example

$$\begin{aligned} f_3 &= \{0000, 0010, 0011, 0110, 0111, 1000, 1010, 1111\} \\ &= \{00x0, 10x0, 0x10, 0x11, x111\} \\ &= \{x0x0, 0x1x, x111\} \end{aligned}$$

Since each x indicates that the corresponding variable can be ignored, because it can be either 0 or 1, the simplest circuit is obtained if $f = \{x0x0, 0x1x, x111\}$, which is equivalent to the expression

$$f_3 = \bar{x}_2\bar{x}_4 + \bar{x}_1x_3 + x_2x_3x_4$$

We derived the same expression in Figure 4.7.

n-Dimensional Cube

A function that has n variables can be mapped onto an n-dimensional cube. Although it is impractical to draw graphical images of cubes that have more than four variables, it is not difficult to extend the ideas introduced above to a general n-variable case. Because visual interpretation is not possible and because we normally use the word *cube* only for

a three-dimensional structure, many people use the word *hypercube* to refer to structures with more than three dimensions. We will continue to use the word *cube* in our discussion.

It is convenient to refer to a cube as being of a certain *size* that reflects the number of vertices in the cube. Vertices have the smallest size. Each variable has a value of 0 or 1 in a vertex. A cube that has an x in one variable position is larger because it consists of two vertices. For example, the cube 1x01 consists of vertices 1001 and 1101. A cube that has two x's consists of four vertices, and so on. A cube that has k x's consists of 2^k vertices.

An n-dimensional cube has 2^n vertices. Two vertices are adjacent if they differ in the value of only one coordinate. Because there are n coordinates (axes in the n-dimensional cube), each vertex is adjacent to n other vertices. The n-dimensional cube contains cubes of lower dimensionality. Cubes of the lowest dimension are vertices. Because their dimension is zero, we will call them 0-*cubes*. Edges are cubes of dimension 1; hence we will call them 1-*cubes*. A side of a three-dimensional cube is a 2-*cube*. An entire three-dimensional cube is a 3-*cube*, and so on. In general, we will refer to a set of 2^k adjacent vertices as a k-*cube*.

From the examples in Figures 4.34 and 4.35, it is apparent that the largest possible k-*cubes* that exist for a given function are equivalent to its prime implicants. Next, we will discuss minimization techniques that use the cubical representation of functions.

4.9 A TABULAR METHOD FOR MINIMIZATION

Cubical representation of logic functions is well suited for implementation of minimization algorithms that can be programmed and run efficiently on computers. Such algorithms are included in modern CAD tools. While the CAD tools can be used effectively without detailed knowledge of how their minimization algorithms are implemented, the reader may find it interesting to gain some insight into how this may be accomplished. In this section we will describe a relatively simple tabular method, which illustrates the main concepts and indicates some of the problems that arise.

A tabular approach for minimization was proposed in the 1950s by Willard Quine [6] and Edward McCluskey [7]. It became popular under the name *Quine-McCluskey method.* While it is not efficient enough to be used in modern CAD tools, it is a simple method that illustrates the key issues. We will present it using the cubical notation discussed in section 4.8.

4.9.1 GENERATION OF PRIME IMPLICANTS

As mentioned in section 4.8, the prime implicants of a given logic function f are the largest possible k-cubes for which $f = 1$. For incompletely specified functions, which include a set of don't-care vertices, the prime implicants are the largest k-cubes for which either $f = 1$ or f is unspecified.

Assume that the initial specification of f is given in terms of minterms for which $f = 1$. Also, let the don't cares be specified as minterms. This allows us to create a list of vertices for which either $f = 1$ or it is a don't care condition. We can compare these vertices in pairwise fashion to see if they can be combined into larger cubes. Then we can attempt to

combine these new cubes into still larger cubes and continue the process until we find the prime implicants.

The basis of the method is the combining property of Boolean algebra

$$x_i x_j + x_i \bar{x}_j = x_i$$

which we used in section 4.8 to develop the cubical representation. If we have two cubes that are identical in all variables (coordinates) except one, for which one cube has the value 0 and the other has 1, then these cubes can be combined into a larger cube. For example, consider $f(x_1, \ldots, x_4) = \{1000, 1001, 1010, 1011\}$. The cubes 1000 and 1001 differ only in variable x_4; they can be combined into a new cube 100x. Similarly, 1010 and 1011 can be combined into 101x. Then we can combine 100x and 101x into a larger cube 10xx, which means that the function can be expressed simply as $f = x_1\bar{x}_2$.

Figure 4.36 shows how we can generate the prime implicants for the function, f, in Figure 4.11. The function is defined as

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 10, 11, 12, 13, 15)$$

There are no don't care conditions. Since larger cubes can be generated only from the minterms that differ in just one variable, we can reduce the number of pairwise comparisons by placing the minterms into groups such that the cubes in each group have the same number of 1s, and sort the groups by the number of 1s. Thus, it will be necessary to compare each cube in a given group only with all cubes in the immediately preceding group. In Figure 4.36, the minterms are ordered in this way in list 1. (Note that we indicated the decimal equivalents of the minterms as well, to facilitate our discussion.) The minterms, which are also called 0-cubes as explained in section 4.8, can be combined into 1-cubes shown in list 2. To make the entries easily understood we indicated the minterms that are combined to form each 1-cube. Next, we check if the 0-cubes are included in the 1-cubes and place a check mark beside each cube that is included. We now generate 2-cubes from the 1-cubes in list 2. The only 2-cube that can be generated is xx00, which is placed in list 3. Again, the check marks are placed against the 1-cubes that are included in the 2-cube. Since there exists just

List 1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
10	1 0 1 0	✓
12	1 1 0 0	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

List 2

0,4	0 x 0 0	✓
0,8	x 0 0 0	✓
8,10	1 0 x 0	
4,12	x 1 0 0	✓
8,12	1 x 0 0	✓
10,11	1 0 1 x	
12,13	1 1 0 x	
11,15	1 x 1 1	
13,15	1 1 x 1	

List 3

0,4,8,12	x x 0 0

Figure 4.36 Generation of prime implicants for the function in Figure 4.11.

one 2-cube, there can be no 3-cubes for this function. The cubes in each list without a check mark are the prime implicants of f. Therefore, the set, P, of prime implicants is

$$P = \{10x0, 101x, 110x, 1x11, 11x1, xx00\}$$
$$= \{p_1, p_2, p_3, p_4, p_5, p_6\}$$

4.9.2 DETERMINATION OF A MINIMUM COVER

Having generated the set of all prime implicants, it is necessary to choose a minimum-cost subset that covers all minterms for which $f = 1$. As a simple measure we will assume that the cost is directly proportional to the number of inputs to all gates, which means to the number of literals in the prime implicants chosen to implement the function.

To find a minimum-cost cover, we construct a *prime implicant cover table* in which there is a row for each prime implicant and a column for each minterm that must be covered. Then we place check marks to indicate the minterms covered by each prime implicant. Figure 4.37*a* shows the table for the prime implicants derived in Figure 4.36. If there is a single check mark in some column of the cover table, then the prime implicant that covers the minterm of this column is *essential* and it must be included in the final cover. Such is the case with p_6, which is the only prime implicant that covers minterms 0 and 4. The next step is to remove the row(s) corresponding to the essential prime implicants and the column(s) covered by them. Hence we remove p_6 and columns 0, 4, 8, and 12, which leads to the table in Figure 4.37*b*.

Now, we can use the concept of *row dominance* to reduce the cover table. Observe that p_1 covers only minterm 10 while p_2 covers both 10 and 11. We say that p_2 *dominates* p_1. Since the cost of p_2 is the same as the cost of p_1, it is prudent to choose p_2 rather than p_1, so we will remove p_1 from the table. Similarly, p_5 dominates p_3, hence we will remove p_3 from the table. Thus, we obtain the table in Figure 4.37*c*. This table indicates that we must choose p_2 to cover minterm 10 and p_5 to cover minterm 13, which also takes care of covering minterms 11 and 15. Therefore, the final cover is

$$C = \{p_2, p_5, p_6\}$$
$$= \{101x, 11x1, xx00\}$$

which means that the minimum-cost implementation of the function is

$$f = x_1\bar{x}_2x_3 + x_1x_2x_4 + \bar{x}_3\bar{x}_4$$

This is the same expression as the one derived in section 4.2.2.

In this example we used the concept of row dominance to reduce the cover table. We removed the dominated rows because they cover fewer minterms and the cost of their prime implicants is the same as the cost of the prime implicants of the dominating rows. However, a dominated row should not be removed if the cost of its prime implicant is less than the cost of the dominating row's prime implicant. An example of this situation can be found in problem 4.25.

The tabular method can be used with don't care conditions as illustrated in the following example.

Prime implicant	Minterm 0	4	8	10	11	12	13	15
$p_1 = 1\ 0\ x\ 0$			✓	✓				
$p_2 = 1\ 0\ 1\ x$				✓	✓			
$p_3 = 1\ 1\ 0\ x$						✓	✓	
$p_4 = 1\ x\ 1\ 1$					✓			✓
$p_5 = 1\ 1\ x\ 1$							✓	✓
$p_6 = x\ x\ 0\ 0$	✓	✓	✓			✓		

(a) Initial prime implicant cover table

Prime implicant	Minterm 10	11	13	15
p_1	✓			
p_2	✓	✓		
p_3			✓	
p_4		✓		✓
p_5			✓	✓

(b) After the removal of essential prime implicants

Prime implicant	Minterm 10	11	13	15
p_2	✓	✓		
p_4		✓		✓
p_5			✓	✓

(c) After the removal of dominated rows

Figure 4.37 Selection of a cover for the function in Figure 4.11.

Example 4.14 The don't care minterms are included in the initial list in the same way as the minterms for which $f = 1$. Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 5, 6, 7, 8, 9, 13) + D(1, 12, 15)$$

We encourage the reader to derive a Karnaugh map for this function as an aid in visual-

izing the derivation that follows. Figure 4.38 depicts the generation of prime implicants, producing the result

$$P = \{00x0, 0x10, 011x, x00x, xx01, 1x0x, x1x1\}$$
$$= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$$

The initial prime implicant cover table is shown in Figure 4.39*a*. The don't care minterms are not included in the table because they do not have to be covered. There are no essential prime implicants. Examining this table, we see that column 8 has check marks in the same rows as column 9. Moreover, column 9 has an additional check mark in row p_5. Hence column 9 dominates column 8. We refer to this as the concept of *column dominance*. When one column dominates another, we can remove the dominating column, which is column 9 in this case. Note that this is in contrast to rows where we remove dominated (rather than dominating) rows. The reason is that when we choose a prime implicant to cover the minterm that corresponds to the dominated column, this prime implicant will also cover the minterm corresponding to the dominating column. In our example, choosing either p_4 or p_6 covers both minterms 8 and 9. Similarly, column 13 dominates column 5, hence column 13 can be deleted.

After removing columns 9 and 13, we obtain the reduced table in Figure 4.39*b*. In this table row p_4 dominates p_6 and row p_7 dominates p_5. This means that p_5 and p_6 can be removed, giving the table in Figure 4.39*c*. Now, p_4 and p_7 are essential to cover minterms 8 and 5, respectively. Thus, the table in Figure 4.39*d* is obtained, from which it is obvious that p_2 covers the remaining minterms 2 and 6. Note that row p_2 dominates both rows p_1 and p_3.

The final cover is

$$C = \{p_2, p_4, p_7\}$$
$$= \{0x10, x00x, x1x1\}$$

List 1

0	0 0 0 0	✓
1	0 0 0 1	✓
2	0 0 1 0	✓
8	1 0 0 0	✓
5	0 1 0 1	✓
6	0 1 1 0	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

List 2

0,1	0 0 0 x	✓
0,2	0 0 x 0	
0,8	x 0 0 0	✓
1,5	0 x 0 1	✓
2,6	0 x 1 0	
1,9	x 0 0 1	✓
8,9	1 0 0 x	✓
8,12	1 x 0 0	✓
5,7	0 1 x 1	✓
6,7	0 1 1 x	
5,13	x 1 0 1	✓
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
7,15	x 1 1 1	✓
13,15	1 1 x 1	✓

List 3

0,1,8,9	x 0 0 x
1,5,9,13	x x 0 1
8,9,12,13	1 x 0 x
5,7,13,15	x 1 x 1

Figure 4.38 Generation of prime implicants for the function in Example 4.14.

Prime implicant	Minterm 0	2	5	6	7	8	9	13
p_1 = 0 0 x 0	✓	✓						
p_2 = 0 x 1 0		✓		✓				
p_3 = 0 1 1 x				✓	✓			
p_4 = x 0 0 x	✓					✓	✓	
p_5 = x x 0 1			✓				✓	✓
p_6 = 1 x 0 x						✓	✓	✓
p_7 = x 1 x 1			✓		✓			✓

(a) Initial prime implicant cover table

Prime implicant	Minterm 0	2	5	6	7	8
p_1 = 0 0 x 0	✓	✓				
p_2 = 0 x 1 0		✓		✓		
p_3 = 0 1 1 x				✓	✓	
p_4 = x 0 0 x	✓					✓
p_5 = x x 0 1			✓			
p_6 = 1 x 0 x						✓
p_7 = x 1 x 1			✓		✓	

(b) After the removal of columns 9 and 13

Prime implicant	Minterm 0	2	5	6	7	8
p_1	✓	✓				
p_2		✓		✓		
p_3				✓	✓	
p_4	✓					✓
p_7			✓		✓	

(c) After the removal of rows p_5 and p_6

Prime implicant	Minterm 2	6
p_1	✓	
p_2	✓	✓
p_3		✓

(d) After including p_4 and p_7 in the cover

Figure 4.39 Selection of a cover for the function in Example 4.14.

and the function is implemented as

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

In Figures 4.37 and 4.39, we used the concept of row and column dominance to reduce the cover table. This is not always possible, as illustrated in the following example.

Example 4.15

Consider the function

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 10, 15) + D(1, 2, 7, 8, 11, 14)$$

The prime implicants for this function are

$$\begin{aligned} P &= \{00xx, x0x0, x01x, xx11, 1x1x\} \\ &= \{p_1, p_2, p_3, p_4, p_5\} \end{aligned}$$

The initial prime implicant cover table is shown in Figure 4.40*a*. There are no essential prime implicants. Also, there are no dominant rows or columns. Moreover, all prime implicants have the same cost because each of them is implemented with two literals. Thus, the table does not provide any clues that can be used to select a minimum-cost cover.

A good practical approach is to use the concept of *branching*, which was introduced in section 4.2.2. We can choose any prime implicant, say p_3, and first choose to include this prime implicant in the final cover. Then we can determine the rest of the final cover in the usual way and compute its cost. Next we try the other possibility by excluding p_3 from the final cover and determine the resulting cost. We compare the costs and choose the less expensive alternative.

Figure 4.40*b* gives the cover table that is left if p_3 is included in the final cover. The table does not include minterms 3 and 10 because they are covered by p_3. The table indicates that a complete cover must include either p_1 or p_2 to cover minterm 0 and either p_4 or p_5 to cover minterm 15. Therefore, a complete cover can be

$$C = \{p_1, p_3, p_4\}$$

The alternative of excluding p_3 leads to the cover table in Figure 4.40*c*. Here, we see that a minimum-cost cover requires only two prime implicants. One possibility is to choose p_1 and p_5. The other possibility is to choose p_2 and p_4. Hence a minimum-cost cover is just

$$\begin{aligned} C_{min} &= \{p_1, p_5\} \\ &= \{00xx, 1x1x\} \end{aligned}$$

The function is realized as

$$f = \bar{x}_1 \bar{x}_2 + x_1 x_3$$

Prime implicant	Minterm 0	3	10	15
$p_1 = 0\ 0\ x\ x$	✓	✓		
$p_2 = x\ 0\ x\ 0$	✓		✓	
$p_3 = x\ 0\ 1\ x$		✓	✓	
$p_4 = x\ x\ 1\ 1$		✓		✓
$p_5 = 1\ x\ 1\ x$			✓	✓

(a) Initial prime implicant cover table

Prime implicant	Minterm 0	15
p_1	✓	
p_2	✓	
p_4		✓
p_5		✓

(b) After including p_3 in the cover

Prime implicant	Minterm 0	3	10	15
p_1	✓	✓		
p_2	✓		✓	
p_4		✓		✓
p_5			✓	✓

(c) After excluding p_3 from the cover

Figure 4.40 Selection of a cover for the function in Example 4.15.

4.9.3 SUMMARY OF THE TABULAR METHOD

The tabular method can be summarized as follows:

1. Starting with a list of cubes that represent the minterms where $f = 1$ or a don't care condition, generate the prime implicants by successive pairwise comparisons of the cubes.

2. Derive a cover table which indicates the minterms where $f = 1$ that are covered by each prime implicant.
3. Include the essential prime implicants (if any) in the final cover and reduce the table by removing both these prime implicants and the covered minterms.
4. Use the concept of row and column dominance to reduce the cover table further. A dominated row is removed only if the cost of its prime implicant is greater than or equal to the cost of the dominating row's prime implicant.
5. Repeat steps 3 and 4 until the cover table is either empty or no further reduction of the table is possible.
6. If the reduced cover table is not empty, then use the branching approach to determine the remaining prime implicants that should be included in a minimum cost cover.

The tabular method illustrates how an algebraic technique can be used to generate the prime implicants. It also shows a simple approach for dealing with the covering problem, to find a minimum-cost cover. The method has some practical limitations. In practice, functions are seldom defined in the form of minterms. They are usually given either in the form of algebraic expressions or as sets of cubes. The need to start the minimization process with a list of minterms means that the expressions or sets have to be expanded into this form. This list may be very large. As larger cubes are generated, there will be numerous comparisons performed and the computation will be slow. Using the cover table to select the optimal set of prime implicants is also computationally intensive when large functions are involved.

Many algebraic techniques have been developed, which aim to reduce the time that it takes to generate the optimal covers. While most of these techniques are beyond the scope of this book, we will briefly discuss one possible approach in the next section. A reader who intends to use the CAD tools, but is not interested in the details of automated minimization, may skip this section without loss of continuity.

4.10 A CUBICAL TECHNIQUE FOR MINIMIZATION

Assume that the initial specification of a function f is given in terms of implicants that are not necessarily either minterms or prime implicants. Then it is convenient to define an operation that will generate other implicants that are not given explicitly in the initial specification, but which will eventually lead to the prime implicants of f. One such possibility is known as the $*$-*product* operation, which is usually pronounced the "star-product" operation. We will refer to it simply as the $*$-*operation*.

$*$-Operation

The $*$-operation provides a simple way of deriving a new cube by combining two cubes that differ in the value of only one variable. Let $A = A_1A_2 \ldots A_n$ and $B = B_1B_2 \ldots B_n$ be two cubes that are implicants of an n-variable function. Thus each coordinate A_i and B_i is specified as having the value 0, 1, or x. There are two distinct steps in the $*$-operation. First, the $*$-operation is evaluated for each pair A_i and B_i, in coordinates $i = 1, 2, \ldots, n$, according to the table in Figure 4.41 Then based on the results of using the table, a set of

A_i \ B_i	0	1	x
0	0	ø	0
1	ø	1	1
x	0	1	x

$A_i * B_i$

Figure 4.41 The coordinate $*$-operation.

rules is applied to determine the overall result of the $*$-operation. The table in Figure 4.41 defines the coordinate $*$-operation, $A_i * B_i$. It specifies the result of $A_i * B_i$ for each possible combination of values of A_i and B_i. This result is the intersection (i.e., the common part) of A and B in this coordinate. Note that when A_i and B_i have the opposite values (0 and 1, or vice versa), the result of the coordinate $*$-operation is indicated by the symbol ø. We say that the intersection of A_i and B_i is empty. Using the table, the complete $*$-operation for A and B is defined as follows:

$$C = A * B, \text{ such that}$$

1. $C = \text{ø}$ *if* $A_i * B_i = \text{ø}$ for more than one i.
2. Otherwise, $C_i = A_i * B_i$ when $A_i * B_i \neq \text{ø}$, and $C_i = \text{x}$ for the coordinate where $A_i * B_i = \text{ø}$.

For example, let $A = \{0\text{x}0\}$ and $B = \{111\}$. Then $A_1 * B_1 = 0 * 1 = \text{ø}$, $A_2 * B_2 = \text{x} * 1 = 1$, and $A_3 * B_3 = 0 * 1 = \text{ø}$. Because the result is ø in two coordinates, it follows from condition 1 that $A * B = \text{ø}$. In other words, these two cubes cannot be combined into another cube, because they differ in two coordinates.

As another example, consider $A = \{11\text{x}\}$ and $B = \{10\text{x}\}$. In this case $A_1 * B_1 = 1 * 1 = 1$, $A_2 * B_2 = 1 * 0 = \text{ø}$, and $A_3 * B_3 = \text{x} * \text{x} = \text{x}$. According to condition 2 above, $C_1 = 1$, $C_2 = \text{x}$, and $C_3 = \text{x}$, which gives $C = A * B = \{1\text{xx}\}$. A larger 2-cube is created from two 1-cubes that differ in one coordinate only.

The result of the $*$-operation may be a smaller cube than the two cubes involved in the operation. Consider $A = \{1\text{x}1\}$ and $B = \{11\text{x}\}$. Then $C = A * B = \{111\}$. Notice that C is included in both A and B, which means that this cube will not be useful in searching for prime implicants. Therefore, it should be discarded by the minimization algorithm.

As a final example, consider $A = \{\text{x}10\}$ and $B = \{0\text{x}1\}$. Then $C = A * B = \{01\text{x}\}$. All three of these cubes are the same size, but C is not included in either A or B. Hence C has to be considered in the search for prime implicants. The reader may find it helpful to draw a Karnaugh map to see how cube C is related to cubes A and B.

Using the $*$-Operation to Find Prime Implicants

The essence of the $*$-operation is to find new cubes from pairs of existing cubes. In particular, it is of interest to find new cubes that are not included in the existing cubes. A procedure for finding the prime implicants may be organized as follows.

Suppose that a function f is specified by means of a set of implicants that are represented as cubes. Let this set be denoted as the cover C^k of f. Let c^i and c^j be any two cubes in C^k. Then apply the $*$-operation to all pairs of cubes in C^k; let G^{k+1} be the set of newly

generated cubes. Hence

$$G^{k+1} = c^i * c^j \text{ for all } c^i, c^j \in C^k$$

Now a new cover for f may be formed by using the cubes in C^k and G^{k+1}. Some of these cubes may be redundant because they are included in other cubes; they should be removed. Let the new cover be

$$C^{k+1} = C^k \cup G^{k+1} - \text{ redundant cubes}$$

where $\cup$ denotes the logical union of two sets, and the minus sign $(-)$ denotes the removal of elements of a set. If $C^{k+1} \neq C^k$, then a new cover C^{k+2} is generated using the same process. If $C^{k+1} = C^k$, then the cubes in the cover are the prime implicants of f. For an n-variable function, it is necessary to repeat the step at most n times.

Redundant cubes that have to be removed are identified through pairwise comparison of cubes. Cube $A = A_1A_2 \ldots A_n$ should be removed if it is included in some cube $B = B_1B_2 \ldots B_n$, which is the case if $A_i = B_i$ or $B_i = \text{x}$ for every coordinate i.

Example 4.16

Consider the function $f(x_1, x_2, x_3)$ of Figure 4.9. Assume that f is initially specified as a set of vertices that correspond to the minterms, m_0, m_1, m_2, m_3, and m_7. Hence let the initial cover be $C^0 = \{000, 001, 010, 011, 111\}$. Using the $*$-operation to generate a new set of cubes, we obtain $G^1 = \{00\text{x}, 0\text{x}0, 0\text{x}1, 01\text{x}, \text{x}11\}$. Then $C^1 = C^0 \cup G^1$ – redundant cubes. Observe that each cube in C^0 is included in one of the cubes in G^1; therefore, all cubes in C^0 are redundant. Thus $C^1 = G^1$.

The next step is to apply the $*$-operation to the cubes in C^1, which yields $G^2 = \{000, 001, 0\text{xx}, 0\text{x}1, 010, 01\text{x}, 011\}$. Note that all of these cubes are included in the cube 0xx; therefore, all but 0xx are redundant. Now it is easy to see that

$$\begin{aligned} C^2 &= C^1 \cup G^2 - \text{redundant terms} \\ &= \{\text{x}11, 0\text{xx}\} \end{aligned}$$

since all cubes of C^1, except x11, are redundant because they are covered by 0xx.

Applying the $*$-operation to C^2 yields $G^3 = \{011\}$ and

$$\begin{aligned} C^3 &= C^2 \cup G^3 - \text{redundant terms} \\ &= \{\text{x}11, 0\text{xx}\} \end{aligned}$$

Since $C^3 = C^2$, the conclusion is that the prime implicants of f are the cubes $\{\text{x}11, 0\text{xx}\}$, which represent the product terms x_2x_3 and $\bar{x}_1$. This is the same set of prime implicants that we derived using a Karnaugh map in Figure 4.9.

Observe that the derivation of prime implicants in this example is similar to the tabular method explained in section 4.9 because the starting point was a function, f, given as a set of minterms.

Example 4.17

As another example, consider the four-variable function of Figure 4.10. Assume that this function is initially specified as the cover $C^0 = \{0101, 1101, 1110, 011\text{x}, \text{x}01\text{x}\}$. Then successive applications of the $*$-operation and removing the redundant terms gives

$$C^1 = \{x01x, x10l, 01x1, x110, 1x10, 0x1x\}$$
$$C^2 = \{x01x, x10l, 01x1, 0x1x, xx10\}$$
$$C^3 = C^2$$

Therefore, the prime implicants are $\bar{x}_2x_3$, $x_2\bar{x}_3x_4$, $\bar{x}_1x_2x_4$, $\bar{x}_1x_3$, and $x_3\bar{x}_4$.

4.10.1 DETERMINATION OF ESSENTIAL PRIME IMPLICANTS

From a cover that consists of all prime implicants, it is necessary to extract a minimal cover. As we saw in section 4.2.2, all *essential* prime implicants must be included in the minimal cover. To find the essential prime implicants, it is useful to define an operation that determines a part of a cube (implicant) that is *not* covered by another cube. One such operation is called the *#-operation* (pronounced the "sharp operation"), which is defined as follows.

#-Operation

Again, let $A = A_1A_2 \cdots A_n$ and $B = B_1B_2 \cdots B_n$ be two cubes (implicants) of an n-variable function. The sharp operation $A\#B$ leaves as a result "that part of A that is not covered by B." Similar to the $*$-operation, the #-operation has two steps: $A_i\#B_i$ is evaluated for each coordinate i, and then a set of rules is applied to determine the overall result. The sharp operation for each coordinate is defined in Figure 4.42. After this operation is performed for all pairs (A_i, B_i), the complete #-operation is defined as follows:

$$C = A\#B, \text{ such that}$$

1. $C = A$ if $A_i\#B_i = \emptyset$ for some i.
2. $C = \emptyset$ if $A_i\#B_i = \varepsilon$ for all i.
3. Otherwise, $C = \bigcup_i (A_1, A_2, \ldots, \bar{B}_i, \ldots, A_n)$, where the union is for all i for which $A_i = x$ and $B_i \neq x$.

The first condition corresponds to the case where cubes A and B do not intersect at all; namely, A and B differ in the value of at least one variable, which means that no part of A is covered by B. For example, let $A = 0x1$ and $B = 11x$. The coordinate #-products are $A_1\#B_1 = \emptyset$, $A_2\#B_2 = 0$, and $A_3\#B_3 = \varepsilon$. Then from rule 1 it follows that $0x1 \# 11x = 0x1$.

A_i \ B_i	0	1	x
0	ε	∅	ε
1	∅	ε	ε
x	1	0	ε

$A_i \# B_i$

Figure 4.42 The coordinate #-operation.

The second condition reflects the case where A is fully covered by B. For example, 0x1 # 0xx = ∅. The third condition is for the case where only a part of A is covered by B. In this case the #-operation generates one or more cubes. Specifically, it generates one cube for each coordinate i that is x in A_i, but is not x in B_i. Each cube generated is identical to A, except that A_i is replaced by $\overline{B}_i$. For example, 0xx # 01x = 00x, and 0xx # 010 = {00x, 0x1}.

We will now show how the #-operation can be used to find the essential prime implicants. Let P be the set of all prime implicants of a given function f. Let p^i denote one prime implicant in the set P and let DC denote the don't-care vertices for f. (We use superscripts to refer to different prime implicants in this section because we are using subscripts to refer to coordinate positions in cubes.) Then p^i is an essential prime implicant if and only if

$$p^i \# (P - p^i) \# DC \neq \emptyset$$

This means that p^i is essential if there exists at least one vertex for which $f = 1$ that is covered by p^i, but not by any other prime implicant. The #-operation is also performed with the set of don't-care cubes because vertices in p^i that correspond to don't-care conditions are not essential to cover. The meaning of $p^i \# (P - p^i)$ is that the #-operation is applied successively to each prime implicant in P. For example, consider $P = \{p^1, p^2, p^3, p^4\}$ and $DC = \{d^1, d^2\}$. To check whether p^3 is essential, we evaluate

$$((((p^3 \# p^1) \# p^2) \# p^4) \# d^1) \# d^2$$

If the result of this expression is not ∅, then p^3 is essential.

Example 4.18

In Example 4.16 we determined that the cubes x11 and 0xx are the prime implicants of the function f in Figure 4.9. We can discover whether each of these prime implicants is essential as follows

$$\text{x11 \# 0xx} = 111 \neq \emptyset$$
$$\text{0xx \# x11} = \{00\text{x}, 0\text{x}0\} \neq \emptyset$$

The cube x11 is essential because it is the only prime implicant that covers the vertex 111, for which $f = 1$. The prime implicant 0xx is essential because it is the only one that covers the vertices 000, 001, and 010. This can be seen in the Karnaugh map in Figure 4.9.

Example 4.19

In Example 4.17 we found that the prime implicants of the function in Figure 4.10 are $P =$ {x01x, x101, 01x1, 0x1x, xx10}. Because this function has no don't cares, we compute

$$\text{x01x \# } (P - \text{x01x}) = 1011 \neq \emptyset$$

This is computed in the following steps: x01x # x101 = x01x, then x01x # 01x1 = x01x, then x01x # 0x1x = 101x, and finally 101x # xx10 = 1011. Similarly, we obtain

$$x101 \# (P - x101) = 1101 \neq \emptyset$$
$$01x1 \# (P - 01x1) = \emptyset$$
$$0x1x \# (P - 0x1x) = \emptyset$$
$$xx10 \# (P - xx10) = 1110 \neq \emptyset$$

Therefore, the essential prime implicants are x01x, x101, and xx10 because they are the only ones that cover the vertices 1011, 1101, and 1110, respectively. This is obvious from the Karnaugh map in Figure 4.10.

When checking whether a cube A is essential, the #-operation with one of the cubes in $P - A$ may generate multiple cubes. If so, then each of these cubes has to be checked using the #-operation with all of the remaining cubes in $P - A$.

4.10.2 COMPLETE PROCEDURE FOR FINDING A MINIMAL COVER

Having introduced the $*$- and #-operations, we can now outline a complete procedure for finding a minimal cover for any n-variable function. Assume that the function f is specified in terms of vertices for which $f = 1$; these vertices are often referred to as the *ON-set* of the function. Also, assume that the don't-care conditions are specified as a *DC-set*. Then the initial cover for f is a union of the ON and DC sets.

Prime implicants of f can be generated using the $*$-operation, as explained in section 4.10. Then the #-operation can be used to find the essential prime implicants as presented in section 4.10.1. If the essential prime implicants cover the entire ON-set, then they form the minimum-cost cover for f. Otherwise, it is necessary to include other prime implicants until all vertices in the ON-set are covered.

A nonessential prime implicant p^i should be deleted if there exists a less-expensive prime implicant p^j that covers all vertices of the ON-set that are covered by p^i. If the remaining nonessential prime implicants have the same cost, then a possible heuristic approach is to arbitrarily select one of them, include it in the cover, and determine the rest of the cover. Then an alternative cover is generated by excluding this prime implicant, and the lower-cost cover is chosen for implementation. We already used this approach, which is often referred to as the *branching* heuristic, in sections 4.2.2 and 4.9.2.

The preceding discussion can be summarized in the form of the following minimization procedure:

1. Let $C^0 = ON \cup DC$ be the initial cover of function f and its don't-care conditions.
2. Find all prime implicants of C^0 using the $*$-operation; let P be this set of prime implicants.
3. Find the essential prime implicants using the #-operation. A prime implicant p^i is essential if $p^i \# (P - p^i) \# DC \neq \emptyset$. If the essential prime implicants cover all vertices of the ON-set, then these implicants form the minimum-cost cover.
4. Delete any nonessential p^i that is more expensive (i.e., a smaller cube) than some other prime implicant p^j if $p^i \# DC \# p^j = \emptyset$.

5. Choose the lowest-cost prime implicants to cover the remaining vertices of the ON-set. Use the branching heuristic on the prime implicants of equal cost and retain the cover with the lowest cost.

Example 4.20

To illustrate the minimization procedure, we will use the function

$$f(x_1, x_2, x_3, x_4, x_5) = \sum m(0, 1, 4, 8, 13, 15, 20, 21, 23, 26, 31) + D(5, 10, 24, 28)$$

To help the reader follow the discussion, this function is also shown in the form of a Karnaugh map in Figure 4.43.

Instead of f being specified in terms of minterms, let us assume that f is given as the following SOP expression

$$f = \bar{x}_1\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4x_5 + \bar{x}_1x_2x_3x_5 + x_1\bar{x}_2x_3x_5 + x_1x_3x_4x_5 + \bar{x}_2x_3\bar{x}_4\bar{x}_5$$

Also, we will assume that don't cares are specified using the expression

$$\text{DC} = x_1x_2\bar{x}_4\bar{x}_5 + \bar{x}_1x_2\bar{x}_3x_4\bar{x}_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4x_5$$

Thus, the ON-set expressed as cubes is

$$\text{ON} = \{0\text{x}000, 11010, 00001, 011\text{x}1, 101\text{x}1, 1\text{x}111, \text{x}0100\}$$

and the don't care set is

$$\text{DC} = \{11\text{x}00, 01010, 00101\}$$

The initial cover C^0 consists of the ON-set and the DC-set:

$$C^0 = \{0\text{x}000, 11010, 00001, 011\text{x}1, 101\text{x}1, 1\text{x}111, \text{x}0100, 11\text{x}00, 01010, 00101\}$$

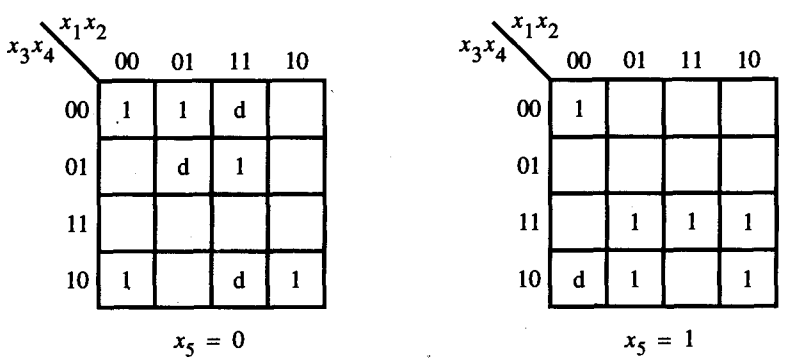


Figure 4.43 The function for Example 4.20.

Using the $*$-operation, the subsequent covers obtained are

$C^1 = \{0x000, 011x1, 101x1, 1x111, x0100, 11x00, 0000x, 00x00, x1000, 010x0, 110x0, x1010, 00x01, x1111, 0x101, 1010x, x0101, 1x100, 0010x\}$

$C^2 = \{0x000, 011x1, 101x1, 1x111, 11x00, x1111, 0x101, 1x100, x010x, 00x0x, x10x0\}$

$C^3 = C^2$

Therefore, $P = C^2$.

Using the #-operation, we find that there are two essential prime implicants: 00x0x (because it is the only one that covers the vertex 00001) and x10x0 (because it is the only one that covers the vertex 11010). The minterms of f covered by these two prime implicants are $m(0, 1, 4, 8, 26)$.

Next, we find that 1x100 can be deleted because the only ON-set vertex that it covers is 10100 (m_{20}), which is also covered by x010x and the cost of this prime implicant is lower. Note that having removed 1x100, the prime implicant x010x becomes essential because none of the other remaining prime implicants covers the vertex 10100. Therefore, x010x has to be included in the final cover. It covers $m(20, 21)$.

There remains to find prime implicants to cover $m(13, 15, 23, 31)$. Using the branching heuristic, the lowest-cost cover is obtained by including the prime implicants 011x1 and 1x111. Thus the final cover is

$$C_{minimum} = \{00x0x, x10x0, x010x, 011x1, 1x111\}$$

The corresponding sum-of-products expression is

$$f = \bar{x}_1\bar{x}_2\bar{x}_4 + x_2\bar{x}_3\bar{x}_5 + \bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_2x_3x_5 + x_1x_3x_4x_5$$

Although this procedure is tedious when performed by hand, it is not difficult to write a computer program to implement the algorithm automatically. The reader should check the validity of our solution by finding the optimal realization from the Karnaugh map in Figure 4.43.

4.11 PRACTICAL CONSIDERATIONS

The purpose of the preceding section was to give the reader some idea about how minimization of logic functions may be automated for use in CAD tools. We chose a scheme that is not too difficult to explain. From the practical point of view, this scheme has some drawbacks. The main difficulty is that the number of cubes that must be considered in the process can be extremely large.

If the goal of minimization is relaxed so that it is not imperative to find a minimum-cost implementation, then it is possible to derive heuristic techniques that produce good results in reasonable time. A technique of this type forms the basis of the widely used Espresso program, which is available from the University of California at Berkeley via the World Wide Web. Espresso is a two-level optimization program. Both input to the program and its output are specified in the format of cubes. Instead of using the $*$-operation to find the

prime implicants, Espresso uses an implicant-expansion technique. (See problem 4.30 for an illustration of the expansion of implicants.) A comprehensive explanation of Espresso is given in [19], while simplified outlines can be found in [3, 12].

The University of California at Berkeley also provides two software programs that can be used for design of multilevel circuits, called MIS [20] and SIS [21]. They allow a user to apply various multilevel optimization techniques to a logic circuit. The user can experiment with different optimization strategies by applying techniques such as factoring and decomposition to all or part of a circuit. SIS also includes the Espresso algorithm for two-level minimization of functions, as well as many other optimization techniques.

Numerous commercial CAD systems are on the market. Four companies whose products are widely used are Cadence Design Systems, Mentor Graphics, Synopsys, and Synplicity. Information on their products is available on the World Wide Web. Each company provides logic synthesis software that can be used to target various types of chips, such as PLDs, gate arrays, standard cells, and custom chips. Because there are many possible ways to synthesize a given circuit, as we saw in the previous sections, each commercial product uses a proprietary logic optimization strategy based on heuristics.

To describe CAD tools, some new terminology has been invented. In particular, we should mention two terms that are widely used in industry: *technology-independent logic synthesis* and *technology mapping*. The first term refers to techniques that are applied when optimizing a circuit without considering the resources available in the target chip. Most of the techniques presented in this chapter are of this type. The second term, technology mapping, refers to techniques that are used to ensure that the circuit produced by logic synthesis can be realized using the logic resources available in the target chip. A good example of technology mapping is the transformation from a circuit in the form of logic operations such as AND and OR into a circuit that consists of only NAND operations. This type of technology mapping is done when targeting a circuit to a gate array that contains only NAND gates. Another example is the translation from logic operations to lookup tables, which is done when targeting a design to an FPGA. It should be noted that the terminology is sometimes used inconsistently. For instance, some CAD systems consider factoring, which was discussed in section 4.6.1, to be technology independent, whereas other systems consider it to be a part of the technology mapping. Still other systems, such as MAX+plusII, do not use these two terms at all, even though they clearly implement both types of techniques. We will not rely on these terms in this book and have mentioned them only for completeness.

The next section provides a more detailed discussion of CAD tools. To give an example of the features provided in these tools, we use the MAX+plusII system that accompanies the book. Of course, different CAD systems offer different features. MAX+plusII synthesizes designs for implementation in PLDs. It includes all the optimization techniques introduced in this chapter.

4.12 CAD Tools

In section 2.9 we introduced the concept of a CAD system and described CAD tools for performing design entry, initial synthesis, and functional simulation. In this section we

introduce the remaining tools in a typical CAD system, which are used for performing logic synthesis and optimization, physical design, and timing simulation. The principles behind such tools are quite general; the details may vary from one system to another. We will discuss the main aspects of the tools in as general a fashion as possible. However, to provide a sufficient degree of reality, we will use illustrative examples based on the Altera MAX+plusII system that is provided with the book. To fully grasp the concepts presented in the following discussion, the reader should go through the material in Tutorials 1 and 2, which are presented in Appendices B and C.

A typical CAD system comprises tools for performing the following tasks:

- *Design entry* allows the designer to enter a description of the desired circuit in the form of truth tables, schematic diagrams, or HDL code.
- *Initial synthesis* generates an initial circuit, based on data entered during the design entry stage.
- *Functional simulation* is used to verify the functionality of the circuit, based on inputs provided by the designer.
- *Logic synthesis and optimization* applies optimization techniques to derive an optimized circuit.
- *Physical design* determines how to implement the optimized circuit in a given target technology, for example, in a PLD chip.
- *Timing simulation* determines the propagation delays that are expected in the implemented circuit.
- *Chip configuration* configures the actual chip to realize the designed circuit.

The first three of these tools are discussed in Chapter 2. The rest are described below.

4.12.1 LOGIC SYNTHESIS AND OPTIMIZATION

The optimization techniques described in this chapter are automatically applied by CAD tools when synthesizing logic circuits. Consider the Verilog code in Figure 4.44. It describes the function f from Figure 4.5*a* in the canonical form, which consists of minterms. We used CAD tools to synthesize f for implementation in a chip. The result obtained was

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

which is the same minimal sum-of-products expression derived in Figure 4.5*a*. This result was displayed in a *report file*, which is produced by the CAD system. The report file includes a set of logic equations that describe the synthesized circuit.

CAD tools often include many optional features that can be invoked by the user. Figure 4.45 shows some of the logic synthesis options provided by MAX+plusII. Although the reader may not recognize all the options shown, the meaning of terms such as minimization, multilevel synthesis, factoring, and decomposition should be obvious at this point. Detailed explanation of various synthesis procedures can be found in specialized texts [5, 22].

The optimized circuit produced by the logic synthesis tools depends both on the type of logic resources available in the target chip and on the particular CAD system that is used. For example, if the target chip is a CPLD, then each logic function in the circuit is expressed in terms of the gates available in a macrocell. For an FPGA that contains lookup tables

```
module func1 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;

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

endmodule
```

Figure 4.44 Verilog code for the function in Figure 4.5*a*.

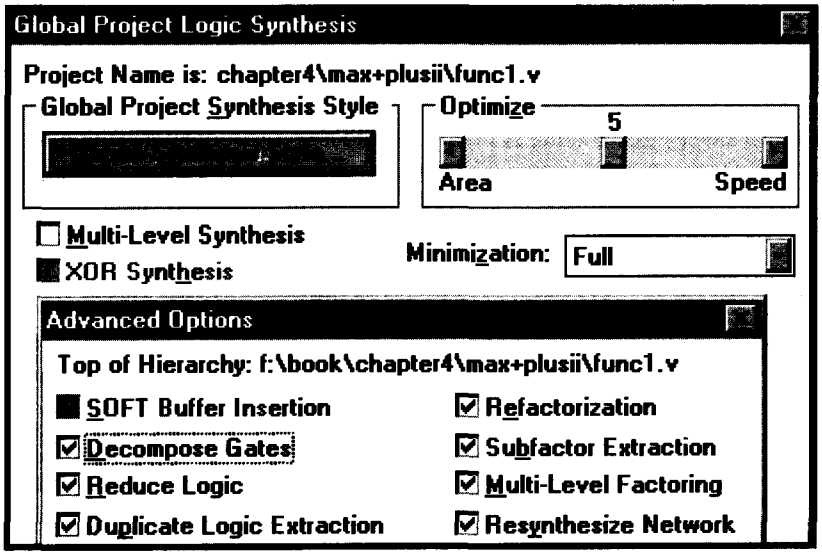


Figure 4.45 Logic synthesis options in MAX+plusII.

(LUTs), the number of inputs to each logic function in the circuit is constrained by the size of the LUTs. If the target chip is a gate array, then the logic functions in the optimized circuit are expressed using only the type of logic cells available in the gate array. Finally, if standard-cell technology is used, then the circuit comprises whatever types of logic cells can be fabricated on the standard-cell chip.

4.12.2 PHYSICAL DESIGN

After logic synthesis the next step in the design flow is to determine exactly how to implement the circuit in the target technology. This step is usually called *physical design*, or *layout synthesis*. There are two main parts to physical design: placement and routing.

A *placement* CAD tool determines where in the target device each logic function in the optimized circuit will be realized. The placement task is highly dependent on the

implementation technology. For example, if a PLD is used for implementation, then the structure of the chip is predefined and the placement tool determines which logic resources in the chip are to be used to realize each logic function in the circuit. In the case of a CPLD, the logic functions are assigned to macrocells. For an FPGA each logic function is assigned to a logic cell.

Continuing with our example, our placement tool realizes the function f from Figure 4.44 in an FPGA as depicted in Figure 4.46. The figure represents a screen capture of a *Floorplan Editor*, which displays the results generated by the physical design tools. The small squares in the diagram represent the logic cells in the FPGA, which are four-input LUTs (see Appendix E). The logic cell at the top left is used to realize the function f. At the bottom of the window, the Floorplan Editor shows the logic expression contained in the LUT for f. Lines are drawn to indicate the input and output connections of this logic cell. They connect to the I/O cells that are used for inputs x_1, x_2, and x_3, as well as for the output f.

After the placement has been completed, the next step is to decide which of the wires in the chip are to be used to realize the required interconnections. This step is called *routing*. Like the placement task, routing is highly dependent on the implementation technology. For a CPLD the programming switches attached to the interconnection wires must be set to connect the macrocells together as needed for the implemented circuit. Similarly, for an FPGA the programming switches are used to connect the logic cells together. If the

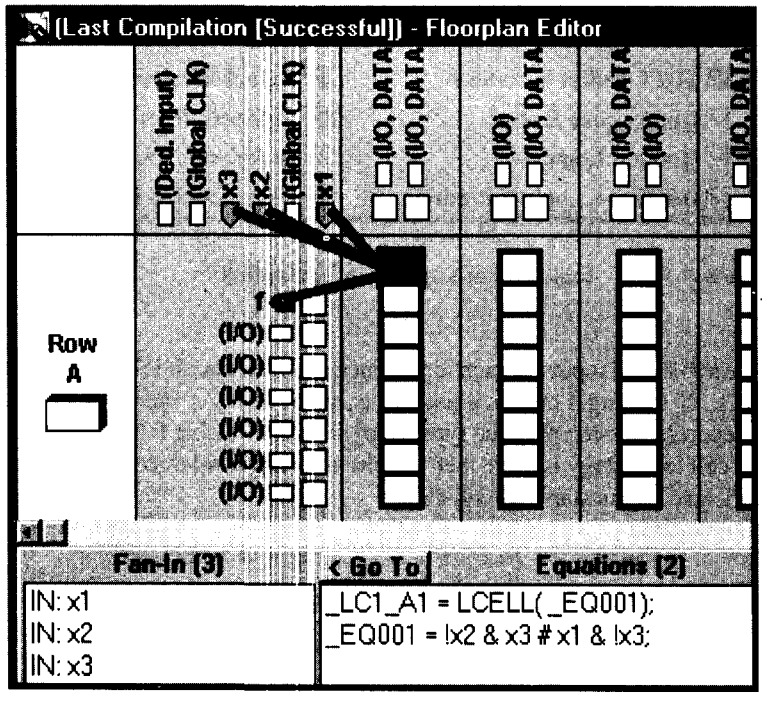


Figure 4.46 The results of physical design for the Verilog code in Figure 4.44.

implementation technology is a gate array or a standard-cell chip, then the routing tool specifies the interconnection wires that are to be fabricated between the rows of logic cells. Some small examples of routing were presented in Chapter 3, in Figures 3.59 and 3.67.

Both the placement and routing tasks can be difficult problems to solve for the CAD tools, especially for the larger devices, such as FPGAs, gate arrays, and standard-cell chips. Much research effort has gone into the development of algorithms for these tasks. Detailed explanations of these algorithms can be found in more specialized books [23, 24].

4.12.3 TIMING SIMULATION

In section 2.9.3 we described functional simulation and said that it is used to ensure that a logic circuit description entered into a CAD system functions as expected by the designer. In functional simulation it is assumed that signal propagation delays through logic gates are negligible. In this section we consider *timing simulation*, which simulates the actual propagation delays in the technology chosen for implementation.

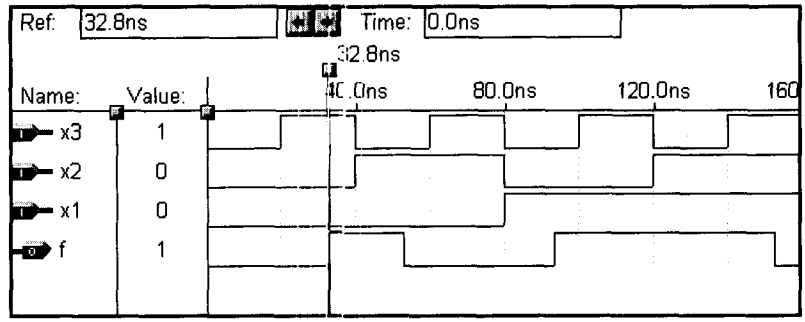
After the physical design tasks are completed, the CAD system has determined exactly how the designed circuit is to be realized in the target technology. It is then possible for the CAD tools to create a model of the circuit that includes all timing aspects of the target chip. The model represents the delays associated with the logic resources in the chip (macrocells or logic cells) and with the interconnection wires.

The results of timing simulation for the function f from Figure 4.44 are shown in Figure 4.47. They were obtained using a timing simulator. The simulator allows the designer to specify a waveform for each of the inputs x_1, x_2, and x_3, and the tool generates the corresponding waveform produced at the output f. Part (*a*) of the figure gives the timing expected when the circuit is implemented in an FPGA. Observe that a heavy vertical line, which is called the *reference line*, is set at the point where f first makes a transition from 0 to 1. The simulator specifies in the box labeled Ref that the reference line is set at 32.8 ns from the start time of the simulation. The change in $x_1x_2x_3$ from 000 to 001 takes place at 20 ns; hence $32.8 - 20 = 12.8$ ns are required for the change in inputs to cause f to change to 1. The reason for the delay at f is that the signals must propagate through the transistor circuits in the FPGA. The timing aspects of transistor circuits are discussed in Chapter 3.

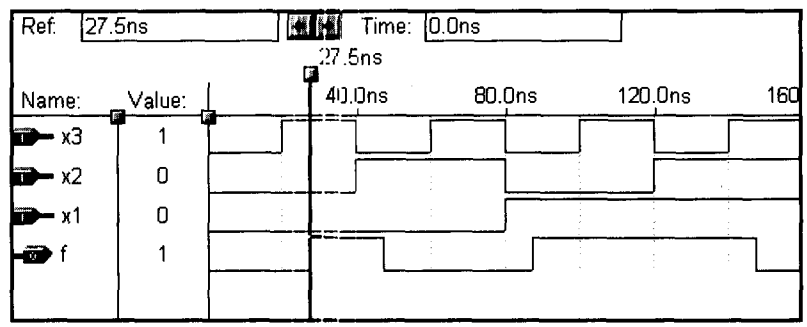
Figure 4.47*b* shows the same simulation for the circuit when it is implemented in a CPLD. Of course, the circuit implements the same function as when implemented in the FPGA, but the timing is different. In the CPLD, f changes 7.5 ns after the inputs change. The speed of a circuit may vary considerably when implemented in different types of chips. Although our example suggests that the CPLD provides much faster speed than the FPGA, the difference is exaggerated because of the small size of the circuit. In general, when larger circuits are implemented, CPLDs and FPGAs provide similar speeds.

4.12.4 SUMMARY OF DESIGN FLOW

Figure 4.48 summarizes the design flow of a complete CAD system. After initial synthesis the logic synthesis tool automatically optimizes the circuit being designed. The physical



(a) Timing in an FPGA



(b) Timing in a CPLD

Figure 4.47 Timing simulation for the Verilog code in Figure 4.44.

design tool then determines exactly how to implement the circuit in the chosen technology. Timing simulation ensures that the implemented circuit meets the required performance. Note that if functional correctness has already been ascertained using functional simulation, as discussed in section 2.9, then the functionality of the circuit need not be verified using timing simulation. However, if functional simulation was not done, then timing simulation can be used to check for proper functionality as well. If timing or functional problems are discovered, they are corrected by returning to the previous steps in the design flow. For functional errors it is necessary to revisit the design entry step. For timing errors it may be possible to correct the problems by using the logic synthesis tool. For example, the window displayed in Figure 4.45 shows a sliding bar that can be used to change the emphasis of the logic synthesis algorithms between circuit cost or circuit speed. Cost is optimized by minimizing the amount of area needed on the chip to implement the circuit. Speed is optimized by minimizing the propagation delay of signals in the circuit. It may

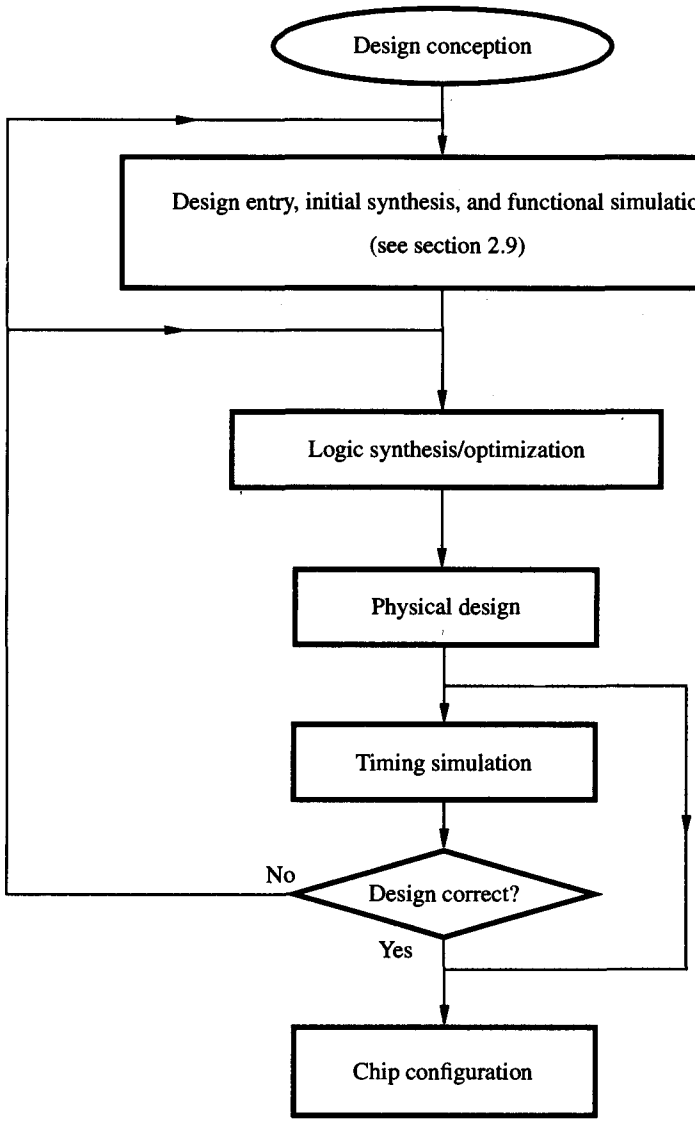


Figure 4.48 A complete CAD system.

also be possible to use a faster speed grade of the selected chip or to select a different type of chip that results in a faster circuit, as in the example from Figure 4.47. If the logic synthesis tool cannot resolve the timing problems, then it is necessary to return to the beginning of the design flow to consider other design alternatives. The final step is to configure the target chip to implement the desired circuit.

4.12.5 EXAMPLES OF CIRCUITS SYNTHESIZED FROM VERILOG CODE

Section 2.10 shows how simple Verilog programs can be written to describe logic functions. This section provides further examples of circuits designed using Verilog code. We show the results of synthesizing the code for implementation in two different types of chips, a CPLD and an FPGA.

Example 4.21 Consider the Verilog code in Figure 4.49. The logic expression for f corresponds to the truth table in Figure 4.1. We derived the minimal sum-of-products form, $f = \bar{x}_3 + x_1\bar{x}_2$, using the Karnaugh map in Figure 4.5*b*. If we compile the Verilog code for implementation in a CPLD, the CAD tools produce the expression

$$f = \bar{x}_3 + x_1\bar{x}_2x_3$$

It is easy to show that this expression is not fully minimized. Using the identity 16*a* in section 2.5, the expression can be reduced to $f = \bar{x}_3 + x_1\bar{x}_2$, which is the minimal form that we derived manually. However, because the circuit is being implemented in a CPLD, the extra literal in the product term $x_1\bar{x}_2x_3$ does not increase the cost. Figure 4.50 shows the expression for f realized in a macrocell. Observe that since the XOR gate in the macrocell is not used for the circuit, one input to the XOR gate is connected to 0.

As we have said before, CAD tools include many options that can affect the results of the synthesis procedure. Some of the options available in MAX+plusII are shown in the window in Figure 4.45. One of the options is called *XOR synthesis*, which is a synthesis technique that attempts to use XOR gates as judiciously as possible. If this option is turned on and the Verilog code in Figure 4.49 is synthesized again, the resulting expression for f becomes

$$f = \bar{x}_3 \oplus x_1\bar{x}_2x_3$$

The reader should verify that this is functionally equivalent to the sum-of-products form given above. The implementation of this expression in a macrocell is depicted in Figure 4.51. The XOR gate is now used as part of the function, with one input connected to $\bar{x}_3$. Since it occupies a single macrocell, the cost of the implementation is the same as for the circuit in Figure 4.50. Although not true in this example, for some logic functions the XOR gates lead to greatly reduced cost. We should note that it is even possible to realize any

```
module example4_21 (x1, x2, x3, f);
  input x1, x2, x3;
  output f;

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

endmodule
```

Figure 4.49 Verilog code for the function in Figure 4.1.

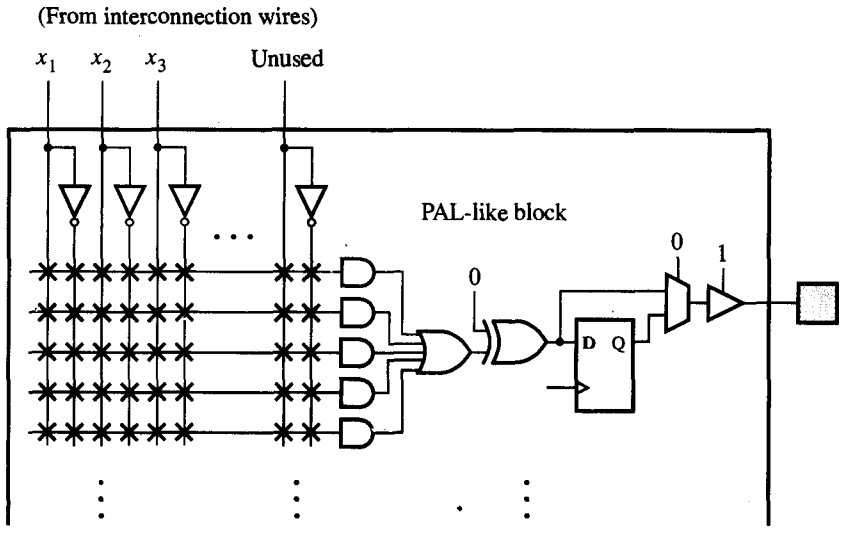


Figure 4.50 Implementation of the Verilog code in Figure 4.49.

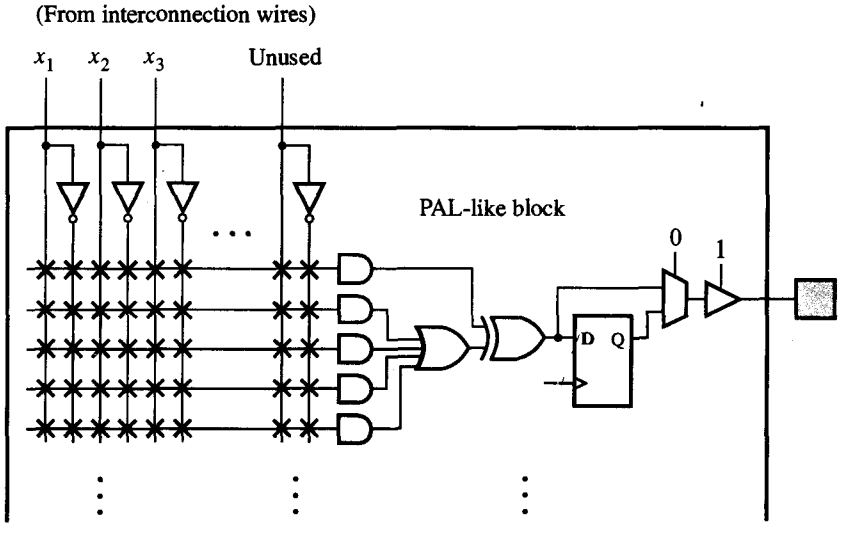


Figure 4.51 Implementation of the Verilog code in Figure 4.49 using XOR synthesis.

arbitrary logic function using only AND and XOR gates [4]. We discuss some typical uses of XOR gates in Chapter 5. As this example illustrates, for any given logic function, several different implementations often have the same cost in a given chip.

Figure 4.52 gives the results of synthesizing the Verilog code in Figure 4.49 into an FPGA. In this case the compiler generates the same sum-of-products form that we derived

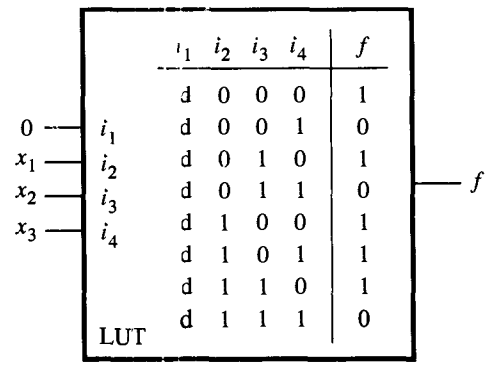


Figure 4.52 The Verilog code in Figure 4.49 implemented in a LUT.

manually. Because the logic cells in the chip are four-input lookup tables, only a single logic cell is needed for this function. The figure shows that the variables x_1, x_2, and x_3 are connected to the LUT inputs called i_2, i_3, and i_4. Input i_1 is not used because the function requires only three inputs. The truth table in the LUT indicates that the unused input is treated as a don't care. Thus only half of the rows in the table are shown, since the other half is identical. The unused LUT input is shown connected to 0 in the figure, but it could just as well be connected to 1.

It is interesting to consider the benefits provided by the optimizations used in logic synthesis. For the implementation in the CPLD, the function was simplified from the original five product terms in the canonical form to just two product terms. However, both the optimized and nonoptimized forms fit into a single macrocell in the chip, and thus they have the same cost (Appendix E shows that the MAX 7000 CPLD has five product terms in each macrocell). Similarly, for the FPGA, since a LUT is used for implementation, it does not matter whether the function is minimized, because it fits in a single LUT. The reason is that our example circuit is very small. For large circuits it is essential to perform the optimization. Examples 4.22 and 4.23 illustrate logic functions for which the cost of implementation is reduced when optimized.

Example 4.22 The Verilog code in Figure 4.53 corresponds to the function f_1 in Figure 4.7. Because there are six product terms in the canonical form, two macrocells would be needed in a MAX 7000 CPLD. When synthesized by the CAD tools, the resulting expression is

$$f = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$$

which is the same as the expression derived in Figure 4.7. Because the optimized expression has only two product terms, it can be realized using just one macrocell and hence results in a lower cost.

When f_1 is synthesized for implementation in an FPGA, the expression generated is the same as for the CPLD. Since the function has only four inputs, it needs just one LUT.

```
module example4_22 (x1, x2, x3, x4, f);
  input x1, x2, x3, x4;
  output f;

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

endmodule
```

Figure 4.53 Verilog code for f_1 in Figure 4.7.

Example 4.23

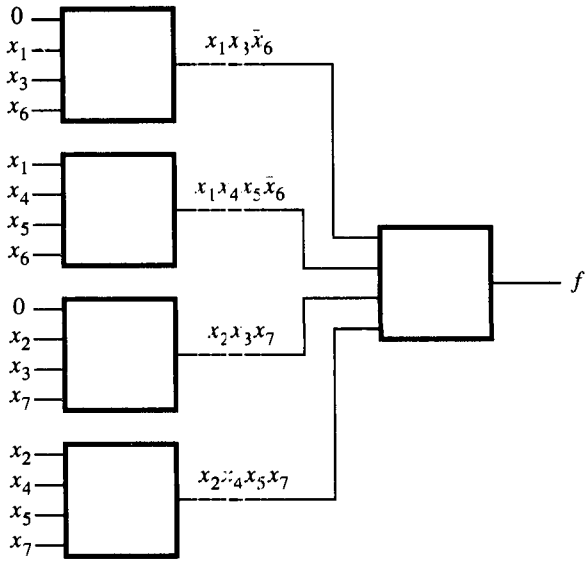
In section 4.6 we used a seven-variable logic function as a motivation for multilevel synthesis. This function is given in the Verilog code in Figure 4.54. The logic expression is in minimal sum-of-products form. When it is synthesized for implementation in a CPLD, no optimizations are performed by the CAD tools. The function requires one macrocell. This function is more interesting when we consider its implementation in an FPGA with four-input LUTs. Because there are seven inputs, more than one LUT is required. If the function is implemented directly as given in the Verilog code, then five LUTs are needed, as depicted in Figure 4.55*a*. Rather than showing the truth table programmed in each LUT, we show the logic function that is implemented at the LUT output. Synthesis results in the following expression:

$$f = (x_1\bar{x}_6 + x_2x_7)(x_3 + x_4x_5)$$

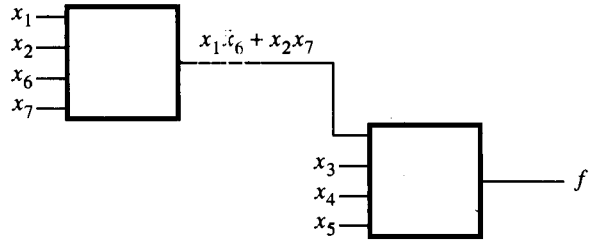
We derived the same expression by using factoring in section 4.6. As illustrated in Figure 4.55*b*, it can be implemented using only two LUTs. One LUT produces the term $S = x_1\bar{x}_6 + x_2x_7$. The other LUT implements the four-input function $f = Sx_3 + Sx_4x_5$.

```
module example4_23 (x1, x2, x3, x4, x5, x6, x7, f);
  input x1, x2, x3, x4, x5, x6, x7;
  output f;

  assign f = (x1 & x3 & ~x6) | (x1 & x4 & x5 & ~x6) |
             (x2 & x3 & x7) | (x2 & x4 & x5 & x7);

endmodule
```

Figure 4.54 Verilog code for the function of section 4.6.



(a) Sum-of-products realization



(b) Factored realization

Figure 4.55 Implementation of the Verilog code in Figure 4.54.

4.13 CONCLUDING REMARKS

This chapter has attempted to provide the reader with an understanding of various aspects of synthesis for logic functions and how synthesis is automated using modern CAD tools. Now that the reader is comfortable with the fundamental concepts, we can examine digital circuits of a more sophisticated nature. The next chapter describes circuits that perform arithmetic operations, which are a key part of computers.

PROBLEMS

4.1 Find the minimum-cost SOP and POS forms for the function $f(x_1, x_2, x_3) = \sum m(1, 2, 3, 5)$.

4.2 Repeat problem 4.1 for the function $f(x_1, x_2, x_3) = \sum m(1, 4, 7) + D(2, 5)$.

4.3 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \Pi M(0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 14, 15)$.

4.4 Repeat problem 4.1 for the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 8, 9, 10, 15) + D(1, 3, 6, 7)$.

4.5 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \Pi M(1, 4, 6, 7, 9, 12,15, 17, 20, 21, 22, 23, 28, 31)$.

4.6 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(0, 1, 3, 4, 6, 8, 9, 11, 13, 14, 16, 19, 20, 21, 22, 24, 25) + D(5, 7, 12, 15, 17, 23)$.

4.7 Repeat problem 4.1 for the function $f(x_1, \ldots, x_5) = \sum m(1, 4, 6, 7, 9, 10, 12, 15, 17, 19, 20, 23, 25, 26, 27, 28, 30, 31) + D(8, 16, 21, 22)$.

4.8 Find 5 three-variable functions for which the product-of-sums form has lower cost than the sum-of-products form.

4.9 A four-variable logic function that is equal to 1 if any three or all four of its variables are equal to 1 is called a *majority* function. Design a minimum-cost circuit that implements this majority function.

4.10 Derive a minimum-cost realization of the four-variable function that is equal to 1 if exactly two or exactly three of its variables are equal to 1; otherwise it is equal to 0.

4.11 Prove or show a counter-example for the statement: If a function f has a unique minimum-cost SOP expression, then it also has a unique minimum-cost POS expression.

4.12 A circuit with two outputs has to implement the following functions

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 6, 7, 9) + D(10, 11)$$

$$g(x_1, \ldots, x_4) = \sum m(2, 4, 9, 10, 15) + D(0, 13, 14)$$

Design the minimum-cost circuit and compare its cost with combined costs of two circuits that implement f and g separately. Assume that the input variables are available in both uncomplemented and complemented forms.

4.13 Repeat problem 4.12 for the following functions

$$f(x_1, \ldots, x_5) = \sum m(1, 4, 5, 11, 27, 28) + D(10, 12, 14, 15, 20, 31)$$

$$g(x_1, \ldots, x_5) = \sum m(0, 1, 2, 4, 5, 8, 14, 15, 16, 18, 20, 24, 26, 28, 31) + D(10, 11, 12, 27)$$

4.14 Implement the logic circuit in Figure 4.23 using NAND gates only.

4.15 Implement the logic circuit in Figure 4.23 using NOR gates only.

4.16 Implement the logic circuit in Figure 4.25 using NAND gates only.

4.17 Implement the logic circuit in Figure 4.25 using NOR gates only.

4.18 Consider the function $f = x_3x_5 + \bar{x}_1x_2x_4 + x_1\bar{x}_2\bar{x}_4 + x_1x_3\bar{x}_4 + \bar{x}_1x_3x_4 + \bar{x}_1x_2x_5 + x_1\bar{x}_2x_5$. Derive a minimum-cost circuit that implements this function using NOT, AND, and OR gates.

4.19 Derive a minimum-cost circuit that implements the function $f(x_1, \ldots, x_4) = \sum m(4, 7, 8, 11) + D(12, 15)$.

4.20 Find the simplest realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 7, 9, 10, 13, 14)$, assuming that the logic gates have a maximum fan-in of two.

4.21 Find the minimum-cost circuit for the function $f(x_1, \ldots, x_4) = \sum m(0, 4, 8, 13, 14, 15)$. Assume that the input variables are available in uncomplemented form only. (Hint: use functional decomposition.)

4.22 Use functional decomposition to find the best implementation of the function $f(x_1, \ldots, x_5) = \sum m(1, 2, 7, 9, 10, 18, 19, 25, 31) + D(0, 15, 20, 26)$. How does your implementation compare with the lowest-cost SOP implementation? Give the costs.

4.23 Use the tabular method discussed in section 4.9 to find a minimum cost SOP realization for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$$

4.24 Repeat problem 4.23 for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 4, 6, 8, 9, 15) + D(3, 7, 11, 13)$$

4.25 Repeat problem 4.23 for the function

$$f(x_1, \ldots, x_4) = \sum m(0, 3, 4, 5, 7, 9, 11) + D(8, 12, 13, 14)$$

4.26 Show that the following distributive-like rules are valid

$$(A \cdot B)\#C = (A\#C) \cdot (B\#C)$$
$$(A + B)\#C = (A\#C) + (B\#C)$$

4.27 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4) = \sum m(0, 2, 4, 5, 7, 8, 9, 15)$.

4.28 Repeat problem 4.27 for the function $f(x_1, \ldots, x_5) = \bar{x}_1\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3 + x_2x_3\bar{x}_4x_5 + x_1\bar{x}_2\bar{x}_3x_4 + x_1x_2x_3x_4\bar{x}_5 + \bar{x}_1x_2x_4\bar{x}_5 + \bar{x}_1\bar{x}_3x_4x_5$.

4.29 Use the cubical representation and the method discussed in section 4.10 to find a minimum-cost SOP realization of the function $f(x_1, \ldots, x_4)$ defined by the ON-set ON = {00x0, 100x, x010, 1111} and the don't-care set DC = {00x1, 011x}.

4.30 In section 4.10.1 we showed how the $*$-product operation can be used to find the prime implicants of a given function f. Another possibility is to find the prime implicants by expanding the implicants in the initial cover of the function. An implicant is *expanded* by removing one literal to create a larger implicant (in terms of the number of vertices covered). A larger implicant is valid only if it does not include any vertices for which $f = 0$. The largest valid implicants obtained in the process of expansion are the prime implicants. Figure P4.1 illustrates the expansion of the implicant $\bar{x}_1x_2x_3$ of the function in Figure 4.9, which is also used in Example 4.16. Note from Figure 4.9 that

$$\bar{f} = x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 + x_1x_2\bar{x}_3$$

In Figure P4.1 the word NO is used to indicate that the expanded term is not valid, because it includes one or more vertices from $\bar{f}$. From the graph it is clear that the largest valid implicants that arise from this expansion are x_2x_3 and $\bar{x}_1$; they are prime implicants of f.

Expand the other four implicants given in the initial cover in Example 4.14 to find all prime implicants of f. What is the relative complexity of this procedure compared to the $*$-product technique?

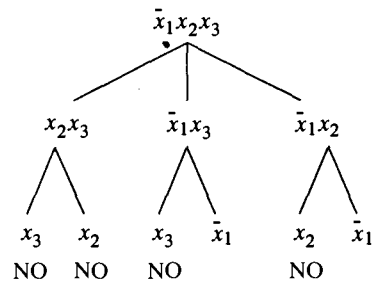


Figure P4.1 Expansion of implicant $\bar{x}_1x_2x_3$.

4.31 Repeat problem 4.30 for the function in Example 4.17. Expand the implicants given in the initial cover C^0.

4.32 Consider the logic expressions

$$f = x_1\bar{x}_2\bar{x}_5 + \bar{x}_1\bar{x}_2\bar{x}_4\bar{x}_5 + x_1x_2x_4x_5 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2x_3x_5 + \bar{x}_2\bar{x}_3x_4\bar{x}_5 + x_1x_2x_3x_4\bar{x}_5$$

$$g = \bar{x}_2x_3\bar{x}_4 + \bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5 + x_1x_3x_4\bar{x}_5 + x_1\bar{x}_2x_4\bar{x}_5 + x_1x_3x_4x_5 + \bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_5 + x_1x_2\bar{x}_3x_4x_5$$

Prove or disprove that $f = g$.

4.33 Consider the circuit in Figure P4.2, which implements functions f and g. What is the cost of this circuit, assuming that the input variables are available in both true and complemented Redesign the circuit to implement the same functions, but at as low a cost as possible. What is the cost of your circuit?

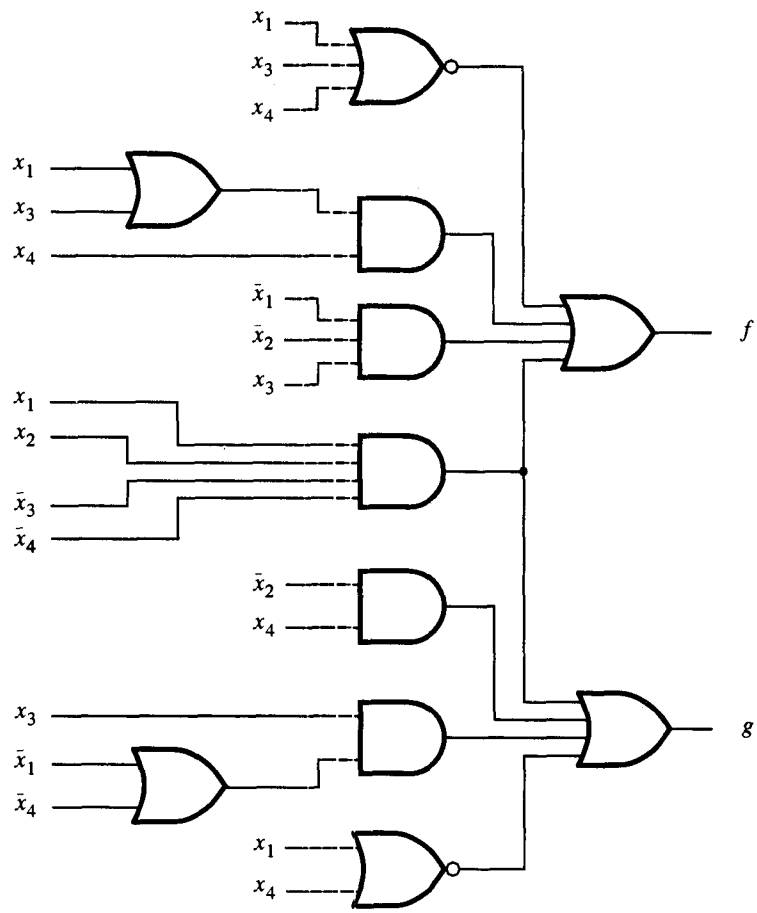


Figure P4.2 Circuit for problem 4.33.

4.34 Repeat problem 4.33 for the circuit in Figure P4.3. Use only NAND gates in your circuit.

4.35 Write Verilog code to implement the circuit in Figure 4.25*b* using the gate level primitives.

4.36 Write Verilog code to implement the circuit in Figure 4.25*b* using the continuous assignment.

4.37 Write Verilog code to implement the circuit in Figure 4.27*c* using the gate level primitives.

4.38 Write Verilog code to implement the circuit in Figure 4.27*c* using the continuous assignment.

4.39 Write Verilog code to implement the circuit in Figure 4.28*b* using the gate level primitives.

4.40 Write Verilog code to implement the circuit in Figure 4.28*b* using the continuous assignment.

4.41 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15)$ using the gate level primitives. Ensure that the resulting circuit is as simple as possible.

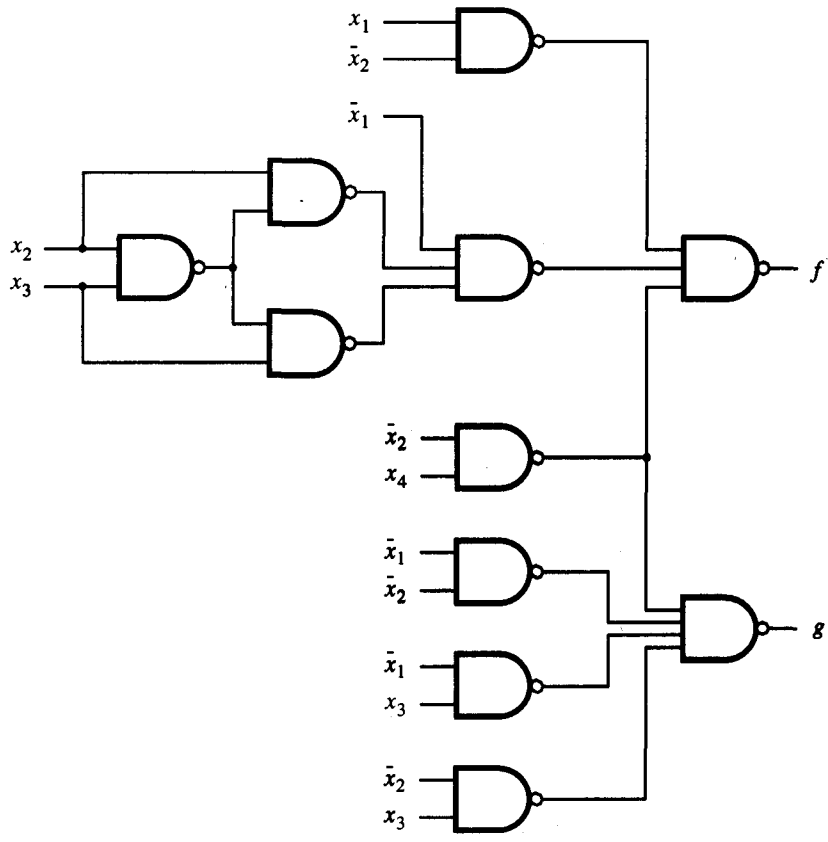


Figure P4.3 Circuit for problem 4.34.

4.42 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \sum m(0, 1, 2, 4, 5, 7, 8, 9, 11, 12, 14, 15)$ using the continuous assignment.

4.43 Repeat problem 4.41 for the function $f(x_1, \ldots, x_4) = \sum m(1, 4, 7, 14, 15) + D(0, 5, 9)$.

4.44 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \sum m(1, 4, 7, 14, 15) + D(0, 5, 9)$ using the continuous assignment.

4.45 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \Pi M(6, 8, 9, 12, 13)$ using the gate level primitives. Ensure that the resulting circuit is as simple as possible.

4.46 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \Pi M(6, 8, 9, 12, 13)$ using the continuous assignment.

4.47 Repeat problem 4.45 for the function $f(x_1, \ldots, x_4) = \Pi M(3, 11, 14) + D(0, 2, 10, 12)$.

4.48 Write Verilog code to implement the function $f(x_1, \ldots, x_4) = \Pi M(3, 11, 14) + D(0, 2, 10, 12)$. using the continuous assignment.

REFERENCES

1. M. Karnaugh, "A Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of AIEE, Communications and Electronics* 72, part 1, November 1953, pp. 593–599.
2. R. L. Ashenhurst, "The Decomposition of Switching Functions," Proc. of the Symposium on the Theory of Switching, 1957, *Vol. 29 of Annals of Computation Laboratory* (Harvard University: Cambridge, MA, 1959), pp. 74–116.
3. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).
4. T. Sasao, *Logic Synthesis and Optimization* (Kluwer: Boston, MA, 1993).
5. S. Devadas, A. Gosh, and K. Keutzer, *Logic Synthesis* (McGraw-Hill: New York, 1994).
6. W. V. Quine, "The Problem of Simplifying Truth Functions," *Amer. Math. Monthly* 59 (1952), pp. 521–531.
7. E. J. McCluskey Jr., "Minimization of Boolean Functions," *Bell System Tech. Journal*, November 1956, pp. 1417–1444.
8. E. J. McCluskey, *Logic Design Principles* (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. J. F. Wakerly, *Digital Design Principles and Practices* (Prentice-Hall: Englewood Cliffs, NJ, 1990).
10. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
11. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed. (West: St. Paul, MN, 1993).
12. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
13. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
14. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
15. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
16. A. Dewey, *Analysis and Design of Digital Systems with VHDL* (PWS Publishing Co.: Boston, MA, 1997).
17. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
18. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
19. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer: Boston, MA, 1984).
20. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Synthesis Optimization System," *IEEE Transactions on Computer-Aided Design*, CAD-6, November 1987, pp. 1062–81.

21. E. M. Sentovic, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Technical Report UCB/ERL M92/41, Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

22. G. De Micheli, *Synthesis and Optimization of Digital Circuits* (McGraw-Hill: New York, 1994).

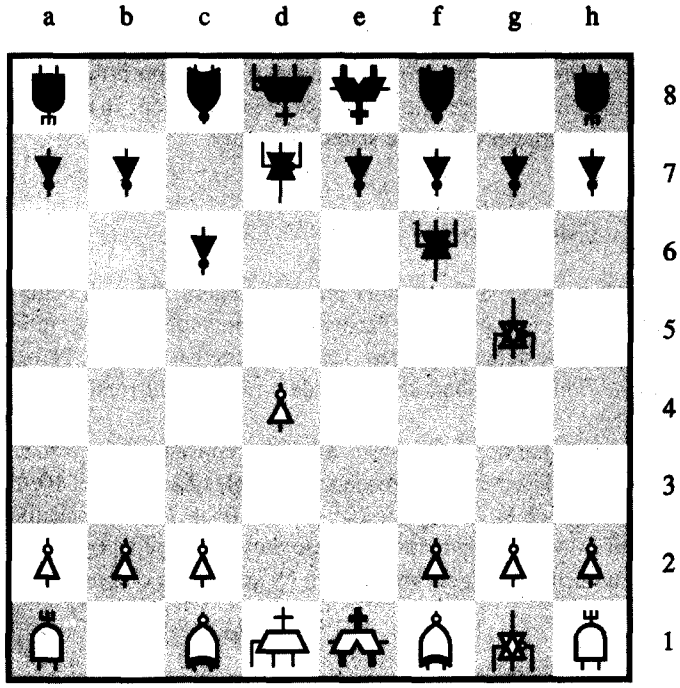
23. N. Sherwani, *Algorithms for VLSI Physical Design Automation* (Kluwer: Boston, MA, 1995).

24. B. Preas and M. Lorenzetti, *Physical Design Automation of VLSI Systems* (Benjamin/Cummings: Redwood City, CA, 1988).

chapter

5

NUMBER REPRESENTATION AND ARITHMETIC CIRCUITS



5. Ne4–g5, Ng8–f6

In this chapter we will discuss logic circuits that perform arithmetic operations. We will explain how numbers can be added, subtracted, and multiplied. We will also show how to write Verilog code to describe the arithmetic circuits. These circuits provide an excellent platform for illustrating the power and versatility of Verilog in specifying complex logic-circuit assemblies. The concepts involved in the design of the arithmetic circuits are easily applied to a wide variety of other circuits.

Before tackling the design of arithmetic circuits, it is necessary to discuss how numbers are represented in digital systems. In the previous chapters we dealt with logic variables in a general way, using variables to represent either the states of switches or some general conditions. Now we will use the variables to represent numbers. Several variables are needed to specify a number, with each variable corresponding to one digit of the number.

5.1 POSITIONAL NUMBER REPRESENTATION

When dealing with numbers and arithmetic operations, it is convenient to use standard symbols. Thus to represent addition we use the plus (+) symbol, and for subtraction we use the minus (−) symbol. In previous chapters we used the + symbol to represent the logical OR operation and − to denote the deletion of an element from a set. Even though we will now use the same symbols for two different purposes, the meaning of each symbol will usually be clear from the context of the discussion. In cases where there may be some ambiguity, the meaning will be stated explicitly.

5.1.1 UNSIGNED INTEGERS

The simplest numbers to consider are the integers. We will begin by considering positive integers and then expand the discussion to include negative integers. Numbers that are positive only are called *unsigned*, and numbers that can also be negative are called *signed*. Representation of numbers that include a radix point (real numbers) is discussed later in the chapter.

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

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

which represents the value

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

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

Because the digits have 10 possible values and each digit is weighted as a power of 10, we say that decimal numbers are *base*-10, or *radix*-10 numbers. Decimal numbers are familiar, convenient, and easy to understand. However, in digital circuits it is not practical to use digits that can assume 10 values. In digital systems we use the binary, or *base*-2,

system in which digits can be 0 or 1. Each binary digit is called a *bit*. In the binary number system, the same positional number representation is used so that

$$B = b_{n-1}b_{n-2}\cdots b_1b_0$$

represents an integer that has the value

$$V(B) = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad [5.1]$$
$$= \sum_{i=0}^{n-1} b_i \times 2^i$$

For example, the binary number 1101 represents the value

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

Because a particular digit pattern has different meanings for different radices, we will indicate the radix 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

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

Note that the range of integers that can be represented by a binary number depends on the number of bits used. For example, with four bits the largest number is $(1111)_2 = (15)_{10}$. An example of a larger number is $(10110111)_2 = (183)_{10}$. In general, using n bits allows representation of 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 of an unsigned integer, 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*.

5.1.2 CONVERSION BETWEEN DECIMAL AND BINARY SYSTEMS

A binary number is converted into a decimal number simply by applying Equation 5.1 and evaluating it using decimal arithmetic. Converting a decimal number into a binary number is not quite as straightforward. The conversion can be performed by successively dividing the decimal number by 2 as follows. Suppose that a decimal number $D = d_{k-1}\cdots d_1d_0$, with a value V, is to be converted into a binary number $B = b_{n-1}\cdots b_2b_1b_0$. Thus

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

If we divide V by 2, the result is

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

The quotient of this integer division is $b_{n-1} \times 2^{n-2} + \cdots + b_2 \times 2 + b_1$, and the remainder is b_0. If the remainder is 0, then $b_0 = 0$; if it is 1, then $b_0 = 1$. Observe that the quotient

is just another binary number, which comprises $n - 1$ bits, rather than n bits. Dividing this number by 2 yields the remainder b_1. The new quotient is

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

Continuing the process of dividing the new quotient by 2, and determining one bit in each step, will produce all bits of the binary number. The process continues until the quotient becomes 0. Figure 5.1 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.

5.1.3 OCTAL AND HEXADECIMAL REPRESENTATIONS

The positional number representation can be used for any radix. If the radix is r, then the number

$$K = k_{n-1}k_{n-2}\cdots k_1k_0$$

has the value

$$V(K) = \sum_{i=0}^{n-1} k_i \times r^i$$

Our interest is limited to those radices that are most practical. We will use decimal numbers because they are used by people, and we will use binary numbers because they are used by computers. In addition, two other radices are useful—8 and 16. Numbers represented with radix 8 are called *octal* numbers, while radix-16 numbers are called *hexadecimal* numbers. In octal representation the digit values range from 0 to 7. In hexadecimal representation

Convert $(857)_{10}$

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

Result is $(1101011001)_2$

Figure 5.1 Conversion from decimal to binary.

(often abbreviated as *hex*), each digit can have one of 16 values. The first 10 are denoted the same as in the decimal system, namely, 0 to 9. Digits that correspond to the decimal values 10, 11, 12, 13, 14, and 15 are denoted by the letters, A, B, C, D, E, and F. Table 5.1 gives the first 18 integers in these number systems.

Table 5.1 Numbers in different systems.

Decimal	Binary	Octal	Hexadecimal
00	00000	00	00
01	00001	01	01
02	00010	02	02
03	00011	03	03
04	00100	04	04
05	00101	05	05
06	00110	06	06
07	00111	07	07
08	01000	10	08
09	01001	11	09
10	01010	12	0A
11	01011	13	0B
12	01100	14	0C
13	01101	15	0D
14	01110	16	0E
15	01111	17	0F
16	10000	20	10
17	10001	21	11
18	10010	22	12

In computers the dominant number system is binary. The reason for using the octal and hexadecimal systems is that they serve as a useful shorthand notation for binary numbers. One octal digit represents three bits. Thus a binary number is converted into an octal number by taking groups of three bits, starting from the least-significant bit, and replacing them with the corresponding octal digit. For example, 101011010111 is converted as

$$\underbrace{101}_{5}\quad\underbrace{011}_{3}\quad\underbrace{010}_{2}\quad\underbrace{111}_{7}$$

which means that $(101011010111)_2 = (5327)_8$. If the number of bits is not a multiple of three, then we add 0s to the left of the most-significant bit. For example, $(10111011)_2 = (273)_8$ because

$$\underbrace{010}_{2}\quad\underbrace{111}_{7}\quad\underbrace{011}_{3}$$

Conversion from octal to binary is just as straightforward; each octal digit is simply replaced by three bits that denote the same value.

Similarly, a hexadecimal digit is represented using four bits. For example, a 16-bit number is represented using four hex digits, as in

$$(1010111100100101)_2 = (AF25)_{16}$$

because

$$\underbrace{1010}_{A} \quad \underbrace{1111}_{F} \quad \underbrace{0010}_{2} \quad \underbrace{0101}_{5}$$

Zeros are added to the left of the most-significant bit if the number of bits is not a multiple of four. For example, $(1101101000)_2 = (368)_{16}$ because

$$\underbrace{0011}_{3} \quad \underbrace{0110}_{6} \quad \underbrace{1000}_{8}$$

Conversion from hexadecimal to binary involves straightforward substitution of each hex digit by four bits that denote the same value.

Binary numbers used in modern computers often have 32 or 64 bits. Written as binary n-tuples (sometimes called bit vectors), such numbers are awkward for people to deal with. It is much simpler to deal with them in the form of 8- or 16-digit hex numbers. Because the arithmetic operations in a digital system usually involve binary numbers, we will focus on circuits that use such numbers. We will sometimes use the hexadecimal representation as a convenient shorthand description.

We have introduced the simplest numbers—unsigned integers. It is necessary to be able to deal with several other types of numbers. We will discuss the representation of signed numbers, fixed-point numbers, and floating-point numbers later in this chapter. But first we will examine some simple circuits that operate on numbers to give the reader a feeling for digital circuits that perform arithmetic operations and to provide motivation for further discussion.

5.2 ADDITION OF UNSIGNED NUMBERS

Binary addition is performed in the same way as decimal addition except that the values of individual digits can be only 0 or 1. The addition of 2 one-bit numbers entails four possible combinations, as indicated in Figure 5.2*a*. Two bits are needed to represent the result of the addition. The right-most bit is called the *sum*, s. The left-most bit, which is produced as a carry-out when both bits being added are equal to 1, is called the *carry*, c. The addition operation is defined in the form of a truth table in part (*b*) of the figure. The sum bit s is the XOR function, which was introduced in section 3.9.1. The carry c is the AND function of inputs x and y. A circuit realization of these functions is shown in Figure 5.2*c*. This circuit, which implements the addition of only two bits, is called a *half-adder*.

A more interesting case is when larger numbers that have multiple bits are involved. Then it is still necessary to add each pair of bits, but for each bit position i, the addition operation may include a *carry-in* from bit position $i - 1$.

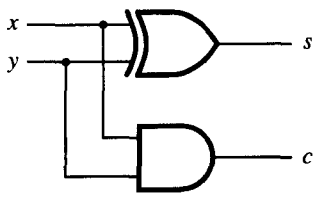
x	0	0	1	1
$+y$	$+0$	$+1$	$+0$	$+1$
$c\ s$	0 0	0 1	0 1	1 0

Carry — Sum

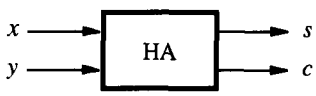
(a) The four possible cases

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table



(c) Circuit



(d) Graphical symbol

Figure 5.2 Half-adder.

Figure 5.3 gives an example of the addition operation. The operands are $X = (01111)_2 = (15)_{10}$ and $Y = (01010)_2 = (10)_{10}$. Note that five bits are used to represent X and Y. Using five bits, it is possible to represent integers in the range from 0 to 31; hence the sum $S = X + Y = (25)_{10}$ can also be denoted as a five-bit integer. Note also the labeling of individual bits, such that $X = x_4x_3x_2x_1x_0$ and $Y = y_4y_3y_2y_1y_0$. The figure shows the carries generated during the addition process. For example, a carry of 0 is generated when x_0 and y_0 are added, a carry of 1 is produced when x_1 and y_1 are added, and so on.

In Chapters 2 and 4 we designed logic circuits by first specifying their behavior in the form of a truth table. This approach is impractical in designing an adder circuit that can add the five-bit numbers in Figure 5.3. The required truth table would have 10 input variables, 5 for each number X and Y. It would have $2^{10} = 1024$ rows! A better approach is to consider the addition of each pair of bits, x_i and y_i, separately.

$X = x_4x_3x_2x_1x_0$	0 1 1 1 1	$(15)_{10}$
$+ Y = y_4y_3y_2y_1y_0$	0 1 0 1 0	$(10)_{10}$
	1 1 1 0 ← Generated carries	
$S = s_4s_3s_2s_1s_0$	1 1 0 0 1	$(25)_{10}$

Figure 5.3 An example of addition.

For bit position 0, there is no carry-in, and hence the addition is the same as for Figure 5.2. For each other bit position i, the addition involves bits x_i and y_i, and a carry-in c_i. The sum and carry-out functions of variables x_i, y_i, and c_i are specified in the truth table in Figure 5.4*a*. The sum bit, s_i, is the modulo-2 sum of x_i, y_i, and c_i. The *carry-out*, c_{i+1}, is equal to 1 if the sum of x_i, y_i, and c_i is equal to either 2 or 3. Karnaugh maps for these functions are shown in part (*b*) of the figure. For the carry-out function the optimal sum-of-products realization is

$$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$$

For the s_i function a sum-of-products realization is

$$s_i = \bar{x}_iy_i\bar{c}_i + x_i\bar{y}_i\bar{c}_i + \bar{x}_i\bar{y}_ic_i + x_iy_ic_i$$

A more attractive way of implementing this function is by using the XOR gates, as explained below.

Use of XOR Gates

The XOR function of two variables is defined as $x_1 \oplus x_2 = \bar{x}_1x_2 + x_1\bar{x}_2$. The preceding expression for the sum bit can be manipulated into a form that uses only XOR operations as follows

$$\begin{aligned} s_i &= (\bar{x}_iy_i + x_i\bar{y}_i)\bar{c}_i + (\bar{x}_i\bar{y}_i + x_iy_i)c_i \\ &= (x_i \oplus y_i)\bar{c}_i + \overline{(x_i \oplus y_i)}c_i \\ &= (x_i \oplus y_i) \oplus c_i \end{aligned}$$

The XOR operation is associative; hence we can write

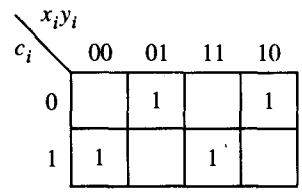
$$s_i = x_i \oplus y_i \oplus c_i$$

Therefore, a single three-input XOR gate can be used to realize s_i.

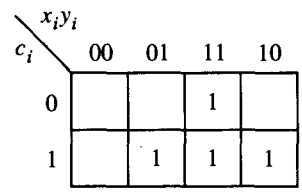
The XOR gate generates as an output a modulo-2 sum of its inputs. The output is equal to 1 if an odd number of inputs have the value 1, and it is equal to 0 otherwise. For this reason the XOR is sometimes referred to as the *odd* function. Observe that the XOR has no minterms that can be combined into a larger product term, as evident from the checkerboard pattern for function s_i in the map in Figure 5.4*b*. The logic circuit implementing the truth table in Figure 5.4*a* is given in Figure 5.4*c*. This circuit is known as a *full-adder*.

c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

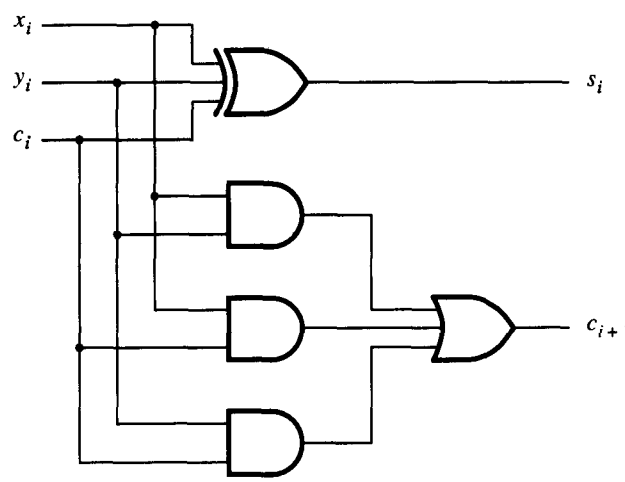


$s_i = x_i \oplus y_i \oplus c_i$



$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$

(b) Karnaugh maps



(c) Circuit

Figure 5.4 Full-adder.

Another interesting feature of XOR gates is that a two-input XOR gate can be thought of as using one input as a control signal that determines whether the true or complemented value of the other input will be passed through the gate as the output value. This is clear from the definition of XOR, where $x_i \oplus y_i = \bar{x}y + x\bar{y}$. Consider x to be the control input. Then if $x = 0$, the output will be equal to the value of y. But if $x = 1$, the output will

be equal to the complement of y. In the derivation above, we used algebraic manipulation to derive $s_i = (x_i \oplus y_i) \oplus c_i$. We could have obtained the same expression immediately by making the following observation. In the top half of the truth table in Figure 5.4*a*, c_i is equal to 0, and the sum function s_i is the XOR of x_i and y_i. In the bottom half of the table, c_i is equal to 1, while s_i is the complemented version of its top half. This observation leads directly to our expression using 2 two-input XOR operations. We will encounter an important example of using XOR gates to pass true or complemented signals under the control of another signal in section 5.3.3.

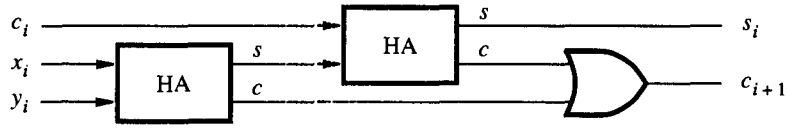
In the preceding discussion we encountered the complement of the XOR operation, which we denoted as $\overline{x \oplus y}$. This operation is used so commonly that it is given the distinct name *XNOR*. A special symbol, $\odot$, is often used to denote the XNOR operation, namely

$$x \odot y = \overline{x \oplus y}$$

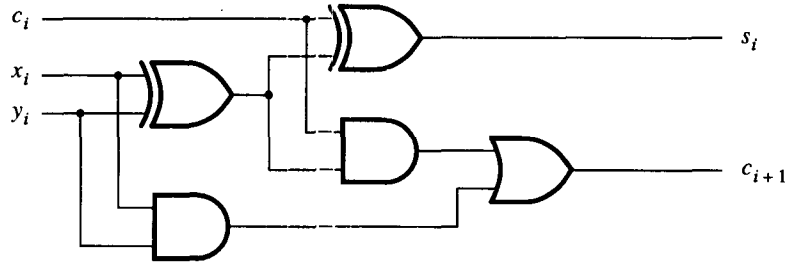
The XNOR is sometimes also referred to as the *coincidence* operation because it produces the output of 1 when its inputs coincide in value; that is, they are both 0 or both 1.

5.2.1 DECOMPOSED FULL-ADDER

In view of the names used for the circuits, one can expect that a full-adder can be constructed using half-adders. This can be accomplished by creating a multilevel circuit of the type discussed in section 4.6.2. The circuit is given in Figure 5.5. It uses two half-adders to form a full-adder. The reader should verify the functional correctness of this circuit.



(a) Block diagram



(b) Detailed diagram

Figure 5.5 A decomposed implementation of the full-adder circuit.

5.2.2 RIPPLE-CARRY ADDER

To perform addition by hand, we start from the least-significant digit and add pairs of digits, progressing to the most-significant digit. If a carry is produced in position i, then this carry is added to the operands in position $i + 1$. The same arrangement can be used in a logic circuit that performs addition. For each bit position we can use a full-adder circuit, connected as shown in Figure 5.6. Note that to be consistent with the customary way of writing numbers, the least-significant bit position is on the right. Carries that are produced by the full-adders propagate to the left.

When the operands X and Y are applied as inputs to the adder, it takes some time before the output sum, S, is valid. Each full-adder introduces a certain delay before its s_i and c_{i+1} outputs are valid. Let this delay be denoted as Δt. Thus the carry-out from the first stage, c_1, arrives at the second stage Δt after the application of the x_0 and y_0 inputs. The carry-out from the second stage, c_2, arrives at the third stage with a $2\Delta t$ delay, and so on. The signal c_{n-1} is valid after a delay of $(n - 1)\Delta t$, which means that the complete sum is available after a delay of $n\Delta t$. Because of the way the carry signals "ripple" through the full-adder stages, the circuit in Figure 5.6 is called a *ripple-carry adder.*

The delay incurred to produce the final sum and carry-out in a ripple-carry adder depends on the size of the numbers. When 32- or 64-bit numbers are used, this delay may become unacceptably high. Because the circuit in each full-adder leaves little room for a drastic reduction in the delay, it may be necessary to seek different structures for implementation of n-bit adders. We will discuss a technique for building high-speed adders in section 5.4.

So far we have dealt with unsigned integers only. The addition of such numbers does not require a carry-in for stage 0. In Figure 5.6 we included c_0 in the diagram so that the ripple-carry adder can also be used for subtraction of numbers, as we will see in section 5.3.

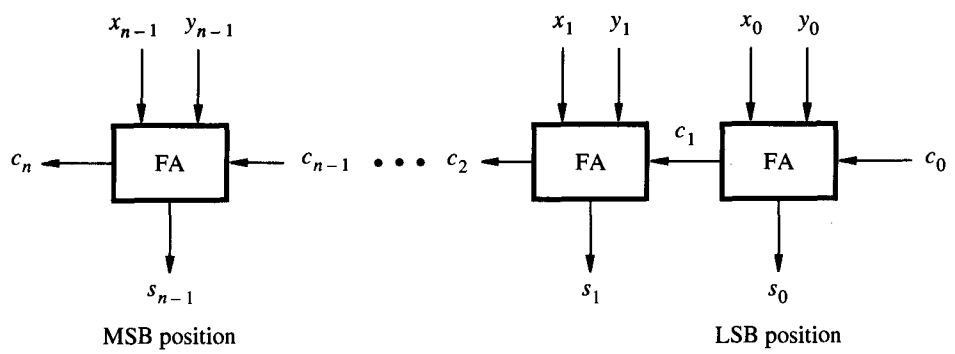


Figure 5.6 An n-bit ripple-carry adder.

5.2.3 DESIGN EXAMPLE

Suppose that we need a circuit that multiplies an eight-bit unsigned number by 3. Let $A = a_7a_6 \cdots a_1a_0$ denote the number and $P = p_9p_8 \cdots p_1p_0$ denote the product $P = 3A$. Note that 10 bits are needed to represent the product.

A simple approach to design the required circuit is to use two ripple-carry adders to add three copies of the number A, as illustrated in Figure 5.7*a*. The symbol that denotes each adder is a commonly used graphical symbol for adders. The letters x_i, y_i, s_i, and c_i indicate the meaning of the inputs and outputs according to Figure 5.6. The first adder produces $A + A = 2A$. Its result is represented as eight sum bits and the carry from the most-significant bit. The second adder produces $2A + A = 3A$. It has to be a nine-bit adder to be able to handle the nine bits of $2A$, which are generated by the first adder. Because the y_i inputs have to be driven only by the eight bits of A, the ninth input y_8 is connected to a constant 0.

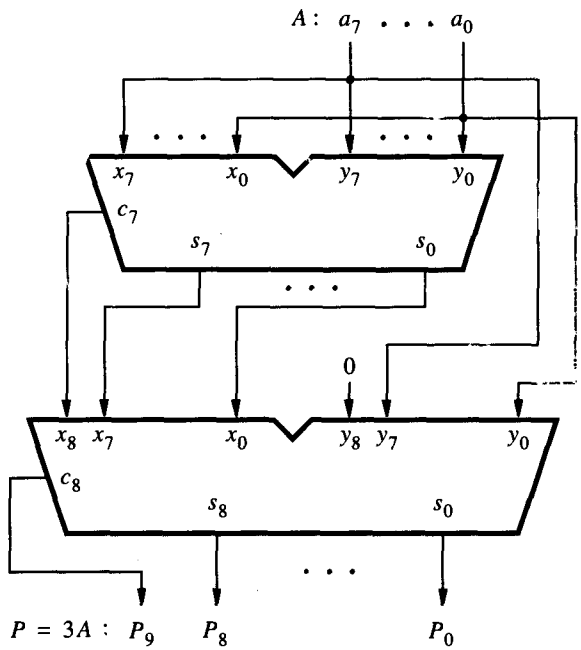
This approach is straightforward, but not very efficient. Because $3A = 2A + A$, we can observe that $2A$ can be generated by shifting the bits of A one bit-position to the left, which gives the bit pattern $a_7a_6a_5a_4a_3a_2a_1a_00$. According to equation 5.1, this pattern is equal to $2A$. Then a single ripple-carry adder suffices for implementing $3A$, as shown in Figure 5.7*b*. This is essentially the same circuit as the second adder in part (*a*) of the figure. Note that the input x_0 is connected to a constant 0. Note also that in the second adder in part (*a*) the value of x_0 is always 0, even though it is driven by the least-significant bit, s_0, of the sum of the first adder. Because $x_0 = y_0 = a_0$ in the first adder, the sum bit s_0 will be 0, whether a_0 is 0 or 1.

5.3 SIGNED NUMBERS

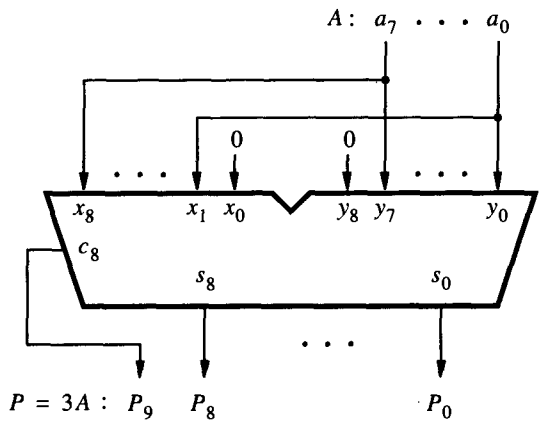
In the decimal system the sign of a number is indicated by a + or − symbol to the left of the most-significant digit. In the binary system the *sign* of a number is denoted by the left-most bit. For a positive number the left-most bit is equal to 0, and for a negative number it is equal to 1. Therefore, in signed numbers the left-most bit represents the sign, and the remaining $n - 1$ bits represent the magnitude, as illustrated in Figure 5.8. It is important to note the difference in the location of the most-significant bit (MSB). In unsigned numbers all bits represent the magnitude of a number; hence all n bits are *significant* in defining the magnitude. Therefore, the MSB is the left-most bit, b_{n-1}. In signed numbers there are $n - 1$ significant bits, and the MSB is in bit position b_{n-2}.

5.3.1 NEGATIVE NUMBERS

Positive numbers are represented using the positional number representation as explained in the previous section. Negative numbers can be represented in three different ways: sign-and-magnitude, 1's complement, and 2's complement.



(a) Naive approach



(b) Efficient design

Figure 5.7 Circuit that multiplies an eight-bit unsigned number by 3.

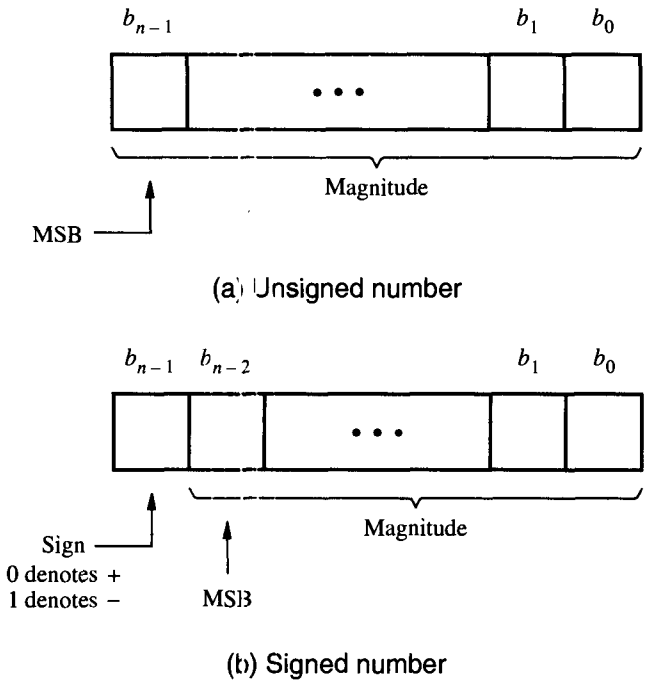


Figure 5.8 Formats for representation of integers.

Sign-and-Magnitude Representation

In the familiar decimal representation, the magnitude of both positive and negative numbers is expressed in the same way. The sign symbol distinguishes a number as being positive or negative. This scheme is called the *sign-and-magnitude* number representation. The same scheme can be used with binary numbers in which case the sign bit is 0 or 1 for positive or negative numbers, respectively. For example, if we use four-bit numbers, then +5 = 0101 and −5 = 1101. Because of its similarity to decimal sign-and-magnitude numbers, this representation is easy to understand. However, as we will see shortly, this representation is not well suited for use in computers. More suitable representations are based on complementary systems, explained below.

1's Complement Representation

In a complementary number system, the negative numbers are defined according to a subtraction operation involving positive numbers. We will consider two schemes for binary numbers: the 1's complement and the 2's complement. In the *1's complement* scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from $2^n - 1$; that is, $K = (2^n - 1) - P$. For example, if $n = 4$, then $K = (2^4 - 1) - P = (15)_{10} - P = (1111)_2 - P$. If we convert +5 to a negative, we get −5 = 1111 − 0101 = 1010. Similarly, +3 = 0011 and −3 = 1111 − 0011 = 1100. Clearly, the 1's complement

can be obtained simply by complementing each bit of the number, including the sign bit. While 1's complement numbers are easy to derive, they have some drawbacks when used in arithmetic operations, as we will see in the next section.

2's Complement Representation

In the 2's complement scheme, a negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n; namely, $K = 2^n - P$. Using our four-bit example, $-5 = 10000 - 0101 = 1011$, and $-3 = 10000 - 0011 = 1101$. Finding 2's complements in this manner requires performing a subtraction operation that involves borrows. However, we can observe that if K_1 is the 1's complement of P and K_2 is the 2's complement of P, then

$$K_1 = (2^n - 1) - P$$
$$K_2 = 2^n - P$$

It follows that $K_2 = K_1 + 1$. Thus a simpler way of finding a 2's complement of a number is to add 1 to its 1's complement because finding a 1's complement is trivial. This is how 2's complement numbers are obtained in logic circuits that perform arithmetic operations.

The reader will need to develop an ability to find 2's complement numbers quickly. There is a simple rule that can be used for this purpose.

Rule for Finding 2's Complements Given a signed number, $B = b_{n-1}b_{n-2}\cdots b_1b_0$, its 2's complement, $K = k_{n-1}k_{n-2}\cdots k_1k_0$, can be found by examining the bits of B from right to left and taking the following action: copy all bits of B that are 0 and the first bit that is 1; then simply complement the rest of the bits.

For example, if $B = 0110$, then we copy $k_0 = b_0 = 0$ and $k_1 = b_1 = 1$, and complement the rest so that $k_2 = \overline{b}_2 = 0$ and $k_3 = \overline{b}_3 = 1$. Hence $K = 1010$. As another example, if $B = 10110100$, then $K = 01001100$. We leave the proof of this rule as an exercise for the reader.

Table 5.2 illustrates the interpretation of all 16 four-bit patterns in the three signed-number representations that we have considered. Note that for both sign-and-magnitude representation and for 1's complement representation there are two patterns that represent the value zero. For 2's complement there is only one such pattern. Also, observe that the range of numbers that can be represented with four bits in 2's complement form is -8 to $+7$, while in the other two representations it is -7 to $+7$.

Using 2's-complement representation, an n-bit number $B = b_{n-1}b_{n-2}\cdots b_1b_0$ represents the value

$$V(B) = (-b_{n-1} \times 2^{n-1}) + b_{n-2} \times 2^{n-2} + \cdots + b_1 \times 2^1 + b_0 \times 2^0 \qquad \textbf{[5.2]}$$

Thus the largest negative number, $100\cdots00$, has the value -2^{n-1}. The largest positive number, $011\cdots11$, has the value $2^{n-1} - 1$.

Table 5.2 Interpretation of four-bit signed integers.

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	+0
1000	−0	−7	−8
1001	−1	−6	−7
1010	−2	−5	−6
1011	−3	−4	−5
1100	−4	−3	−4
1101	−5	−2	−3
1110	−6	−1	−2
1111	−7	−0	−1

5.3.2 ADDITION AND SUBTRACTION

To assess the suitability of different number representations, it is necessary to investigate their use in arithmetic operations—particularly in addition and subtraction. We can illustrate the good and bad aspects of each representation by considering very small numbers. We will use four-bit numbers, consisting of a sign bit and three significant bits. Thus the numbers have to be small enough so that the magnitude of their sum can be expressed in three bits, which means that the sum cannot exceed the value 7.

Addition of positive numbers is the same for all three number representations. It is actually the same as the addition of unsigned numbers discussed in section 5.2. But there are significant differences when negative numbers are involved. The difficulties that arise become apparent if we consider operands with different combinations of signs.

Sign-and-Magnitude Addition

If both operands have the same sign, then the addition of sign-and-magnitude numbers is simple. The magnitudes are added, and the resulting sum is given the sign of the operands. However, if the operands have opposite signs, the task becomes more complicated. Then it is necessary to subtract the smaller number from the larger one. This means that logic circuits that compare and subtract numbers are also needed. We will see shortly that it is possible to perform subtraction without the need for this circuitry. For this reason, the sign-and-magnitude representation is not used in computers.

1's Complement Addition

An obvious advantage of the 1's complement representation is that a negative number is generated simply by complementing all bits of the corresponding positive number. Figure 5.9 shows what happens when two numbers are added. There are four cases to consider in terms of different combinations of signs. As seen in the top half of the figure, the computation of $5 + 2 = 7$ and $(-5) + 2 = (-3)$ is straightforward; a simple addition of the operands gives the correct result. Such is not the case with the other two possibilities. Computing $5 + (-2) = 3$ produces the bit vector 10010. Because we are dealing with four-bit numbers, there is a carry-out from the sign-bit position. Also, the four bits of the result represent the number 2 rather than 3, which is a wrong result. Interestingly, if we take the carry-out from the sign-bit position and add it to the result in the least-significant bit position, the new result is the correct sum of 3. This correction is indicated in blue in the figure. A similar situation arises when adding $(-5) + (-2) = (-7)$. After the initial addition the result is wrong because the four bits of the sum are 0111, which represents +7 rather than −7. But again, there is a carry-out from the sign-bit position, which can be used to correct the result by adding it in the LSB position, as shown in Figure 5.9.

The conclusion from these examples is that the addition of 1's complement numbers may or may not be simple. In some cases a correction is needed, which amounts to an extra addition that must be performed. Consequently, the time needed to add two 1's complement numbers may be twice as long as the time needed to add two unsigned numbers.

2's Complement Addition

Consider the same combinations of numbers as used in the 1's complement example. Figure 5.10 indicates how the addition is performed using 2's complement numbers. Adding $5 + 2 = 7$ and $(-5) + 2 = (-3)$ is straightforward. The computation $5 + (-2) = 3$ generates the correct four bits of the result, namely 0011. There is a carry-out from the sign-bit position, which we can simply ignore. The fourth case is $(-5) + (-2) = (-7)$. Again, the four bits of the result, 1001, give the correct sum (−7). In this case also, the carry-out from the sign-bit position can be ignored.

(+5)	0 1 0 1	(−5)	1 0 1 0
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+7)	0 1 1 1	(−3)	1 1 0 0
(+5)	0 1 0 1	(−5)	1 0 1 0
+ (−2)	+ 1 1 0 1	+ (−2)	+ 1 1 0 1
(+3)	1 0 0 1 0	(−7)	1 0 1 1 1
	↳ 1		↳ 1
	0 0 1 1		1 0 0 0

Figure 5.9 Examples of 1's complement addition.

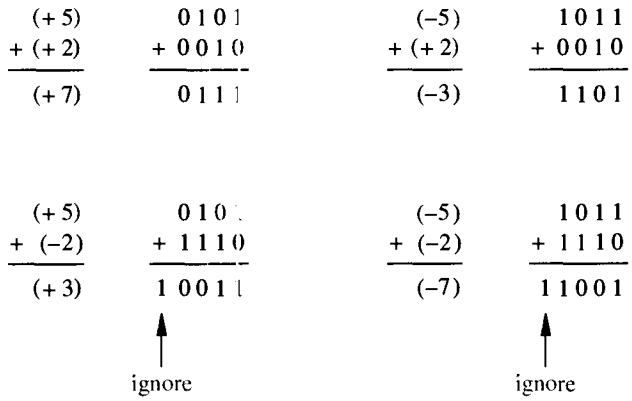


Figure 5.10 Examples of 2's complement addition.

As illustrated by these examples, the addition of 2's complement numbers is very simple. When the numbers are added, the result is always correct. If there is a carry-out from the sign-bit position, it is simply ignored. Therefore, the addition process is the same, regardless of the signs of the operands. It can be performed by an adder circuit, such as the one shown in Figure 5.6. Hence the 2's complement notation is highly suitable for the implementation of addition operations. We will now consider its use in subtraction operations.

2's Complement Subtraction

The easiest way of performing subtraction is to negate the subtrahend and add it to the minuend. This is done by finding the 2's complement of the subtrahend and then performing the addition. Figure 5.11 illustrates the process. The operation $5 - (+2) = 3$ involves finding the 2's complement of +2, which is 1110. When this number is added to 0101, the result is $0011 = (+3)$ and a carry-out from the sign-bit position occurs, which is ignored. A similar situation arises for $(-5) - (+2) = (-7)$. In the remaining two cases there is no carry-out, and the result is correct.

As a graphical aid to visualize the addition and subtraction examples in Figures 5.10 and 5.11, we can place all possible four-bit patterns on a modulo-16 circle given in Figure 5.12. If these bit patterns represented unsigned integers, they would be numbers 0 to 15. If they represent 2's-complement integers, then the numbers range from −8 to +7, as shown. The addition operation is done by stepping in the clockwise direction by the magnitude of the number to be added. For example, $-5 + 2$ is determined by starting at 1011 $(= -5)$ and moving clockwise two steps, giving the result 1101 $(= -3)$. Subtraction is performed by stepping in the counterclockwise direction. For example, $-5 - (+2)$ is determined by starting at 1011 and moving counterclockwise two steps, which gives 1001 $(= -7)$.

The key conclusion of this section is that the subtraction operation can be realized as the addition operation, using a 2's complement of the subtrahend, regardless of the signs of

(+5)	0101		0101
−(+2)	−0010	⇨	+1110
(+3)			10011
			↑ ignore
(−5)	1011		1011
−(+2)	−0010	⇨	+1110
(−7)			11001
			↑ ignore
(+5)	0101		0101
−(−2)	−1110	⇨	+0010
(+7)			0111
(−5)	1011		1011
−(−2)	−1110	⇨	+0010
(−3)			1101

Figure 5.11 Examples of 2's complement subtraction.

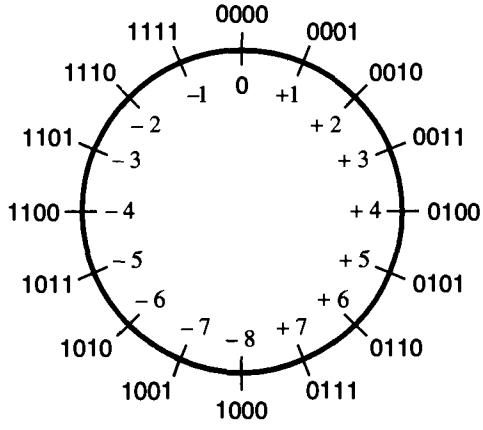


Figure 5.12 Graphical interpretation of four-bit 2's complement numbers.

the two operands. Therefore, it should be possible to use the same adder circuit to perform both addition and subtraction.

5.3.3 ADDER AND SUBTRACTOR UNIT

The only difference between performing addition and subtraction is that for subtraction it is necessary to use the 2's complement of one operand. Let X and Y be the two operands, such that Y serves as the subtrahend in subtraction. From section 5.3.1 we know that a 2's complement can be obtained by adding 1 to the 1's complement of Y. Adding 1 in the least-significant bit position can be accomplished simply by setting the carry-in bit c_0 to 1. A 1's complement of a number is obtained by complementing each of its bits. This could be done with NOT gates, but we need a more flexible circuit where we can use the true value of Y for addition and its complement for subtraction.

In section 5.2 we explained that two-input XOR gates can be used to choose between true and complemented versions of an input value, under the control of the other input. This idea can be applied in the design of the adder/subtractor unit as follows. Assume that there exists a control signal that chooses whether addition or subtraction is to be performed. Let this signal be called $\overline{\text{Add}}$/Sub. Also, let its value be 0 for addition and 1 for subtraction. To indicate this fact, we placed a bar over Add. This is a commonly used convention, where a bar over a name means that the action specified by the name is to be taken if the control signal has the value 0. Now let each bit of Y be connected to one input of an XOR gate, with the other input connected to $\overline{\text{Add}}$/Sub. The outputs of the XOR gates represent Y if $\overline{\text{Add}}$/Sub = 0, and they represent the 1's complement of Y if $\overline{\text{Add}}$/Sub = 1. This leads to the circuit in Figure 5.13. The main part of the circuit is an n-bit adder, which can be implemented using the ripple-carry structure of Figure 5.6. Note that the control signal $\overline{\text{Add}}$/Sub is also

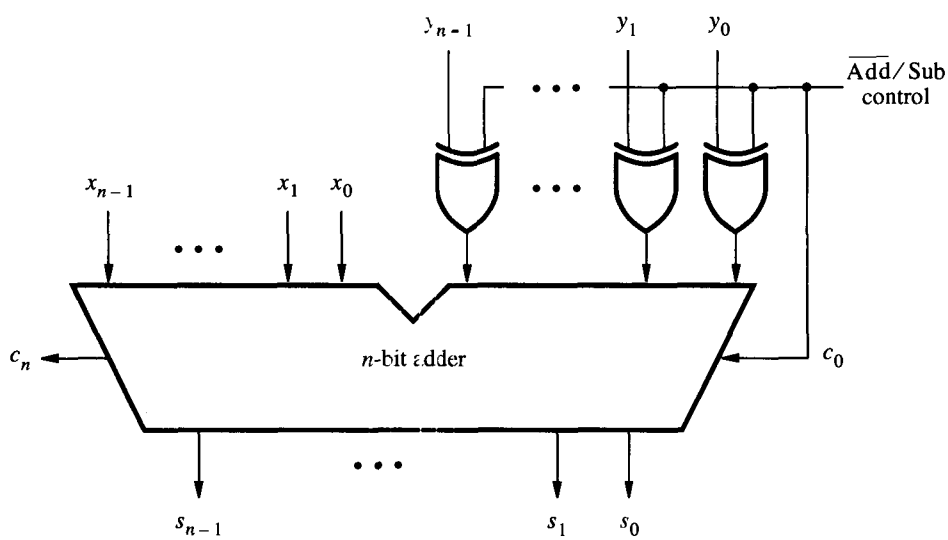


Figure 5.13 Adder/subtractor unit.

connected to the carry-in c_0. This makes $c_0 = 1$ when subtraction is to be performed, thus adding the 1 that is needed to form the 2's complement of Y. When the addition operation is performed, we will have $c_i = 0$.

The combined adder/subtractor unit is a good example of an important concept in the design of logic circuits. It is useful to design circuits to be as flexible as possible and to exploit common portions of circuits for as many tasks as possible. This approach minimizes the number of gates needed to implement such circuits, and it reduces the wiring complexity substantially.

5.3.4 RADIX-COMPLEMENT SCHEMES

The idea of performing a subtraction operation by addition of a complement of the subtrahend is not restricted to binary numbers. We can gain some insight into the workings of the 2's complement scheme by considering its counterpart in the decimal number system. Consider the subtraction of two-digit decimal numbers. Computing a result such as $74 - 33 = 41$ is simple because each digit of the subtrahend is smaller than the corresponding digit of the minuend; therefore, no borrow is needed in the computation. But computing $74 - 36 = 38$ is not as simple because a borrow is needed in subtracting the least-significant digit. If a borrow occurs, the computation becomes more complicated.

Suppose that we restructure the required computation as follows

$$\begin{aligned} 74 - 36 &= 74 + 100 - 100 - 36 \\ &= 74 + (100 - 36) - 100 \end{aligned}$$

Now two subtractions are needed. Subtracting 36 from 100 still involves borrows. But noting that $100 = 99 + 1$, these borrows can be avoided by writing

$$\begin{aligned} 74 - 36 &= 74 + (99 + 1 - 36) - 100 \\ &= 74 + (99 - 36) + 1 - 100 \end{aligned}$$

The subtraction in parentheses does not require borrows; it is performed by subtracting each digit of the subtrahend from 9. We can see a direct correlation between this expression and the one used for 2's complement, as reflected in the circuit in Figure 5.13. The operation $(99 - 36)$ is analogous to complementing the subtrahend Y to find its 1's complement, which is the same as subtracting each bit from 1. Using decimal numbers, we find the *9's complement* of the subtrahend by subtracting each digit from 9. In Figure 5.13 we add the carry-in of 1 to form the 2's complement of Y. In our decimal example we perform $(99 - 36) + 1 = 64$. Here 64 is the 10's complement of 36. For an n-digit decimal number, N, its *10's complement*, K_{10}, is defined as $K_{10} = 10^n - N$, while its 9's complement, K_9, is $K_9 = (10^n - 1) - N$.

Thus the required subtraction $(74 - 36)$ can be performed by addition of the 10's complement of the subtrahend, as in

$$\begin{aligned} 74 - 36 &= 74 + 64 - 100 \\ &= 138 - 100 \\ &= 38 \end{aligned}$$

The subtraction 138 − 100 is trivial because it means that the leading digit in 138 is simply deleted. This is analogous to ignoring the carry-out from the circuit in Figure 5.13, as discussed for the subtraction examples in Figure 5.11.

Example 5.1 Suppose that A and B are n-digit decimal numbers. Using the above 10's-complement approach, B can be subtracted from A as follows:

$$A - B = A + (10^n - B) - 10^n$$

If $A \geq B$, then the operation $A + (10^n - B)$ produces a carry-out of 1. This carry is equivalent to 10^n; hence it can be simply ignored.

But if $A < B$, then the operation $A + (10^n - B)$ produces a carry-out of 0. Let the result obtained be M, so that

$$A - B = M - 10^n$$

We can rewrite this as

$$10^n - (B - A) = M$$

The left side of this equation is the 10's complement of $(B - A)$. The 10's complement of a positive number represents a negative number that has the same magnitude. Hence M correctly represents the negative value obtained from the computation $A - B$ when $A < B$. This concept is illustrated in the examples that follow.

Example 5.2 When dealing with binary signed numbers we use 0 in the left-most bit position to denote a positive number and 1 to denote a negative number. If we wanted to build hardware that operates on signed decimal numbers, we could use a similar approach. Let 0 in the left-most digit position denote a positive number and let 9 denote a negative number. Note that 9 is the 9's complement of 0 in the decimal system, just as 1 is the 1's complement of 0 in the binary system.

Thus, using three-digit signed numbers, $A = 045$ and $B = 027$ are positive numbers with magnitudes 45 and 27, respectively. The number B can be subtracted from A as follows

$$\begin{aligned} A - B &= 045 - 027 \\ &= 045 + 1000 - 1000 - 027 \\ &= 045 + (999 - 027) + 1 - 1000 \\ &= 045 + 972 + 1 - 1000 \\ &= 1018 - 1000 \\ &= 018 \end{aligned}$$

This gives the correct answer of +18.

Next consider the case where the minuend has lower value than the subtrahend. This is illustrated by the computation

$$\begin{aligned} B - A &= 027 - 045 \\ &= 027 + 1000 - 1000 - 045 \\ &= 027 + (999 - 045) + 1 - 1000 \\ &= 027 + 954 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

From this expression it appears that we still need to perform the subtraction $982 - 1000$. But as seen in Example 5.1, this can be rewritten as

$$\begin{aligned} 982 &= 1000 + B - A \\ &= 1000 - (A - B) \end{aligned}$$

Therefore, 982 is the negative number that results when forming the 10's complement of $(A - B)$. From the previous computation we know that $(A - B) = 018$, which denotes +18. Thus the signed number 982 is the 10's complement representation of -18, which is the required result.

---

et $C = 955$ and $D = 973$; hence the values of C and D are -45 and -27, respectively. The number D can be subtracted from C as follows

$$\begin{aligned} C - D &= 955 - 973 \\ &= 955 + 1000 - 1000 - 973 \\ &= 955 + (999 - 973) + 1 - 1000 \\ &= 955 + 026 + 1 - 1000 \\ &= 982 - 1000 \end{aligned}$$

The number 982 is the 10's complement representation of -18, which is the correct result.

Consider now the case $D - A$, where $D = 973$ and $A = 045$:

$$\begin{aligned} D - A &= 973 - 045 \\ &= 973 + 1000 - 1000 - 045 \\ &= 973 + (999 - 045) + 1 - 1000 \\ &= 973 + 954 + 1 - 1000 \\ &= 1928 - 1000 \\ &= 928 \end{aligned}$$

The result 928 is the 10's complement representation of -72.

These examples illustrate that signed numbers can be subtracted without using a subtraction operation that involves borrows. The only subtraction needed is in forming the 9's complement of the subtrahend, in which case each digit is simply subtracted from 9.

Thus a circuit that forms the 9's complement, combined with a normal adder circuit, will suffice for both addition and subtraction of decimal signed numbers. A key point is that the hardware needs to deal only with n digits if n-digit numbers are used. Any carry that may be generated from the left-most digit position is simply ignored.

The concept of subtracting a number by adding its radix-complement is general. If the radix is r, then the r's complement, K_r, of an n-digit number, N, is determined as $K_r = r^n - N$. The $(r-1)$'s complement, K_{r-1}, is defined as $K_{r-1} = (r^n - 1) - N$; it is computed simply by subtracting each digit of N from the value $(r-1)$. The $(r-1)$'s complement is referred to as the *diminished-radix complement*. Circuits for forming the $(r-1)$'s complements are simpler than those for general subtraction that involves borrows. The circuits are particularly simple in the binary case, where the 1's complement requires just inverting each bit.

Example 5.4 In Figure 5.11 we illustrated the subtraction operation on binary numbers given in 2's-complement representation. Consider the computation $(+5) - (+2) = (+3)$, using the approach discussed above. Each number is represented by a four-bit pattern. The value 2^4 is represented as 10000. Then

$$\begin{aligned} 0101 - 0010 &= 0101 + (10000 - 0010) - 10000 \\ &= 0101 + (1111 - 0010) + 1 - 10000 \\ &= 0101 + 1101 + 1 - 10000 \\ &= 10011 - 10000 \\ &= 0011 \end{aligned}$$

Because $5 > 2$, there is a carry from the fourth bit position. It represents the value 2^4, denoted by the pattern 10000.

Example 5.5 Consider now the computation $(+2) - (+5) = (-3)$, which gives

$$\begin{aligned} 0010 - 0101 &= 0010 + (10000 - 0101) - 10000 \\ &= 0010 + (1111 - 0101) + 1 - 10000 \\ &= 0010 + 1010 + 1 - 10000 \\ &= 1101 - 10000 \end{aligned}$$

Because $2 < 5$, there is no carry from the fourth bit position. The answer, 1101, is the 2's-complement representation of -3. Note that

$$\begin{aligned} 1101 &= 10000 + 0010 - 0101 \\ &= 10000 - (0101 - 0010) \\ &= 10000 - 0011 \end{aligned}$$

indicating that 1101 is the 2's complement of 0011 (+3).

Example 5.6

Finally, consider the case where the subtrahend is a negative number. The computation $(+5) - (-2) = (+7)$ is done as follows

$$\begin{aligned} 0101 - 1110 &= 0101 + (10000 - 1110) - 10000 \\ &= 0101 + (1111 - 1110) + 1 - 10000 \\ &= 0101 + 0001 + 1 - 10000 \\ &= 0111 - 10000 \end{aligned}$$

While $5 > (-2)$, the pattern 1110 is greater than the pattern 0101 when the patterns are treated as unsigned numbers. Therefore, there is no carry from the fourth bit position. The answer 0111 is the 2's complement representation of +7. Note that

$$\begin{aligned} 0111 &= 10000 + 0101 - 1110 \\ &= 10000 - (1110 - 0101) \\ &= 10000 - 1001 \end{aligned}$$

and 1001 represents −7.

5.3.5 ARITHMETIC OVERFLOW

The result of addition or subtraction is supposed to fit within the significant bits used to represent the numbers. If n bits are used to represent signed numbers, then the result must be in the range -2^{n-1} to $2^{n-1} - 1$. If the result does not fit in this range, then we say that *arithmetic overflow* has occurred. To ensure the correct operation of an arithmetic circuit, it is important to be able to detect the occurrence of overflow.

Figure 5.14 presents the four cases where 2's-complement numbers with magnitudes of 7 and 2 are added. Because we are using four-bit numbers, there are three significant bits, b_{2-0}. When the numbers have opposite signs, there is no overflow. But if both numbers have the same sign, the magnitude of the result is 9, which cannot be represented with just three significant bits; therefore, overflow occurs. The key to determining whether overflow

(+7)	0 1 1 1	(−7)	1 0 0 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+9)	1 0 0 1	(−5)	1 0 1 1
	$c_4 = 0$		$c_4 = 0$
	$c_3 = 1$		$c_3 = 0$
(+7)	0 1 1 1	(−7)	1 0 0 1
+ (−2)	+ 1 1 1 0	+ (−2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(−9)	1 0 1 1 1
	$c_4 = 1$		$c_4 = 1$
	$c_3 = 1$		$c_3 = 0$

Figure 5.14 Examples for determination of overflow.

occurs is the carry-out from the MSB position, called c_3 in the figure, and from the sign-bit position, called c_4. The figure indicates that overflow occurs when these carry-outs have different values, and a correct sum is produced when they have the same value. Indeed, this is true in general for both addition and subtraction of 2's-complement numbers. As a quick check of this statement, consider the examples in Figure 5.10 where the numbers are small enough so that overflow does not occur in any case. In the top two examples in the figure, there is a carry-out of 0 from both sign and MSB positions. In the bottom two examples, there is a carry-out of 1 from both positions. Therefore, for the examples in Figures 5.10 and 5.14, the occurrence of overflow is detected by

$$\begin{aligned} \text{Overflow} &= c_3\bar{c}_4 + \bar{c}_3 c_4 \\ &= c_3 \oplus c_4 \end{aligned}$$

For n-bit numbers we have

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Thus the circuit in Figure 5.13 can be modified to include overflow checking with the addition of one XOR gate.

5.3.6 PERFORMANCE ISSUES

When buying a digital system, such as a computer, the buyer pays particular attention to the performance that the system is expected to provide and to the cost of acquiring the system. Superior performance usually comes at a higher cost. However, a large increase in performance can often be achieved at a modest increase in cost. A commonly used indicator of the value of a system is its *price/performance ratio.*

The addition and subtraction of numbers are fundamental operations that are performed frequently in the course of a computation. The speed with which these operations are performed has a strong impact on the overall performance of a computer. In light of this, let us take a closer look at the speed of the adder/subtractor unit in Figure 5.13. We are interested in the largest delay from the time the operands X and Y are presented as inputs, until the time all bits of the sum S and the final carry-out, c_n, are valid. Most of this delay is caused by the n-bit adder circuit. Assume that the adder is implemented using the ripple-carry structure in Figure 5.6 and that each full-adder stage is the circuit in Figure 5.4*c*. The delay for the carry-out signal in this circuit, Δt, is equal to two gate delays. From section 5.2.2 we know that the final result of the addition will be valid after a delay of $n\Delta t$, which is equal to $2n$ gate delays. In addition to the delay in the ripple-carry path, there is also a delay in the XOR gates that feed either the true or complemented value of Y to the adder inputs. If this delay is equal to one gate delay, then the total delay of the circuit in Figure 5.13 is $2n+1$ gate delays. For a large n, say $n = 32$ or $n = 64$, the delay would lead to unacceptably poor performance. Therefore, it is important to find faster circuits to perform addition.

The speed of any circuit is limited by the longest delay along the paths through the circuit. In the case of the circuit in Figure 5.13, the longest delay is along the path from the y_i input, through the XOR gate and through the carry circuit of each adder stage. The

longest delay is often referred to as the *critical-path delay*, and the path that causes this delay is called the *critical path.*

5.4 FAST ADDERS

The performance of a large digital system is dependent on the speed of circuits that form its various functional units. Obviously, better performance can be achieved using faster circuits. This can be accomplished by using superior (usually newer) technology in which the delays in basic gates are reduced. But it can also be accomplished by changing the overall structure of a functional unit, which may lead to even more impressive improvement. In this section we will discuss an alternative for implementation of an n-bit adder, which substantially reduces the time needed to add numbers.

5.4.1 CARRY-LOOKAHEAD ADDER

To reduce the delay caused by the effect of carry propagation through the ripple-carry adder, we can attempt to evaluate quickly for each stage whether the carry-in from the previous stage will have a value 0 or 1. If a correct evaluation can be made in a relatively short time, then the performance of the complete adder will be improved.

From Figure 5.4*b* the carry-out function for stage i can be realized as

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

If we factor this expression as

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

then it can be written as

$$c_{i+1} = g_i + p_i c_i \qquad \textbf{[5.3]}$$

where

$$g_i = x_i y_i$$
$$p_i = x_i + y_i$$

The function g_i is equal to 1 when both inputs x_i and y_i are equal to 1, regardless of the value of the incoming carry to this stage, c_i. Since in this case stage i is guaranteed to generate a carry-out, g is called the *generate* function. The function p_i is equal to 1 when at least one of the inputs x_i and y_i is equal to 1. In this case a carry-out is produced if $c_i = 1$. The effect is that the carry-in of 1 is propagated through stage i; hence p_i is called the *propagate* function.

Expanding the expression 5.3 in terms of stage $i - 1$ gives

$$\begin{aligned} c_{i+1} &= g_i + p_i(g_{i-1} + p_{i-1} c_{i-1}) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} c_{i-1} \end{aligned}$$

The same expansion for other stages, ending with stage 0, gives

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_2 p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \quad \textbf{[5.4]}$$

This expression represents a two-level AND-OR circuit in which c_{i+1} is evaluated very quickly. An adder based on this expression is called a *carry-lookahead adder.*

To appreciate the physical meaning of expression 5.4, it is instructive to consider its effect on the construction of a fast adder in comparison with the details of the ripple-carry adder. We will do so by examining the detailed structure of the two stages that add the least-significant bits, namely, stages 0 and 1. Figure 5.15 shows the first two stages of a ripple-carry adder in which the carry-out functions are implemented as indicated in expression 5.3. Each stage is essentially the circuit from Figure 5.4*c* except that an extra OR gate is used (which produces the p_i signal), instead of an AND gate because we factored the sum-of-products expression for c_{i+1}.

The slow speed of the ripple-carry adder is caused by the long path along which a carry signal must propagate. In Figure 5.15 the critical path is from inputs x_0 and y_0 to the output c_2. It passes through five gates, as highlighted in blue. The path in other stages of an n-bit adder is the same as in stage 1. Therefore, the total delay along the critical path is $2n + 1$.

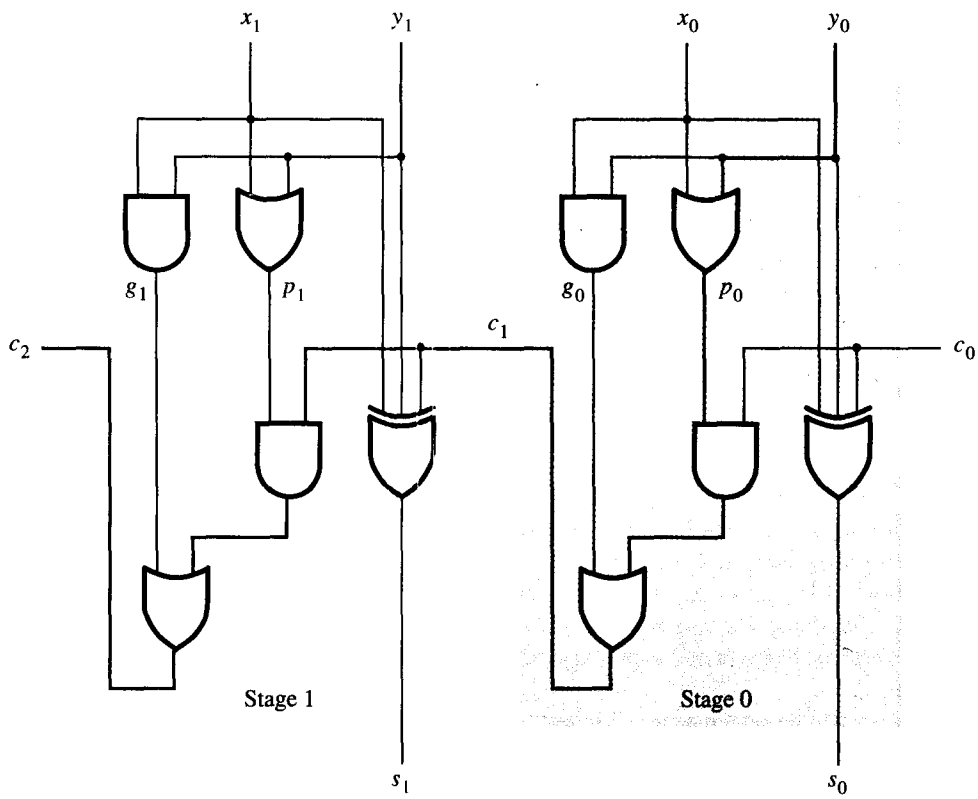


Figure 5.15 A ripple-carry adder based on Expression 5.3.

Figure 5.16 gives the first two stages of the carry-lookahead adder, using expression 5.4 to implement the carry-out functions. Thus

$$c_1 = g_0 + p_0c_0$$
$$c_2 = g_1 + p_1g_0 + p_1p_0c_0$$

The critical path for producing the c_2 signal is highlighted in blue. In this circuit, c_2 is produced just as quickly as c_1, after a total of three gate delays. Extending the circuit to n bits, the final carry-out signal c_n would also be produced after only three gate delays because expression 5.4 is just a large two-level (AND-OR) circuit.

The total delay in the n-bit carry-lookahead adder is four gate delays. The values of all g_i and p_i signals are determined after one gate delay. It takes two more gate delays to evaluate all carry signals. Finally, it takes one more gate delay (XOR) to generate all sum bits. The key to the good performance of the adder is quick evaluation of carry signals.

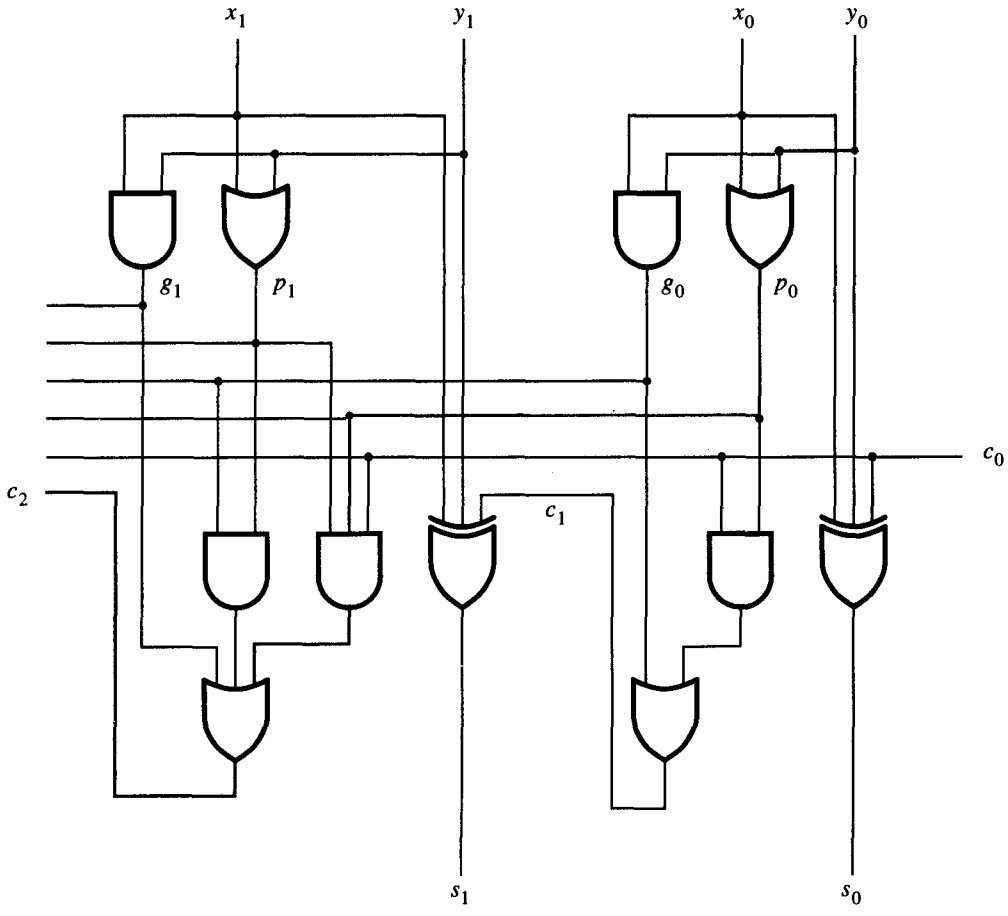


Figure 5.16 The first two stages of a carry-lookahead adder.

The complexity of an n-bit carry-lookahead adder increases rapidly as n becomes larger. To reduce the complexity, we can use a *hierarchical* approach in designing large adders. Suppose that we want to design a 32-bit adder. We can divide this adder into 4 eight-bit blocks, such that bits b_{7-0} are block 0, bits b_{15-8} are block 1, bits b_{23-16} are block 2, and bits b_{31-24} are block 3. Then we can implement each block as an eight-bit carry-lookahead adder. The carry-out signals from the four blocks are c_8, c_{16}, c_{24}, and c_{32}. Now we have two possibilities. We can connect the four blocks as four stages in a ripple-carry adder. Thus while carry-lookahead is used within each block, the carries ripple between the blocks. This circuit is illustrated in Figure 5.17.

Instead of using a ripple-carry approach between blocks, a faster circuit can be designed in which a second-level carry-lookahead is performed to produce quickly the carry signals between blocks. The structure of this "hierarchical carry-lookahead adder" is shown in Figure 5.18. Each block in the top row includes an eight-bit carry-lookahead adder, based on generate signals, g_i, and propagate signals, p_i, for each stage in the block, as discussed before. However, instead of producing a carry-out signal from the most-significant bit of the block, each block produces generate and propagate signals for the entire block. Let G_j and P_j denote these signals for each block j. Now G_j and P_j can be used as inputs to a second-level carry-lookahead circuit, at the bottom of Figure 5.18, which evaluates all carries between blocks. We can derive the block generate and propagate signals for block 0 by examining the expression for c_8

$$c_8 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

The last term in this expression specifies that, if all eight propagate functions are 1, then the carry-in c_0 is propagated through the entire block. Hence

$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0$$

The rest of the terms in the expression for c_8 represent all other cases when the block produces a carry-out. Thus

$$G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \cdots + p_7p_6p_5p_4p_3p_2p_1g_0$$

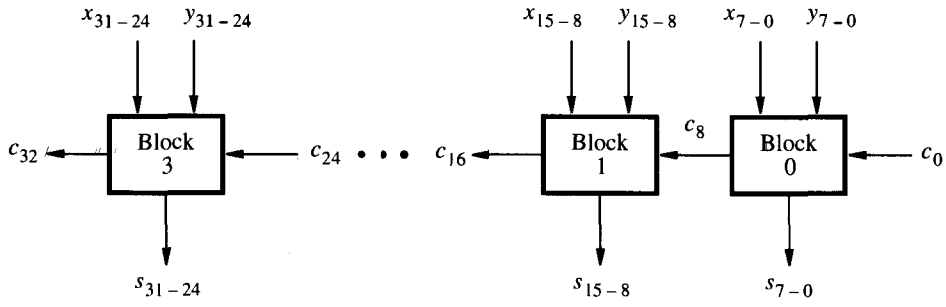


Figure 5.17 A hierarchical carry-lookahead adder with ripple-carry between blocks.

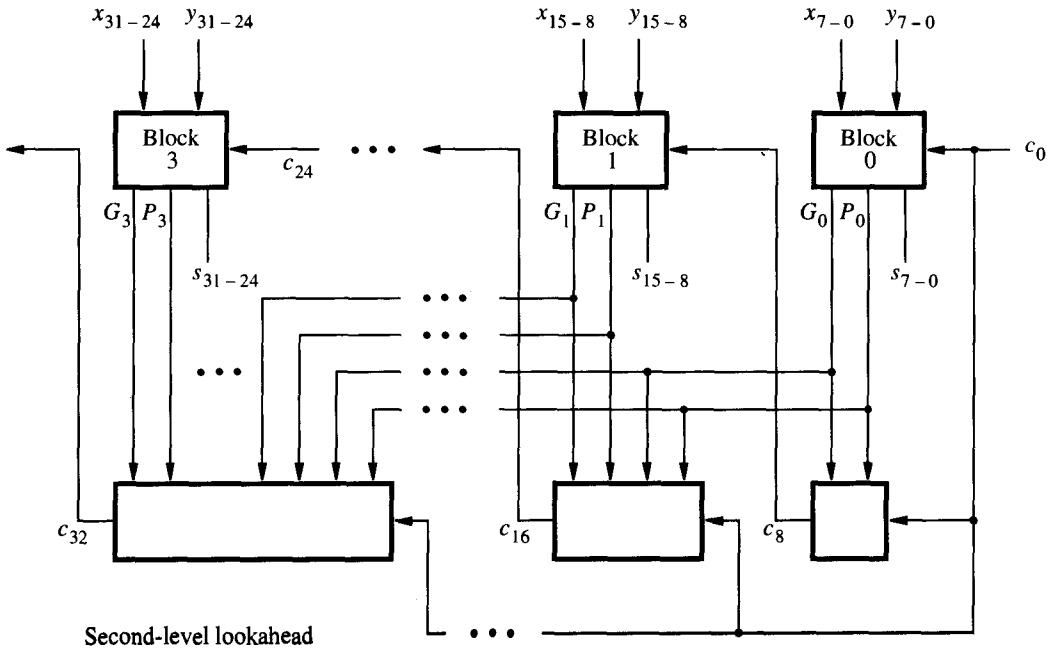


Figure 5.18 A hierarchical carry-lookahead adder.

The expression for c_8 in the hierarchical adder is given by

$$c_8 = G_0 + P_0 c_0$$

For block 1 the expressions for G_1 and P_1 have the same form as for G_0 and P_0 except that each subscript i is replaced by $i + 8$. The expressions for G_2, P_2, G_3, and P_3 are derived in the same way. The expression for the carry-out of block 1, c_{16}, is

$$\begin{aligned} c_{16} &= G_1 + P_1 c_8 \\ &= G_1 + P_1 G_0 + P_1 P_0 c_0 \end{aligned}$$

Similarly, the expressions for c_{24} and c_{32} are

$$c_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_{32} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

Using this scheme, it takes two more gate delays to produce the carry signals c_8, c_{16}, and c_{24} than the time needed to generate the G_j and P_j functions. Therefore, since G_j and P_j require three gate delays, c_8, c_{16}, and c_{24} are available after five gate delays. The time needed to add two 32-bit numbers involves these five gate delays plus two more to produce the internal carries in blocks 1, 2, and 3, plus one more gate delay (XOR) to generate each sum bit. This gives a total of eight gate delays.

In section 5.3.5 we determined that it takes $2n+1$ gate delays to add two numbers using a ripple-carry adder. For 32-bit numbers this implies 65 gate delays. It is clear that the carry-lookahead adder offers a large performance improvement. The trade-off is much greater complexity of the required circuit.

Technology Considerations

The preceding delay analysis assumes that gates with any number of inputs can be used. We know from Chapters 3 and 4 that the technology used to implement the gates limits the fan-in to a rather small number of inputs. Therefore the reality of fan-in constraints must be taken into account. To illustrate this problem, consider the expressions for the first eight carries:

$$\begin{aligned}
c_1 &= g_0 + p_0c_0 \\
c_2 &= g_1 + p_1g_0 + p_1p_0c_0 \\
&\vdots \\
c_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4g_3 + p_7p_6p_5p_4p_3g_2 \\
&\quad + p_7p_6p_5p_4p_3p_2g_1 + p_7p_6p_5p_4p_3p_2p_1g_0 + p_7p_6p_5p_4p_3p_2p_1p_0c_0
\end{aligned}$$

Suppose that the maximum fan-in of the gates is four inputs. Then it is impossible to implement all of these expressions with a two-level AND-OR circuit. The biggest problem is c_8, where one of the AND gates requires nine inputs; moreover, the OR gate also requires nine inputs. To meet the fan-in constraint, we can rewrite the expression for c_8 as

$$\begin{aligned}
c_8 = (g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4) + [(p_7p_6p_5p_4)(g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0)] \\
+ (p_7p_6p_5p_4)(p_3p_2p_1p_0)c_0
\end{aligned}$$

To implement this expression we need 11 AND gates and three OR gates. The propagation delay in generating c_8 consists of one gate delay to develop all g_i and p_i, two gate delays to produce the sum-of-products terms in parentheses, one gate delay to form the product term in square brackets, and one delay for the final ORing of terms. Hence c_8 is valid after five gate delays, rather than the three gates delays that would be needed without the fan-in constraint.

Because fan-in limitations reduce the speed of the carry-lookahead adder, some devices that are characterized by low fan-in include dedicated circuitry for implementation of fast adders. Examples of such devices include FPGAs whose logic blocks are based on lookup tables.

Before we leave the topic of the carry-lookahead adder, we should consider an alternative implementation of the structure in Figure 5.16. The same functionality can be achieved by using the circuit in Figure 5.19. In this case stage 0 is implemented using the circuit of Figure 5.5 in which 2 two-input XOR gates are used to generate the sum bit, rather than having 1 three-input XOR gate. The output of the first XOR gate can also serve as the propagate signal p_0. Thus the corresponding OR gate in Figure 5.16 is not needed. Stage 1 is constructed using the same approach.

The circuits in Figures 5.16 and 5.19 require the same number of gates. But is one of them better in some way? The answer must be sought by considering the specific aspects of the technology that is used to implement the circuits. If a CPLD or an FPGA is used, such as

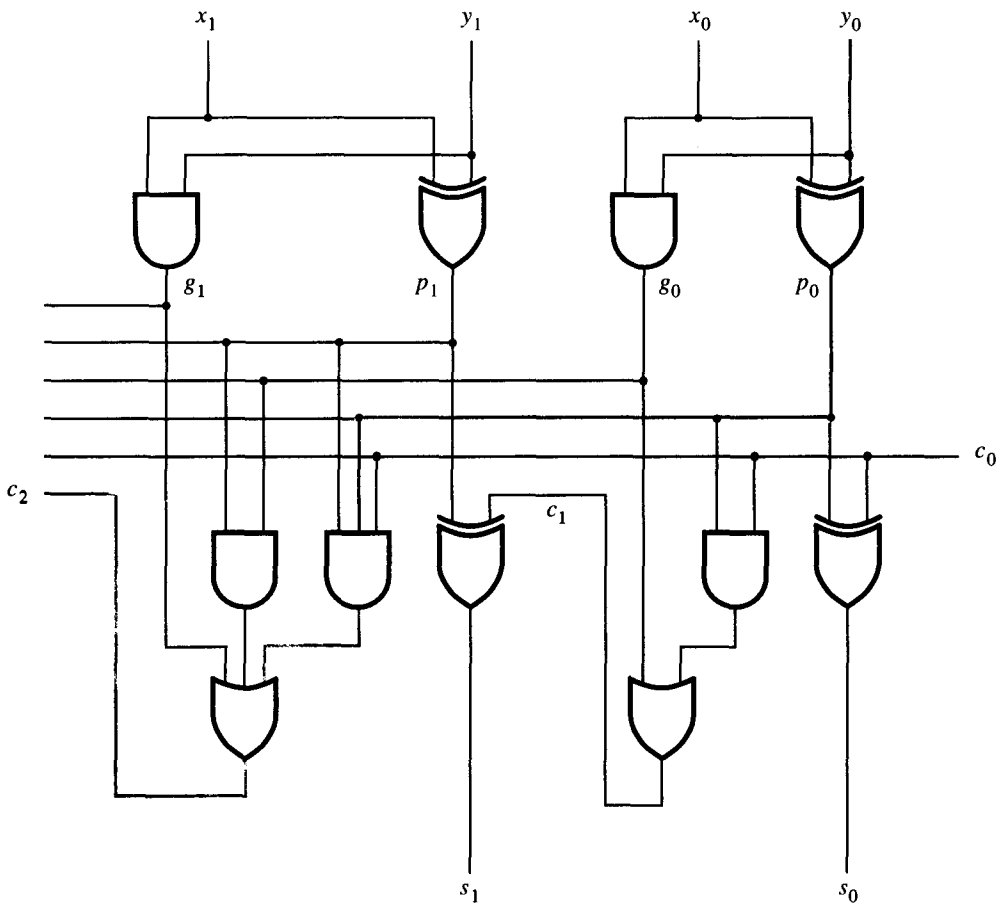


Figure 5.19 An alternative design for a carry-lookahead adder.

those in Figures 3.33 and 3.39, then it does not matter which circuit is chosen. A three-input XOR function can be realized by one macrocell in the CPLD, using the sum-of-products expression

$$s_i = x_i\bar{y}_i\bar{c}_i + \bar{x}_i y_i \bar{c}_i + \bar{x}_i\bar{y}_i c_i + x_i y_i c_i$$

because the macrocell allows for implementation of four product terms.

In the FPGA any three-input function can be implemented in a single logic cell; hence it is easy to realize a three-input XOR. However, suppose that we want to build a carry-lookahead adder on a custom chip. If the XOR gate is constructed using the approach discussed in section 3.9.1, then a three-input XOR would actually be implemented using 2 two-input XOR gates, as we have done for the sum bits in Figure 5.19. Therefore, if the first XOR gate realizes the function $x_i \oplus y_i$, which is also the propagate function p_i, then it is obvious that the alternative in Figure 5.19 is more attractive. The important point of this

discussion is that optimization of logic circuits may depend on the target technology. The CAD tools take this fact into account.

The carry-lookahead adder is a well-known concept. There exist standard chips that implement a portion of the carry-lookahead circuitry. They are called *carry-lookahead generators*. CAD tools often include predesigned subcircuits for adders, which designers can use to design larger units.

5.5 DESIGN OF ARITHMETIC CIRCUITS USING CAD TOOLS

In this section we show how the arithmetic circuits can be designed by using CAD tools. Two different design methods are discussed: using schematic capture and using Verilog code.

5.5.1 DESIGN OF ARITHMETIC CIRCUITS USING SCHEMATIC CAPTURE

An obvious way to design an arithmetic circuit via schematic capture is to draw a schematic that contains the necessary logic gates. For example, to create an n-bit adder, we could first draw a schematic that represents a full-adder. Then an n-bit ripple-carry adder could be created by drawing a higher-level schematic that connects together n *instances* of the full-adder. A hierarchical schematic created in this manner would look like the circuit shown in Figure 5.6. We could also use this methodology to create an adder/subtractor circuit, such as the circuit depicted in Figure 5.13.

The main problem with this approach is that it is cumbersome, especially when the number of bits is large. This problem is even more apparent if we consider creating a schematic for a carry-lookahead adder. As shown in section 5.4.1, the carry circuitry in each stage of the carry-lookahead adder becomes increasingly more complex. Hence it is necessary to draw a separate schematic for each stage of the adder. A better approach for creating arithmetic circuits via schematic capture is to use predefined subcircuits.

We mentioned in section 2.9.1 that schematic capture tools provide a library of graphical symbols that represent basic logic gates. These gates are used to create schematics of relatively simple circuits. In addition to basic gates, most schematic capture tools also provide a library of commonly used circuits, such as adders. Each circuit is provided as a module that can be imported into a schematic and used as part of a larger circuit. In some CAD systems the modules are referred to as *macrofunctions*, or *megafunctions*.

There are two main types of macrofunctions: technology dependent and technology independent. A *technology-dependent macrofunction* is designed to suit a specific type of chip. For example, in section 5.4.1 we described an expression for a carry-lookahead adder that was designed to meet a fan-in constraint of four-input gates. A macrofunction that implements this expression would be technology specific. A *technology-independent macrofunction* can be implemented in any type of chip. A macrofunction for an adder that represents different circuits for different types of chips is a technology-independent macrofunction.

A good example of a library of macrofunctions is the *Library of Parameterized Modules (LPM)* that is included as part of the MAX+plusII CAD system. Each module in the library is technology independent. Also, each module is *parameterized*, which means that it can be used in a variety of ways. For example, the LPM library includes an n-bit adder module, named *lpm_add_sub*. The number of bits, n, is set by a parameter called LPM_WIDTH.

A schematic containing the *lpm_add_sub* module is given in Figure 5.20. The module has several associated parameters, which are configured by using the CAD tools. The two most important parameters for the purposes of our discussion are named LPM_WIDTH and LPM_REPRESENTATION. As mentioned above LPM_WIDTH specifies the number of bits in the adder. The LPM_REPRESENTATION parameter specifies whether signed or unsigned integers are used. This affects only the part of the module that determines when arithmetic overflow occurs, as discussed in section 5.3.5. For the schematic shown, LPM_WIDTH = 16, and signed numbers are used. The module can perform addition or subtraction, determined by the input *add_sub*. Thus the module represents an adder/subtractor circuit, such as the one shown in Figure 5.13.

The numbers to be added by the *lpm_add_sub* module are connected to the terminals called *dataa*[ ] and *datab*[ ]. The square brackets in these names mean that they represent multibit numbers. In the schematic *dataa*[ ] and *datab*[ ] are connected to the 16-bit input signals $X[15..0]$ and $Y[15..0]$. The meaning of the syntax $X[15..0]$ is that the signal X represents 16 bits, named $X[15]$, $X[14]$, ..., $X[0]$. The *lpm_add_sub* module produces the sum on the terminal called *result*[ ], which is connected to the output $S[15..0]$. Figure 5.20 also shows that the carry-in is set to *Gnd*. The *add_sub* input is connected to logic high, which configures the module as an adder. (Note that the logic high voltage is called V_{CC} in the schematic displayed by the CAD tool. We use V_{DD} to denote this voltage level in the book, but V_{CC} is also used in the literature.) Finally, the carry-out from the module is connected to the signal *Cout*, and the overflow terminal is connected to *Overflow*.

Besides the convenience in creating the schematic, the *lpm_add_sub* macrofunction has another advantage. It allows the logic synthesis algorithms provided by the CAD system to

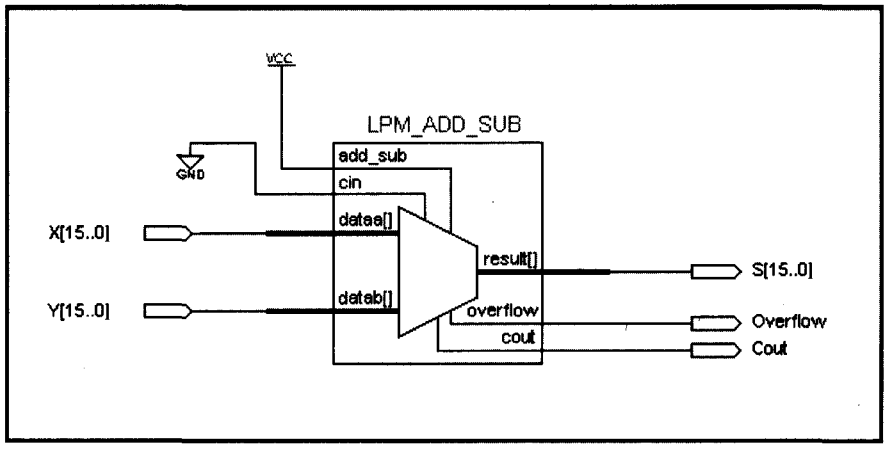


Figure 5.20 Schematic using an LPM adder/subtractor module.

generate different circuits for different chips and different optimization goals. For instance, if the speed of the adder is not crucial, but it is important to keep the cost of the circuit low, then the CAD system may generate a ripple-carry adder to implement the *lpm_add_sub* module. But if speed is important, then a fast adder, such as the carry-lookahead adder, can be generated. Some chips, such as FPGAs, include special-purpose circuitry for implementing fast adders, as we mentioned in section 5.4.1. Using a technology-independent macrofunction allows the CAD system to generate a circuit that makes use of such special-purpose circuitry.

Examples of the results obtained when synthesizing a circuit from the schematic for implementation in an FPGA are shown in Figures 5.21 and 5.22. In Figure 5.21 the logic synthesis was performed with the goal of minimizing the cost of the circuit, as opposed to the speed. This results in a ripple-carry adder. The figure shows a screen capture of the timing simulator. The values of the 16-bit signals X, Y, and S are shown in the simulation output as hexadecimal numbers. At the beginning of the simulation, both X and Y are set to 0000. After 50 ns, Y is changed to 0001. The correct sum is generated quickly, after about 13 ns, because the carry needs to ripple through only one stage of the adder in this case. The next change in the inputs occurs at 150 ns, when X changes to 3FFF. To produce the new sum, which is 4000, the adder must wait for its carry signals to ripple from the first stage to the last stage. This is seen in the simulation output as a sequence of rapid changes in the value of S, eventually settling at the correct sum. Observe that the simulator's reference line, the heavy vertical line in the figure, shows that the correct sum is produced 204.5 ns from the start of the simulation. Because the change in inputs happened at 150 ns, the adder takes $204.5 - 150 = 54.5$ ns to compute the sum. At 250 ns, X changes to 7FFF, which causes the sum to be 8000. This sum is too large for a positive 16-bit signed number; hence *Overflow* is set to 1 to indicate the arithmetic overflow.

Figure 5.22 shows the same simulation input for the schematic in Figure 5.20 but with the synthesized circuit optimized for speed. In this case the adder produced by the CAD tools makes use of the dedicated carry-logic circuitry in the FPGA. In this adder it takes about 23 ns to produce the sum.

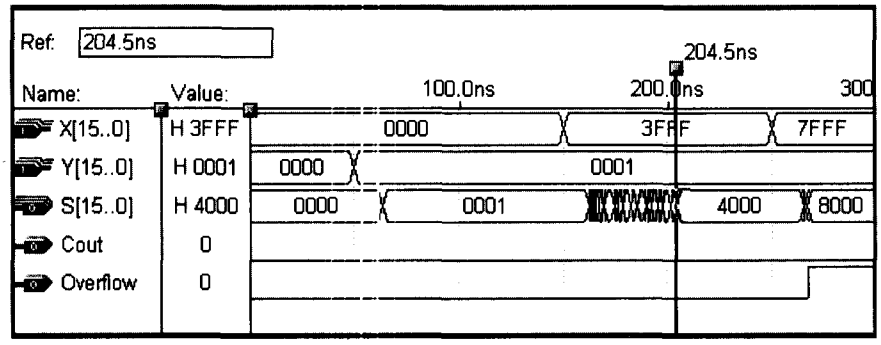


Figure 5.21 Simulation results for the LPM adder optimized for cost.

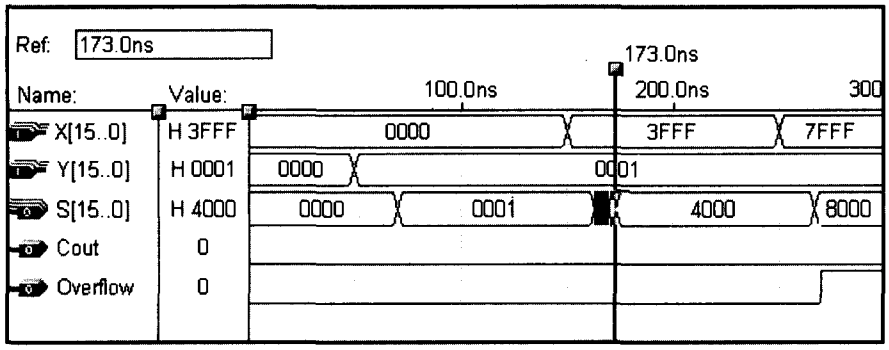


Figure 5.22 Simulation results for the LPM adder optimized for speed.

5.5.2 DESIGN OF ARITHMETIC CIRCUITS USING VERILOG

We said in section 5.5.1 that an obvious way to create an *n*-bit adder is to draw a hierarchical schematic that contains *n* full-adders. This approach can also be followed by using Verilog, by first creating a Verilog module for a full-adder and then defining a higher-level module that uses *n instances* of the full-adder. As a first attempt at designing arithmetic circuits using Verilog, we will show how to write the hierarchical code for a ripple-carry adder.

Suppose that we wish to implement the full-adder circuit given in Figure 5.4*c*, which has the inputs *Cin*, *x*, and *y*, and produces the outputs *s* and *Cout*. One way of specifying this circuit in Verilog is to use the gate level primitives as shown in Figure 5.23. Each of the three AND gates in the circuit is defined by a separate statement. Verilog allows combining such statements into a single statement as shown in Figure 5.24. In this case, commas are used to separate the definition of each AND gate.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  xor (s, x, y, Cin);
  and (z1, x, y);
  and (z2, x, Cin);
  and (z3, y, Cin);
  or (Cout, z1, z2, z3);

endmodule
```

Figure 5.23 Verilog code for the full-adder using gate level primitives.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  xor (s, x, y, Cin);
  and (z1, x, y),
      (z2, x, Cin),
      (z3, y, Cin);
  or (Cout, z1, z2, z3);

endmodule
```

Figure 5.24 Another version of Verilog code from Figure 5.23.

Another possibility is to use functional expressions as indicated in Figure 5.25. The XOR operation is denoted by the ^ sign. Again, it is possible to combine the two continuous assignment statements into a single statement as shown in Figure 5.26.

Both of the above approaches result in the same full-adder circuit being synthesized. We can now create a separate Verilog module for the ripple-carry adder, which instantiates

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  assign s = x ^ y ^ Cin;
  assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 5.25 Verilog code for the full-adder using continuous assignment.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  assign s = x ^ y ^ Cin,
         Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 5.26 Another version of Verilog code from Figure 5.25.

the *fulladd* module as a subcircuit. One method of doing this is shown in Figure 5.27. The module comprises the code for a four-bit ripple-carry adder, named *adder4*. One of the four-bit numbers to be added is represented by the four signals x_3, x_2, x_1, x_0, and the other number is represented by y_3, y_2, y_1, y_0. The sum is represented by s_3, s_2, s_1, s_0. The circuit incorporates a carry input, *carryin*, into the least-significant bit position and a carry output, *carryout*, from the most-significant bit position.

The four-bit adder in Figure 5.27 is described using four *instantiation* statements. Each statement begins with the name of the module, *fulladd*, that is being instantiated. Next comes an *instance name*, which can be any legal Verilog name. The instance names must be unique. The least-significant stage in the adder is named *stage0* and the most-significant stage is *stage3*. The signal names in the *adder4* module that are to be connected to each input and output port on the *fulladd* module are then listed. These signals are listed in the same order as in the *fulladd* module, namely the order *Cin*, *x*, *y*, *s*, *Cout*.

The signal names associated with each instance of the *fulladd* module implicitly specify how the full-adders are connected together. For example, the carry-out of the *stage0* instance is connected to the carry-in of the *stage1* instance. The synthesized circuit has the same structure as the one shown in Figure 5.6. The *fulladd* module may be included in the same Verilog source code file as the *adder4* module, as we have done in Figure 5.27, but it may also comprise a separate file. In the latter case, the location of the file *fulladd* has to be indicated to the compiler.

```
module adder4 (carryin, x3, x2, x1, x0, y3, y2, y1, y0, s3, s2, s1, s0, carryout);
   input carryin, x3, x2, x1, x0, y3, y2, y1, y0;
   output s3, s2, s1, s0, carryout;

   fulladd stage0 (carryin, x0, y0, s0, c1);
   fulladd stage1 (c1, x1, y1, s1, c2);
   fulladd stage2 (c2, x2, y2, s2, c3);
   fulladd stage3 (c3, x3, y3, s3, carryout);

endmodule

module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

Figure 5.27 Verilog code for a four-bit adder.

5.5.3 USING VECTORED SIGNALS

In Figure 5.27 each of the four-bit inputs and the four-bit output of the adder is represented using single-bit signals. A more convenient approach is to use multibit signals, called *vectors*, to represent the numbers. Just as a number is represented in a logic circuit as signals on multiple wires, it can be represented in Verilog code as a multibit vector. An example of an input vector is

```
input [3:0] W;
```

This statement defines W to be a four-bit vector. Its individual bits can be referred to by using an index value in square brackets. Thus, the most-significant bit (MSB) is referred to as $W[3]$ and the least-significant bit (LSB) is $W[0]$. A two-bit vector that consists of the two middle bits of W is denoted as $W[2:1]$. The symbol W refers to the entire vector.

Using vectors we can specify the four-bit adder as shown in Figure 5.28. In addition to the input vectors X and Y, and output vector S, we chose to define the carry signals between the full-adder stages as a three-bit vector $C[3:1]$. Note that the carry into *stage0* is still called *carryin*, while the carry from *stage3* is called *carryout*.

For specifying signals that are neither inputs nor outputs of a module, which are used only for internal connections within the module, Verilog provides the **wire** type. In the *adder4* module we need three internal carry signals, which are defined as a three-bit vector in the statement

```
wire [3:1] C;
```

In Figure 5.28, signal C[1] is used to connect the carry output of the full-adder in stage 0 to the carry input of the full-adder in stage 1. Similarly, C[2] and C[3] are used to connect the other stages of the adder.

The vector specification gives the bit width in square brackets, as in $W[3:0]$. The bit width is specified using the index of the MSB first and the LSB last. Hence, $W[3]$ is MSB

```
module adder4 (carryin, X, Y, S, carryout);
  input carryin;
  input [3:0] X, Y;
  output [3:0] S;
  output carryout;
  wire [3:1] C;

  fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
  fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
  fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
  fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Figure 5.28 A four-bit adder using vectors.

and $W[0]$ is LSB. A reverse ordering can also be used. For example, $Z[0:3]$ defines a four-bit vector in which $Z[0]$ is its MSB and $Z[3]$ is its LSB. The terminology MSB and LSB is natural when vectors are used to represent numbers. In other cases, the bit-select index merely identifies a particular bit in a vector.

5.5.4 USING A GENERIC SPECIFICATION

The approach in designing a ripple-carry adder presented in Figure 5.28 is rather restrictive because the resulting circuit is of a predetermined size of four bits. A similar adder that could add 32-bit numbers would require Verilog code with 32 instances of the full-adder subcircuit defined in separate statements. From the designer's point of view, it is preferable to define a module that could be used to implement an adder of any size, where the size may be given as a parameter.

Verilog allows the use of general parameters that can be given a specific value as desired. For example, an n-bit vector representing a number may be given as $W[n-1:0]$. If n is defined in the Verilog statement

parameter n = 4;

then the bit range of W is [3:0].

The ripple-carry adder in Figure 5.6 can be described using the logic expressions

$$s_k = x_k \oplus y_k \oplus c_k$$
$$c_{k+1} = x_k y_k + x_k c_k + y_k c_k$$

for $k = 0, 1, \ldots, n-1$. Instead of instantiating full-adders as in Figure 5.28, these expressions can be used in Verilog to specify the desired adder.

Figure 5.29 shows Verilog code that defines an n-bit adder. The inputs X and Y, and the output sum S, are declared to be n-bit vectors. To simplify the use of carry signals in the adder circuit, we defined a vector C that has $n + 1$ bits. Bit $C[0]$ is the carry into the LSB position, while $C[n]$ is the carry from the MSB position. Hence $C[0]$ = *carryin* and *carryout* = $C[n]$ in terms of the n-bit adder.

To specify the repetitive structure of the ripple-carry adder, Figure 5.29 introduces the Verilog **for** statement. Like the **if-else** statement introduced in section 2.10.2, the **for** statement is a procedural statement that must be placed inside an **always** block, as shown in the figure. As explained in section 2.10.2, any signal that is assigned a value by a statement within an **always** block must retain this value until it is again re-evaluated by changes in the sensitivity variables given in the **always** statement. Such signals are declared to be of **reg** type; they are *carryout*, S, and C signals in Figure 5.29. The sensitivity variables are X, Y, and *carryin*.

In our example, the **for** loop consists of two statements delineated by **begin** and **end**. These statements define the sum and carry functions for the adder stage that corresponds to the value of the loop variable k. The range of k is from 0 to $n - 1$ and its value is incremented by 1 for each pass through the loop. The Verilog **for** statement is similar to the **for** loop in the C programming language. However, the C operators ++ and −− do not exist in Verilog, hence incrementing or decrementing of the loop variable must be given as

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   reg [n-1:0] S;
   reg carryout;
   reg [n:0] C;
   integer k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure 5.29 A generic specification of a ripple-carry adder.

$k = k+1$ or $k = k-1$, rather than k++ or k−−. Note that k is declared to be an integer and it is used to control the number of iterations of the **for** loop; it does not represent a physical connection in the circuit. The effect of the **for** loop is to repeat the statements inside the loop for each loop iteration. For instance, if k were set to 2 in this example, then the **for** loop would be equivalent to the four statements

```
S[0] = X[0] ^ Y[0] ^ C[0];
C[1] = (X[0] & Y[0]) | (X[0] & C[0]) | (Y[0] & C[0]);
S[1] = X[1] ^ Y[1] ^ C[1];
C[2] = (X[1] & Y[1]) | (X[1] & C[1]) | (Y[1] & C[1]);
```

Since the value of n is 32, as declared in the **parameter** statement, the code in the figure implements a 32-bit adder.

5.5.5 NETS AND VARIABLES IN VERILOG

A logic circuit is modeled in Verilog by a collection of interconnected logic elements and/or by procedural statements that describe its behavior. Connections between logic elements are defined using *nets*. Signals produced by procedural statements are referred to as *variables*.

Nets

A net represents a node in a circuit. Nets can be of different types. For synthesis purposes the only important nets are of **wire** type, which we used in section 5.5.3. A **wire** connects an output of one logic element in a circuit to an input of another logic element. It can be a *scalar* that represents a single connection or a *vector* that represents multiple connections. For example, in Figure 5.27, carry signals c_3, c_2, and c_1 are scalars that model the connections between the full-adder modules. The specific connections are defined by the way the full-adder modules are instantiated. In Figure 5.28, the same carry signals are defined as a three-bit vector C. Observe that in Figure 5.27 the carry signals are not explicitly declared to be of **wire** type. The reason is that nets do not have to be declared in the code because Verilog syntax assumes that all signals are nets by default. Of course, the code in the figure would also be correct if we include in it the declaration

```
wire c3, c2, c1;
```

In Figure 5.28 it is necessary to declare the existence of vector C; otherwise, the Verilog compiler would not be able to determine that $C[3]$, $C[2]$, and $C[1]$ are the constituent signals of C. Since these signals are nets, the vector C is declared to be of **wire** type.

Another type of nets are the **tri** nets. This keyword declares a net to be of tri-state type, which indicates that the signals may have the high-impedance value, z, in addition to logic 0 and 1. Nets of **tri** type are treated in the same way as the **wire** type. They are used only to enhance the readability of code that includes tri-state gates. We will discuss the application of **tri** nets in section 7.14 in Chapter 7.

Variables

Verilog provides *variables* to allow a circuit to be described in terms of its behavior. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten by a subsequent assignment statement. There are two types of variables: **reg** and **integer**. As mentioned in section 2.10.2, all signals that are assigned a value using procedural statements must be declared as variables by using the **reg** or **integer** keywords. The scalar *carryout* and the vectors S and C in Figure 5.29 are examples of the **reg** type. The loop variable k in the same figure illustrates the **integer** type. It serves as a loop index. Such variables are useful for describing a circuit's behavior; they do not usually correspond directly to signals in the resulting circuit.

Further discussion of nets and variables is given in section A.6 in Appendix A.

5.5.6 ARITHMETIC ASSIGNMENT STATEMENTS

Arithmetic operations are used so often that it is convenient to have them incorporated directly into a hardware description language. Verilog implements such operations using arithmetic assignment statements and vectors. If the following vectors are defined

```
input  [n-1:0] X, Y;
output [n-1:0] S;
```

then, the arithmetic assignment statement

$$S = X + Y;$$

represents an n-bit adder.

In addition to the + operator, which is used for addition, Verilog also provides other arithmetic operators. The Verilog operators are discussed fully in sections 6.6.5 of Chapter 6 and A.7 of Appendix A. The complete Verilog code that includes the preceding statement is given in Figure 5.30. Since there is a single statement in the **always** block, it is not necessary to include the **begin** and **end** delimiters. This code defines a circuit that generates the n sum bits, but it does not include the carry-out signal. Also, it does not provide the arithmetic overflow signal. One way in which these signals can be added is given in Figure 5.31.

The carry-out from the MSB position, $n-1$, can be derived from the values of x_{n-1}, y_{n-1}, and s_{n-1} by using the expression

$$carryout = x_{n-1}y_{n-1} + x_{n-1}\bar{s}_{n-1} + y_{n-1}\bar{s}_{n-1}$$

(Note that this is just a normal logic expression in which the + sign represents the OR operation.) The expression for arithmetic overflow was defined in section 5.3.4 as $c_n \oplus c_{n-1}$. In our case, c_n corresponds to *carryout*, but there is no direct way of accessing c_{n-1}, which is the carry from bit-position $n-2$. It can be shown that the expression $x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$ corresponds to c_{n-1}, thus

$$overflow = carryout \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$$

The reader should verify the validity of these expressions as an exercise.

Another way of including the carry-out and overflow signals is shown in Figure 5.32. The (n+1)-bit vector named *Sum* is used. The extra bit, *Sum*[n], becomes the carry-out from bit-position $n-1$ in the adder. The statement used to assign the sum of *X*, *Y*, and *carryin* to the *Sum* signal uses an unusual syntax. The meaning of the terms in brackets, namely {1'b0, X} and {1'b0, Y}, is that a 0 is concatenated on the left of the n-bit vectors *X* and *Y* to create (n+1)-bit vectors. In Verilog the { , } operator is called the *concatenate*

```
module addern (carryin, X, Y, S);
  parameter n = 32;
  input carryin;
  input [n-1:0] X, Y;
  output [n-1:0] S;
  reg [n-1:0] S;

  always @(X or Y or carryin)
    S = X + Y + carryin;

endmodule
```

Figure 5.30 Specification of an n-bit adder using arithmetic assignment.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout, overflow;
   reg [n–1:0] S;
   reg carryout, overflow;

   always @(X or Y or carryin)
   begin
      S = X + Y + carryin;
      carryout = (X[n–1] & Y[n–1]) | (X[n–1] & ~S[n–1]) | (Y[n–1] & ~S[n–1]);
      overflow = carryout ^ X[n–1] ^ Y[n–1] ^ S[n–1];
   end

endmodule
```

Figure 5.31 An n-bit adder with carry-out and overflow signals.

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n–1:0] X, Y;
   output [n–1:0] S;
   output carryout, overflow;
   reg [n–1:0] S;
   reg carryout, overflow;
   reg [n:0] Sum;

   always @(X or Y or carryin)
   begin
      Sum = {1'b0, X} + {1'b0, Y} + carryin;
      S = Sum[n–1:0];
      carryout = Sum[n];
      overflow = carryout ^ X[n–1] ^ Y[n–1] ^ S[n–1];
   end

endmodule
```

Figure 5.32 An alternative specification of n-bit adder with carry-out and overflow signals.

operator. If A is an m-bit vector and B is a k-bit vector, then {A, B} creates an $(m+k)$-bit vector comprising A as its most-significant m bits and B as its least-significant k bits. The notation 1'b0 represents a one-bit binary number that has the value 0. The reason that the concatenate operator is used in Figure 5.32 is to cause *Sum*[n] to be equivalent to the carry from bit position $n-1$. In effect, we created $x_n = y_n = 0$ so that

$$Sum[n] = 0 + 0 + c_{n-1}$$

This example is useful because it provides a simple introduction to the concept of concatenation. But we could have written simply

```
Sum = X + Y + carryin;
```

Because *Sum* is an $(n+1)$-bit vector, the summation will be performed as if X and Y were $(n+1)$-bit vectors in which 0s are padded on the left.

Another detail to observe from the figure is the statement

```
S = Sum[n-1:0];
```

This assigns the lower n bits of *Sum* to the output sum S. The next statement assigns the carry-out from the addition, *Sum*[n], to the output signal *carryout*.

We show the code in Figures 5.31 and 5.32 to illustrate some features of Verilog in the context of adder design. In general, a given design task can be performed using different approaches, as we will see throughout the book. Let us attempt another specification of the n-bit adder. In Figure 5.32 we use an $(n+1)$-bit vector, *Sum*, as an intermediate signal needed to produce the n bits of S and the carry-out from the adder stage $n-1$. This requires two Verilog statements that extract the desired bits from *Sum*. We showed how concatenation can be used to pad a 0 to vectors X and Y, but pointed out that this is not necessary because a vector is automatically padded with 0s if it is involved in an arithmetic operation that produces a result of greater bit size. We can use concatenation more effectively on the left side of the addition statement by concatenating *carryout* to the S vector so that

```
{carryout, S} = X + Y + carryin;
```

Then there is no need for the *Sum* signal and the Verilog code is simplified as indicated in Figure 5.33. Since both figures, 5.32 and 5.33, describe the same behavior of the adder, the Verilog compiler is likely to generate the same circuit for either figure. The code in Figure 5.33 is simpler and more elegant.

Note that the same approach can be used to specify a full-adder circuit, as shown in Figure 5.34. Unlike the specifications in Figures 5.23 to 5.26, which define the structure of the full-adder in terms of basic logic operations, in this case the code describes its behavior and the Verilog compiler implements the suitable details using the target technology.

When the Verilog compiler translates the code, it generates an adder circuit to implement the + operator. When using the MAX+plusII CAD system, the adder used by the compiler is actually the *lpm_add_sub* module shown in Figure 5.20. The compiler automatically sets the parameters for the module so that it represents an n-bit adder corresponding to the value of n declared in a **parameter** statement. The logic synthesis algorithms can generate different circuits for different goals, such as cost or speed optimization, as discussed in

```
module addern (carryin, X, Y, S, carryout, overflow);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout, overflow;
   reg [n-1:0] S;
   reg carryout, overflow;

   always @(X or Y or carryin)
   begin
      {carryout, S} = X + Y + carryin;
      overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
   end

endmodule
```

Figure 5.33 Simplified complete specification of *n*-bit adder.

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;
   reg s, Cout;

   always @(x or y or Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

Figure 5.34 Behavioral specification of a full-adder.

section 4.12. For completeness, we should also mention that the *lpm_add_sub* module can be directly instantiated in Verilog code, in a similar way that the *fulladd* component was instantiated in Figure 5.28. An example of this is given in section A.12, in Appendix A.

5.5.7 REPRESENTATION OF NUMBERS IN VERILOG CODE

Numbers can be given as constants in Verilog code. They can be given as binary (b), octal (o), hexadecimal (h), or decimal (d) numbers. Their size can be either fixed or unspecified. For sized numbers the format is

<size_in_bits>'<radix_identifier><significant_digits>

The size is a decimal number that gives the number of bits needed, the radix is identified using letters b, o, h, or d, and the digits are given in the notation of the radix used. For example, the decimal number 2217 can be represented using 12 bits as follows

```
12'b100010101001
12'o4251
12'h8A9
12'd2217
```

Unsized numbers are given without specifying the size. For example, the decimal number 278 may be given as

```
'b100010110
'o426
'h116
278
```

For decimal numbers it is not necessary to give the radix identifier d. When an unsized number is used in an expression the Verilog compiler gives it a certain size, which is typically the same as the size of the other operand(s) in the expression.

Negative numbers are represented by placing the minus sign in front. Thus, if −5 is specified as −4'b101, it will be interpreted as a four-bit 2's-complement of 5, which is 1011.

The specified size may exceed the number of bits that are actually needed to represent a given number. In this case, the final representation is padded to the left to yield the required size. However, if there are more digits than can fit into the number of bits given as the size, the extra digits will be ignored.

To improve readability, it is possible to use the underscore character. Instead of writing 12'b100010101001, it is easier to visualize the same number as 12'b1000_1010_1001.

When numbers are used in the design of a circuit they become signals carried by wires and manipulated by logic gates. So far, we have implicitly assumed that signals in a digital circuit have values 0 and 1. In Chapter 3 we saw that a wire may be in a high-impedance state, z. In Chapter 4 we sometimes used x to denote a signal that could be either 0 or 1. These possibilities are included in Verilog, where any signal can have four possible values: 0, 1, z, and x. The symbol x is used to denote an unknown logic value, which could be any of 0, 1, or z. A constant used in a Verilog design may be given as 8'hz3, which is the same as 8'bzzzz0011. The specification 8'bx or 8'hx denotes an unknown eight-bit number. The z and x values cannot be used with the decimal radix. When padding occurs, it is normally 0s that are padded to the left. But if the leftmost digit given is z or x, then these values are padded to the left. We should note that either lower- or upper-case letters can be used for the signal values z and x, as well as the radix identifiers b, o, h, and d.

Numbers represented by vectors of different bit sizes can be used in arithmetic operations. Suppose that A is an eight-bit vector and B is a four-bit vector. Then the statement

```
C = A + B;
```

will generate an eight-bit sum vector C. The result will be correct if B is a positive number.

However, if B is a negative number expressed in 2's complement representation, the result will be incorrect because 0s will be padded on the left to make B an eight-bit vector for the purpose of the addition operation. The value of a positive number does not change if 0s are appended as the most-significant bits; the value of a negative number does not change if 1s are appended as the most-significant bits. Such replication of the sign bit is called *sign extension*. Therefore, for correct operation it is necessary to use a sign-extended version of B, which can be accomplished with concatenation in the statement

```
C = A + {4{B[3]}, B};
```

The notation 4{B[3]} denotes that the bit B[3] is replicated four times; it is equivalent to writing {B[3], B[3], B[3], B[3]}. This is referred to as the *replication* operator, which is discussed in section 6.6.5 in Chapter 6. If we want to generate a carry-out signal from bit position 7, then we could adopt the approach in Figure 5.32 by using the statement

```
C = {1'b0, A} + {1'b0, 4{B[3]}, B};
```

5.6 MULTIPLICATION

Before we discuss the general issue of multiplication, we should note that a binary number, B, can be multiplied by 2 simply by adding a zero to the right of its least-significant bit. This effectively moves all bits of B to the left, and we say that B is *shifted* left by one bit position. Thus if $B = b_{n-1}b_{n-2} \cdots b_1b_0$, then $2 \times B = b_{n-1}b_{n-2} \cdots b_1b_0 0$. (We have already used this fact in section 5.2.3.) Similarly, a number is multiplied by 2^k by shifting it left by k bit positions. This is true for both unsigned and signed numbers.

We should also consider what happens if a binary number is shifted right by k bit positions. According to the positional number representation, this action divides the number by 2^k. For unsigned numbers the shifting amounts to adding k zeros to the left of the most-significant bit. For example, if B is an unsigned number, then $B \div 2 = 0b_{n-1}b_{n-2} \cdots b_2b_1$. Note that bit b_0 is lost when shifting to the right. For signed numbers it is necessary to preserve the sign. This is done by shifting the bits to the right and filling from the left with the value of the sign bit. Hence if B is a signed number, then $B \div 2 = b_{n-1}b_{n-1}b_{n-2} \cdots b_2b_1$. For instance, if $B = 011000 = (24)_{10}$, then $B \div 2 = 001100 = (12)_{10}$ and $B \div 4 = 000110 = (6)_{10}$. Similarly, if $B = 101000 = -(24)_{10}$, then $B \div 2 = 110100 = -(12)_{10}$ and $B \div 4 = 111010 = -(6)_{10}$. The reader should also observe that the smaller the positive number, the more 0s there are to the left of the first 1, while for a negative number there are more 1s to the left of the first 0.

Now we can turn our attention to the general task of multiplication. Two binary numbers can be multiplied using the same method as we use for decimal numbers. We will focus our discussion on multiplication of unsigned numbers. Figure 5.35*a* shows how multiplication is performed manually, using four-bit numbers. Each multiplier bit is examined from right to left. If a bit is equal to 1, an appropriately shifted version of the multiplicand is added to form a *partial product*. If the multiplier bit is equal to 0, then nothing is added. The sum of all shifted versions of the multiplicand is the desired product. Note that the product occupies eight bits.

Multiplicand M	(14)	1 1 1 0
Multiplier Q	(11)	× 1 0 1 1
		1 1 1 0
		1 1 1 0
		0 0 0 0
		1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

(a) Multiplication by hand

Multiplicand M	(11)	1 1 1 0
Multiplier Q	(14)	× 1 0 1 1
Partial product 0		1 1 1 0
		+ 1 1 1 0
Partial product 1		1 0 1 0 1
		+ 0 0 0 0
Partial product 2		0 1 0 1 0
		+ 1 1 1 0
Product P	(154)	1 0 0 1 1 0 1 0

(b) Multiplication for implementation in hardware

Figure 5.35 Multiplication of unsigned numbers.

The same scheme can be used to design a multiplier circuit. We will stay with four-bit numbers to keep the discussion simple. Let the multiplicand, multiplier, and product be denoted as $M = m_3m_2m_1m_0$, $Q = q_3q_2q_1q_0$, and $P = p_7p_6p_5p_4p_3p_2p_1p_0$, respectively. One simple way of implementing the multiplication scheme is to use a sequential approach, where an eight-bit adder is used to compute partial products. As a first step, the bit q_0 is examined. If $q_0 = 1$, then M is added to the initial partial product, which is initialized to 0. If $q_0 = 0$, then 0 is added to the partial product. Next q_1 is examined. If $q_1 = 1$, then the value $2 \times M$ is added to the partial product. The value $2 \times M$ is created simply by shifting M one bit position to the left. Similarly, $4 \times M$ is added to the partial product if $q_2 = 1$, and $8 \times M$ is added if $q_3 = 1$. We will show in Chapter 10 how such a circuit may be implemented.

This sequential approach leads to a relatively slow circuit, primarily because a single eight-bit adder is used to perform all additions needed to generate the partial products and the final product. A much faster circuit can be obtained if multiple adders are used to compute the partial products.

5.6.1 ARRAY MULTIPLIER FOR UNSIGNED NUMBERS

Figure 5.35*b* indicates how multiplication may be performed by using multiple adders. In each step a four-bit adder is used to compute the new partial product. Note that as the computation progresses, the least-significant bits are not affected by subsequent additions; hence they can be passed directly to the final product, as indicated by blue arrows. Of course, these bits are a part of the partial products as well.

A fast multiplier circuit can be designed using an array structure that is similar to the organization in Figure 5.35*b*. Consider a 4 × 4 example, where the multiplicand and multiplier are $M = m_3m_2m_1m_0$ and $Q = q_3q_2q_1q_0$, respectively. The partial product 0, $PP0 = pp0_3\ pp0_2\ pp0_1\ pp0_0$, can be generated using the AND of q_0 with each bit of M. Thus

$$PP0 = m_3q_0\ \ m_2q_0\ \ m_1q_0\ \ m_0q_0$$

Partial product 1, $PP1$, is generated using the AND of q_1 with M and adding it to $PP0$ as follows

$PP0$:		0	$pp0_3$	$pp0_2$	$pp0_1$	$pp0_0$
	+	m_3q_1	m_2q_1	m_1q_1	m_0q_0	0
$PP1$:		$pp1_4$	$pp1_3$	$pp1_2$	$pp1_1$	$pp1_0$

Similarly, partial product 2, $PP2$, is generated using the AND of q_2 with M and adding to $PP1$, and so on.

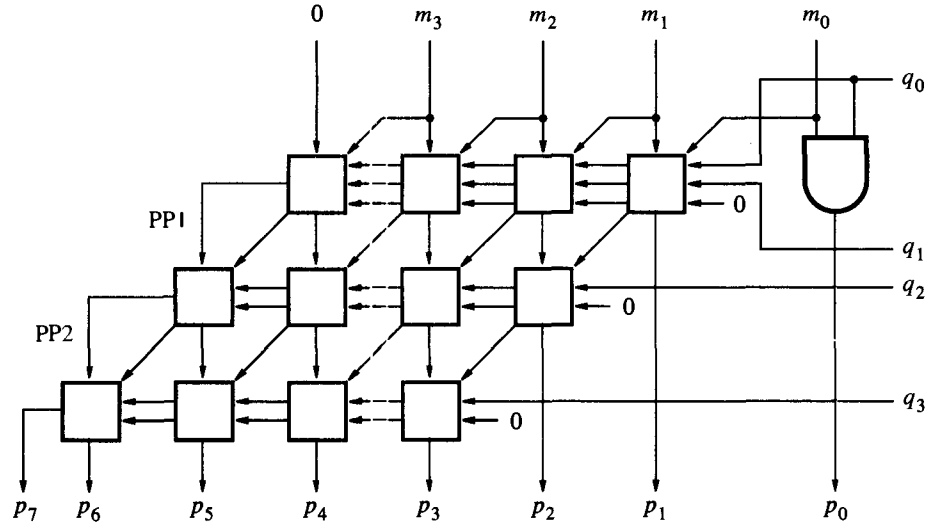
A circuit that implements the preceding operations is arranged in an array, as shown in Figure 5.36*a*. There are two types of blocks in the array. Part (*b*) of the figure shows the details of the blocks in the top row, and part (*c*) shows the block used in the second and third rows. Observe that the shifted versions of the multiplicand are provided by routing the m_k signals diagonally from one block to another. The full-adder included in each block implements a ripple-carry adder to generate each partial product. It is possible to design even faster multipliers by using other types of adders [1].

5.6.2 MULTIPLICATION OF SIGNED NUMBERS

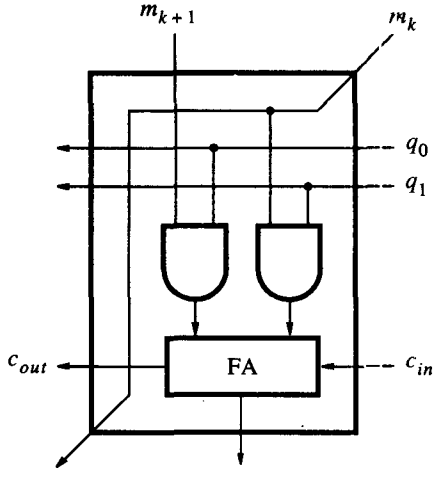
Multiplication of unsigned numbers illustrates the main issues involved in the design of multiplier circuits. Multiplication of signed numbers is somewhat more complex.

If the multiplier operand is positive, it is possible to use essentially the same scheme as for unsigned numbers. For each bit of the multiplier operand that is equal to 1, a properly shifted version of the multiplicand must be added to the partial product. The multiplicand can be either positive or negative.

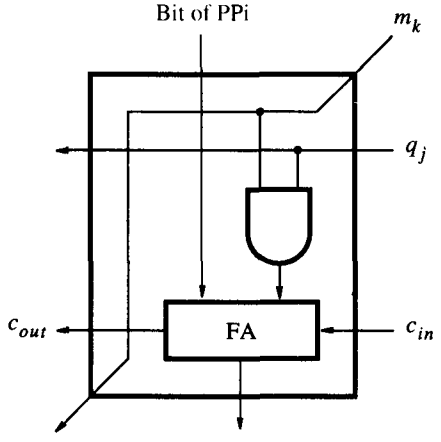
Since shifted versions of the multiplicand are added to the partial products, it is important to ensure that the numbers involved are represented correctly. For example, if the two right-most bits of the multiplier are both equal to 1, then the first addition must produce the partial product $PP1 = M + 2M$, where M is the multiplicand. If $M = m_{n-1}m_{n-2}\cdots m_1m_0$, then $PP1 = m_{n-1}m_{n-2}\cdots m_1m_0 + m_{n-1}m_{n-2}\cdots m_1m_00$. The



(a) Structure of the circuit



(b) A block in the top row



(c) A block in the bottom two rows

Figure 5.36 A 4 × 4 multiplier circuit.

adder that performs this addition comprises circuitry that adds two operands of equal length. Since shifting the multiplicand to the left, to generate $2M$, results in one of the operands having $n + 1$ bits, the required addition has to be performed using the second operand, M, represented also as an $(n + 1)$-bit number. An n-bit signed number is represented as an $(n + 1)$-bit number by using sign extension, that is, by replicating the sign bit as the new left-most bit. Thus $M = m_{n-1}m_{n-2} \cdots m_1m_0$ is represented using $(n + 1)$ bits as $M = m_{n-1}m_{n-1}m_{n-2} \cdots m_1m_0$.

When a shifted version of the multiplicand is added to a partial product, overflow has to be avoided. Hence the new partial product must be larger by one extra bit. Figure 5.37*a* illustrates the process of multiplying two positive numbers. The sign-extended bits are shown in blue. Part (*b*) of the figure involves a negative multiplicand. Note that the resulting product has $2n$ bits in both cases.

For a negative multiplier operand, it is possible to convert both the multiplier and the multiplicand into their 2's complements because this will not change the value of the result. Then the scheme for a positive multiplier can be used.

Multiplicand M	(+14)	0 1 1 1 0
Multiplier Q	(+11)	× 0 1 0 1 1
Partial product 0		0 0 0 1 1 1 0
		+ 0 0 1 1 1 0
Partial product 1		0 0 1 0 1 0 1
		+ 0 0 0 0 0 0
Partial product 2		0 0 0 1 0 1 0
		+ 0 0 1 1 1 0
Partial product 3		0 0 1 0 0 1 1
		+ 0 0 0 0 0 0
Product P	(+154)	0 0 1 0 0 1 1 0 1 0

(a) Positive multiplicand

Multiplicand M	(−14)	1 0 0 1 0
Multiplier Q	(+11)	× 0 1 0 1 1
Partial product 0		1 1 1 0 0 1 0
		+ 1 1 0 0 1 0
Partial product 1		1 1 0 1 0 1 1
		+ 0 0 0 0 0 0
Partial product 2		1 1 1 0 1 0 1
		+ 1 1 0 0 1 0
Partial product 3		1 1 0 1 1 0 0
		+ 0 0 0 0 0 0
Product P	(−154)	1 1 0 1 1 0 0 1 1 0

(b) Negative multiplicand

Figure 5.37 Multiplication of signed numbers.

We have presented a relatively simple scheme for multiplication of signed numbers. There exist other techniques that are more efficient but also more complex. We will not pursue these techniques, but an interested reader may consult reference [1].

We have discussed circuits that perform addition, subtraction, and multiplication. Another arithmetic operation that is needed in computer systems is division. Circuits that perform division are more complex; we will present an example in Chapter 10. Various techniques for performing division are usually discussed in books on the subject of computer organization, and can be found in references [1, 2].

5.7 OTHER NUMBER REPRESENTATIONS

In the previous sections we dealt with binary integers represented in the positional number representation. Other types of numbers are also used in digital systems. In this section we will discuss briefly three other types: fixed-point, floating-point, and binary-coded decimal numbers.

5.7.1 FIXED-POINT NUMBERS

A *fixed-point* number consists of integer and fraction parts. It can be written in the positional number representation as

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

The value of the number is

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

The position of the radix point is assumed to be fixed; hence the name fixed-point number. If the radix point is not shown, then it is assumed to be to the right of the least-significant digit, which means that the number is an integer.

Logic circuits that deal with fixed-point numbers are essentially the same as those used for integers. We will not discuss them separately.

5.7.2 FLOATING-POINT NUMBERS

Fixed-point numbers have a range that is limited by the significant digits used to represent the number. For example, if we use eight digits and a sign to represent decimal integers, then the range of values that can be represented is 0 to ±99999999. If eight digits are used to represent a fraction, then the representable range is 0.00000001 to ±0.99999999. In scientific applications it is often necessary to deal with numbers that are very large or very small. Instead of using the fixed-point representation, which would require many significant digits, it is better to use the floating-point representation in which numbers are

represented by a *mantissa* comprising the significant digits and an *exponent* of the radix R. The format is

$$Mantissa \times R^{Exponent}$$

The numbers are often *normalized*, such that the radix point is placed to the right of the first nonzero digit, as in 5.234×10^{43} or 6.31×10^{-28}.

Binary floating-point representation has been standardized by the Institute of Electrical and Electronic Engineers (IEEE) [3]. Two sizes of formats are specified in this standard—a *single-precision* 32-bit format and a *double-precision* 64-bit format. Both formats are illustrated in Figure 5.38.

Figure 5.38*a* depicts the single-precision format. The left-most bit is the sign bit—0 for positive and 1 for negative numbers. There is an 8-bit exponent field, E, and a 23-bit mantissa field, M. The exponent is with respect to the radix 2. Because it is necessary to be able to represent both very large and very small numbers, the exponent can be either positive or negative. Instead of simply using an 8-bit signed number as the exponent, which would allow exponent values in the range -128 to 127, the IEEE standard specifies the exponent in the *excess-127* format. In this format the value 127 is added to the value of the actual exponent so that

$$Exponent = E - 127$$

In this way E becomes a positive integer. This format is convenient for adding and subtracting floating-point numbers because the first step in these operations involves comparing the exponents to determine whether the mantissas must be appropriately shifted to add/subtract

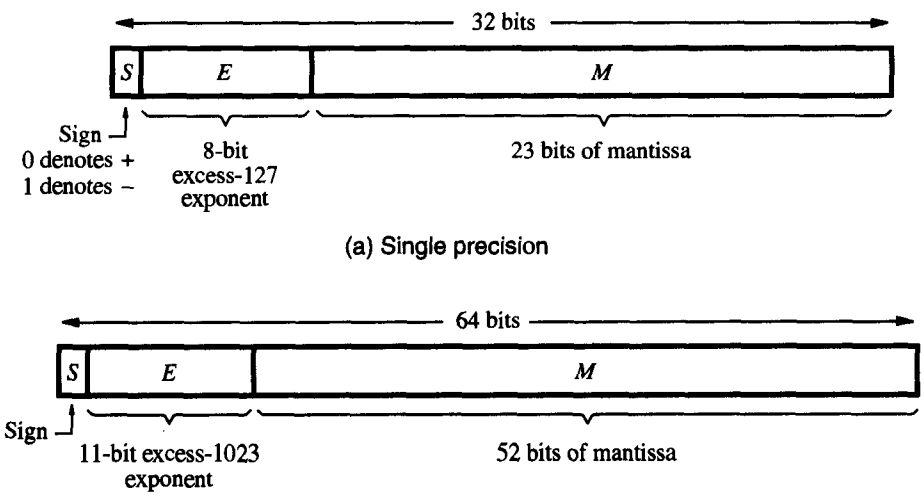


Figure 5.38 IEEE Standard floating-point formats.

the significant bits. The range of E is 0 to 255. The extreme values of $E = 0$ and $E = 255$ are taken to denote the exact zero and infinity, respectively. Therefore, the normal range of the exponent is -126 to 127, which is represented by the values of E from 1 to 254.

The mantissa is represented using 23 bits. The IEEE standard calls for a normalized mantissa, which means that the most-significant bit is always equal to 1. Thus it is not necessary to include this bit explicitly in the mantissa field. Therefore, if M is the bit vector in the mantissa field, the actual value of the mantissa is $1.M$, which gives a 24-bit mantissa. Consequently, the floating-point format in Figure 5.35*a* represents the number

$$\text{Value} = \pm 1.M \times 2^{E-127}$$

The size of the mantissa field allows the representation of numbers that have the precision of about seven decimal digits. The exponent field range of 2^{-126} to 2^{127} corresponds to about $10^{\pm 38}$.

Double-Precision Floating-Point Format

Figure 5.38*b* shows the double-precision format, which uses 64 bits. Both the exponent and mantissa fields are larger. This format allows greater range and precision of numbers. The exponent field has 11 bits, and it specifies the exponent in the *excess-1023* format, where

$$\textit{Exponent} = E - 1023$$

The range of E is 0 to 2047, but again the values $E = 0$ and $E = 2047$ are used to indicate the exact 0 and infinity, respectively. Thus the normal range of the exponent is -1022 to 1023, which is represented by the values of E from 1 to 2046.

The mantissa field has 52 bits. Since the mantissa is assumed to be normalized, its actual value is again $1.M$. Therefore, the value of a floating-point number is

$$\text{Value} = \pm 1.M \times 2^{E-1023}$$

This format allows representation of numbers that have the precision of about 16 decimal digits and the range of approximately $10^{\pm 308}$.

Arithmetic operations using floating-point operands are significantly more complex than signed integer operations. Because this is a rather specialized domain, we will not elaborate on the design of logic circuits that can perform such operations. For a more complete discussion of floating-point operations, the reader may consult references [1, 2].

5.7.3 BINARY-CODED-DECIMAL REPRESENTATION

In digital systems it is possible to represent decimal numbers simply by encoding each digit in binary form. This is called the *binary-coded-decimal (BCD)* representation. Because there are 10 digits to encode, it is necessary to use four bits per digit. Each digit is encoded by the binary pattern that represents its unsigned value, as shown in Table 5.3. Note that only 10 of the 16 available patterns are used in BCD, which means that the remaining 6 patterns should not occur in logic circuits that operate on BCD operands; these patterns are usually treated as don't-care conditions in the design process. BCD representation was used in some early computers as well as in many handheld calculators. Its main virtue is

Table 5.3 Binary-coded decimal digits.

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

that it provides a format that is convenient when numerical information is to be displayed on a simple digit-oriented display. Its drawbacks are complexity of circuits that perform arithmetic operations and the fact that six of the possible code patterns are wasted.

Even though the importance of BCD representation has diminished, it is still encountered. To give the reader an indication of the complexity of the required circuits, we will consider BCD addition in some detail.

BCD Addition

The addition of two BCD digits is complicated by the fact that the sum may exceed 9, in which case a correction will have to be made. Let $X = x_3x_2x_1x_0$ and $Y = y_3y_2y_1y_0$ represent the two BCD digits and let $S = s_3s_2s_1s_0$ be the desired sum digit, $S = X + Y$. Obviously, if $X + Y \leq 9$, then the addition is the same as the addition of 2 four-bit unsigned binary numbers. But, if $X + Y > 9$, then the result requires two BCD digits. Moreover, the four-bit sum obtained from the four-bit adder may be incorrect.

There are two cases where some correction has to be made: when the sum is greater than 9 but no carry-out is generated using four bits, and when the sum is greater than 15 so that a carry-out is generated using four bits. Figure 5.39 illustrates these cases. In the first case the four-bit addition yields $7 + 5 = 12 = Z$. To obtain a correct BCD result, we must generate $S = 2$ and a carry-out of 1. The necessary correction is apparent from the fact that the four-bit addition is a modulo-16 scheme, whereas decimal addition is a modulo-10 scheme. Therefore, a correct decimal digit can be generated by adding 6 to the result of four-bit addition whenever this result exceeds 9. Thus we can arrange the computation as follows

$$Z = X + Y$$

If $Z \leq 9$, then $S = Z$ and carry-out $= 0$

if $Z > 9$, then $X = Z + 6$ and carry-out $= 1$

X	0111	7
+ Y	+ 0101	+ 5
Z	1100	12
	+ 0110	
carry →	10010	
	S = 2	

X	1000	8
+ Y	+ 1001	+ 9
Z	10001	17
	+ 0110	
carry →	10111	
	S = 7	

Figure 5.39 Addition of BCD digits.

The second example in Figure 5.39 shows what happens when $X + Y > 15$. In this case the four least-significant bits of Z represent the digit 1, which is wrong. But a carry is generated, which corresponds to the value 16, that must be taken into account. Again adding 6 to the intermediate sum Z provides the necessary correction.

Figure 5.40 gives a block diagram of a one-digit BCD adder that is based on this scheme. The block that detects whether $Z > 9$ produces an output signal, Adjust, which controls the multiplexer that provides the correction when needed. A second four-bit adder generates the corrected sum bits. If Adjust = 0, then $S = Z + 0$; if Adjust = 1, then $S = Z + 6$ and carry-out = 1.

The one-digit BCD adder can be specified in Verilog code by describing its behavior as shown in Figure 5.41. Inputs X and Y, and output S are defined as four-bit numbers. The intermediate sum, Z, is defined as a five-bit number. The **if-else** statement is used to provide the adjustment explained above; hence it is not necessary to use an explicit *Adjust* signal. To verify the functional correctness of the code, we performed a functional simulation. An example of the obtained results is given in Figure 5.42.

If we wish to derive a circuit to implement the block diagram in Figure 5.40 by hand, instead of by using Verilog, then the following approach can be used. To define the *Adjust* function, we can observe that the intermediate sum will exceed 9 if the carry-out from the four-bit adder is equal to 1, or if $z_3 = 1$ and either z_2 or z_1 (or both) are equal to 1. Hence the logic expression for this function is

$$\text{Adjust} = \text{Carry-out} + z_3(z_2 + z_1)$$

Instead of implementing another complete four-bit adder to perform the correction, we can use a simpler circuit because the addition of constant 6 does not require the full capability of a four-bit adder. Note that the least-significant bit of the sum, s_0, is not affected at all; hence $s_0 = z_0$. A two-bit adder may be used to develop bits s_2 and s_1. Bit s_3 is the same as

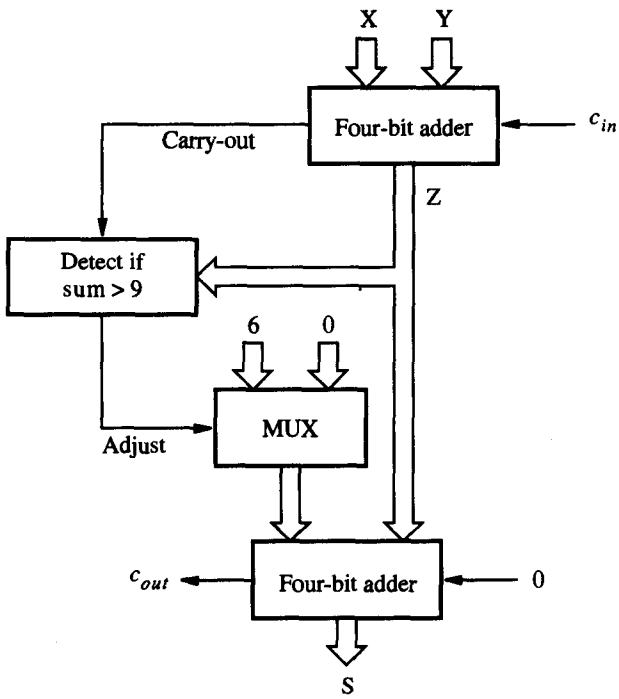


Figure 5.40 Block diagram for a one-digit BCD adder.

```
module bcdadd (Cin, X, Y, S, Cout);
   input Cin;
   input [3:0] X, Y;
   output [3:0] S;
   output Cout;
   reg [3:0] S;
   reg Cout;
   reg [4:0] Z;

   always @(X or Y or Cin)
   begin
      Z = X + Y + Cin;
      if (Z < 10)
         {Cout, S} = Z;
      else
         {Cout, S} = Z + 6;
   end

endmodule
```

Figure 5.41 Verilog code for a one-digit BCD adder.

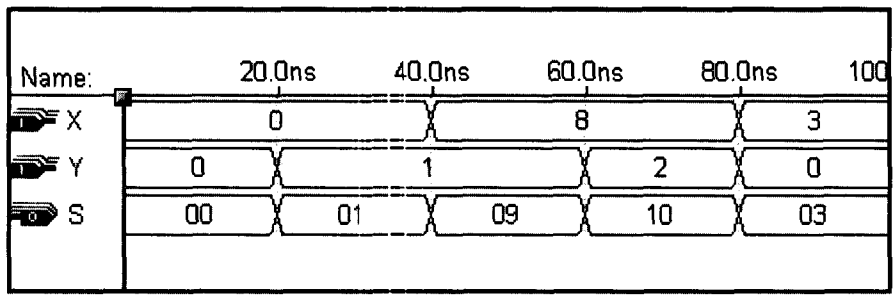


Figure 5.42 Functional simulation of the Verilog code in Figure 5.41.

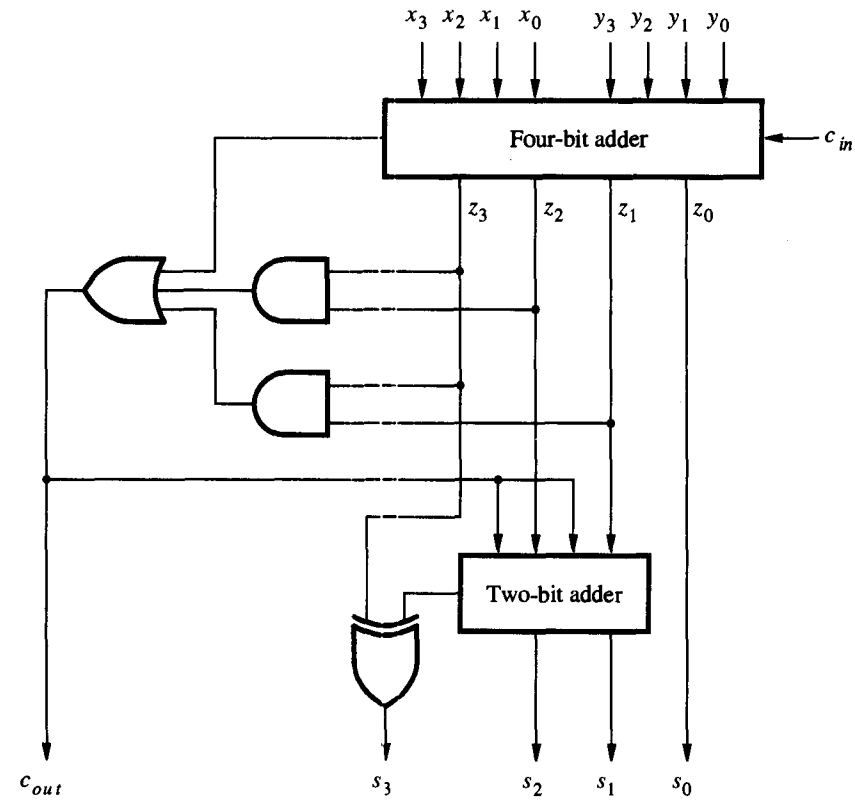


Figure 5.43 Circuit for a one-digit BCD adder.

z_3 if the carry-out from the two-bit adder is 0, and it is equal to $\overline{z}_3$ if this carry-out is equal to 1. A complete circuit that implements this scheme is shown in Figure 5.43. Using the one-digit BCD adder as a basic block, it is possible to build larger BCD adders in the same way as a binary full-adder is used to build larger ripple-carry binary adders.

Subtraction of BCD numbers can be handled with the radix-complement approach. Just as we use 2's complement representation to deal with negative binary numbers, we can use 10's complement representation to deal with decimal numbers. We leave the development of such a scheme as an exercise for the reader (see problem 5.19).

5.8 ASCII CHARACTER CODE

The most popular code for representing information in digital systems is used for both letters and numbers, as well as for some control characters. It 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 5.4.

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

Characters that are either letters of the alphabet or numbers are referred to as *alphanumeric* characters. In addition to these characters, 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, say, printer or display, should return to the left-most column.

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

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

Parity

The concept of *parity* is widely used in digital systems for error-checking purposes. When digital information is transmitted from one point to another, perhaps by long wires, it is possible for some bits to become corrupted during the transmission process. For example, the sender may transmit a bit whose value is equal to 1, but the receiver observes a bit whose value is 0. Suppose that a data item consists of n bits. A simple error-checking mechanism can be implemented by including an extra bit, p, which indicates the parity of the n-bit item. Two kinds of parity can be used. For *even parity* the p bit is given the value such that the total number of 1s in the $n + 1$ transmitted bits (comprising the n-bit data and the parity

Table 5.4 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	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	
1111	SI	US	/	?	O	—	o	DEL

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

Bit positions of code format = | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

bit p) is even. For *odd parity* the p bit is given the value that makes the total number of 1s odd. The sender generates the p bit based on the n-bit data item that is to be transmitted. The receiver checks whether the parity of the received item is correct.

Parity generating and checking circuits can be realized with XOR gates. For example, for a four-bit data item consisting of bits $x_3x_2x_1x_0$, the even parity bit can be generated as

$$p = x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

At the receiving end the checking is done using

$$c = p \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

If $c = 0$, then the received item shows the correct parity. If $c = 1$, then an error has occurred. Note that observing $c = 0$ is not a guarantee that the received item is correct. If two or any even number of bits have their values inverted during the transmission, the parity of the data item will not be changed; hence the error will not be detected. But if an odd number of bits are corrupted, then the error will be detected.

The attractiveness of parity checking lies in its simplicity. There exist other more sophisticated schemes that provide more reliable error-checking mechanisms [4]. We will discuss parity circuits again in section 9.3.

PROBLEMS

5.1 Determine the decimal values of the following unsigned numbers:
(a) $(0111011110)_2$
(b) $(1011100111)_2$
(c) $(3751)_8$
(d) $(A25F)_{16}$
(e) $(F0F0)_{16}$

5.2 Determine the decimal values of the following 1's complement numbers:
(a) 0111011110
(b) 1011100111
(c) 1111111110

5.3 Determine the decimal values of the following 2's complement numbers:
(a) 0111011110
(b) 1011100111
(c) 1111111110

5.4 Convert the decimal numbers 73, 1906, −95, and −1630 into signed 12-bit numbers in the following representations:
(a) Sign and magnitude
(b) 1's complement
(c) 2's complement

5.5 Perform the following operations involving eight-bit 2's complement numbers and indicate whether arithmetic overflow occurs. Check your answers by converting to decimal sign-and-magnitude representation.

00110110	01110101	11011111
+01000101	+11011110	+10111000
00110110	01110101	11010011
−00101011	−11010110	−11101100

5.6 Prove that the XOR operation is associative, which means that $x_i \oplus (y_i \oplus z_i) = (x_i \oplus y_i) \oplus z_i$.

5.7 Show that the circuit in Figure 5.5 implements the full-adder specified in Figure 5.4*a*.

5.8 Prove the validity of the simple rule for finding the 2's complement of a number, which was presented in section 5.3. Recall that the rule states that scanning a number from right to left, all 0s and the first 1 are copied; then all remaining bits are complemented.

5.9 Prove the validity of the expression Overflow = $c_n \oplus c_{n-1}$ for addition of n-bit signed numbers.

5.10 In section 5.5.4 we stated that a carry-out signal, c_k, from bit position $k-1$ of an adder circuit can be generated as $c_k = x_k \oplus y_k \oplus s_k$, where x_k and y_k are inputs and s_k is the sum bit. Verify the correctness of this statement.

5.11 Consider the circuit in Figure P5.1. Can this circuit be used as one stage in a carry-ripple adder? Discuss the pros and cons.

5.12 Determine the number of gates needed to implement an n-bit carry-lookahead adder, assuming no fan-in constraints. Use AND, OR, and XOR gates with any number of inputs.

5.13 Determine the number of gates needed to implement an eight-bit carry-lookahead adder assuming that the maximum fan-in for the gates is four.

5.14 In Figure 5.18 we presented the structure of a hierarchical carry-lookahead adder. Show the complete circuit for a four-bit version of this adder, built using 2 two-bit blocks.

5.15 What is the critical delay path in the multiplier in Figure 5.33? What is the delay along this path in terms of the number of gates?

5.16 (a) Write a Verilog module to describe the circuit block in Figure 5.36*b*. Use the CAD tools to synthesize a circuit from the code and verify its functional correctness.
(b) Write a Verilog module to describe the circuit block in Figure 5.36*c*. Use the CAD tools to synthesize a circuit from the code and verify its functional correctness.
(c) Write a Verilog module to describe the 4 × 4 multiplier shown in Figure 5.36*a*. Your code should be hierarchical and should use the subcircuits designed in parts (a) and (b). Synthesize a circuit from the code and verify its functional correctness.

5.17 Consider the Verilog code in Figure P5.2. Given the relationship between the signals IN and OUT, what is the functionality of the circuit described by the code? Comment on whether or not this code represents a good style to use for the functionality that it represents.

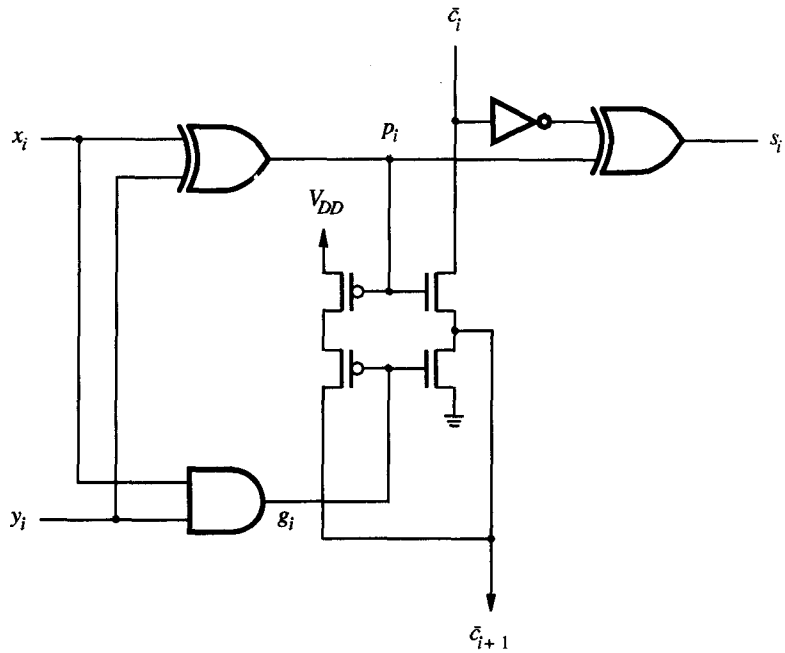


Figure P5.1 Circuit for problem 5.11.

```
module problem5_17 (IN, OUT);
   input [3:0] IN;
   output [3:0] OUT;
   reg [3:0] OUT;

   always @(IN)
      if (IN == 4'b0101)  OUT = 4'b0001;
      else if (IN == 4'b0110)  OUT = 4'b0010;
      else if (IN == 4'b0111)  OUT = 4'b0011;
      else if (IN == 4'b1001)  OUT = 4'b0010;
      else if (IN == 4'b1010)  OUT = 4'b0100;
      else if (IN == 4'b1011)  OUT = 4'b0110;
      else if (IN == 4'b1101)  OUT = 4'b0011;
      else if (IN == 4'b1110)  OUT = 4'b0110;
      else if (IN == 4'b1111)  OUT = 4'b1001;
      else  OUT = 4'b0000;

endmodule
```

Figure P5.2 The code for problem 5.17.

5.18 Design a circuit that generates the 9's complement of a BCD digit. Note that the 9's complement of d is $9 - d$.

5.19 Derive a scheme for performing subtraction using BCD operands. Show a block diagram for the subtractor circuit.
Hint: Subtraction can be performed easily if the operands are in the 10's complement (radix complement) representation. In this representation the sign digit is 0 for a positive number and 9 for a negative number.

5.20 Write complete Verilog code for the circuit that you derived in problem 5.19.

5.21 Suppose that we want to determine how many of the bits in a three-bit unsigned number are equal to 1. Design the simplest circuit that can accomplish this task.

5.22 Repeat problem 5.21 for a six-bit unsigned number.

5.23 Repeat problem 5.21 for an eight-bit unsigned number.

5.24 Show a graphical interpretation of three-digit decimal numbers, similar to Figure 5.12. The left-most digit is 0 for positive numbers and 9 for negative numbers. Verify the validity of your answer by trying a few examples of addition and subtraction.

5.25 In a ternary number system there are three digits: 0, 1, and 2. Figure P5.3 defines a ternary half-adder. Design a circuit that implements this half-adder using binary-encoded signals, such that two bits are used for each ternary digit. Let $A = a_1a_0$, $B = b_1b_0$, and $Sum = s_1s_0$; note that *Carry* is just a binary signal. Use the following encoding: $00 = (0)_3$, $01 = (1)_3$, and $10 = (2)_3$. Minimize the cost of the circuit.

A B	*Carry*	*Sum*
0 0	0	0
0 1	0	1
0 2	0	2
1 0	0	1
1 1	0	2
1 2	1	0
2 0	0	2
2 1	1	0
2 2	1	1

Figure P5.3 Ternary half-adder.

5.26 Design a ternary full-adder circuit, using the approach described in problem 5.25.

5.27 Consider the subtractions $26 - 27 = 99$ and $18 - 34 = 84$. Using the concepts presented in section 5.3.4, explain how these answers (99 and 84) can be interpreted as the correct signed results of these subtractions.

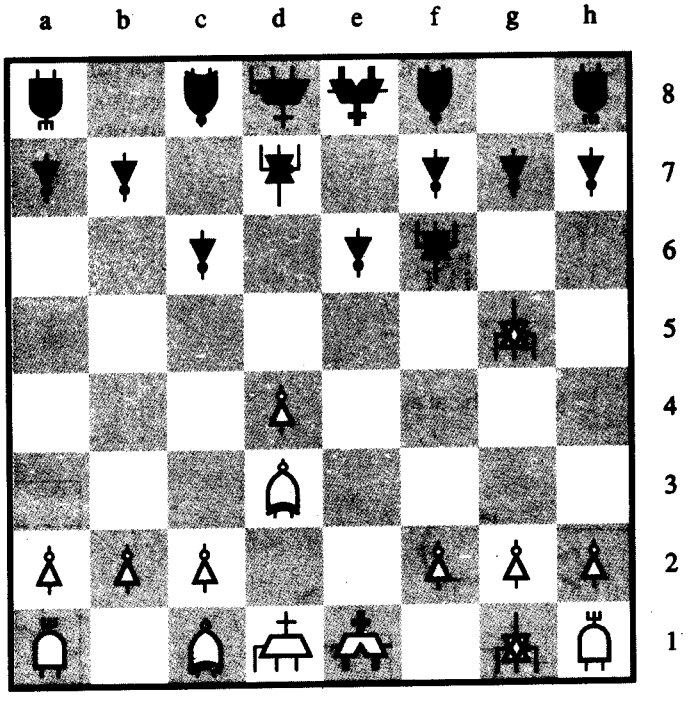
REFERENCES

1. V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, *Computer Organization*, 5th ed. (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2nd ed. (Morgan Kaufmann: San Francisco, CA, 1998).
3. Institute of Electrical and Electronic Engineers (IEEE), "A Proposed Standard for Floating-Point Arithmetic," *Computer* 14, no. 3 (March 1981), pp. 51–62.
4. W. W. Peterson and E. J. Weldon Jr., *Error-Correcting Codes*, 2nd ed. (MIT Press: Boston, MA, 1972).

chapter

6

COMBINATIONAL-CIRCUIT BUILDING BLOCKS



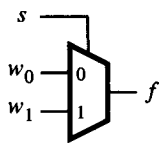
6. Bf1–d3, e7–e6

Previous chapters have introduced the basic techniques for design of logic circuits. In practice, a few types of logic circuits are often used as building blocks in larger designs. This chapter discusses a number of these blocks and gives examples of their use. The chapter also includes a major section on Verilog, which describes several key features of the language.

6.1 MULTIPLEXERS

Multiplexers were introduced briefly in Chapters 2 and 3. A multiplexer circuit has a number of data inputs, one or more select inputs, and one output. It passes the signal value on one of the data inputs to the output. The data input is selected by the values of the select inputs. Figure 6.1 shows a 2-to-1 multiplexer. Part (*a*) gives the symbol commonly used. The *select* input, s, chooses as the output of the multiplexer either input w_0 or w_1. The multiplexer's functionality can be described in the form of a truth table as shown in part (*b*) of the figure. Part (*c*) gives a sum-of-products implementation of the 2-to-1 multiplexer, and part (*d*) illustrates how it can be constructed with transmission gates.

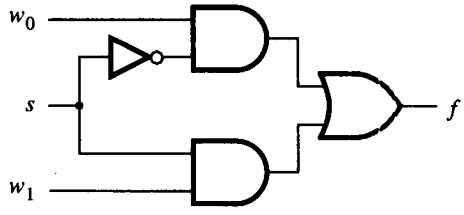
Figure 6.2*a* depicts a larger multiplexer with four data inputs, $w_0, \ldots, w_3$, and two select inputs, s_1 and s_0. As shown in the truth table in part (*b*) of the figure, the two-bit number represented by s_1s_0 selects one of the data inputs as the output of the multiplexer. A sum-of-products implementation of the 4-to-1 multiplexer appears in Figure 6.2*c*. It



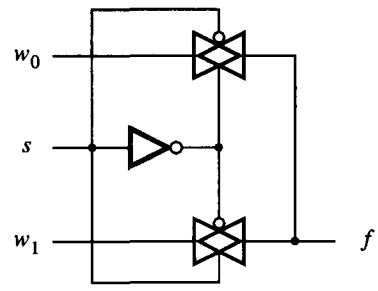
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

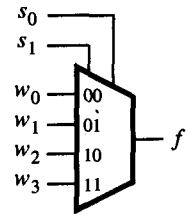


(c) Sum-of-products circuit



(d) Circuit with transmission gates

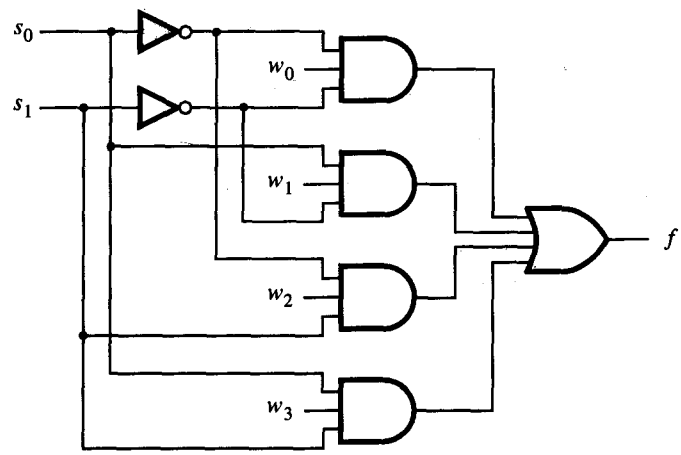
Figure 6.1 A 2-to-1 multiplexer



(a) Graphical symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 6.2 A 4-to-1 multiplexer.

realizes the multiplexer function

$$f = \bar{s}_1\bar{s}_0w_0 + \bar{s}_1s_0w_1 + s_1\bar{s}_0w_2 + s_1s_0w_3$$

It is possible to build larger multiplexers using the same approach. Usually, the number of data inputs, n, is an integer power of two. A multiplexer that has n data inputs, $w_0, \ldots, w_{n-1}$, requires $\lceil \log_2 n \rceil$ select inputs. Larger multiplexers can also be constructed from smaller multiplexers. For example, the 4-to-1 multiplexer can be built using three 2-to-1 multiplexers as illustrated in Figure 6.3. If the 4-to-1 multiplexer is implemented using transmission gates, then the structure in this figure is always used. Figure 6.4 shows how a 16-to-1 multiplexer is constructed with five 4-to-1 multiplexers.

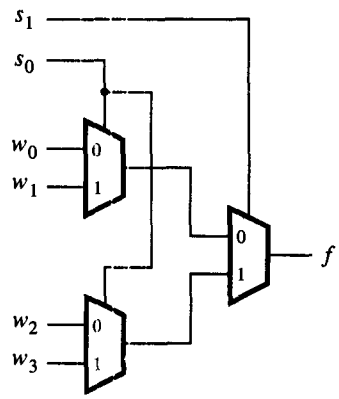


Figure 6.3 Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

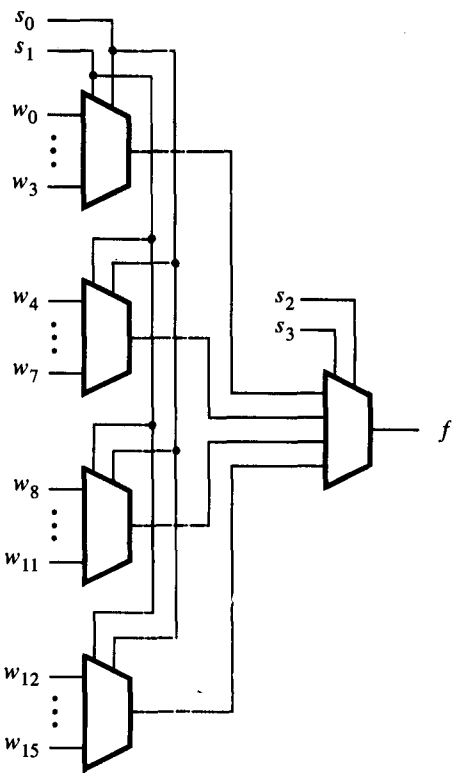


Figure 6.4 A 16-to-1 multiplexer.

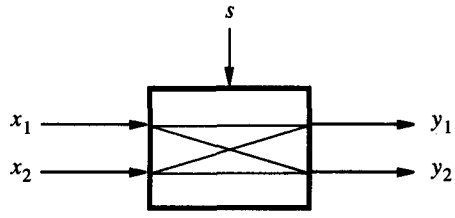
Example 6.1

Figure 6.5 shows a circuit that has two inputs, x_1 and x_2, and two outputs, y_1 and y_2. As indicated by the blue lines, the function of the circuit is to allow either of its inputs to be connected to either of its outputs, under the control of another input, s. A circuit that has n inputs and k outputs, whose sole function is to provide a capability to connect any input to any output, is usually referred to as an $n \times k$ crossbar switch. Crossbars of various sizes can be created, with different numbers of inputs and outputs. When there are two inputs and two outputs, it is called a 2×2 crossbar.

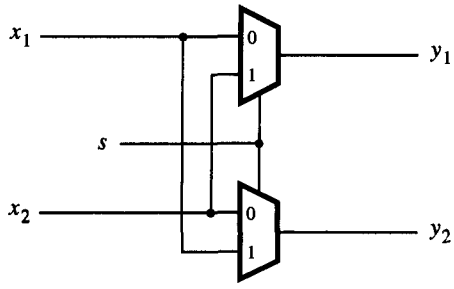
Figure 6.5*b* shows how the 2×2 crossbar can be implemented using 2-to-1 multiplexers. The multiplexer select inputs are controlled by the signal s. If $s = 0$, the crossbar connects x_1 to y_1 and x_2 to y_2, while if $s = 1$, the crossbar connects x_1 to y_2 and x_2 to y_1. Crossbar switches are useful in many practical applications in which it is necessary to be able to connect one set of wires to another set of wires, where the connection pattern changes from time to time.

Example 6.2

We introduced field-programmable gate array (FPGA) chips in section 3.6.5. Figure 3.39 depicts a small FPGA that is programmed to implement a particular circuit. The logic blocks in the FPGA have two inputs, and there are four tracks in each routing channel. Each of the programmable switches that connects a logic block input or output to an interconnection wire is shown as an X. A small part of Figure 3.39 is reproduced in Figure 6.6*a*. For clarity, the figure shows only a single logic block and the interconnection wires and switches associated with its input terminals.

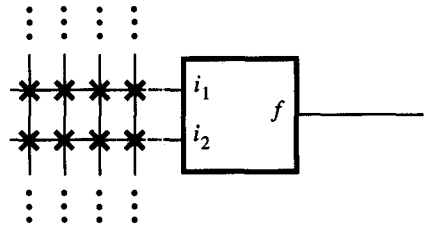


(a) A 2x2 crossbar switch

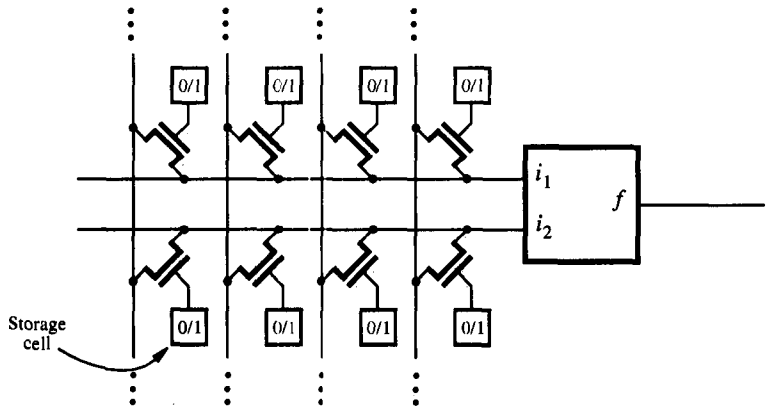


(b) Implementation using multiplexers

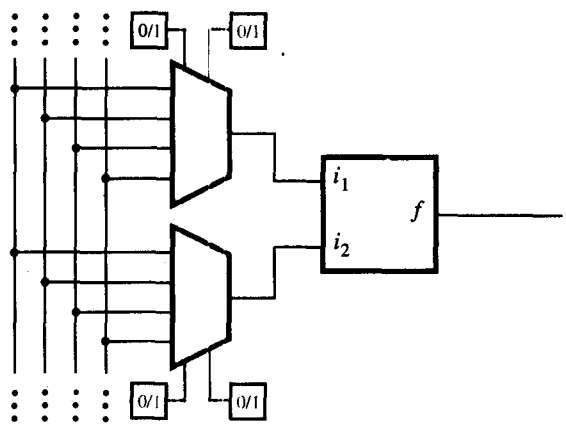
Figure 6.5 A practical application of multiplexers.



(a) Part of the FPGA in Figure 3.39



(b) Implementation using pass transistors



(c) Implementation using multiplexers

Figure 6.6 Implementing programmable switches in an FPGA.

One way in which the programmable switches can be implemented is illustrated in Figure 6.6*b*. Each X in part (*a*) of the figure is realized using an NMOS transistor controlled by a storage cell. This type of programmable switch was also shown in Figure 3.68. We described storage cells briefly in section 3.6.5 and will discuss them in more detail in section 10.1. Each cell stores a single logic value, either 0 or 1, and provides this value as the output of the cell. Each storage cell is built by using several transistors. Thus the eight cells shown in the figure use a significant amount of chip area.

The number of storage cells needed can be reduced by using multiplexers, as shown in Figure 6.6*c*. Each logic block input is fed by a 4-to-1 multiplexer, with the select inputs controlled by storage cells. This approach requires only four storage cells, instead of eight. In commercial FPGAs the multiplexer-based approach is usually adopted.

6.1.1 SYNTHESIS OF LOGIC FUNCTIONS USING MULTIPLEXERS

Multiplexers are useful in many practical applications, such as those described above. They can also be used in a more general way to synthesize logic functions. Consider the example in Figure 6.7*a*. The truth table defines the function $f = w_1 \oplus w_2$. This function can be implemented by a 4-to-1 multiplexer in which the values of f in each row of the truth table are connected as constants to the multiplexer data inputs. The multiplexer select inputs are driven by w_1 and w_2. Thus for each valuation of $w_1 w_2$, the output f is equal to the function value in the corresponding row of the truth table.

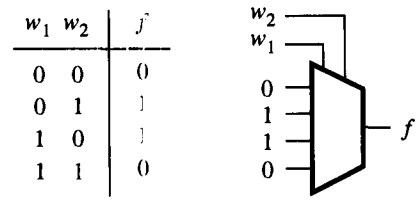
The above implementation is straightforward, but it is not very efficient. A better implementation can be derived by manipulating the truth table as indicated in Figure 6.7*b*, which allows f to be implemented by a single 2-to-1 multiplexer. One of the input signals, w_1 in this example, is chosen as the select input of the 2-to-1 multiplexer. The truth table is redrawn to indicate the value of f for each value of w_1. When $w_1 = 0, f$ has the same value as input w_2, and when $w_1 = 1, f$ has the value of $\overline{w}_2$. The circuit that implements this truth table is given in Figure 6.7*c*. This procedure can be applied to synthesize a circuit that implements any logic function.

Example 6.3

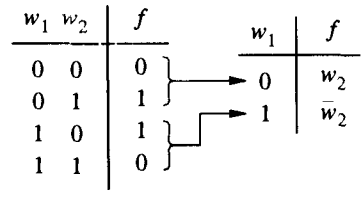
Figure 6.8*a* gives the truth table for the three-input majority function, and it shows how the truth table can be modified to implement the function using a 4-to-1 multiplexer. Any two of the three inputs may be chosen as the multiplexer select inputs. We have chosen w_1 and w_2 for this purpose, resulting in the circuit in Figure 6.8*b*.

Example 6.4

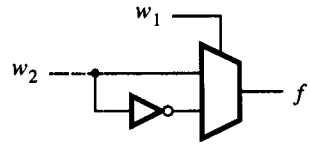
Figure 6.9*a* indicates how the function $f = w_1 \oplus w_2 \oplus w_3$ can be implemented using 2-to-1 multiplexers. When $w_1 = 0, f$ is equal to the XOR of w_2 and w_3, and when $w_1 = 1, f$ is the XNOR of w_2 and w_3. The left multiplexer in the circuit produces $w_2 \oplus w_3$, using the result from Figure 6.7, and the right multiplexer uses the value of w_1 to select either $w_2 \oplus w_3$ or its



(a) Implementation using a 4-to-1 multiplexer



(b) Modified truth table



(c) Circuit

Figure 6.7 Synthesis of a logic function using mutiplexers.

complement. Note that we could have derived this circuit directly by writing the function as $f = (w_2 \oplus w_3) \oplus w_1$.

Figure 6.10 gives an implementation of the three-input XOR function using a 4-to-1 multiplexer. Choosing w_1 and w_2 for the select inputs results in the circuit shown.

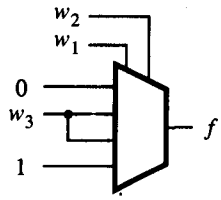
6.1.2 MULTIPLEXER SYNTHESIS USING SHANNON'S EXPANSION

Figures 6.8 through 6.10 illustrate how truth tables can be interpreted to implement logic functions using multiplexers. In each case the inputs to the multiplexers are the constants 0 and 1, or some variable or its complement. Besides using such simple inputs, it is possible to connect more complex circuits as inputs to a multiplexer, allowing functions to be synthesized using a combination of multiplexers and other logic gates. Suppose that we

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table

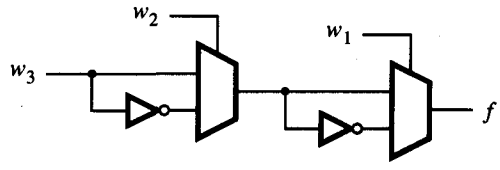


(b) Circuit

Figure 6.8 Implementation of the three-input majority function using a 4-to-1 multiplexer.

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	$w_2 \oplus w_3$
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	$\overline{w_2 \oplus w_3}$
1	1	0	0	
1	1	1	1	

(a) Truth table

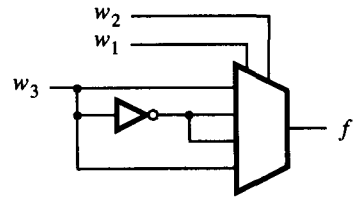


(b) Circuit

Figure 6.9 Three-input XOR implemented with 2-to-1 multiplexers.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\overline{w}_3$
0	1	1	0	
1	0	0	1	$\overline{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



(b) Circuit

Figure 6.10 Three-input XOR implemented with a 4-to-1 multiplexer.

want to implement the three-input majority function in Figure 6.8 using a 2-to-1 multiplexer in this way. Figure 6.11 shows an intuitive way of realizing this function. The truth table can be modified as shown on the right. If $w_1 = 0$, then $f = w_2w_3$, and if $w_1 = 1$, then $f = w_2 + w_3$. Using w_1 as the select input for a 2-to-1 multiplexer leads to the circuit in Figure 6.11*b*.

This implementation can be derived using algebraic manipulation as follows. The function in Figure 6.11*a* is expressed in sum-of-products form as

$$f = \overline{w}_1w_2w_3 + w_1\overline{w}_2w_3 + w_1w_2\overline{w}_3 + w_1w_2w_3$$

It can be manipulated into

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(\overline{w}_2w_3 + w_2\overline{w}_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

which corresponds to the circuit in Figure 6.11*b*.

Multiplexer implementations of logic functions require that a given function be decomposed in terms of the variables that are used as the select inputs. This can be accomplished by means of a theorem proposed by Claude Shannon [1].

Shannon's Expansion Theorem Any Boolean function $f(w_1, \ldots, w_n)$ can be written in the form

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

This expansion can be done in terms of any of the n variables. We will leave the proof of the theorem as an exercise for the reader (see problem 6.9).

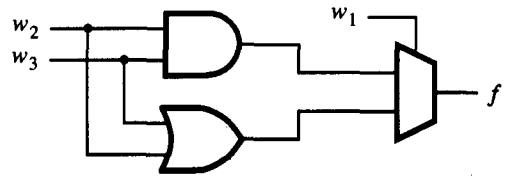
To illustrate its use, we can apply the theorem to the three-input majority function, which can be written as

$$f(w_1, w_2, w_3) = w_1w_2 + w_1w_3 + w_2w_3$$

w_1	w_2	w_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

w_1	f
0	w_2w_3
1	$w_2 + w_3$

(a) Truth table



(b) Circuit

Figure 6.11 The three-input majority function implemented using a 2-to-1 multiplexer.

Expanding this function in terms of w_1 gives

$$f = \overline{w}_1(w_2w_3) + w_1(w_2 + w_3)$$

which is the expression that we derived above.

For the three-input XOR function, we have

$$\begin{aligned} f &= w_1 \oplus w_2 \oplus w_3 \\ &= \overline{w}_1 \cdot (w_2 \oplus w_3) + w_1 \cdot \overline{(w_2 \oplus w_3)} \end{aligned}$$

which gives the circuit in Figure 6.9*b*.

In Shannon's expansion the term $f(0, w_2, \ldots, w_n)$ is called the *cofactor* of f with respect to $\overline{w}_1$; it is denoted in shorthand notation as $f_{\overline{w}_1}$. Similarly, the term $f(1, w_2, \ldots, w_n)$ is called the cofactor of f with respect to w_1, written f_{w_1}. Hence we can write

$$f = \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}$$

In general, if the expansion is done with respect to variable w_i, then f_{w_i} denotes $f(w_1, \ldots, w_{i-1}, 1, w_{i+1}, \ldots, w_n)$ and

$$f(w_1, \ldots, w_n) = \overline{w}_i f_{\overline{w}_i} + w_i f_{w_i}$$

The complexity of the logic expression may vary, depending on which variable, w_i, is used, as illustrated in Example 6.5.

Example 6.5 For the function $f = \overline{w}_1 w_3 + w_2 \overline{w}_3$, decomposition using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (w_3 + w_2) + w_1 (w_2 \overline{w}_3) \end{aligned}$$

Using w_2 instead of w_1 produces

$$\begin{aligned} f &= \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2} \\ &= \overline{w}_2 (\overline{w}_1 w_3) + w_2 (\overline{w}_1 + \overline{w}_3) \end{aligned}$$

Finally, using w_3 gives

$$\begin{aligned} f &= \overline{w}_3 f_{\overline{w}_3} + w_3 f_{w_3} \\ &= \overline{w}_3 (w_2) + w_3 (\overline{w}_1) \end{aligned}$$

The results generated using w_1 and w_2 have the same cost, but the expression produced using w_3 has a lower cost. In practice, the CAD tools that perform decompositions of this type try a number of alternatives and choose the one that produces the best result.

Shannon's expansion can be done in terms of more than one variable. For example, expanding a function in terms of w_1 and w_2 gives

$$\begin{aligned} f(w_1, \ldots, w_n) = \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) + \\ w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \end{aligned}$$

This expansion gives a form that can be implemented using a 4-to-1 multiplexer. If Shannon's expansion is done in terms of all n variables, then the result is the canonical sum-of-products form, which was defined in section 2.6.1.

Example 6.6 Assume that we wish to implement the function

$$f = \overline{w}_1 \overline{w}_3 + w_1 w_2 + w_1 w_3$$

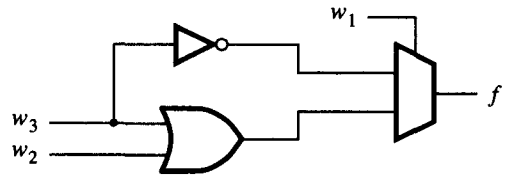
using a 2-to-1 multiplexer and any other necessary gates. Shannon's expansion using w_1 gives

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1 (\overline{w}_3) + w_1 (w_2 + w_3) \end{aligned}$$

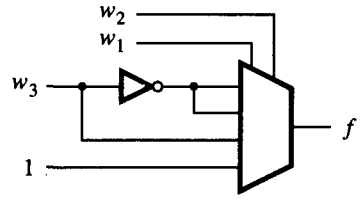
The corresponding circuit is shown in Figure 6.12*a*. Assume now that we wish to use a 4-to-1 multiplexer instead. Further decomposition using w_2 gives

$$\begin{aligned} f &= \overline{w}_1 \overline{w}_2 f_{\overline{w}_1 \overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1 \overline{w}_2 f_{w_1 \overline{w}_2} + w_1 w_2 f_{w_1 w_2} \\ &= \overline{w}_1 \overline{w}_2 (\overline{w}_3) + \overline{w}_1 w_2 (\overline{w}_3) + w_1 \overline{w}_2 (w_3) + w_1 w_2 (1) \end{aligned}$$

The circuit is shown in Figure 6.12*b*.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 6.12 The circuits synthesized in Example 6.6.

Example 6.7

Consider the three-input majority function

$$f = w_1w_2 + w_1w_3 + w_2w_3$$

We wish to implement this function using only 2-to-1 multiplexers. Shannon's expansion using w_1 yields

$$\begin{aligned} f &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3 + w_2w_3) \\ &= \overline{w}_1(w_2w_3) + w_1(w_2 + w_3) \end{aligned}$$

Let $g = w_2w_3$ and $h = w_2 + w_3$. Expansion of both g and h using w_2 gives

$$\begin{aligned} g &= \overline{w}_2(0) + w_2(w_3) \\ h &= \overline{w}_2(w_3) + w_2(1) \end{aligned}$$

The corresponding circuit is shown in Figure 6.13. It is equivalent to the 4-to-1 multiplexer circuit derived using a truth table in Figure 6.8.

Example 6.8

In section 3.6.5 we said that most FPGAs use lookup tables for their logic blocks. Assume that an FPGA exists in which each logic block is a three-input lookup table (3-LUT). Because it stores a truth table, a 3-LUT can realize any logic function of three variables. Using Shannon's expansion, any four-variable function can be realized with at most three 3-LUTs. Consider the function

$$f = \overline{w}_2w_3 + \overline{w}_1w_2\overline{w}_3 + w_2\overline{w}_3w_4 + w_1\overline{w}_2\overline{w}_4$$

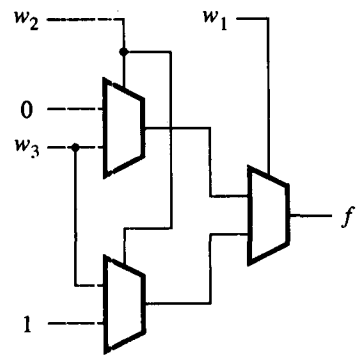
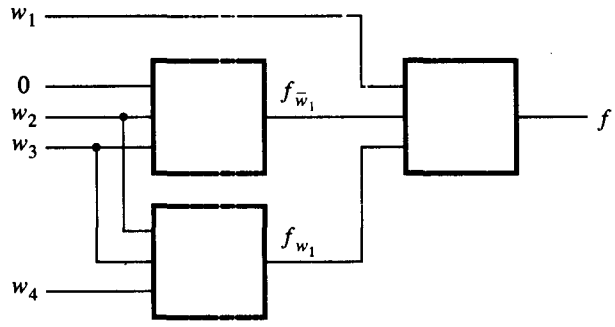


Figure 6.13 The circuit synthesized in Example 6.7.

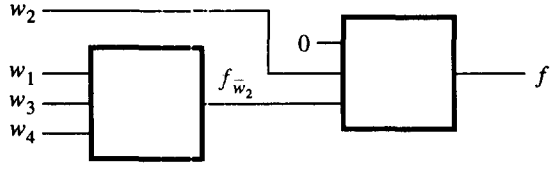
Expansion in terms of w_1 produces

$$\begin{aligned} f &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\ &= \overline{w}_1(\overline{w}_2 w_3 + w_2\overline{w}_3 + w_2\overline{w}_3 w_4) + w_1(\overline{w}_2 w_3 + w_2\overline{w}_3 w_4 + \overline{w}_2\overline{w}_4) \\ &= \overline{w}_1(\overline{w}_2 w_3 + w_2\overline{w}_3) + w_1(\overline{w}_2 w_3 + w_2\overline{w}_3 w_4 + \overline{w}_2\overline{w}_4) \end{aligned}$$

A circuit with three 3-LUTs that implements this expression is shown in Figure 6.14*a*. Decomposition of the function using w_2, instead of w_1, gives



(a) Using three 3-LUTs



(b) Using two 3-LUTs

Figure 6.14 Circuits synthesized in Example 6.8.

$$f = \overline{w}_2 f_{\overline{w}_2} + w_2 f_{w_2}$$
$$= \overline{w}_2(w_3 + w_1\overline{w}_4) + w_2(\overline{w}_1\overline{w}_3 + \overline{w}_3 w_4)$$

Observe that $\overline{f}_{\overline{w}_2} = f_{w_2}$; hence only two 3-LUTs are needed, as illustrated in Figure 6.14*b*. The LUT on the right implements the two-variable function $\overline{w}_2 f_{\overline{w}_2} + w_2 \overline{f}_{\overline{w}_2}$.

Since it is possible to implement any logic function using multiplexers, general-purpose chips exist that contain multiplexers as their basic logic resources. Both Actel Corporation [2] and QuickLogic Corporation [3] offer FPGAs in which the logic block comprises an arrangement of multiplexers. Texas Instruments offers gate array chips that have multiplexer-based logic blocks [4].

6.2 DECODERS

Decoder circuits are used to decode encoded information. A binary decoder, depicted in Figure 6.15, is a logic circuit with n inputs and 2^n outputs. Only one output is asserted at a time, and each output corresponds to one valuation of the inputs. The decoder also has an enable input, *En*, that is used to disable the outputs; if $En = 0$, then none of the decoder outputs is asserted. If $En = 1$, the valuation of $w_{n-1} \cdots w_1 w_0$ determines which of the outputs is asserted. An n-bit binary code in which exactly one of the bits is set to 1 at a time is referred to as *one-hot encoded*, meaning that the single bit that is set to 1 is deemed to be "hot." The outputs of a binary decoder are one-hot encoded.

A 2-to-4 decoder is given in Figure 6.16. The two data inputs are w_1 and w_0. They represent a two-bit number that causes the decoder to assert one of the outputs $y_0, \ldots, y_3$. Although a decoder can be designed to have either active-high or active-low outputs, in Figure 6.16 active-high outputs are assumed. Setting the inputs $w_1 w_0$ to 00, 01, 10, or 11 causes the output y_0, y_1, y_2, or y_3 to be set to 1, respectively. A graphical symbol for the decoder is given in part (*b*) of the figure, and a logic circuit is shown in part (*c*).

Larger decoders can be built using the sum-of-products structure in Figure 6.16*c*, or else they can be constructed from smaller decoders. Figure 6.17 shows how a 3-to-8 decoder is built with two 2-to-4 decoders. The w_2 input drives the enable inputs of the two decoders. The top decoder is enabled if $w_2 = 0$, and the bottom decoder is enabled if $w_2 = 1$. This concept can be applied for decoders of any size. Figure 6.18 shows how five 2-to-4 decoders can be used to construct a 4-to-16 decoder. Because of its treelike structure, this type of circuit is often referred to as a *decoder tree*.

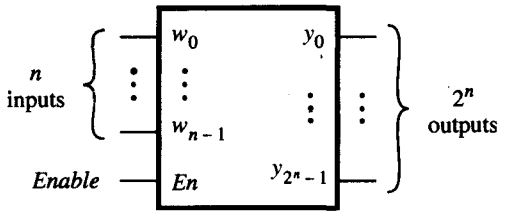
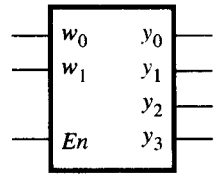


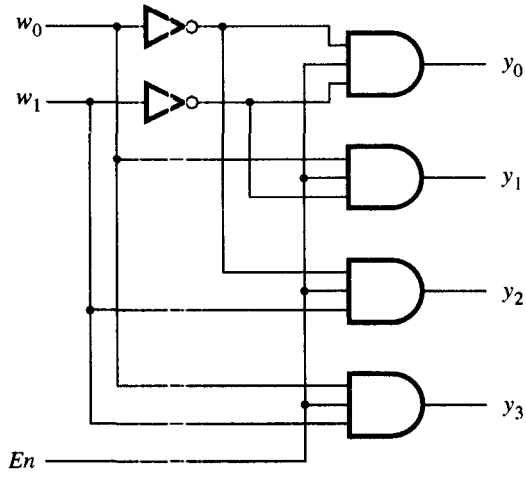
Figure 6.15 An n-to-2^n binary decoder.

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



(b) Graphical symbol



(c) Logic circuit

Figure 6.16 A 2-to-4 decoder.

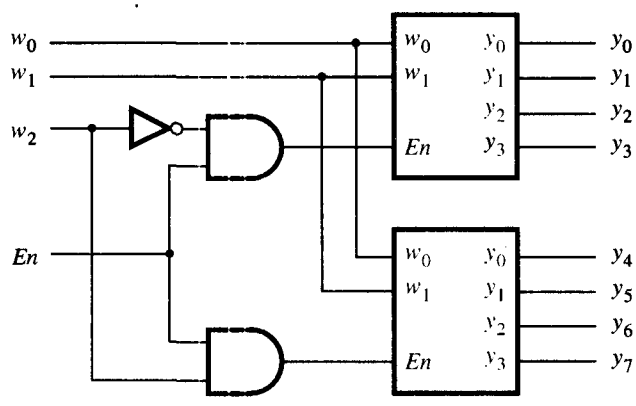


Figure 6.17 A 3-to-8 decoder using two 2-to-4 decoders.

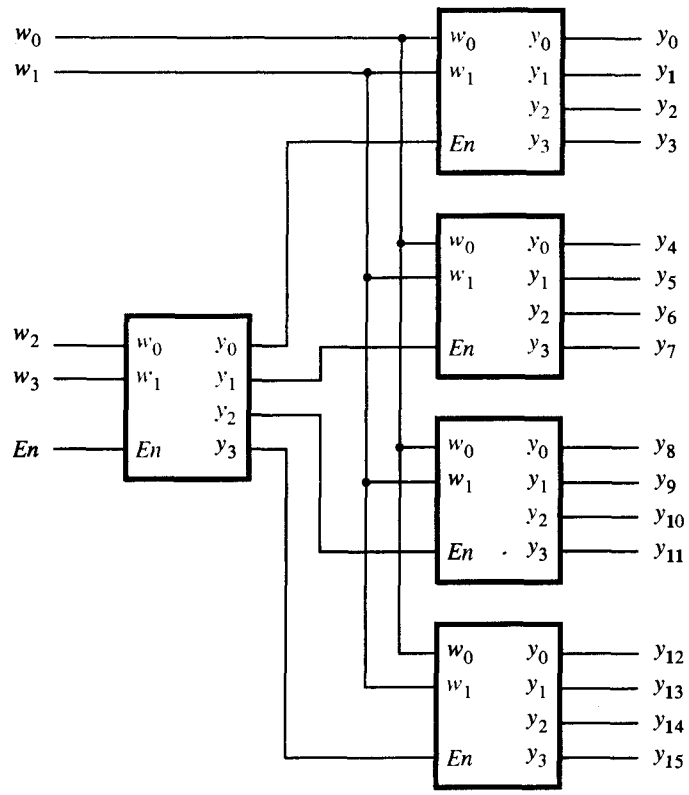


Figure 6.18 A 4-to-16 decoder built using a decoder tree.

Example 6.9

Decoders are useful for many practical purposes. In Figure 6.2*c* we showed the sum-of-products implementation of the 4-to-1 multiplexer, which requires AND gates to distinguish the four different valuations of the select inputs s_1 and s_0. Since a decoder evaluates the values on its inputs, it can be used to build a multiplexer as illustrated in Figure 6.19. The enable input of the decoder is not needed in this case, and it is set to 1. The four outputs of the decoder represent the four valuations of the select inputs.

Example 6.10

In Figure 3.59 we showed how a 2-to-1 multiplexer can be constructed using two tri-state buffers. This concept can be applied to any size of multiplexer, with the addition of a decoder. An example is shown in Figure 6.20. The decoder enables one of the tri-state buffers for each valuation of the select lines, and that tri-state buffer drives the output, f, with the selected data input. We have now seen that multiplexers can be implemented in various ways. The choice of whether to employ the sum-of-products form, transmission gates, or tri-state buffers depends on the resources available in the chip being used. For

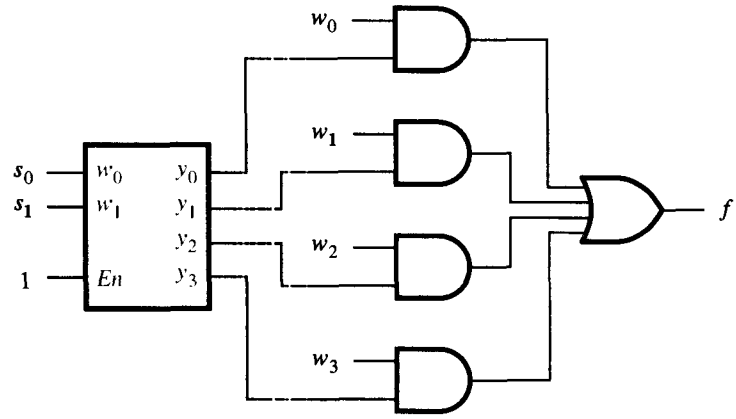


Figure 6.19 A 4-to-1 multiplexer built using a decoder.

instance, most FPGAs that use lookup tables for their logic blocks do not contain tri-state buffers. Hence multiplexers must be implemented in the sum-of-products form using the lookup tables (see problem 6.15).

6.2.1 DEMULTIPLEXERS

We showed in section 6.1 that a multiplexer has one output, n data inputs, and $\lceil \log_2 n \rceil$ select inputs. The purpose of the multiplexer circuit is to *multiplex* the n data inputs onto the single data output under control of the select inputs. A circuit that performs the opposite function, namely, placing the value of a single data input onto multiple data outputs, is

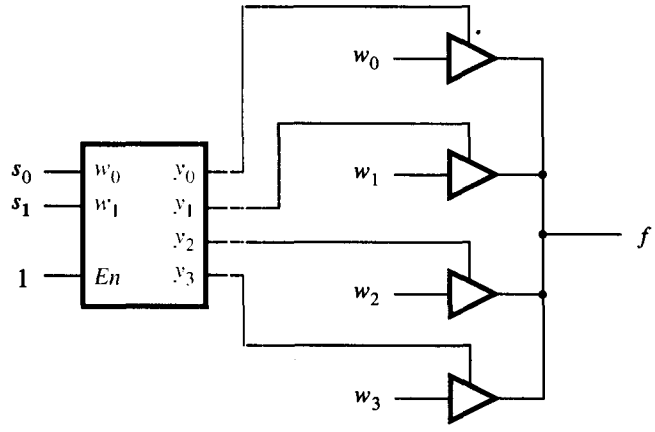


Figure 6.20 A 4-to-1 multiplexer built using a decoder and tri-state buffers.

called a *demultiplexer*. The demultiplexer can be implemented using a decoder circuit. For example, the 2-to-4 decoder in Figure 6.16 can be used as a 1-to-4 demultiplexer. In this case the *En* input serves as the data input for the demultiplexer, and the y_0 to y_3 outputs are the data outputs. The valuation of w_1w_0 determines which of the outputs is set to the value of *En*. To see how the circuit works, consider the truth table in Figure 6.16*a*. When $En = 0$, all the outputs are set to 0, including the one selected by the valuation of w_1w_0. When $En = 1$, the valuation of w_1w_0 sets the appropriate output to 1.

In general, an n-to-2^n decoder circuit can be used as a 1-to-n demultiplexer. However, in practice decoder circuits are used much more often as decoders rather than as demultiplexers. In many applications the decoder's *En* input is not actually needed; hence it can be omitted. In this case the decoder always asserts one of its data outputs, $y_0, \ldots, y_{2^n-1}$, according to the valuation of the data inputs, $w_{n-1} \ldots w_0$. Example 6.11 uses a decoder that does not have the *En* input.

Example 6.11

One of the most important applications of decoders is in memory blocks, which are used to store information. Such memory blocks are included in digital systems, such as computers, where there is a need to store large amounts of information electronically. One type of memory block is called a *read-only memory* (ROM). A ROM consists of a collection of storage cells, where each cell permanently stores a single logic value, either 0 or 1. Figure 6.21 shows an example of a ROM block. The storage cells are arranged in 2^m rows with n

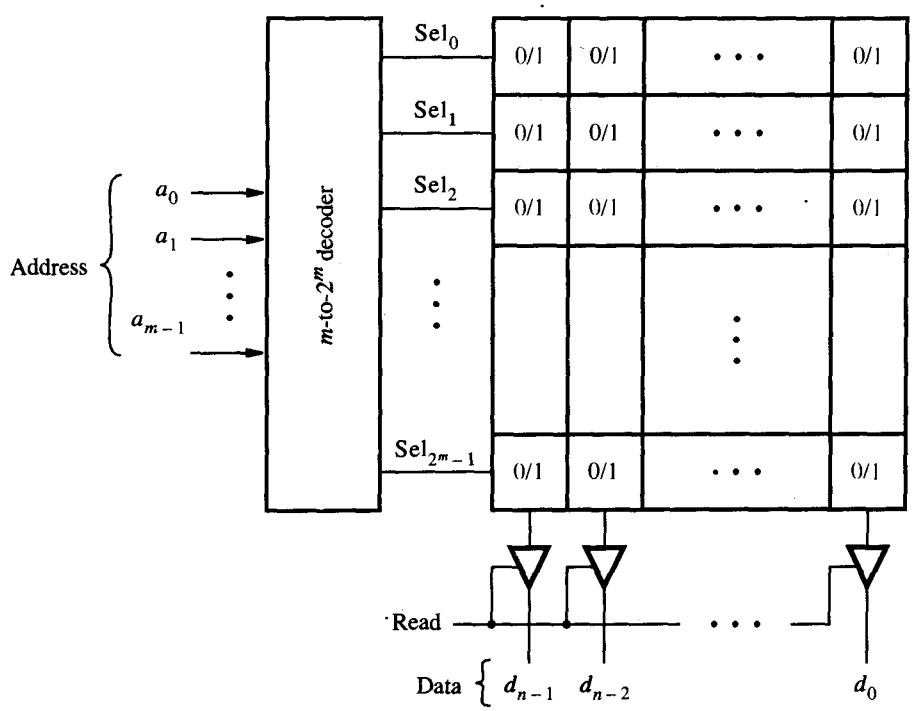


Figure 6.21 A $2^m \times n$ read-only memory (ROM) block.

cells per row. Thus each row stores n bits of information. The location of each row in the ROM is identified by its *address*. In the figure the row at the top of the ROM has address 0, and the row at the bottom has address $2^m - 1$. The information stored in the rows can be accessed by asserting the select lines, Sel_0 to Sel_{2^m-1}. As shown in the figure, a decoder with m inputs and 2^m outputs is used to generate the signals on the select lines. Since the inputs to the decoder choose the particular address (row) selected, they are called the *address* lines. The information stored in the row appears on the data outputs of the ROM, $d_{n-1}, \ldots, d_0$, which are called the *data* lines. Figure 6.21 shows that each data line has an associated tri-state buffer that is enabled by the ROM input named *Read*. To access, or *read*, data from the ROM, the address of the desired row is placed on the address lines and *Read* is set to 1.

Many different types of memory blocks exist. In a ROM the stored information can be read out of the storage cells, but it cannot be changed (see problem 6.31). Another type of ROM allows information to be both read out of the storage cells and stored, or *written*, into them. Reading its contents is the normal operation, whereas writing requires a special procedure. Such a memory block is called a programmable ROM (PROM). The storage cells in a PROM are usually implemented using EEPROM transistors. We discussed EEPROM transistors in section 3.10 to show how they are used in PLDs. Other types of memory blocks are discussed in section 10.1.

6.3 ENCODERS

An encoder performs the opposite function of a decoder. It encodes given information into a more compact form.

6.3.1 BINARY ENCODERS

A *binary encoder* encodes information from 2^n inputs into an n-bit code, as indicated in Figure 6.22. Exactly one of the input signals should have a value of 1, and the outputs present the binary number that identifies which input is equal to 1. The truth table for a 4-to-2 encoder is provided in Figure 6.23*a*. Observe that the output y_0 is 1 when either input w_1 or w_3 is 1, and output y_1 is 1 when input w_2 or w_3 is 1. Hence these outputs can be generated by the circuit in Figure 6.23*b*. Note that we assume that the inputs are one-hot

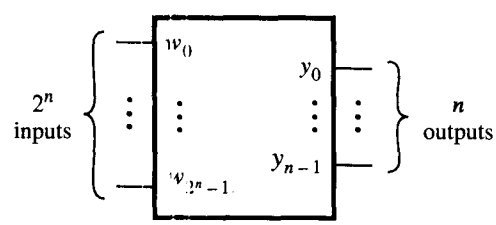
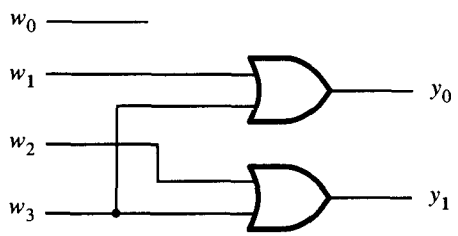


Figure 6.22 A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Figure 6.23 A 4-to-2 binary encoder.

encoded. All input patterns that have multiple inputs set to 1 are not shown in the truth table, and they are treated as don't-care conditions.

Encoders are used to reduce the number of bits needed to represent given information. A practical use of encoders is for transmitting information in a digital system. Encoding the information allows the transmission link to be built using fewer wires. Encoding is also useful if information is to be stored for later use because fewer bits need to be stored.

6.3.2 PRIORITY ENCODERS

Another useful class of encoders is based on the priority of input signals. In a *priority encoder* each input has a priority level associated with it. The encoder outputs indicate the active input that has the highest priority. When an input with a high priority is asserted, the other inputs with lower priority are ignored. The truth table for a 4-to-2 priority encoder is shown in Figure 6.24. It assumes that w_0 has the lowest priority and w_3 the highest. The outputs y_1 and y_0 represent the binary number that identifies the highest priority input set to 1. Since it is possible that none of the inputs is equal to 1, an output, z, is provided to indicate this condition. It is set to 1 when at least one of the inputs is equal to 1. It is set to 0 when all inputs are equal to 0. The outputs y_1 and y_0 are not meaningful in this case, and hence the first row of the truth table can be treated as a don't-care condition for y_1 and y_0.

The behavior of the priority encoder is most easily understood by first considering the last row in the truth table. It specifies that if input w_3 is 1, then the outputs are set to $y_1y_0 = 11$. Because w_3 has the highest priority level, the values of inputs w_2, w_1, and w_0

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 6.24 Truth table for a 4-to-2 priority encoder.

do not matter. To reflect the fact that their values are irrelevant, w_2, w_1, and w_0 are denoted by the symbol x in the truth table. The second-last row in the truth table stipulates that if $w_2 = 1$, then the outputs are set to $y_1y_0 = 10$, but only if $w_3 = 0$. Similarly, input w_1 causes the outputs to be set to $y_1y_0 = 01$ only if both w_3 and w_2 are 0. Input w_0 produces the outputs $y_1y_0 = 00$ only if w_0 is the only input that is asserted.

A logic circuit that implements the truth table can be synthesized by using the techniques developed in Chapter 4. However, a more convenient way to derive the circuit is to define a set of intermediate signals, $i_0, \ldots, i_3$, based on the observations above. Each signal, i_k, is equal to 1 only if the input with the same index, w_k, represents the highest-priority input that is set to 1. The logic expressions for $i_0, \ldots, i_3$ are

$$i_0 = \overline{w}_3\overline{w}_2\overline{w}_1w_0$$
$$i_1 = \overline{w}_3\overline{w}_2w_1$$
$$i_2 = \overline{w}_3w_2$$
$$i_3 = w_3$$

Using the intermediate signals, the rest of the circuit for the priority encoder has the same structure as the binary encoder in Figure 6.23, namely

$$y_0 = i_1 + i_3$$
$$y_1 = i_2 + i_3$$

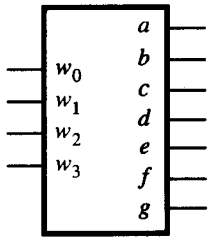
The output z is given by

$$z = i_1 + i_2 + i_3 + i_4$$

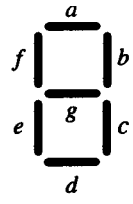
6.4 CODE CONVERTERS

The purpose of the decoder and encoder circuits is to convert from one type of input encoding to a different output encoding. For example, a 3-to-8 binary decoder converts from a binary number on the input to a one-hot encoding at the output. An 8-to-3 binary encoder performs the opposite conversion. There are many other possible types of code converters. One common example is a BCD-to-7-segment decoder, which converts one binary-coded decimal (BCD) digit into information suitable for driving a digit-oriented

display. As illustrated in Figure 6.25*a*, the circuit converts the BCD digit into seven signals that are used to drive the segments in the display. Each segment is a small light-emitting diode (LED), which glows when driven by an electrical signal. The segments are labeled from *a* to *g* in the figure. The truth table for the BCD-to-7-segment decoder is given in Figure 6.25*c*. For each valuation of the inputs $w_3, \ldots, w_0$, the seven outputs are set to display the appropriate BCD digit. Note that the last 6 rows of a complete 16-row truth table are not shown. They represent don't-care conditions because they are not legal BCD codes and will never occur in a circuit that deals with BCD data. A circuit that implements the truth table can be derived using the synthesis techniques discussed in Chapter 4. Finally, we should note that although the word *decoder* is traditionally used for this circuit, a more appropriate term is *code converter*. The term *decoder* is more appropriate for circuits that produce one-hot encoded outputs.



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

(c) Truth table

Figure 6.25 A BCD-to-7-segment display code converter.

6.5 ARITHMETIC COMPARISON CIRCUITS

Chapter 5 presented arithmetic circuits that perform addition, subtraction, and multiplication of binary numbers. Another useful type of arithmetic circuit compares the relative sizes of two binary numbers. Such a circuit is called a *comparator*. This section considers the design of a comparator that has two n-bit inputs, A and B, which represent unsigned binary numbers. The comparator produces three outputs, called $AeqB$, $AgtB$, and $AltB$. The $AeqB$ output is set to 1 if A and B are equal. The $AgtB$ output is 1 if A is greater than B, and the $AltB$ output is 1 if A is less than B.

The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of A and B. However, even for moderate values of n, the truth table is large. A better approach is to derive the comparator circuit by considering the bits of A and B in pairs. We can illustrate this by a small example, where $n = 4$.

Let $A = a_3a_2a_1a_0$ and $B = b_3b_2b_1b_0$. Define a set of intermediate signals called i_3, i_2, i_1, and i_0. Each signal, i_k, is 1 if the bits of A and B with the same index are equal. That is, $i_k = \overline{a_k \oplus b_k}$. The comparator's $AeqB$ output is then given by

$$AeqB = i_3i_2i_1i_0$$

An expression for the $AgtB$ output can be derived by considering the bits of A and B in the order from the most-significant bit to the least-significant bit. The first bit-position, k, at which a_k and b_k differ determines whether A is less than or greater than B. If $a_k = 0$ and $b_k = 1$, then $A < B$. But if $a_k = 1$ and $b_k = 0$, then $A > B$. The $AgtB$ output is defined by

$$AgtB = a_3\bar{b}_3 + i_3a_2\bar{b}_2 + i_3i_2a_1\bar{b}_1 + i_3i_2i_1a_0\bar{b}_0$$

The i_k signals ensure that only the first digits, considered from the left to the right, of A and B that differ determine the value of $AgtB$.

The $AltB$ output can be derived by using the other two outputs as

$$AltB = \overline{AeqB + AgtB}$$

A logic circuit that implements the four-bit comparator circuit is shown in Figure 6.26. This approach can be used to design a comparator for any value of n.

6.6 VERILOG FOR COMBINATIONAL CIRCUITS

Having presented a number of useful building block circuits, we will now consider how such circuits can be described in Verilog. Rather than using gates or logic expressions, we will specify the circuits in terms of their behavior. We will also give a more rigorous description of previously used behavioral Verilog constructs and introduce some new ones.

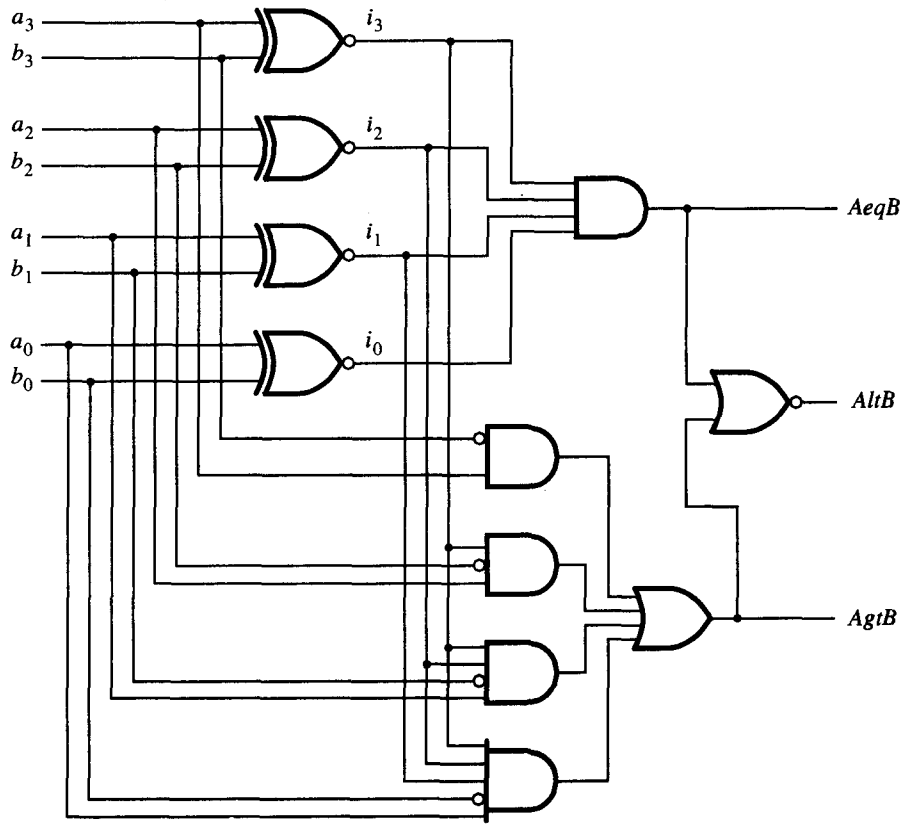


Figure 6.26 A four-bit comparator circuit.

6.6.1 THE CONDITIONAL OPERATOR

In a logic circuit it is often necessary to choose between several possible signals or values based on the state of some condition. A typical example is a multiplexer circuit in which the output is equal to the data input signal chosen by the valuation of the select inputs. For simple implementation of such choices Verilog provides a *conditional* operator (?:) which assigns one of two values depending on a conditional expression. It involves three operands used in the syntax

conditional_expression ? true_expression : false_expression

If the conditional expression evaluates to 1 (true), then the value of true_expression is chosen; otherwise, the value of false_expression is chosen. For example, the statement

A = (B < C) ? (D + 5) : (D + 2);

```
module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output f;

  assign f = s ? w1 : w0;

endmodule
```

Figure 6.27 A 2-to-1 multiplexer specified using the conditional operator.

means that if B is less than C, the value of A will be $D + 5$ or else A will have the value $D+2$. We used parentheses in the expression to improve readability; they are not necessary. The conditional operator can be used both in continuous assignment statements and in procedural statements inside an **always** block.

Example 6.12 A 2-to-1 multiplexer can be defined using the conditional operator in an **assign** statement as shown in Figure 6.27. The module, named *mux2to1*, has the inputs w_0, w_1, and s, and the output f. The signal s is used for the selection criterion. The output f is equal to w_1 if the select input s has the value 1; otherwise, f is equal to w_0. Figure 6.28 shows how the same multiplexer can be defined by using the conditional operator inside an **always** block.

The same approach can be used to define a 4-to-1 multiplexer by nesting the conditional operators as indicated in Figure 6.29. The module is named *mux4to1*. Its two select inputs, which are called s_1 and s_0 in Figure 6.2, are represented by the two-bit vector S. The first conditional expression tests the value of bit s_1. If $s_1 = 1$, then s_0 is tested and f is set to w_3 if $s_0 = 1$ and f is set to w_2 if $s_0 = 0$. This corresponds to the third and fourth rows of the truth table in Figure 6.2*b*. Similarly, if $s_1 = 0$ the conditional operator on the right chooses $f = w_1$ if $s_0 = 1$ and $f = w_0$ if $s_0 = 0$, thus realizing the first two rows of the truth table.

```
module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output f;
  reg f;

  always @(w0 or w1 or s)
    f = s ? w1 : w0;

endmodule
```

Figure 6.28 An alternative specification of a 2-to-1 multiplexer using the conditional operator.

```
module mux4to1 (w0, w1, w2, w3, S, f);
  input w0, w1, w2, w3;
  input [1:0] S;
  output f;

  assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

Figure 6.29 A 4-to-1 multiplexer specified using the conditional operator.

6.6.2 THE IF-ELSE STATEMENT

We have already used the **if-else** statement in previous chapters. It has the syntax

```
if (conditional_expression) statement;
else statement;
```

The conditional expression may use the operators given in Table A.1. If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed or else the second statement (or a block of statements) is executed.

Example 6.13

Figure 6.30 shows how the **if-else** statement can be used to describe a 2-to-1 multiplexer. The **if** clause states that f is assigned the value of w_0 when $s = 0$. Else, f is assigned the value of w_1.

The **if-else** statement can be used to implement larger multiplexers. A 4-to-1 multiplexer is shown in Figure 6.31. The **if-else** clauses set f to the value of one of the inputs $w_0, \ldots, w_3$,

```
module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output f;
  reg f;

  always @(w0 or w1 or s)
    if (s == 0)
      f = w0;
    else
      f = w1;

endmodule
```

Figure 6.30 Code for a 2-to-1 multiplexer using the **if-else** statement.

```
module mux4to1 (w0, w1, w2, w3, S, f);
  input w0, w1, w2, w3;
  input [1:0] S;
  output f;
  reg f;

  always @(w0 or w1 or w2 or w3 or S)
    if (S == 2'b00)
      f = w0;
    else if (S == 2'b01)
      f = w1;
    else if (S == 2'b10)
      f = w2;
    else if (S == 2'b11)
      f = w3;

endmodule
```

Figure 6.31 Code for a 4-to-1 multiplexer using the **if-else** statement.

depending on the valuation of S. Compiling the code results in the circuit shown in Figure 6.2*c*.

Another way of defining the same circuit is presented in Figure 6.32. In this case, a four-bit vector W is defined instead of single-bit signals w_0, w_1, w_2, and w_3. Also, the four different values of S are specified as decimal rather than binary numbers.

Example 6.14 Figure 6.4 shows how a 16-to-1 multiplexer is built using five 4-to-1 multiplexers. Figure 6.33 presents Verilog code for this circuit using five instantiations of the *mux4to1* module. The data inputs to the *mux16to1* module are the 16-bit vector W, and the select inputs are the four-bit vector $S16$. In the Verilog code signal names are needed for the outputs of the four 4-to-1 multiplexers on the left of Figure 6.4. A four-bit signal named M is used for this purpose. The first multiplexer instantiated, *Mux1*, corresponds to the multiplexer at the top left of Figure 6.4. Its first four ports, which correspond to $w_0, \ldots, w_3$ in Figure 6.31, are driven by the signals $W[0], \ldots, W[3]$. The syntax $S16[1:0]$ is used to attach the signals $S16[1]$ and $S16[0]$ to the two-bit S port of the *mux4to1* module. The $M[0]$ signal is connected to the multiplexer's output port. Similarly, *Mux2*, *Mux3*, and *Mux4* are instantiations of the next three multiplexers on the left. The multiplexer on the right of Figure 6.4 is instantiated as *Mux5*. The signals $M[0], \ldots, M[3]$ are connected to its data inputs, and bits $S16[3]$ and $S16[2]$, which are specified by the syntax $S16[3:2]$, are attached to the select inputs. The output port generates the *mux16to1* output f. Compiling the code results in the multiplexer function

$$f = \bar{s}_3\bar{s}_2\bar{s}_1\bar{s}_0w_0 + \bar{s}_3\bar{s}_2\bar{s}_1s_0w_1 + \bar{s}_3\bar{s}_2s_1\bar{s}_0w_2 + \cdots + s_3s_2s_1\bar{s}_0w_{14} + s_3s_2s_1s_0w_{15}$$

```
module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output f;
  reg f;

  always @(W or S)
    if (S == 0)
      f = W[0];
    else if (S == 1)
      f = W[1];
    else if (S == 2)
      f = W[2];
    else if (S == 3)
      f = W[3];

endmodule
```

Figure 6.32 Alternative specification of a 4-to-1 multiplexer.

```
module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output f;
  wire [0:3] M;

  mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
  mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
  mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
  mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
  mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule
```

Figure 6.33 Hierarchical code for a 16-to-1 multiplexer.

Since the *mux4to1* module is being instantiated in the code of Figure 6.33, it is necessary to either include the code of Figure 6.32 in the same file as the *mux16to1* module or place the *mux4to1* module in a separate file in the same directory, or a directory with a specified path so that the Verilog compiler can find it. Observe that if the code in Figure 6.31 were used as the required *mux4to1* module, then we would have to list the ports separately, as in $W[0]$, $W[1]$, $W[2]$, $W[3]$, rather than as the vector $W[0:3]$.

6.6.3 THE CASE STATEMENT

The **if-else** statement provides the means for choosing an alternative based on the value of an expression. When there are many possible alternatives, the code based on this statement may become awkward to read. Instead, it is often possible to use the Verilog **case** statement which is defined as

```
case (expression)
   alternative1: statement;
   alternative2: statement;
   .
   .
   .
   alternativej: statement;
   [default: statement;]
endcase
```

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only) is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included. Otherwise, the Verilog compiler will synthesize memory elements to deal with the unspecified possibilities; we will discuss this issue in Chapter 7.

Example 6.15 The **case** statement can be used to define a 4-to-1 multiplexer as shown in Figure 6.34. The four values that the select vector S can have are given as decimal numbers, but they could also be given as binary numbers.

```
module mux4to1 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output f;
   reg f;

   always @(W or S)
      case (S)
         0: f = W[0];
         1: f = W[1];
         2: f = W[2];
         3: f = W[3];
      endcase

endmodule
```

Figure 6.34 A 4-to-1 multiplexer defined using the **case** statement.

Example 6.16

Figure 6.35 shows how a **case** statement can be used to describe the truth table for a 2-to-4 binary decoder. The module is called *dec2to4*. The data inputs are the two-bit vector W, and the enable input is En. The four outputs are represented by the four-bit vector Y.

In the truth table for the decoder in Figure 6.16*a*, the inputs are listed in the order $En\ w_1\ w_0$. To represent these three signals in the controlling expression, the Verilog code uses the concatenate operator to combine the En and W signals into a three-bit vector. The four alternatives in the **case** statement correspond to the truth table in Figure 6.16*a* where $En = 1$, and the decoder outputs have the same patterns as in the first four rows of the truth table. The last clause uses the **default** keyword and sets the decoder outputs to 0000, because it represents all other cases, namely those where $En = 0$.

Example 6.17

The 2-to-4 decoder can be specified using a combination of **if-else** and **case** statements as given in Figure 6.36. The **case** alternatives are evaluated if $En = 1$; otherwise, all four bits of the output Y are set to the value 0.

Example 6.18

The tree structure of the 4-to-16 decoder in Figure 6.18 can be defined as shown in Figure 6.37. The inputs are a four-bit vector W and an enable signal En. The outputs are represented by the 16-bit vector Y. The circuit uses five instances of the 2-to-4 decoder defined in either Figure 6.35 or 6.36. The outputs of the leftmost decoder in Figure 6.18 are denoted as the four-bit vector M in Figure 6.37.

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output [0:3] Y;
   reg [0:3] Y;

   always @(W or En)
      case ({En, W})
         3'b100: Y = 4'b1000;
         3'b101: Y = 4'b0100;
         3'b110: Y = 4'b0010;
         3'b111: Y = 4'b0001;
         default: Y = 4'b0000;
      endcase

endmodule
```

Figure 6.35 Verilog code for a 2-to-4 binary decoder.

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output [0:3] Y;
   reg [0:3] Y;

   always @(W or En)
   begin
      if (En == 0)
         Y = 4'b0000;
      else
         case (W)
            0: Y = 4'b1000;
            1: Y = 4'b0100;
            2: Y = 4'b0010;
            3: Y = 4'b0001;
         endcase
   end

endmodule
```

Figure 6.36 Alternative code for a 2-to-4 binary decoder.

```
module dec4to16 (W, Y, En);
   input [3:0] W;
   input En;
   output [0:15] Y;
   wire [0:3] M;

   dec2to4 Dec1 (W[3:2], M[0:3], En);
   dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
   dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
   dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
   dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

Figure 6.37 Verilog code for a 4-to-16 decoder.

Example 6.19 Another example of a **case** statement is given in Figure 6.38. The module, *seg7*, represents the BCD-to-7-segment decoder in Figure 6.25. The BCD input is the four-bit vector named *bcd*, and the seven outputs are the seven-bit vector named *leds*. The **case** alternatives are listed so that they resemble the truth table in Figure 6.25*c*. Note that there is a comment to the right of the case statement, which labels the seven outputs with the letters

```
module seg7 (bcd, leds);
  input [3:0] bcd;
  output [1:7] leds;
  reg [1:7] leds;

  always @(bcd)
    case (bcd)      //abcdefg
      0: leds = 7'b1111110;
      1: leds = 7'b0110000;
      2: leds = 7'b1101101;
      3: leds = 7'b1111001;
      4: leds = 7'b0110011;
      5: leds = 7'b1011011;
      6: leds = 7'b1011111;
      7: leds = 7'b1110000;
      8: leds = 7'b1111111;
      9: leds = 7'b1111011;
      default: leds = 7'bx;
    endcase

endmodule
```

Figure 6.38 Code for a BCD-to-7-segment decoder.

from *a* to *g*. These labels indicate to the reader the correlation between the bits of the *leds* vector in the Verilog code and the seven segments in Figure 6.25*b*. The final **case** alternative sets all seven bits of *leds* to *x*. Recall that *x* is used in Verilog to denote a don't-care condition. This alternative represents the don't-care conditions discussed for Figure 6.25, which are the cases where the *bcd* input does not represent a valid BCD digit.

Example 6.20

An arithmetic logic unit (ALU) is a logic circuit that performs various Boolean and arithmetic operations on n-bit operands. In section 3.5 we discussed a family of standard chips called the 7400-series chips. We said that some of these chips contain basic logic gates, and others provide commonly used logic circuits. One example of an ALU is the chip called the 74381. Table 6.1 specifies the functionality of this chip. It has 2 four-bit data inputs, A and B, a three-bit select input, S, and a four-bit output, F. As the table shows, F is defined by various arithmetic or Boolean operations on the inputs A and B. In this table + means arithmetic addition, and − means arithmetic subtraction. To avoid confusion, the table uses the words XOR, OR, and AND for the Boolean operations. Each Boolean operation is done in a bitwise fashion. For example, $F = A$ AND B produces the four-bit result $f_0 = a_0b_0$, $f_1 = a_1b_1, f_2 = a_2b_2$, and $f_3 = a_3b_3$.

Figure 6.39 shows how the functionality of the 74381 ALU can be described in Verilog code. The case statement shown corresponds directly to Table 6.1. To check the functionality of the code, we synthesized a circuit for implementation in a PLD, and show a timing

Table 6.1 The functionality of the 74381 ALU.

Operation	Inputs s_2 s_1 s_0	Outputs F
Clear	0 0 0	0 0 0 0
B–A	0 0 1	$B - A$
A–B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

```
// 74381 ALU
module alu (s, A, B, F);
   input [2:0] s;
   input [3:0] A, B;
   output [3:0] F;
   reg [3:0] F;

   always @(s or A or B)
      case (s)
         0: F = 4'b0000;
         1: F = B - A;
         2: F = A - B;
         3: F = A + B;
         4: F = A ^ B;
         5: F = A | B;
         6: F = A & B;
         7: F = 4'b1111;
      endcase

endmodule
```

Figure 6.39 Code that represents the functionality of the 74381 ALU chip.

simulation in Figure 6.40. For each valuation of s, the circuit generates the appropriate Boolean or arithmetic operation.

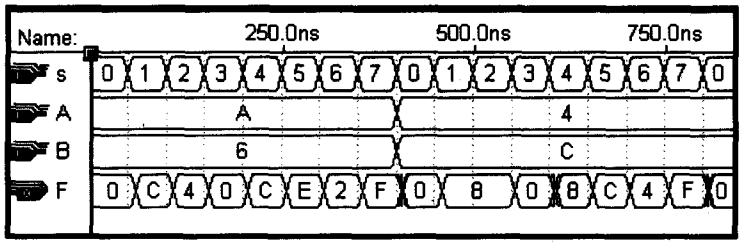


Figure 6.40 Timing simulation for the code in Figure 6.39.

The Casex and Casez Statements

In the **case** statement it is possible to use the logic values 0, 1, z, and x in the **case** alternatives. A bit-by-bit comparison is used to determine the match between the expression and one of the alternatives.

Verilog provides two variants of the **case** statement that treat the z and x values in a different way. The **casez** statement treats all z values in the case alternatives and the controlling expression as don't cares. The **casex** statement treats all z and x values as don't cares.

Example 6.21

Figure 6.41 gives Verilog code for the priority encoder defined in Figure 6.24. The desired priority scheme is realized by using a **casex** statement. The first alternative specifies that, if the input w_3 is 1, then the output is set to $y_1y_0 = 3$. This assignment does not depend on the values of inputs w_2, w_1, or w_0; hence their values do not matter. The other alternatives in the **casex** statement are evaluated only if $w_3 = 0$. The second alternative states that if w_2 is 1, then $y_1y_0 = 2$. If $w_2 = 0$, then the next alternative results in $y_1y_0 = 1$ if $w_1 = 1$. If $w_3 = w_2 = w_1 = 0$ and $w_0 = 1$, then the fourth alternative results in $y_1y_0 = 0$.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is set to 1 at the start of the **always** block. If none of the four alternatives matches the value of W, then the **default** clause is executed. It consists of a two-statement block that resets z to 0 and indicates that the Y output can be set to any pattern because it will be ignored.

6.6.4 THE FOR LOOP

If the structure of a desired circuit exhibits a certain regularity, it may be possible to define the circuit using a **for** loop. We introduced the **for** loop in section 5.5.4, where it was useful in a generic specification of a ripple-carry adder. The **for** loop has the syntax

for (initial_index; terminal_index; increment) statement;

```
module priority (W, Y, z);
  input [3:0] W;
  output [1:0] Y;
  output z;
  reg [1:0] Y;
  reg z;

  always @(W)
  begin
    z = 1;
    casex (W)
      4'b1xxx: Y = 3;
      4'b01xx: Y = 2;
      4'b001x: Y = 1;
      4'b0001: Y = 0;
      default: begin
                 z = 0;
                 Y = 2'bx;
               end
    endcase
  end

endmodule
```

Figure 6.41 Verilog code for a priority encoder.

A loop control variable, which has to be of type **integer**, is set to the value given as the initial index. It is used in the statement or a block of statements delineated by **begin** and **end** keywords. After each iteration, the control variable is changed as defined in the increment. The iterations end after the control variable has reached the terminal index.

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit. In Figure 5.29 the **for** loop was used to define a cascade of full-adder subcircuits to form an n-bit ripple-carry adder. The **for** loop can be used to define many other structures as illustrated by the next two examples.

Example 6.22 Figure 6.42 shows how the **for** loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the **if-else** statement four times, for $k = 0, \ldots, 3$. The first loop iteration sets $y_0 = 1$ if $W = 0$ and $En = 1$. Similarly, the other three iterations set the values of y_1, y_2, and y_3 according to the values of W and En.

This arrangement can be used to specify a large n-to-2^n decoder simply by increasing the sizes of vectors W and Y accordingly, and making $n - 1$ be the terminal index value of k.

```
module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output [0:3] Y;
   reg [0:3] Y;
   integer k;

   always @(W or En)
      for (k = 0; k <= 3; k = k+1)
         if ((W == k) && (En == 1))
            Y[k] = 1;
         else
            Y[k] = 0;

endmodule
```

Figure 6.42 A 2-to-4 binary decoder specified using the **for** loop.

Example 6.23 The priority encoder of Figure 6.24 can be defined by the Verilog code in Figure 6.43. In the **always** block, the output bits y_1 and y_0 are first set to the don't-care state and z is cleared to 0. Then, if one or more of the four inputs $w_3, \ldots, w_0$ is equal to 1, the **for** loop will set the valuation of y_1y_0 to match the index of the highest priority input that has the value 1. Note that each successive iteration through the loop corresponds to a higher priority. Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment. Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of Y established during the previous iterations.

6.6.5 VERILOG OPERATORS

In this section we discuss the Verilog operators that are useful for synthesizing logic circuits. Table 6.2 lists these operators in groups that reflect the type of operation performed. A more complete listing of the operators is given in Table A.1.

To illustrate the results produced by the various operators, we will use three-bit vectors $A[2:0]$, $B[2:0]$ and $C[2:0]$, as well as scalars f and w.

Bitwise Operators

Bitwise operators operate on individual bits of operands. The ~ operator forms the 1's complement of the operand such that the statement

```
C = ~A;
```

produces the result $c_2 = \overline{a}_2$, $c_1 = \overline{a}_1$, and $c_0 = \overline{a}_0$, where a_i and c_i are the bits of the vectors A and C.

```
module priority (W, Y, z);
   input [3:0] W;
   output [1:0] Y;
   output z;
   reg [1:0] Y;
   reg z;
   integer k;

   always @(W)
   begin
      Y = 2'bx;
      z = 0;
      for (k = 0; k < 4; k = k+1)
         if (W[k])
         begin
            Y = k;
            z = 1;
         end
   end

endmodule
```

Figure 6.43 A priority encoder specified using the **for** loop.

Other bitwise operators operate on pairs of bits. The statement

```
C = A & B;
```

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. Similarly, the | and ^ operators perform bitwise OR and XOR operations. The ^~ operator, which can also be written as ~^, produces the XNOR such that

```
C = A ~^ B;
```

gives $c_2 = \overline{a_2 \oplus b_2}$, $c_1 = \overline{a_1 \oplus b_1}$, and $c_0 = \overline{a_0 \oplus b_0}$. If the operands are of unequal size, then the shorter operand is extended by padding 0s to the left.

A scalar function may be assigned a value as a result of a bitwise operation on two vector operands. In this case, it is only the least-significant bits of the operands that are involved in the operation. Hence the statement

```
f = A ^ B;
```

yields $f = a_0 \oplus b_0$.

The bitwise operations may involve operands that include the unknown logic value x. Then the operations are performed according to the truth tables in Figure 6.44. For example, if P = 4'b101x and Q = 4'b1001, then P & Q = 4'b100x while P | Q = 4'b1011.

Table 6.2 Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1
Arithmetic	+	Addition	2
	-	Subtraction	2
	-	2's complement	1
	*	Multiplication	2
	/	Division	2
Relational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal to	2
	<=	Less than or equal to	2
Equality	==	Logical equality	2
	!=	Logical inequality	2
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

Logical Operators

The ! operator has the same effect on a scalar operand as the ~ operator. Thus, f = !w = ~w. But the effect on a vector operand is different, namely if

```
f = !A;
```

then $f = \overline{a_2 + a_1 + a_0}$.

The && operator implements the AND operation such that

```
f = A && B;
```

produces $f = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0)$. Similarly, using the || operator in

```
f = A || B;
```

gives $f = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)$.

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Figure 6.44 Truth tables for bitwise operators.

Reduction Operators

The reduction operators perform an operation on the bits of a single vector operand and produce a one-bit result. Using the & operator in

```
f = &A;
```

produces $f = a_2 \cdot a_1 \cdot a_0$. Similarly,

```
f = ^A;
```

gives $f = a_2 \oplus a_1 \oplus a_0$, and so on. As an example of reduction operator use, consider the parity function discussed in section 5.8. The XOR circuit that computes the parity bit, p, of an n-bit vector X can be defined with the statement

```
p = ^X;
```

Arithmetic Operators

We have already encountered the arithmetic operators in Chapter 5. They perform standard arithmetic operations. Thus

```
C = A + B;
```

puts the three-bit sum of A plus B into C, while

```
C = A - B;
```

puts the difference of A and B into C. The operation

```
C = -A;
```

places the 2's complement of A into C.

The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the Verilog compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

Relational Operators

The relational operators are typically used as conditions in **if-else** and **for** statements. These operators function in the same way as the corresponding operators in the C programming language. An expression that uses the relational operators returns the value 1 if it is evaluated as true, and the value 0 if evaluated as false. If there are any x (unknown) or z bits in the operands, then the expression takes the value x.

Example 6.24 The use of relational operators in the **if-else** statement is illustrated in Figure 6.45. The defined circuit is the four-bit comparator described in section 6.5.

Equality Operators

The expression (A == B) is evaluated as true if A is equal to B and false otherwise. The != operator has the opposite effect. The result is ambiguous (x) if either operand contains x or z values.

Shift Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,

```
B = A << 1;
```

```
module compare (A, B, AeqB, AgtB, AltB);
  input [3:0] A, B;
  output AeqB, AgtB, AltB;
  reg AeqB, AgtB, AltB;

  always @(A or B)
  begin
    AeqB = 0;
    AgtB = 0;
    AltB = 0;
    if (A == B)
      AeqB = 1;
    else if (A > B)
      AgtB = 1;
    else
      AltB = 1;
  end

endmodule
```

Figure 6.45 Verilog code for a four-bit comparator.

results in $b_2 = a_1$, $b_1 = a_0$, and $b_0 = 0$. Similarly,

```
B = A >> 2;
```

yields $b_2 = b_1 = 0$ and $b_0 = a_2$.

Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example,

```
D = {A, B};
```

defines the six-bit vector $D = a_2a_1a_0b_2b_1b_0$. Similarly, the concatenation

```
E = {3'b111, A, 2'b00};
```

produces the eight-bit vector $E = 111a_2a_1a_000$.

Replication Operator

This operator allows repetitive concatenation of the same vector, which is replicated the number of times indicated in the replication constant. For example, {3{A}} is equivalent to writing {A, A, A}. The specification {4{2'b10}} produces the eight-bit vector 10101010.

The replication operator may be used in conjunction with the concatenate operator. For instance, {2{A}, 3{B}} is equivalent to {A, A, B, B, B}. We introduced the concatenate and replication operators in section 5.5.6 and illustrated their use in specifying the adder circuits.

Conditional Operator

The conditional operator is discussed fully in section 6.6.1.

Operator Precedence

The Verilog operators are assumed to have the precedence indicated in Table 6.3. The order of precedence is from top to bottom; operators in the top row have the highest precedence and those in the bottom row have the lowest precedence. The operators listed in the same row have the same precedence.

The designer can use parentheses to change the precedence of operators in Verilog code or remove any possible misinterpretation. It is a good practice to use parentheses to make the code unambiguous and easy to read.

6.6.6 THE GENERATE CONSTRUCT

The Verilog-2001 standard provides a **generate** loop capability that can be used to create multiple instances of subcircuits, which is not available in the previous Verilog standard. A subcircuit may be defined in a block of statements delineated by the **generate** and **endgenerate** keywords. The subcircuit is instantiated multiple times using a generate-index variable. This variable is defined using the **genvar** keyword and it can have only positive integer values. It is not possible to use an index declared as a normal **integer** variable.

Table 6.3 Precedence of Verilog operators.

Operator type	Operator symbols	Precedence
Complement	! ~ −	Highest procedence
Arithmetic	* /	
	+ −	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~&	
	^ ~^	
	| ~|	
Logical	&&	
	||	
Conditional	?:	Lowest precedence

Example 6.25

Figure 6.46 shows how the **generate** construct can be used to specify an n-bit ripple-carry adder. The subcircuit is a full-adder defined structurally in terms of primitive gates as introduced in Figure 5.23. The **for** loop causes the full-adder block to be instantiated n times.

In this example, the **for** statement is used in the **generate** block to control the selection of the generated objects. The **generate** block can also contain **if-else** and **case** statements to determine which objects are generated.

The **generate** construct is a powerful mechanism that enhances the design capability of Verilog. Because it is a recent addition to the Verilog standard, it is not supported by some synthesis tools.

6.6.7 TASKS AND FUNCTIONS

In high-level programming languages it is possible to use subroutines and functions to avoid replicating specific routines that may be needed in several places of a given program. Verilog provides similar capabilities, known as tasks and functions. They can be used to modularize large designs and make the Verilog code easier to understand.

Verilog Task

A task is declared by the keyword **task** and it comprises a block of statements that ends with the keyword **endtask**. The task must be included in the module that calls it. It may have input and output ports. These are not the ports of the module that contains the task,

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 32;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   wire [n:0] C;

   genvar k;
   assign C[0] = carryin;
   assign carryout = C[n];
   generate
      for (k = 0; k < n; k = k+1)
      begin: fulladd_stage
         wire z1, z2, z3;  //wires within full-adder
         xor (S[k], X[k], Y[k], C[k]);
         and (z1, X[k], Y[k]);
         and (z2, X[k], C[k]);
         and (z3, Y[k], C[k]);
         or (C[k+1], z1, z2, z3);
      end
   endgenerate

endmodule
```

Figure 6.46 Using the **generate** loop to define an n-bit ripple-carry adder.

which are used to make external connections to the module. The task ports are used only to pass values between the module and the task.

Example 6.26 In Figure 6.33 we showed the Verilog code for a 16-to-1 multiplexer that instantiates five copies of a 4-to-1 multiplexer circuit given in a separate module named *mux4to1*. The same circuit can be specified using the task approach as shown in Figure 6.47. Observe the key differences. The task *mux4to1* is included in the module *mux16to1*. It is called from an **always** block by means of an appropriate **case** statement. The output of a task must be a variable, hence g is of **reg** type.

Verilog Function

A function is declared by the keyword **function** and it comprises a block of statements that ends with the keyword **endfunction**. The function must have at least one input and it returns a single value that is placed where the function is invoked.

```
module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output f;
  reg f;

  always @(W or S16)
    case (S16[3:2])
      0: mux4to1 (W[0:3], S16[1:0], f);
      1: mux4to1 (W[4:7], S16[1:0], f);
      2: mux4to1 (W[8:11], S16[1:0], f);
      3: mux4to1 (W[12:15], S16[1:0], f);
    endcase

  // Task that specifies a 4-to-1 multiplexer
  task mux4to1;
    input [0:3] X;
    input [1:0] S4;
    output g;
    reg g;

    case (S4)
      0: g = X[0];
      1: g = X[1];
      2: g = X[2];
      3: g = X[3];
    endcase
  endtask

endmodule
```

Figure 6.47 Use of a task in Verilog code.

Example 6.27

Figure 6.48 shows how the code in Figure 6.47 can be written to use a function. The Verilog compiler essentially inserts the body of the function at each place where it is called. Hence the clause

```
0: f = mux4to1 (W[0:3], S16[1:0]);
```

becomes

```
0: case (S16[1:0])
     0: f = W[0];
     1: f = W[1];
     2: f = W[2];
     3: f = W[3];
   endcase
```

```
module mux16to1 (W, S16, f);
  input [0:15] W;
  input [3:0] S16;
  output f;
  reg f;

  // Function that specifies a 4-to-1 multiplexer
  function mux4to1;
    input [0:3] X;
    input [1:0] S4;

    case (S4)
      0: mux4to1 = X[0];
      1: mux4to1 = X[1];
      2: mux4to1 = X[2];
      3: mux4to1 = X[3];
    endcase
  endfunction

  always @(W or S16)
    case (S16[3:2])
      0: f = mux4to1 (W[0:3], S16[1:0]);
      1: f = mux4to1 (W[4:7], S16[1:0]);
      2: f = mux4to1 (W[8:11], S16[1:0]);
      3: f = mux4to1 (W[12:15], S16[1:0]);
    endcase

endmodule
```

Figure 6.48 The code from Figure 6.47 using a function.

The function serves as a convenience that makes the mux16to1 module compact and easier to read.

A Verilog function can invoke another function but it cannot call a Verilog task. A task may call another task and it may invoke a function. In Figure 6.47 we defined the task after the **always** block that calls it. In contrast, in Figure 6.48 we defined the function before the **always** block that invokes it. Both possibilities are allowed in the Verilog standard for both tasks and functions. However, some tools require functions to be defined before the statements that invoke them.

6.7 CONCLUDING REMARKS

This chapter has introduced a number of circuit building blocks. Examples using these blocks to construct larger circuits will be presented in Chapters 7 and 10. To describe the building block circuits efficiently, several Verilog constructs have been introduced. In many cases a given circuit can be described in various ways, using different constructs. A circuit that can be described using an **if-else** statement can also be described using a **case** statement or perhaps a **for** loop. In general, there are no strict rules that dictate when one style should be preferred over another. With experience the user develops a sense for which types of statements work well in a particular design situation. Personal preference also influences how the code is written.

Verilog is not a programming language, and Verilog code should not be written as if it were a computer program. The statements discussed in this chapter can be used to create large, complex circuits. A good way to design such circuits is to construct them using well-defined modules, in the manner that we illustrated for the multiplexers, decoders, encoders, and so on. Additional examples using the Verilog statements introduced in this chapter are given in Chapters 7 and 8. In Chapter 10 we provide a number of examples of using Verilog code to describe larger digital systems. For more information on Verilog, the reader can consult more specialized books [5-11].

In the next chapter we introduce logic circuits that include the ability to store logic signal values in memory elements.

PROBLEMS

6.1 Show how the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 4, 5, 7)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

6.2 Show how the function $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$ can be implemented using a 3-to-8 binary decoder and an OR gate.

6.3 Consider the function $f = \overline{w}_1\overline{w}_3 + w_2\overline{w}_3 + \overline{w}_1 w_2$. Use the truth table to derive a circuit for f that uses a 2-to-1 multiplexer.

6.4 Repeat problem 6.3 for the function $f = \overline{w}_2\overline{w}_3 + w_1 w_2$.

6.5 For the function $f(w_1, w_2, w_3) = \sum m(0, 2, 3, 6)$, use Shannon's expansion to derive an implementation using a 2-to-1 multiplexer and any other necessary gates.

6.6 Repeat problem 6.5 for the function $f(w_1, w_2, w_3) = \sum m(0, 4, 6, 7)$.

6.7 Consider the function $f = \overline{w}_2 + \overline{w}_1\overline{w}_3 + w_1 w_3$. Show how repeated application of Shannon's expansion can be used to derive the minterms of f.

6.8 Repeat problem 6.7 for $f = w_2 + \overline{w}_1\overline{w}_3$.

6.9 Prove Shannon's expansion theorem presented in section 6.1.2.

6.10 Section 6.1.2 shows Shannon's expansion in sum-of-products form. Using the principle of duality, derive the equivalent expression in product-of-sums form.

6.11 Consider the function $f = \overline{w}_1\overline{w}_2 + \overline{w}_2\overline{w}_3 + w_1w_2w_3$. Give a circuit that implements f using the minimal number of two-input LUTs. Show the truth table implemented inside each LUT.

6.12 For the function in problem 6.11, the cost of the minimal sum-of-products expression is 14, which includes four gates and 10 inputs to the gates. Use Shannon's expansion to derive a multilevel circuit that has a lower cost and give the cost of your circuit.

6.13 Consider the function $f(w_1, w_2, w_3, w_4) = \sum m(0, 1, 3, 6, 8, 9, 14, 15)$. Derive an implementation using the minimum possible number of three-input LUTs.

6.14 Give two examples of logic functions with five inputs, $w_1, \ldots, w_5$, that can be realized using 2 four-input LUTs.

6.15 Assume that an FPGA exists in which the logic blocks are 4-LUTs. What is the minimum number of 4-LUTs needed to construct a 4-to-1 multiplexer with select inputs s_1 and s_0 and data inputs w_3, w_2, w_1, and w_0? Show two different circuits that can implement the desired multiplexer using the minimum number of 4-LUTs. Indicate the logic function implemented on the output of each LUT.

6.16 Actel Corporation manufactures an FPGA family called Act 1, which has the multiplexer-based logic block illustrated in Figure P6.1. Show how the function $f = w_2\overline{w}_3 + w_1w_3 + \overline{w}_2w_3$ can be implemented using only one Act 1 logic block.

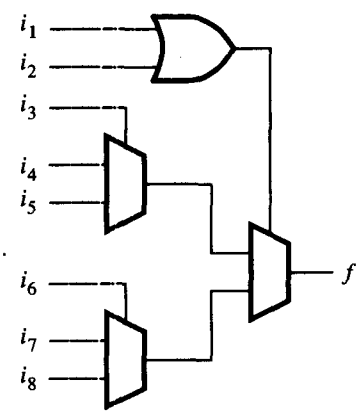


Figure P6.1 The Actel Act 1 logic block.

6.17 Show how the function $f = w_1\overline{w}_3 + \overline{w}_1w_3 + w_2\overline{w}_3 + w_1\overline{w}_2$ can be realized using Act 1 logic blocks. Note that there are no NOT gates in the chip; hence complements of signals have to be generated using the multiplexers in the logic block.

6.18 Consider the Verilog code in Figure P6.2. What type of circuit does the code represent? Comment on whether or not the style of code used is a good choice for the circuit it represents.

```
module problem6_18 (W, En, y0, y1, y2, y3);
  input [1:0] W;
  input En;
  output y0, y1, y2, y3;
  reg y0, y1, y2, y3;

  always @(W or En)
  begin
    y0 = 0;
    y1 = 0;
    y2 = 0;
    y3 = 0;
    if (En)
      if (W == 0)  y0 = 1;
      else if (W == 1)  y1 = 1;
      else if (W == 2)  y2 = 1;
      else  y3 = 1;
  end

endmodule
```

Figure P6.2 Code for problem 6.18.

6.19 Write Verilog code that represents the function in problem 6.2, using a **case** statement.

6.20 Write Verilog code for a 4-to-2 binary encoder.

6.21 Write Verilog code for an 8-to-3 binary encoder.

6.22 Figure P6.3 shows a modified version of the code for a 2-to-4 decoder in Figure 6.42. This code is almost correct but contains one error. What is the error?

6.23 Derive the circuit for an 8-to-3 priority encoder.

6.24 Using a **casex** statement, write Verilog code for an 8-to-3 priority encoder.

6.25 Repeat problem 6.24, using a **for** loop.

6.26 Create a Verilog module named *if2to4* that represents a 2-to-4 binary decoder using an **if-else** statement. Create a second module named *h3to8* that represents the 3-to-8 binary decoder in Figure 6.17 using two instances of the *if2to4* module.

```
module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output [0:3] Y;
  reg [0:3] Y;
  integer k;

  always @(W or En)
    for (k = 0; k <= 3; k = k+1)
      if (W == k)
        Y[k] = En;

endmodule
```

Figure P6.3 Code for problem 6.22.

6.27 Create a Verilog module named *h6to64* that represents a 6-to-64 binary decoder. Use the treelike structure in Figure 6.18, in which the 6-to-64 decoder is built using nine instances of the *h3to8* decoder created in problem 6.26.

6.28 Write Verilog code that represents the circuit in Figure 6.19. Use the *dec2to4* module in Figure 6.35 as a subcircuit in your code.

6.29 Derive minimal sum-of-products expressions for the outputs a, b, and c of the 7-segment display in Figure 6.25.

6.30 Derive minimal sum-of-products expressions for the outputs d, e, f, and g of the 7-segment display in Figure 6.25.

6.31 Figure 6.21 shows a block diagram of a ROM. A circuit that implements a small ROM, with four rows and four columns, is depicted in Figure P6.4. Each X in the figure represents a switch that determines whether the ROM produces a 1 or 0 when that location is read.
(a) Show how a switch (X) can be realized using a single NMOS transistor.
(b) Draw the complete 4×4 ROM circuit, using your switches from part (*a*). The ROM should be programmed to store the bits 0101 in row 0 (the top row), 1010 in row 1, 1100 in row 2, and 0011 in row 3 (the bottom row).
(c) Show how each (X) can be implemented as a programmable switch (as opposed to providing either a 1 or 0 permanently), using an EEPROM cell as shown in Figure 3.64. Briefly describe how the storage cell is used.

6.32 Show the complete circuit for a ROM using the storage cells designed in Part (*a*) of problem 6.31 that realizes the logic functions

$$d_3 = a_0 \oplus a_1$$

$$d_2 = \overline{a_0 \oplus a_1}$$

$$d_1 = a_0 a_1$$

$$d_0 = a_0 + a_1$$

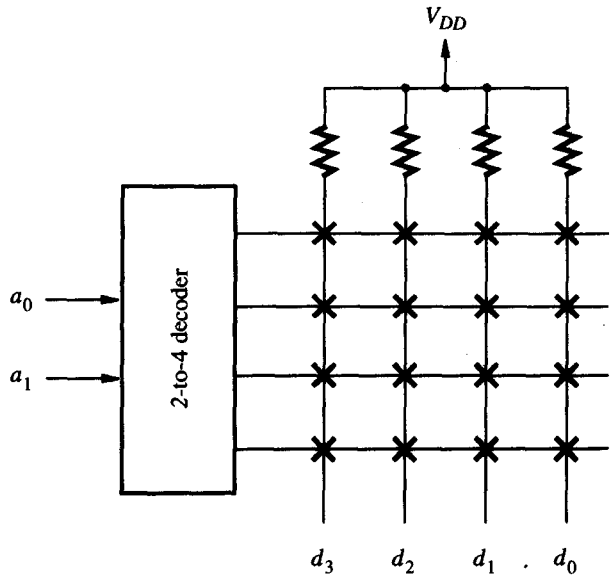


Figure P6.4 A 4 × 4 ROM circuit.

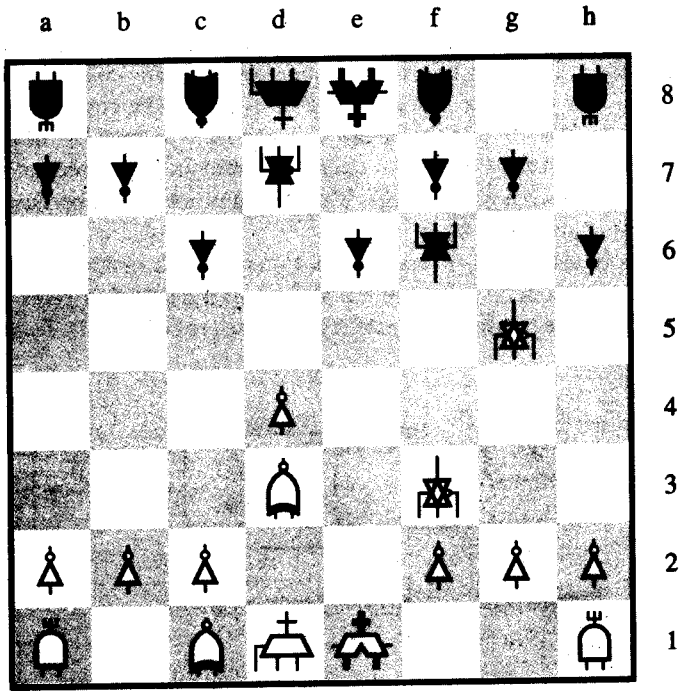
REFERENCES

1. C. E. Shannon, "Symbolic Analysis of Relay and Switching Circuits," *Transactions AIEE* 57 (1938), pp. 713–723.
2. Actel Corporation, "MX FPGA Data Sheet," *http://www.actel.com.*
3. QuickLogic Corporation, "pASIC 3 FPGA Data Sheet," *http://www.quicklogic.com.*
4. R. Landers, S. Mahant-Shetti, and C. Lemonds, "A Multiplexer-Based Architecture for High-Density, Low Power Gate Arrays," *IEEE Journal of Solid-State Circuits* 30, no. 4 (April 1995).
5. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
6. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice-Hall: Upper Saddle River, NJ, 1996).
7. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
8. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
9. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
10. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
11. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

chapter

7

FLIP-FLOPS, REGISTERS, COUNTERS, AND A SIMPLE PROCESSOR



7. Ng1–f3, h7–h6

In previous chapters we considered combinational circuits where the value of each output depends solely on the values of signals applied to the inputs. There exists another class of logic circuits in which the values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit. Such circuits include storage elements that store the values of logic signals. The contents of the storage elements are said to represent the *state* of the circuit. When the circuit's inputs change values, the new input values either leave the circuit in the same state or cause it to change into a new state. Over time the circuit changes through a sequence of states as a result of changes in the inputs. Circuits that behave in this way are referred to as *sequential circuits.*

In this chapter we will introduce circuits that can be used as storage elements. But first, we will motivate the need for such circuits by means of a simple example. Suppose that we wish to control an alarm system, as shown in Figure 7.1. The alarm mechanism responds to the control input $On/\overline{Off}$. It is turned on when $On/\overline{Off} = 1$, and it is off when $On/\overline{Off} = 0$. The desired operation is that the alarm turns on when the sensor generates a positive voltage signal, *Set*, in response to some undesirable event. Once the alarm is triggered, it must remain active even if the sensor output goes back to zero. The alarm is turned off manually by means of a *Reset* input. The circuit requires a memory element to remember that the alarm has to be active until the *Reset* signal arrives.

Figure 7.2 gives a rudimentary memory element, consisting of a loop that has two inverters. If we assume that $A = 0$, then $B = 1$. The circuit will maintain these values indefinitely. We say that the circuit is in the *state* defined by these values. If we assume that $A = 1$, then $B = 0$, and the circuit will remain in this second state indefinitely. Thus the circuit has two possible states. This circuit is not useful, because it lacks some practical means for changing its state.

A more useful circuit is shown in Figure 7.3. It includes a mechanism for changing the state of the circuit in Figure 7.2, using two transmission gates of the type discussed in section 3.9. One transmission gate, $TG1$, is used to connect the *Data* input terminal to point

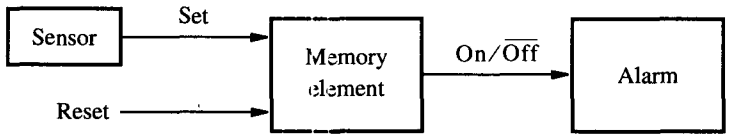


Figure 7.1 Control of an alarm system.

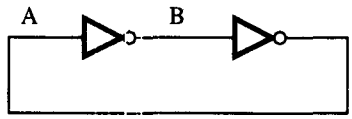


Figure 7.2 A simple memory element.

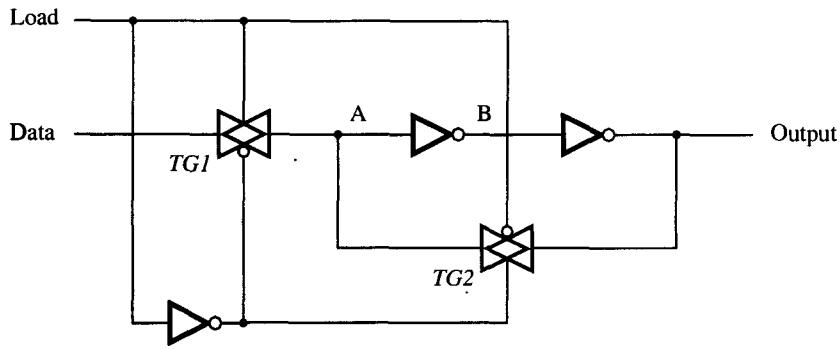


Figure 7.3 A controlled memory element.

A in the circuit. The second; *TG*2, is used as a switch in the *feedback loop* that maintains the state of the circuit. The transmission gates are controlled by the *Load* signal. If *Load* = 1, then *TG*1 is on and the point *A* will have the same value as the *Data* input. Since the value presently stored at *Output* may not be the same value as *Data*, the feedback loop is broken by having *TG*2 turned off when *Load* = 1. When *Load* changes to zero, then *TG*1 turns off and *TG*2 turns on. The feedback path is closed and the memory element will retain its state as long as *Load* = 0. This memory element cannot be applied directly to the system in Figure 7.1, but it is useful for many other applications, as we will see later.

7.1 BASIC LATCH

Instead of using the transmission gates, we can construct a similar circuit using ordinary logic gates. Figure 7.4 presents a memory element built with NOR gates. Its inputs, *Set* and *Reset*, provide the means for changing the state, Q, of the circuit. A more usual way of drawing this circuit is given in Figure 7.5*a*, where the two NOR gates are said to be connected in cross-coupled style. The circuit is referred to as a *basic latch*. Its behavior is described by the truth table in Figure 7.5*b*. When both inputs, *R* and *S*, are equal to 0 the latch maintains its existing state. This state may be either $Q_a = 0$ and $Q_b = 1$, or $Q_a = 1$ and $Q_b = 0$, which is indicated in the truth table by stating that the Q_a and Q_b outputs have values 0/1 and 1/0, respectively. Observe that Q_a and Q_b are complements of each other in this case. When $R = 0$ and $S = 1$, the latch is *set* into a state where $Q_a = 1$ and $Q_b = 0$. When $R = 1$ and $S = 0$, the latch is *reset* into a state where $Q_a = 0$ and $Q_b = 1$. The fourth possibility is to have $R = S = 1$. In this case both Q_a and Q_b will be 0.

Figure 7.5*c* gives a timing diagram for the latch, assuming that the propagation delay through the NOR gates is negligible. Of course, in a real circuit the changes in the waveforms would be delayed according to the propagation delays of the gates. We assume that initially $Q_a = 0$ and $Q_b = 1$. The state of the latch remains unchanged until time t_2,

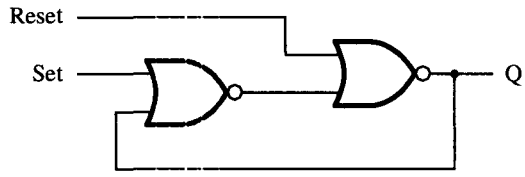
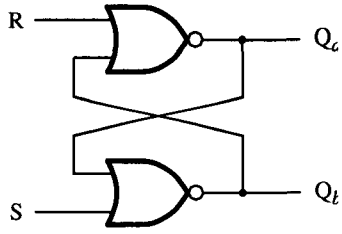


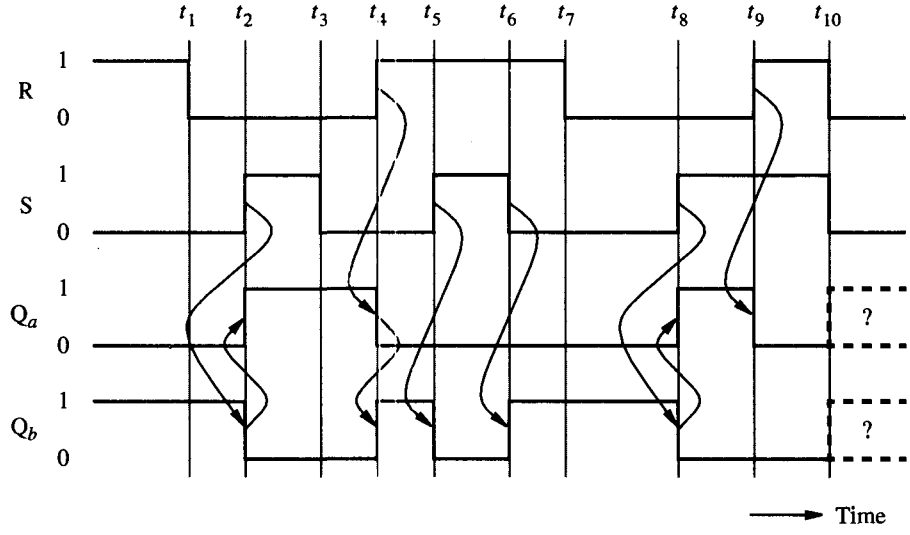
Figure 7.4 A memory element with NOR gates.



(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(No change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Truth table



(c) Timing diagram

Figure 7.5 A basic latch built with NOR gates.

when S becomes equal to 1, causing Q_b to change to 0, which in turn causes Q_a to change to 1. The causality relationship is indicated by the arrows in the diagram. When S goes to 0 at t_3, there is no change in the state because both S and R are then equal to 0. At t_4 we have $R = 1$, which causes Q_a to go to 0, which in turn causes Q_b to go to 1. At t_5 both S and R are equal to 1, which forces both Q_a and Q_b to be equal to 0. As soon as S returns to 0, at t_6, Q_b becomes equal to 1 again. At t_8 we have $S = 1$ and $R = 0$, which causes $Q_b = 0$ and $Q_a = 1$. An interesting situation occurs at t_{10}. From t_9 to t_{10} we have $Q_a = Q_b = 0$ because $R = S = 1$. Now if both R and S change to 0 at t_{10}, both Q_a and Q_b will go to 1. But having both Q_a and Q_b equal to 1 will immediately force $Q_a = Q_b = 0$. There will be an oscillation between $Q_a = Q_b = 0$ and $Q_a = Q_b = 1$. If the delays through the two NOR gates are exactly the same, the oscillation will continue indefinitely. In a real circuit there will invariably be some difference in the delays through these gates, and the latch will eventually settle into one of its two stable states, but we don't know which state it will be. This uncertainty is indicated in the waveforms by dashed lines.

The oscillations discussed above illustrate that even though the basic latch is a simple circuit, careful analysis has to be done to fully appreciate its behavior. In general, any circuit that contains one or more feedback paths, such that the state of the circuit depends on the propagation delays through logic gates, has to be designed carefully. We discuss timing issues in detail in Chapter 9.

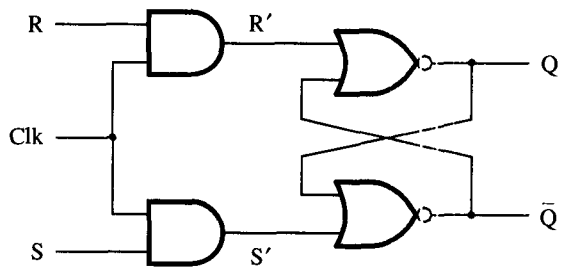
The latch in Figure 7.5*a* can perform the functions needed for the memory element in Figure 7.1, by connecting the *Set* signal to the S input and *Reset* to the R input. The Q_a output provides the desired $On/\overline{Off}$ signal. To initialize the operation of the alarm system, the latch is reset. Thus the alarm is off. When the sensor generates the logic value 1, the latch is set and Q_a becomes equal to 1. This turns on the alarm mechanism. If the sensor output returns to 0, the latch retains its state where $Q_a = 1$; hence the alarm remains turned on. The only way to turn off the alarm is by resetting the latch, which is accomplished by making the *Reset* input equal to 1.

7.2 GATED SR LATCH

In section 7.1 we saw that the basic SR latch can serve as a useful memory element. It remembers its state when both the S and R inputs are 0. It changes its state in response to changes in the signals on these inputs. The state changes occur at the time when the changes in the signals occur. If we cannot control the time of such changes, then we don't know when the latch may change its state.

In the alarm system of Figure 7.1, it may be desirable to be able to enable or disable the entire system by means of a control input, *Enable*. Thus when enabled, the system would function as described above. In the disabled mode, changing the *Set* input from 0 to 1 would not cause the alarm to turn on. The latch in Figure 7.5*a* cannot provide the desired operation. But the latch circuit can be modified to respond to the input signals S and R only when *Enable* = 1. Otherwise, it would maintain its state.

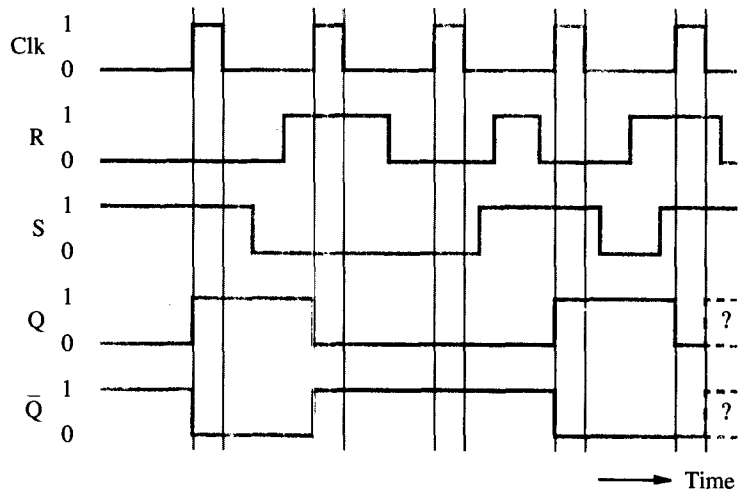
The modified circuit is depicted in Figure 7.6*a*. It includes two AND gates that provide the desired control. When the control signal *Clk* is equal to 0, the S' and R' inputs to the



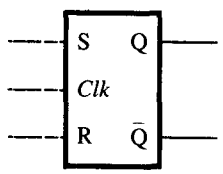
(a) Circuit

Clk	S	R	Q(t + 1)
0	x	x	Q(t) (No change)
1	0	0	Q(t) (No change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Truth table



(c) Timing diagram



(d) Graphical symbol

Figure 7.6 Gated SR latch.

latch will be 0, regardless of the values of signals S and R. Hence the latch will maintain its existing state as long as $Clk = 0$. When Clk changes to 1, the S' and R' signals will be the same as the S and R signals, respectively. Therefore, in this mode the latch will behave as we described in section 7.1. Note that we have used the name Clk for the control signal that allows the latch to be set or reset, rather than call it the *Enable* signal. The reason is that

such circuits are often used in digital systems where it is desirable to allow the changes in the states of memory elements to occur only at well-defined time intervals, as if they were controlled by a clock. The control signal that defines these time intervals is usually called the *clock* signal. The name *Clk* is meant to reflect this nature of the signal.

Circuits of this type, which use a control signal, are called *gated latches*. Because our circuit exhibits set and reset capability, it is called a *gated SR latch*. Figure 7.6*b* describes its behavior. It defines the state of the Q output at time $t + 1$, namely, $Q(t + 1)$, as a function of the inputs S, R, and *Clk*. When $Clk = 0$, the latch will remain in the state it is in at time t, that is, $Q(t)$, regardless of the values of inputs S and R. This is indicated by specifying $S = x$ and $R = x$, where x means that the signal value can be either 0 or 1. (Recall that we already used this notation in Chapter 4.) When $Clk = 1$, the circuit behaves as the basic latch in Figure 7.5. It is set by $S = 1$ and reset by $R = 1$. The last row of the truth table, where $S = R = 1$, shows that the state $Q(t + 1)$ is undefined because we don't know whether it will be 0 or 1. This corresponds to the situation described in section 7.1 in conjunction with the timing diagram in Figure 7.5 at time t_{10}. At this time both S and R inputs go from 1 to 0, which causes the oscillatory behavior that we discussed. If $S = R = 1$, this situation will occur as soon as *Clk* goes from 1 to 0. To ensure a meaningful operation of the gated SR latch, it is essential to avoid the possibility of having both the S and R inputs equal to 1 when *Clk* changes from 1 to 0.

A timing diagram for the gated SR latch is given in Figure 7.6*c*. It shows *Clk* as a periodic signal that is equal to 1 at regular time intervals to suggest that this is how the clock signal usually appears in a real system. The diagram presents the effect of several combinations of signal values. Observe that we have labeled one output as Q and the other as its complement $\overline{Q}$, rather than Q_a and Q_b as in Figure 7.5. Since the undefined mode, where $S = R = 1$, must be avoided in practice, the normal operation of the latch will have the outputs as complements of each other. Moreover, we will often say that the latch is *set* when $Q = 1$, and it is *reset* when $Q = 0$. A graphical symbol for the gated SR latch is given in Figure 7.6*d*.

7.2.1 GATED SR LATCH WITH NAND GATES

So far we have implemented the basic latch with cross-coupled NOR gates. We can also construct the latch with NAND gates. Using this approach, we can implement the gated SR latch as depicted in Figure 7.7. The behavior of this circuit is described by the truth table

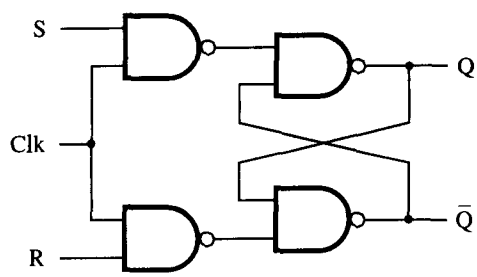


Figure 7.7 Gated SR latch with NAND gates.

in Figure 7.6*b*. Note that in this circuit, the clock is gated by NAND gates, rather than by AND gates. Note also that the *S* and *R* inputs are reversed in comparison with the circuit in Figure 7.6*a*. The circuit with NAND gates requires fewer transistors than the circuit with AND gates. We will use the circuit in Figure 7.7, in preference to the circuit in Figure 7.6*a*.

7.3 GATED D LATCH

In section 7.2 we presented the gated SR latch and showed how it can be used as the memory element in the alarm system of Figure 7.1. This latch is useful for many other applications. In this section we describe another gated latch that is even more useful in practice. It has a single data input, called *D*, and it stores the value on this input, under the control of a clock signal. It is called a *gated D latch*.

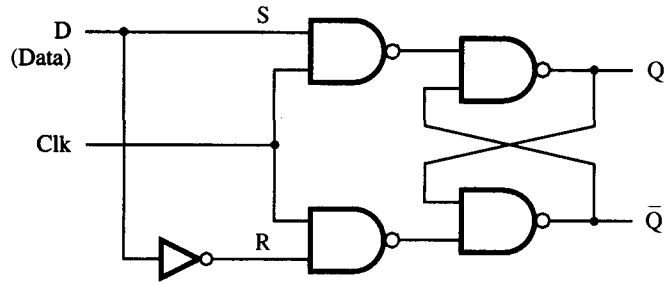
To motivate the need for a gated D latch, consider the adder/subtractor unit discussed in Chapter 5 (Figure 5.13). When we described how that circuit is used to add numbers, we did not discuss what is likely to happen with the sum bits that are produced by the adder. Adder/subtractor units are often used as part of a computer. The result of an addition or subtraction operation is often used as an operand in a subsequent operation. Therefore, it is necessary to be able to remember the values of the sum bits generated by the adder until they are needed again. We might think of using the basic latches to remember these bits, one bit per latch. In this context, instead of saying that a latch remembers the value of a bit, it is more illuminating to say that the latch *stores* the value of the bit or simply "stores the bit." We should think of the latch as a storage element.

But can we obtain the desired operation using the basic latches? We can certainly reset all latches before the addition operation begins. Then we would expect that by connecting a sum bit to the *S* input of a latch, the latch would be set to 1 if the sum bit has the value 1; otherwise, the latch would remain in the 0 state. This would work fine if all sum bits are 0 at the start of the addition operation and, after some propagation delay through the adder, some of these bits become equal to 1 to give the desired sum. Unfortunately, the propagation delays that exist in the adder circuit cause a big problem in this arrangement. Suppose that we use a ripple-carry adder. When the *X* and *Y* inputs are applied to the adder, the sum outputs may alternate between 0 and 1 a number of times as the carries ripple through the circuit. This situation was illustrated in the timing diagram in Figure 5.21. The problem is that if we connect a sum bit to the *S* input of a latch, then if the sum bit is temporarily a 1 and then settles to 0 in the final result, the latch will remain set to 1 erroneously.

The problem caused by the alternating values of the sum bits in the adder could be solved by using the gated SR latches, instead of the basic latches. Then we could arrange that the clock signal is 0 during the time needed by the adder to produce a correct sum. After allowing for the maximum propagation delay in the adder circuit, the clock should go to 1 to store the values of the sum bits in the gated latches. As soon as the values have been stored, the clock can return to 0, which ensures that the stored values will be retained until the next time the clock goes to 1. To achieve the desired operation, we would also have to reset all latches to 0 prior to loading the sum-bit values into these latches. This is

an awkward way of dealing with the problem, and it is preferable to use the gated D latches instead.

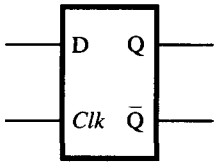
Figure 7.8*a* shows the circuit for a gated D latch. It is based on the gated SR latch, but instead of using the S and R inputs separately, it has just one data input, D. For convenience we have labeled the points in the circuit that are equivalent to the S and R inputs. If $D = 1$, then $S = 1$ and $R = 0$, which forces the latch into the state Q = 1. If $D = 0$, then $S = 0$ and $R = 1$, which causes Q = 0. Of course, the changes in state occur only when $Clk = 1$.



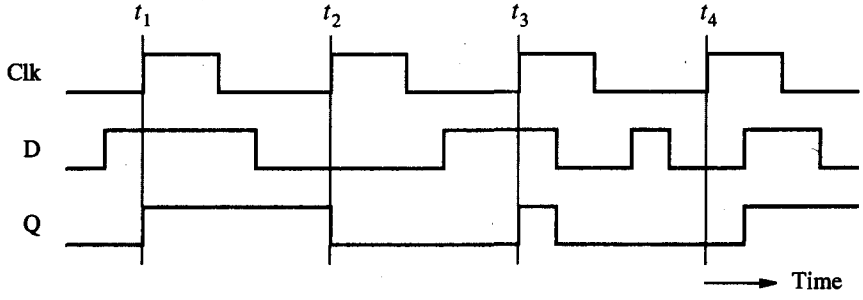
(a) Circuit

Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.8 Gated D latch.

It is important to observe that in this circuit it is impossible to have the troublesome situation where $S = R = 1$. In the gated D latch, the output Q merely tracks the value of the input D while $Clk = 1$. As soon as Clk goes to 0, the state of the latch is frozen until the next time the clock signal goes to 1. Therefore, the gated D latch stores the value of the D input seen at the time the clock changes from 1 to 0. Figure 7.8 also gives the truth table, the graphical symbol, and the timing diagram for the gated D latch.

The timing diagram illustrates what happens if the D signal changes while $Clk = 1$. During the third clock pulse, starting at t_3, the output Q changes to 1 because $D = 1$. But midway through the pulse D goes to 0, which causes Q to go to 0. This value of Q is stored when Clk changes to 0. Now no further change in the state of the latch occurs until the next clock pulse, at t_4. The key point to observe is that as long as the clock has the value 1, the Q output follows the D input. But when the clock has the value 0, the Q output cannot change. In Chapter 3 we saw that the logic values are implemented as low and high voltage levels. Since the output of the gated D latch is controlled by the level of the clock input, the latch is said to be *level sensitive*. The circuits in Figures 7.6 through 7.8 are level sensitive. We will show in section 7.4 that it is possible to design storage elements for which the output changes only at the point in time when the clock changes from one value to the other. Such circuits are said to be *edge triggered*.

At this point we should reconsider the circuit in Figure 7.3. Careful examination of that circuit shows that it behaves in exactly the same way as the circuit in Figure 7.8*a*. The *Data* and *Load* inputs correspond to the D and Clk inputs, respectively. The *Output*, which has the same signal value as point A, corresponds to the Q output. Point B corresponds to $\overline{Q}$. Therefore, the circuit in Figure 7.3 is also a gated D latch. An advantage of this circuit is that it can be implemented using fewer transistors than the circuit in Figure 7.8*a*.

7.3.1 EFFECTS OF PROPAGATION DELAYS

In the previous discussion we ignored the effects of propagation delays. In practical circuits it is essential to take these delays into account. Consider the gated D latch in Figure 7.8*a*. It stores the value of the D input that is present at the time the clock signal changes from 1 to 0. It operates properly if the D signal is stable (that is, not changing) at the time Clk goes from 1 to 0. But it may lead to unpredictable results if the D signal also changes at this time. Therefore, the designer of a logic circuit that generates the D signal must ensure that this signal is stable when the critical change in the clock signal takes place.

Figure 7.9 illustrates the critical timing region. The minimum time that the D signal must be stable prior to the negative edge of the Clk signal is called the *setup time*, t_{su}, of the latch. The minimum time that the D signal must remain stable after the negative edge of the Clk signal is called the *hold time*, t_h, of the latch. The values of t_{su} and t_h depend on the technology used. Manufacturers of integrated circuit chips provide this information on the data sheets that describe their chips. Typical values for CMOS technology are $t_{su} = 3$ ns and $t_h = 2$ ns. We will give examples of how setup and hold times affect the speed of operation of circuits in section 7.13. The behavior of storage elements when setup or hold times are violated is discussed in section 10.3.3.

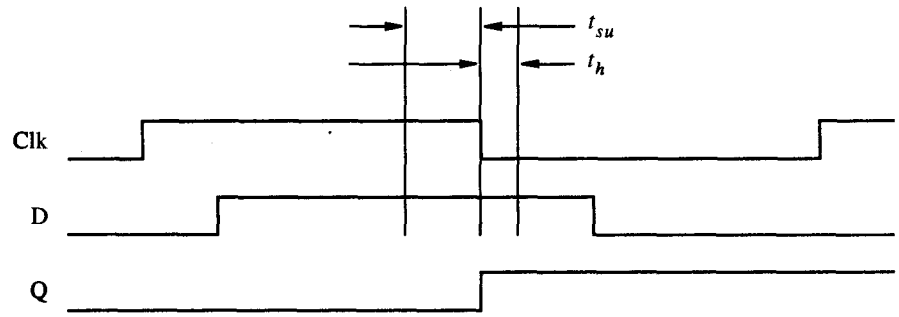


Figure 7.9 Setup and hold times.

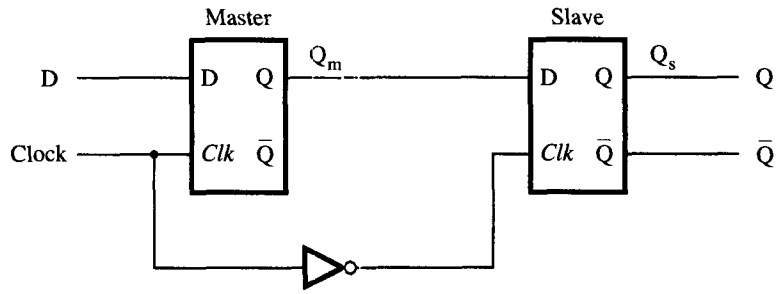
7.4 MASTER-SLAVE AND EDGE-TRIGGERED D FLIP-FLOPS

In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the period when the clock signal is active (equal to 1 in our examples). As we will see in sections 7.8 and 7.9, there is also a need for storage elements that can change their states no more than once during one clock cycle. We will discuss two types of circuits that exhibit such behavior.

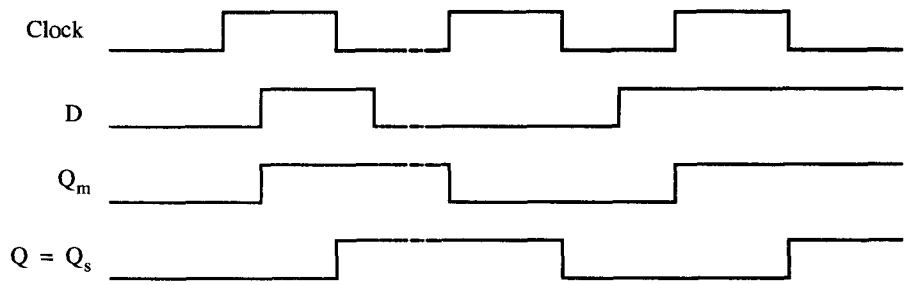
7.4.1 MASTER-SLAVE D FLIP-FLOP

Consider the circuit given in Figure 7.10*a*, which consists of two gated D latches. The first, called *master*, changes its state while *Clock* = 1. The second, called *slave*, changes its state while *Clock* = 0. The operation of the circuit is such that when the clock is high, the master tracks the value of the *D* input signal and the slave does not change. Thus the value of Q_m follows any changes in *D*, and the value of Q_s remains constant. When the clock signal changes to 0, the master stage stops following the changes in the *D* input. At the same time, the slave stage responds to the value of the signal Q_m and changes state accordingly. Since Q_m does not change while *Clock* = 0, the slave stage can undergo at most one change of state during a clock cycle. From the external observer's point of view, namely, the circuit connected to the output of the slave stage, the master-slave circuit changes its state at the negative-going edge of the clock. The *negative edge* is the edge where the clock signal changes from 1 to 0. Regardless of the number of changes in the *D* input to the master stage during one clock cycle, the observer of the Q_s signal will see only the change that corresponds to the *D* input at the negative edge of the clock.

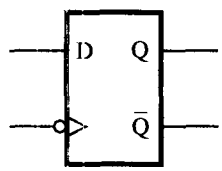
The circuit in Figure 7.10 is called a *master-slave D flip-flop*. The term *flip-flop* denotes a storage element that changes its output state at the edge of a controlling clock signal. The timing diagram for this flip-flop is shown in Figure 7.10*b*. A graphical symbol is given in Figure 7.10*c*. In the symbol we use the > mark to denote that the flip-flop responds to the "active edge" of the clock. We place a bubble on the clock input to indicate that the active edge for this particular circuit is the negative edge.



(a) Circuit



(b) Timing diagram

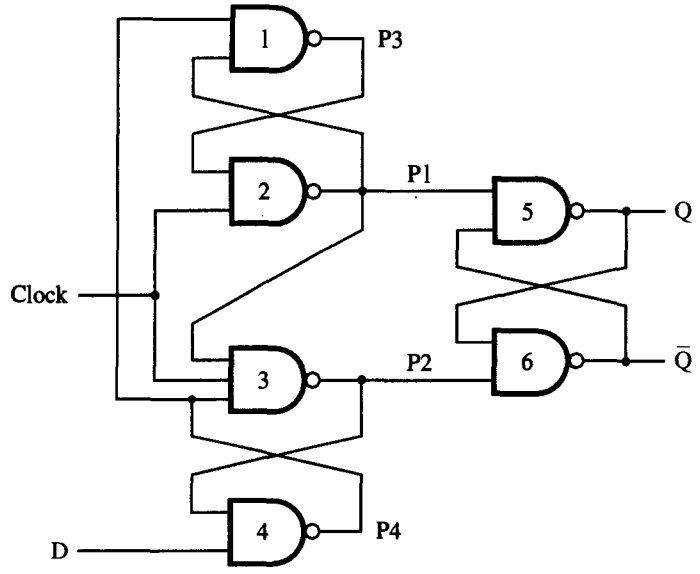


(c) Graphical symbol

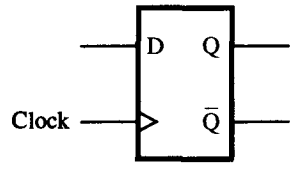
Figure 7.10 Master-slave D flip-flop.

7.4.2 EDGE-TRIGGERED D FLIP-FLOP

The output of the master-slave D flip-flop in Figure 7.10*a* responds on the negative edge of the clock signal. The circuit can be changed to respond to the positive clock edge by connecting the slave stage directly to the clock and the master stage to the complement of the clock. A different circuit that accomplishes the same task is presented in Figure 7.11*a*.



(a) Circuit



(b) Graphical symbol

Figure 7.11 A positive-edge-triggered D flip-flop.

It requires only six NAND gates and, hence, fewer transistors. The operation of the circuit is as follows. When *Clock* = 0, the outputs of gates 2 and 3 are high. Thus $P1 = P2 = 1$, which maintains the output latch, comprising gates 5 and 6, in its present state. At the same time, the signal $P3$ is equal to D, and $P4$ is equal to its complement $\overline{D}$. When *Clock* changes to 1, the following changes take place. The values of $P3$ and $P4$ are transmitted through gates 2 and 3 to cause $P1 = \overline{D}$ and $P2 = D$, which sets $Q = D$ and $\overline{Q} = \overline{D}$. To operate reliably, $P3$ and $P4$ must be stable when *Clock* changes from 0 to 1. Hence the setup time of the flip-flop is equal to the delay from the D input through gates 4 and 1 to $P3$. The hold time is given by the delay through gate 3 because once $P2$ is stable, the changes in D no longer matter.

For proper operation it is necessary to show that, after *Clock* changes to 1, any further changes in D will not affect the output latch as long as *Clock* = 1. We have to consider two cases. Suppose first that $D = 0$ at the positive edge of the clock. Then $P2 = 0$, which will

keep the output of gate 4 equal to 1 as long as *Clock* = 1, regardless of the value of the *D* input. The second case is if *D* = 1 at the positive edge of the clock. Then *P*1 = 0, which forces the outputs of gates 1 and 3 to be equal to 1, regardless of the *D* input. Therefore, the flip-flop ignores changes in the *D* input while *Clock* = 1.

Figure 7.11*b* gives a graphical symbol for this flip-flop. The clock input indicates that the positive edge of the clock is the active edge. A similar circuit, constructed with NOR gates, can be used as a negative-edge-triggered flip-flop.

Level-Sensitive versus Edge-Triggered Storage Elements

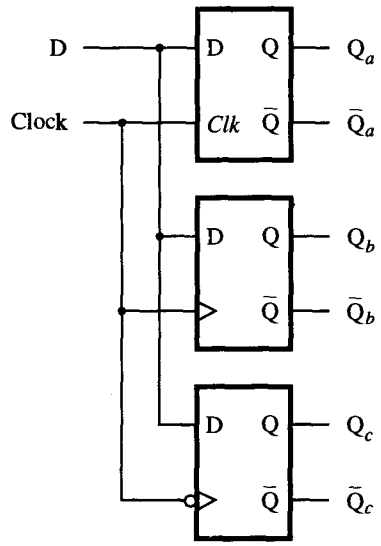
Figure 7.12 shows three different types of storage elements that are driven by the same data and clock inputs. The first element is a gated D latch, which is level sensitive. The second one is a positive-edge-triggered D flip-flop, and the third one is a negative-edge-triggered D flip-flop. To accentuate the differences between these storage elements, the *D* input changes its values more than once during each half of the clock cycle. Observe that the gated D latch follows the *D* input as long as the clock is high. The positive-edge-triggered flip-flop responds only to the value of *D* when the clock changes from 0 to 1. The negative-edge-triggered flip-flop responds only to the value of *D* when the clock changes from 1 to 0.

7.4.3 D FLIP-FLOPS WITH CLEAR AND PRESET

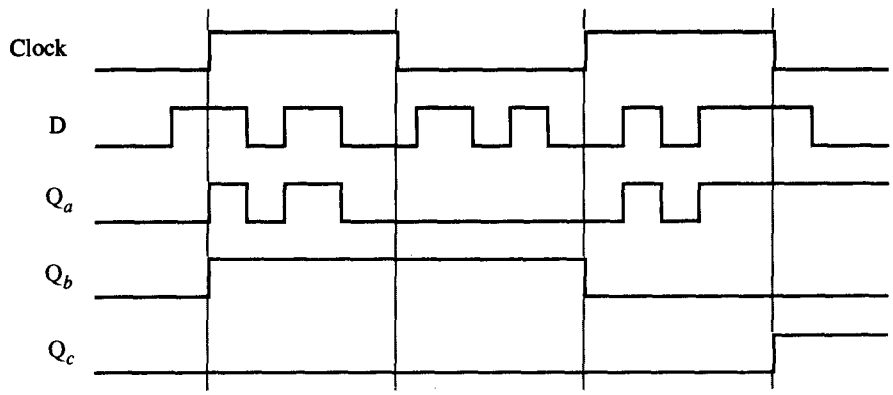
Flip-flops are often used for implementation of circuits that can have many possible states, where the response of the circuit depends not only on the present values of the circuit's inputs but also on the particular state that the circuit is in at that time. We will discuss a general form of such circuits in Chapter 8. A simple example is a counter circuit that counts the number of occurrences of some event, perhaps passage of time. We will discuss counters in detail in section 7.9. A counter comprises a number of flip-flops, whose outputs are interpreted as a number. The counter circuit has to be able to increment or decrement the number. It is also important to be able to force the counter into a known initial state (count). Obviously, it must be possible to clear the count to zero, which means that all flip-flops must have Q = 0. It is equally useful to be able to preset each flip-flop to Q = 1, to insert some specific count as the initial value in the counter. These features can be incorporated into the circuits of Figures 7.10 and 7.11 as follows.

Figure 7.13*a* shows an implementation of the circuit in Figure 7.10*a* using NAND gates. The master stage is just the gated D latch of Figure 7.8*a*. Instead of using another latch of the same type for the slave stage, we can use the slightly simpler gated SR latch of Figure 7.7. This eliminates one NOT gate from the circuit.

A simple way of providing the clear and preset capability is to add an extra input to each NAND gate in the cross-coupled latches, as indicated in blue. Placing a 0 on the *Clear* input will force the flip-flop into the state Q = 0. If *Clear* = 1, then this input will have no effect on the NAND gates. Similarly, *Preset* = 0 forces the flip-flop into the state Q = 1, while *Preset* = 1 has no effect. To denote that the *Clear* and *Preset* inputs are active when their value is 0, we placed an overbar on the names in the figure. We should note that the circuit that uses this flip-flop should not try to force both *Clear* and *Preset* to 0 at the same time. A graphical symbol for this flip-flop is shown in Figure 7.13*b*.



(a) Circuit

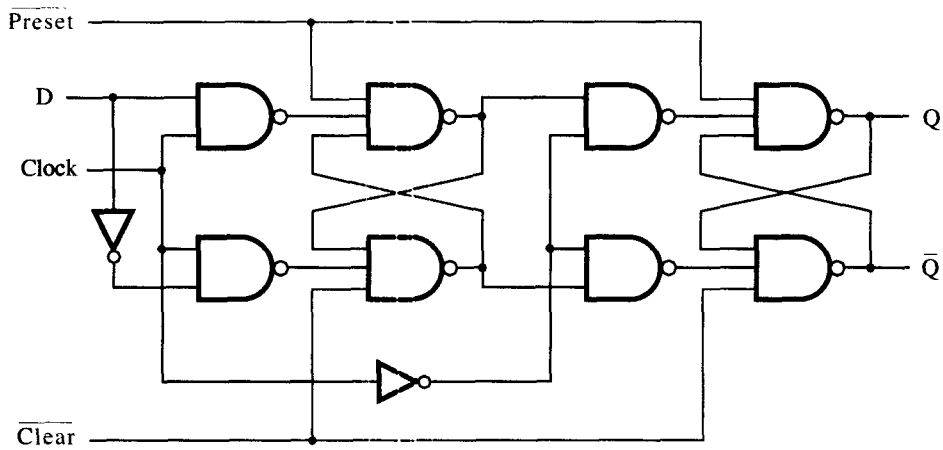


(b) Timing diagram

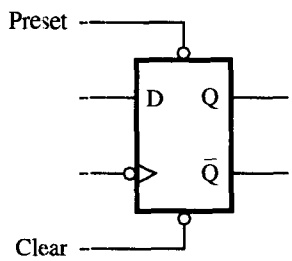
Figure 7.12 Comparison of level-sensitive and edge-triggered D storage elements.

A similar modification can be done on the edge-triggered flip-flop of Figure 7.11*a*, as indicated in Figure 7.14*a*. Again, both *Clear* and *Preset* inputs are active low. They do not disturb the flip-flop when they are equal to 1.

In the circuits in Figures 7.13*a* and 7.14*a*, the effect of a low signal on either the *Clear* or *Preset* input is immediate. For example, if *Clear* = 0 then the flip-flop goes into the state Q = 0 immediately, regardless of the value of the clock signal. In such a circuit, where the *Clear* signal is used to clear a flip-flop without regard to the clock signal, we say that the



(a) Circuit



(b) Graphical symbol

Figure 7.13 Master-slave D flip-flop with *Clear* and *Preset*.

flip-flop has an *asynchronous clear*. In practice, it is often preferable to clear the flip-flops on the active edge of the clock. Such *synchronous clear* can be accomplished as shown in Figure 7.15. The flip-flop operates normally when the *Clear* input is equal to 1. But if *Clear* goes to 0, then on the next positive edge of the clock the flip-flop will be cleared to 0. We will examine the clearing of flip-flops in more detail in section 7.10.

7.5 T FLIP-FLOP

The D flip-flop is a versatile storage element that can be used for many purposes. By including some simple logic circuitry to drive its input, the D flip-flop may appear to be a different type of storage element. An interesting modification is presented in Figure 7.16*a*. This circuit uses a positive-edge-triggered D flip-flop. The *feedback* connections make the input signal D equal to either the value of Q or $\overline{Q}$ under the control of the signal that is labeled T. On each positive edge of the clock, the flip-flop may change its state $Q(t)$. If

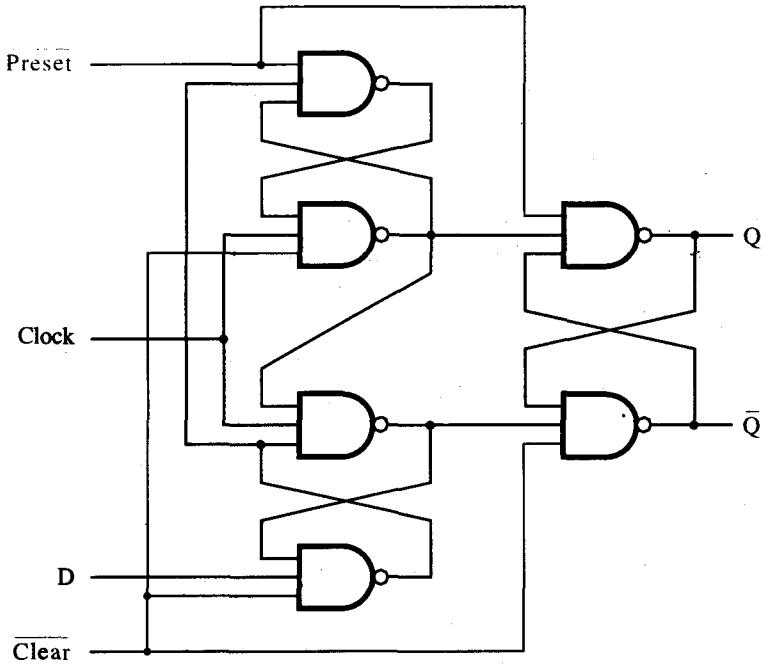


Figure 7.14 Positive-edge-triggered D flip-flop with *Clear* and *Preset*.

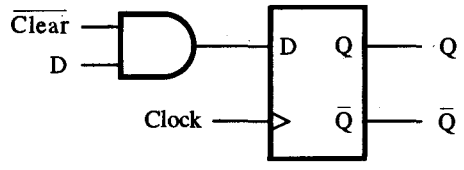
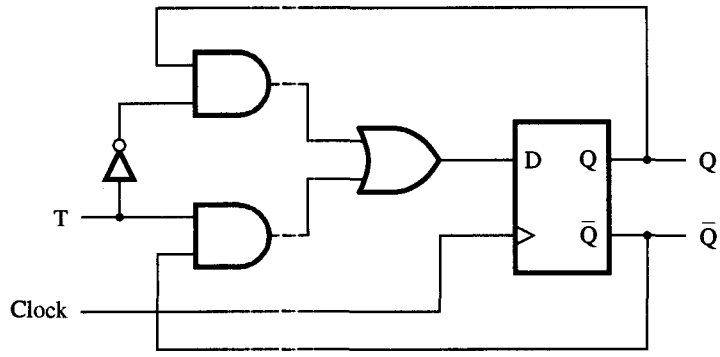


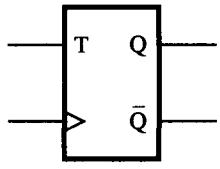
Figure 7.15 Synchronous reset for a D flip-flop.



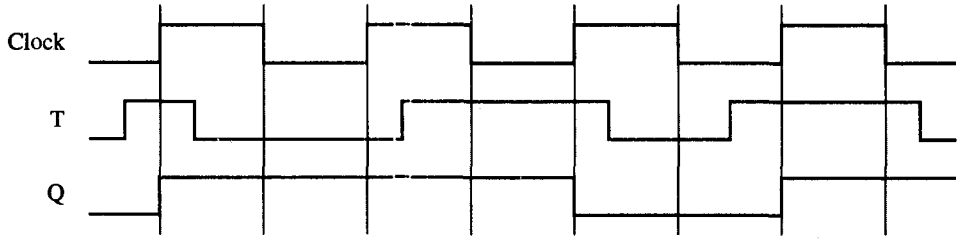
(a) Circuit

T	Q(t + 1)
0	Q(t)
1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol



(d) Timing diagram

Figure 7.16 T flip-flop.

$T = 0$, then $D = Q$ and the state will remain the same, that is, $Q(t + 1) = Q(t)$. But if $T = 1$, then $D = \overline{Q}$ and the new state will be $Q(t + 1) = \overline{Q}(t)$. Therefore, the overall operation of the circuit is that it retains its present state if $T = 0$, and it reverses its present state if $T = 1$.

The operation of the circuit is specified in the form of a truth table in Figure 7.16*b*. Any circuit that implements this truth table is called a *T flip-flop*. The name T flip-flop derives from the behavior of the circuit, which "toggles" its state when $T = 1$. The toggle feature makes the T flip-flop a useful element for building counter circuits, as we will see in section 7.9.

7.5.1 CONFIGURABLE FLIP-FLOPS

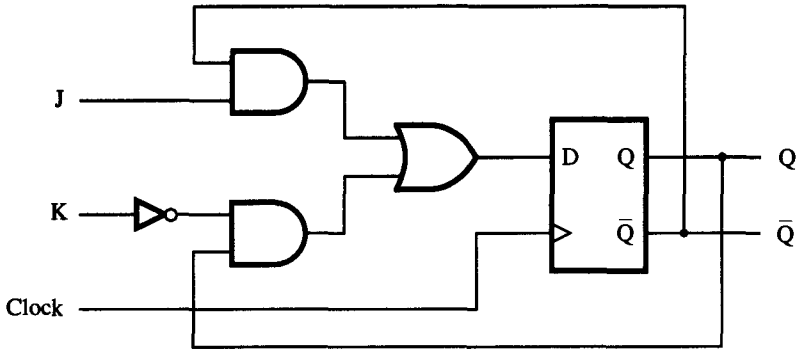
For some circuits one type of flip-flop may lead to a more efficient implementation than a different type of flip-flop. In general purpose chips like PLDs, the flip-flops that are provided are sometimes *configurable*, which means that a flip-flop circuit can be configured to be either D, T, or some other type. For example, in some PLDs the flip-flops can be configured as either D or T types (see problems 7.6 and 7.8).

7.6 JK FLIP-FLOP

Another interesting circuit can be derived from Figure 7.16*a*. Instead of using a single control input, T, we can use two inputs, J and K, as indicated in Figure 7.17*a*. For this circuit the input D is defined as

$$D = J\overline{Q} + \overline{K}Q$$

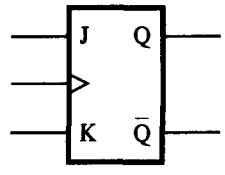
A corresponding truth table is given in Figure 7.17*b*. The circuit is called a *JK flip-flop*. It combines the behaviors of SR and T flip-flops in a useful way. It behaves as the SR flip-flop,



(a) Circuit

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\overline{Q}(t)$

(b) Truth table



(c) Graphical symbol

Figure 7.17 JK flip-flop.

where $J = S$ and $K = R$, for all input values except $J = K = 1$. For the latter case, which has to be avoided in the SR flip-flop, the JK flip-flop toggles its state like the T flip-flop.

The JK flip-flop is a versatile circuit. It can be used for straight storage purposes, just like the D and SR flip-flops. But it can also serve as a T flip-flop by connecting the J and K inputs together.

7.7 SUMMARY OF TERMINOLOGY

We have used the terminology that is quite common. But the reader should be aware that different interpretations of the terms *latch* and *flip-flop* can be found in the literature. Our terminology can be summarized as follows:

Basic latch is a feedback connection of two NOR gates or two NAND gates, which can store one bit of information. It can be set to 1 using the S input and reset to 0 using the R input.

Gated latch is a basic latch that includes input gating and a control input signal. The latch retains its existing state when the control input is equal to 0. Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock. We considered two types of gated latches:

- **Gated SR latch** uses the S and R inputs to set the latch to 1 or reset it to 0, respectively.
- **Gated D latch** uses the D input to force the latch into a state that has the same logic value as the D input.

A **flip-flop** is a storage element based on the gated latch principle, which can have its output state changed only on the edge of the controlling clock signal. We considered two types:

- **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs.
- **Master-slave flip-flop** is built with two gated latches. The master stage is active during half of the clock cycle, and the slave stage is active during the other half. The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage. Master-slave flip-flops can be edge-triggered or level sensitive. If the master stage is a gated D latch, then it behaves as an edge-triggered flip-flop. If the master stage is a gated SR latch, then the flip-flop is level sensitive (see problem 7.19).

7.8 REGISTERS

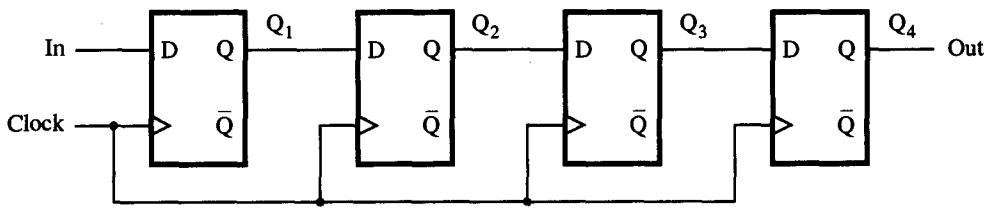
A flip-flop stores one bit of information. When a set of n flip-flops is used to store n bits of information, such as an n-bit number, we refer to these flip-flops as a *register*. A common clock is used for each flip-flop in a register, and each flip-flop operates as described in the

previous sections. The term register is merely a convenience for referring to n-bit structures consisting of flip-flops.

7.8.1 SHIFT REGISTER

In section 5.6 we explained that a given number is multiplied by 2 if its bits are shifted one bit position to the left and a 0 is inserted as the new least-significant bit. Similarly, the number is divided by 2 if the bits are shifted one bit-position to the right. A register that provides the ability to shift its contents is called a *shift register*.

Figure 7.18*a* shows a four-bit shift register that is used to shift its contents one bit-position to the right. The data bits are loaded into the shift register in a serial fashion using the *In* input. The contents of each flip-flop are transferred to the next flip-flop at each positive edge of the clock. An illustration of the transfer is given in Figure 7.18*b*, which shows what happens when the signal values at *In* during eight consecutive clock cycles are 1, 0, 1, 1, 1, 0, 0, and 0, assuming that the initial state of all flip-flops is 0.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 7.18 A simple shift register.

To implement a shift register, it is necessary to use either edge-triggered or master-slave flip-flops. The level-sensitive gated latches are not suitable, because a change in the value of *In* would propagate through more than one latch during the time when the clock is equal to 1.

7.8.2 PARALLEL-ACCESS SHIFT REGISTER

In computer systems it is often necessary to transfer n-bit data items. This may be done by transmitting all bits at once using n separate wires, in which case we say that the transfer is performed in *parallel*. But it is also possible to transfer all bits using a single wire, by performing the transfer one bit at a time, in n consecutive clock cycles. We refer to this scheme as *serial* transfer. To transfer an n-bit data item serially, we can use a shift register that can be loaded with all n bits in parallel (in one clock cycle). Then during the next n clock cycles, the contents of the register can be shifted out for serial transfer. The reverse operation is also needed. If bits are received serially, then after n clock cycles the contents of the register can be accessed in parallel as an n-bit item.

Figure 7.19 shows a four-bit shift register that allows the parallel access. Instead of using the normal shift register connection, the D input of each flip-flop is connected to

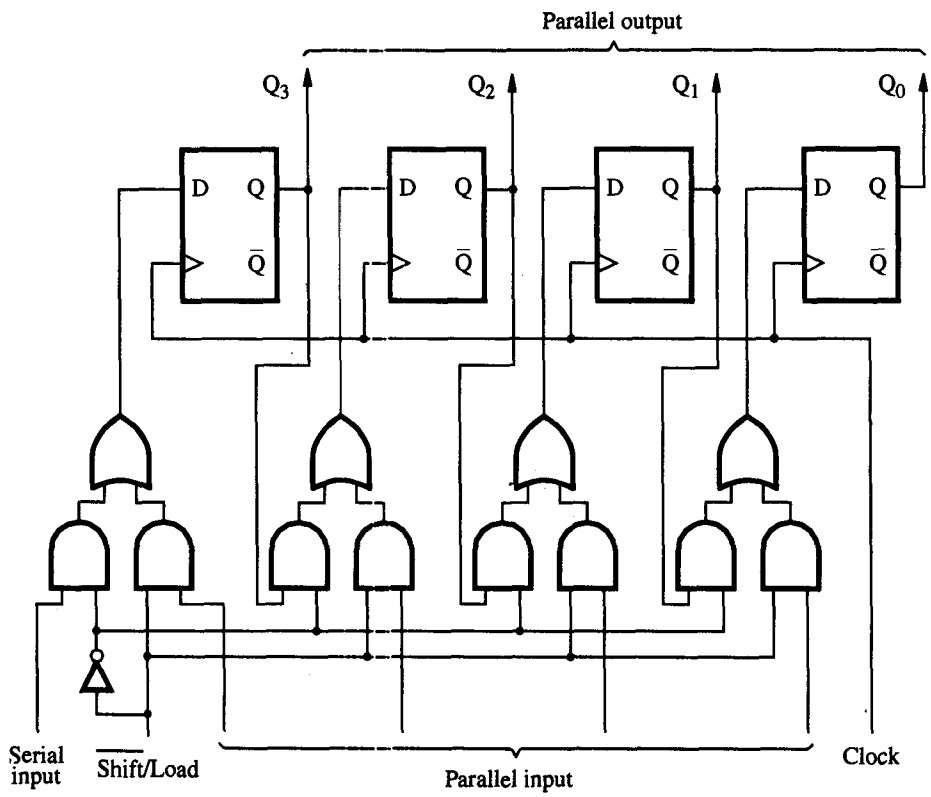


Figure 7.19 Parallel-access shift register.

two different sources. One source is the preceding flip-flop, which is needed for the shift-register operation. The other source is the external input that corresponds to the bit that is to be loaded into the flip-flop as a part of the parallel-load operation. The control signal $\overline{Shift}/Load$ is used to select the mode of operation. If $\overline{Shift}/Load = 0$, then the circuit operates as a shift register. If $\overline{Shift}/Load = 1$, then the parallel input data are loaded into the register. In both cases the action takes place on the positive edge of the clock.

In Figure 7.19 we have chosen to label the flip-flops outputs as $Q_3, \ldots, Q_0$ because shift registers are often used to hold binary numbers. The contents of the register can be accessed in parallel by observing the outputs of all flip-flops. The flip-flops can also be accessed serially, by observing the values of Q_0 during consecutive clock cycles while the contents are being shifted. A circuit in which data can be loaded in series and then accessed in parallel is called a series-to-parallel converter. Similarly, the opposite type of circuit is a parallel-to-series converter. The circuit in Figure 7.19 can perform both of these functions.

7.9 COUNTERS

In Chapter 5 we dealt with circuits that perform arithmetic operations. We showed how adder/subtractor circuits can be designed, either using a simple cascaded (ripple-carry) structure that is inexpensive but slow or using a more complex carry-lookahead structure that is both more expensive and faster. In this section we examine special types of addition and subtraction operations, which are used for the purpose of counting. In particular, we want to design circuits that can increment or decrement a count by 1. Counter circuits are used in digital systems for many purposes. They may count the number of occurrences of certain events, generate timing intervals for control of various tasks in a system, keep track of time elapsed between specific events, and so on.

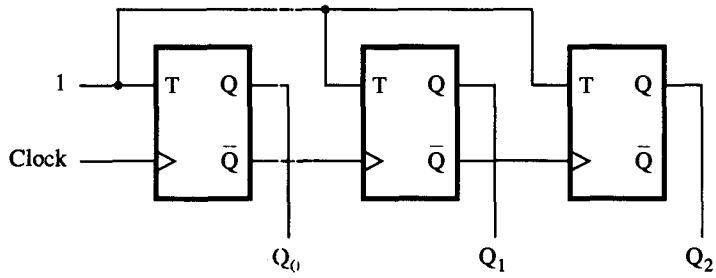
Counters can be implemented using the adder/subtractor circuits discussed in Chapter 5 and the registers discussed in section 7.8. However, since we only need to change the contents of a counter by 1, it is not necessary to use such elaborate circuits. Instead, we can use much simpler circuits that have a significantly lower cost. We will show how the counter circuits can be designed using T and D flip-flops.

7.9.1 ASYNCHRONOUS COUNTERS

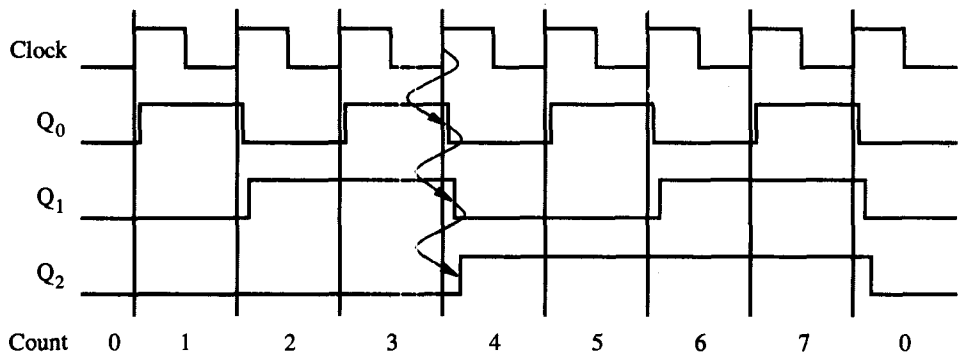
The simplest counter circuits can be built using T flip-flops because the toggle feature is naturally suited for the implementation of the counting operation.

Up-Counter with T Flip-Flops

Figure 7.20*a* gives a three-bit counter capable of counting from 0 to 7. The clock inputs of the three flip-flops are connected in cascade. The *T* input of each flip-flop is connected to a constant 1, which means that the state of the flip-flop will be reversed (toggled) at each positive edge of its clock. We are assuming that the purpose of this circuit is to count the number of pulses that occur on the primary input called *Clock*. Thus the clock input of the first flip-flop is connected to the *Clock* line. The other two flip-flops have their clock inputs driven by the $\overline{Q}$ output of the preceding flip-flop. Therefore, they toggle their state



(a) Circuit



(b) Timing diagram

Figure 7.20 A three-bit up-counter.

whenever the preceding flip-flop changes its state from Q = 1 to Q = 0, which results in a positive edge of the $\overline{Q}$ signal.

Figure 7.20*b* shows a timing diagram for the counter. The value of Q_0 toggles once each clock cycle. The change takes place shortly after the positive edge of the *Clock* signal. The delay is caused by the propagation delay through the flip-flop. Since the second flip-flop is clocked by $\overline{Q}_0$, the value of Q_1 changes shortly after the negative edge of the Q_0 signal. Similarly, the value of Q_2 changes shortly after the negative edge of the Q_1 signal. If we look at the values $Q_2Q_1Q_0$ as the count, then the timing diagram indicates that the counting sequence is 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, and so on. This circuit is a modulo-8 counter. Because it counts in the upward direction, we call it an *up-counter*.

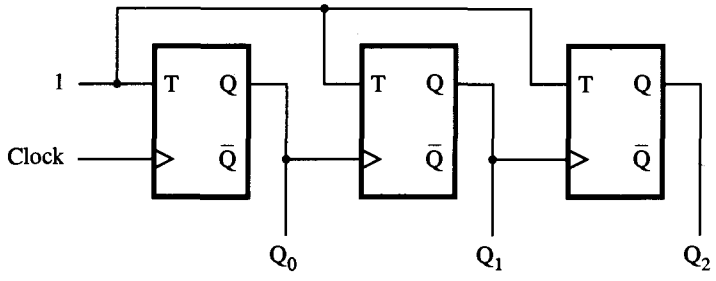
The counter in Figure 7.20*a* has three *stages*, each comprising a single flip-flop. Only the first stage responds directly to the *Clock* signal; we say that this stage is *synchronized* to the clock. The other two stages respond after an additional delay. For example, when *Count* = 3, the next clock pulse will cause the *Count* to go to 4. As indicated by the arrows in the timing diagram in Figure 7.20*b*, this change requires the toggling of the states of all three flip-flops. The change in Q_0 is observed only after a propagation delay from the positive edge of *Clock*. The Q_1 and Q_2 flip-flops have not yet changed; hence for a brief

time the count is $Q_2Q_1Q_0 = 010$. The change in Q_1 appears after a second propagation delay, at which point the count is 000. Finally, the change in Q_2 occurs after a third delay, at which point the stable state of the circuit is reached and the count is 100. This behavior is similar to the rippling of carries in the ripple-carry adder circuit of Figure 5.6. The circuit in Figure 7.20*a* is an *asynchronous counter*, or a *ripple counter*.

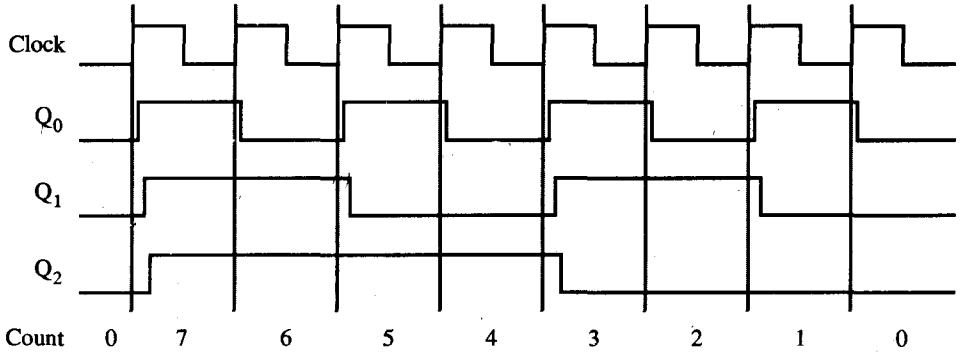
Down-Counter with T Flip-Flops

A slight modification of the circuit in Figure 7.20*a* is presented in Figure 7.21*a*. The only difference is that in Figure 7.21*a* the clock inputs of the second and third flip-flops are driven by the Q outputs of the preceding stages, rather than by the $\overline{Q}$ outputs. The timing diagram, given in Figure 7.21*b*, shows that this circuit counts in the sequence 0, 7, 6, 5, 4, 3, 2, 1, 0, 7, and so on. Because it counts in the downward direction, we say that it is a *down-counter*.

It is possible to combine the functionality of the circuits in Figures 7.20*a* and 7.21*a* to form a counter that can count either up or down. Such a counter is called an *up/down-counter*. We leave the derivation of this counter as an exercise for the reader (problem 7.16).



(a) Circuit



(b) Timing diagram

Figure 7.21 A three-bit down-counter.

7.9.2 SYNCHRONOUS COUNTERS

The asynchronous counters in Figures 7.20*a* and 7.21*a* are simple, but not very fast. If a counter with a larger number of bits is constructed in this manner, then the delays caused by the cascaded clocking scheme may become too long to meet the desired performance requirements. We can build a faster counter by clocking all flip-flops at the same time, using the approach described below.

Synchronous Counter with T Flip-Flops

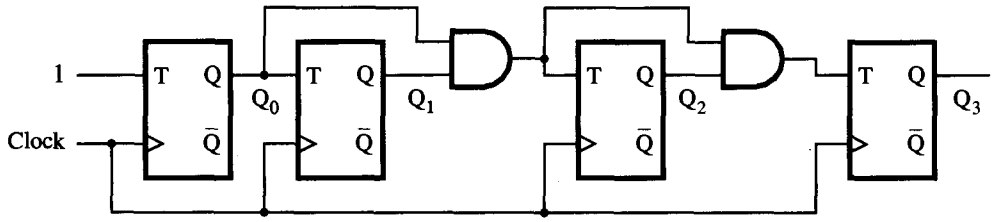
Table 7.1 shows the contents of a three-bit up-counter for eight consecutive clock cycles, assuming that the count is initially 0. Observing the pattern of bits in each row of the table, it is apparent that bit Q_0 changes on each clock cycle. Bit Q_1 changes only when $Q_0 = 1$. Bit Q_2 changes only when both Q_1 and Q_0 are equal to 1. In general, for an n-bit up-counter, a given flip-flop changes its state only when all the preceding flip-flops are in the state $Q = 1$. Therefore, if we use T flip-flops to realize the counter, then the T inputs are defined as

$$T_0 = 1$$
$$T_1 = Q_0$$
$$T_2 = Q_0Q_1$$
$$T_3 = Q_0Q_1Q_2$$
$$\vdots$$
$$T_n = Q_0Q_1 \cdots Q_{n-1}$$

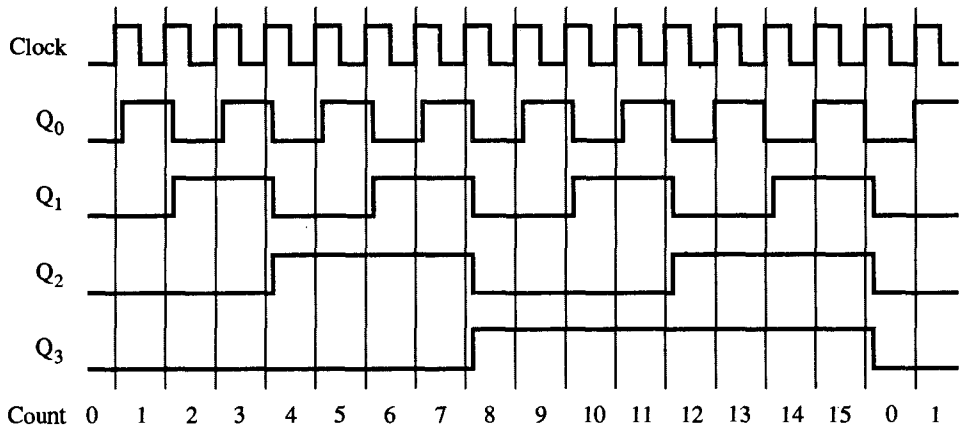
An example of a four-bit counter based on these expressions is given in Figure 7.22*a*. Instead of using AND gates of increased size for each stage, which may lead to fan-in problems, we use a factored arrangement, as shown in the figure. This arrangement does not slow down the response of the counter, because all flip-flops change their states after a

Table 7.1 Derivation of the synchronous up-counter.

Clock cycle	Q_2	Q_1	Q_0	
0	0	0	0	
1	0	0	1	
2	0	1	0	Q_1 changes
3	0	1	1	
4	1	0	0	Q_1 changes, Q_2 changes
5	1	0	1	
6	1	1	0	Q_1 changes
7	1	1	1	
8	0	0	0	Q_1 changes, Q_2 changes



(a) Circuit



(b) Timing diagram

Figure 7.22 A four-bit synchronous up-counter.

propagation delay from the positive edge of the clock. Note that a change in the value of Q_0 may have to propagate through several AND gates to reach the flip-flops in the higher stages of the counter, which requires a certain amount of time. This time must not exceed the clock period. Actually, it must be less than the clock period minus the setup time for the flip-flops.

Figure 7.22*b* gives a timing diagram. It shows that the circuit behaves as a modulo-16 up-counter. Because all changes take place with the same delay after the active edge of the *Clock* signal, the circuit is called a *synchronous counter.*

Enable and Clear Capability

The counters in Figures 7.20 through 7.22 change their contents in response to each clock pulse. Often it is desirable to be able to inhibit counting, so that the count remains in its present state. This may be accomplished by including an *Enable* control signal, as indicated in Figure 7.23. The circuit is the counter of Figure 7.22, where the *Enable* signal controls directly the *T* input of the first flip-flop. Connecting the *Enable* also to the AND-

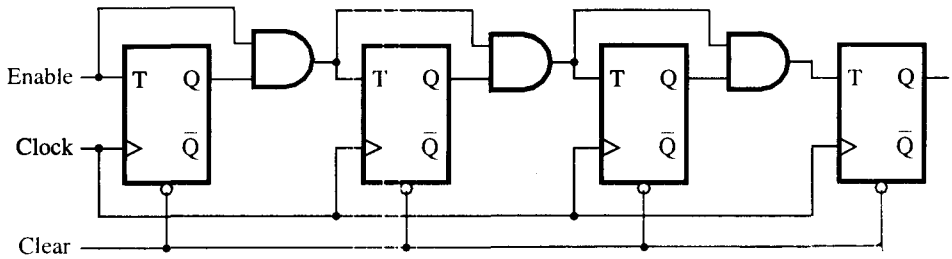


Figure 7.23 Inclusion of Enable and Clear capability.

gate chain means that if *Enable* = 0, then all T inputs will be equal to 0. If *Enable* = 1, then the counter operates as explained previously.

In many applications it is necessary to start with the count equal to zero. This is easily achieved if the flip-flops can be cleared, as explained in section 7.4.3. The clear inputs on all flip-flops can be tied together and driven by a *Clear* control input.

Synchronous Counter with D Flip-Flops

While the toggle feature makes T flip-flops a natural choice for the implementation of counters, it is also possible to build counters using other types of flip-flops. The JK flip-flops can be used in exactly the same way as the T flip-flops because if the J and K inputs are tied together, a JK flip-flop becomes a T flip-flop. We will now consider using D flip-flops for this purpose.

It is not obvious how D flip-flops can be used to implement a counter. We will present a formal method for deriving such circuits in Chapter 8. Here we will present a circuit structure that meets the requirements but will leave the derivation for Chapter 8. Figure 7.24 gives a four-bit up-counter that counts in the sequence 0, 1, 2, . . . , 14, 15, 0, 1, and so on. The count is indicated by the flip-flop outputs $Q_3Q_2Q_1Q_0$. If we assume that *Enable* = 1, then the D inputs of the flip-flops are defined by the expressions

$$D_0 = \overline{Q}_0 = 1 \oplus Q_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1Q_0$$
$$D_3 = Q_3 \oplus Q_2Q_1Q_0$$

For a larger counter the ith stage is defined by

$$D_i = Q_i \oplus Q_{i-1}Q_{i-2} \cdots Q_1Q_0$$

We will show how to derive these equations in Chapter 8.

We have included the *Enable* control signal so that the counter counts the clock pulses only if *Enable* = 1. In effect, the above equations are modified to implement the circuit in the figure as follows

$$D_0 = Q_0 \oplus Enable$$
$$D_1 = Q_1 \oplus Q_0 \cdot Enable$$

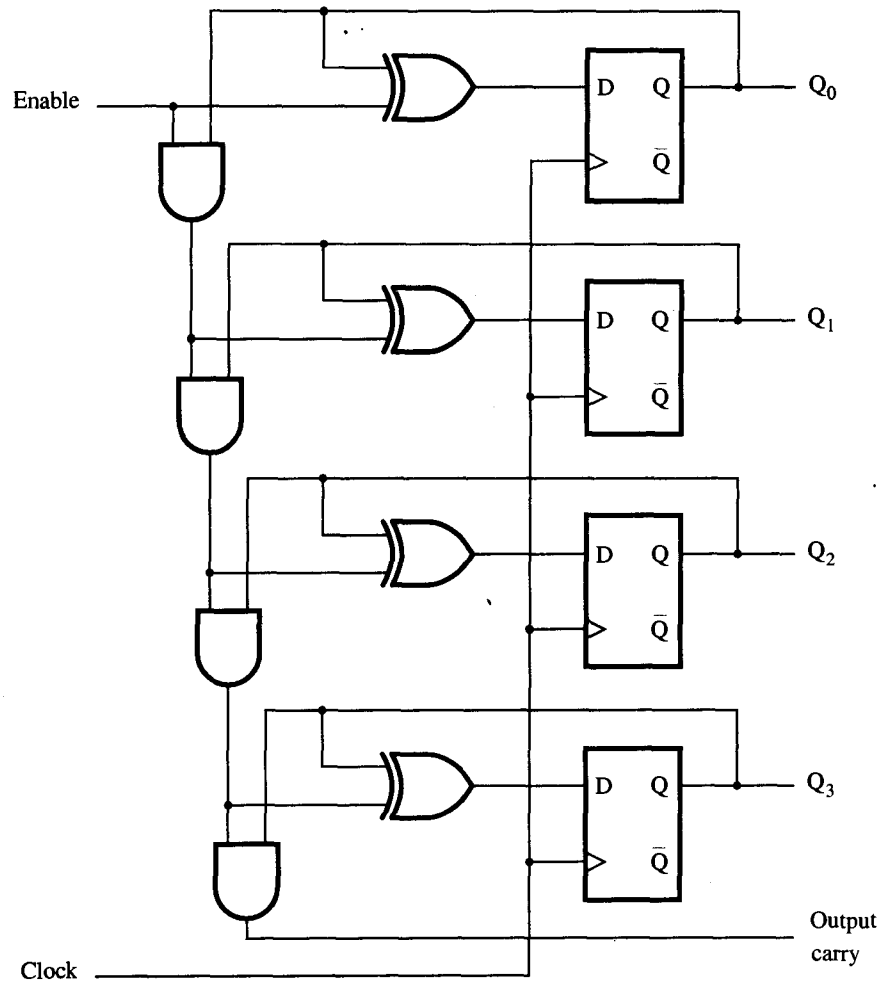


Figure 7.24 A four-bit counter with D flip-flops.

$$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot \textit{Enable}$$
$$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot \textit{Enable}$$

The operation of the counter is based on our observation for Table 7.1 that the state of the flip-flop in stage i changes only if all preceding flip-flops are in the state $Q = 1$. This makes the output of the AND gate that feeds stage i equal to 1, which causes the output of the XOR gate connected to D_i to be equal to $\overline{Q}_i$. Otherwise, the output of the XOR gate provides $D_i = Q_i$, and the flip-flop remains in the same state. This resembles the carry propagation in a carry-lookahead adder circuit (see section 5.4); hence the AND-gate chain can be thought of as the *carry chain.* Even though the circuit is only a four-bit counter, we have included an extra AND that produces the "output carry." This signal makes it easy to concatenate two such four-bit counters to create an eight-bit counter.

Finally, the reader should note that the counter in Figure 7.24 is essentially the same as the circuit in Figure 7.23. We showed in Figure 7.16*a* that a T flip-flop can be formed from a D flip-flop by providing the extra gating that gives

$$D = Q\overline{T} + \overline{Q}T$$
$$= Q \oplus T$$

Thus in each stage in Figure 7.24, the D flip-flop and the associated XOR gate implement the functionality of a T flip-flop.

7.9.3 COUNTERS WITH PARALLEL LOAD

Often it is necessary to start counting with the initial count being equal to 0. This state can be achieved by using the capability to clear the flip-flops as indicated in Figure 7.23. But sometimes it is desirable to start with a different count. To allow this mode of operation, a counter circuit must have some inputs through which the initial count can be loaded. Using the *Clear* and *Preset* inputs for this purpose is a possibility, but a better approach is discussed below.

The circuit of Figure 7.24 can be modified to provide the parallel-load capability as shown in Figure 7.25. A two-input multiplexer is inserted before each D input. One input to the multiplexer is used to provide the normal counting operation. The other input is a data bit that can be loaded directly into the flip-flop. A control input, *Load*, is used to choose the mode of operation. The circuit counts when $Load = 0$. A new initial value, $D_3D_2D_1D_0$, is loaded into the counter when $Load = 1$.

7.10 RESET SYNCHRONIZATION

We have already mentioned that it is important to be able to clear, or *reset*, the contents of a counter prior to commencing a counting operation. This can be done using the clear capability of the individual flip-flops. But we may also be interested in resetting the count to 0 during the normal counting process. An n-bit up-counter functions naturally as a modulo-2^n counter. Suppose that we wish to have a counter that counts modulo some base that is not a power of 2. For example, we may want to design a modulo-6 counter, for which the counting sequence is 0, 1, 2, 3, 4, 5, 0, 1, and so on.

The most straightforward approach is to recognize when the count reaches 5 and then reset the counter. An AND gate can be used to detect the occurrence of the count of 5. Actually, it is sufficient to ascertain that $Q_2 = Q_0 = 1$, which is true only for 5 in our desired counting sequence. A circuit based on this approach is given in Figure 7.26*a*. It uses a three-bit synchronous counter of the type depicted in Figure 7.25. The parallel-load feature of the counter is used to reset its contents when the count reaches 5. The resetting action takes place at the positive clock edge after the count has reached 5. It involves loading $D_2D_1D_0 = 000$ into the flip-flops. As seen in the timing diagram in Figure 7.26*b*,

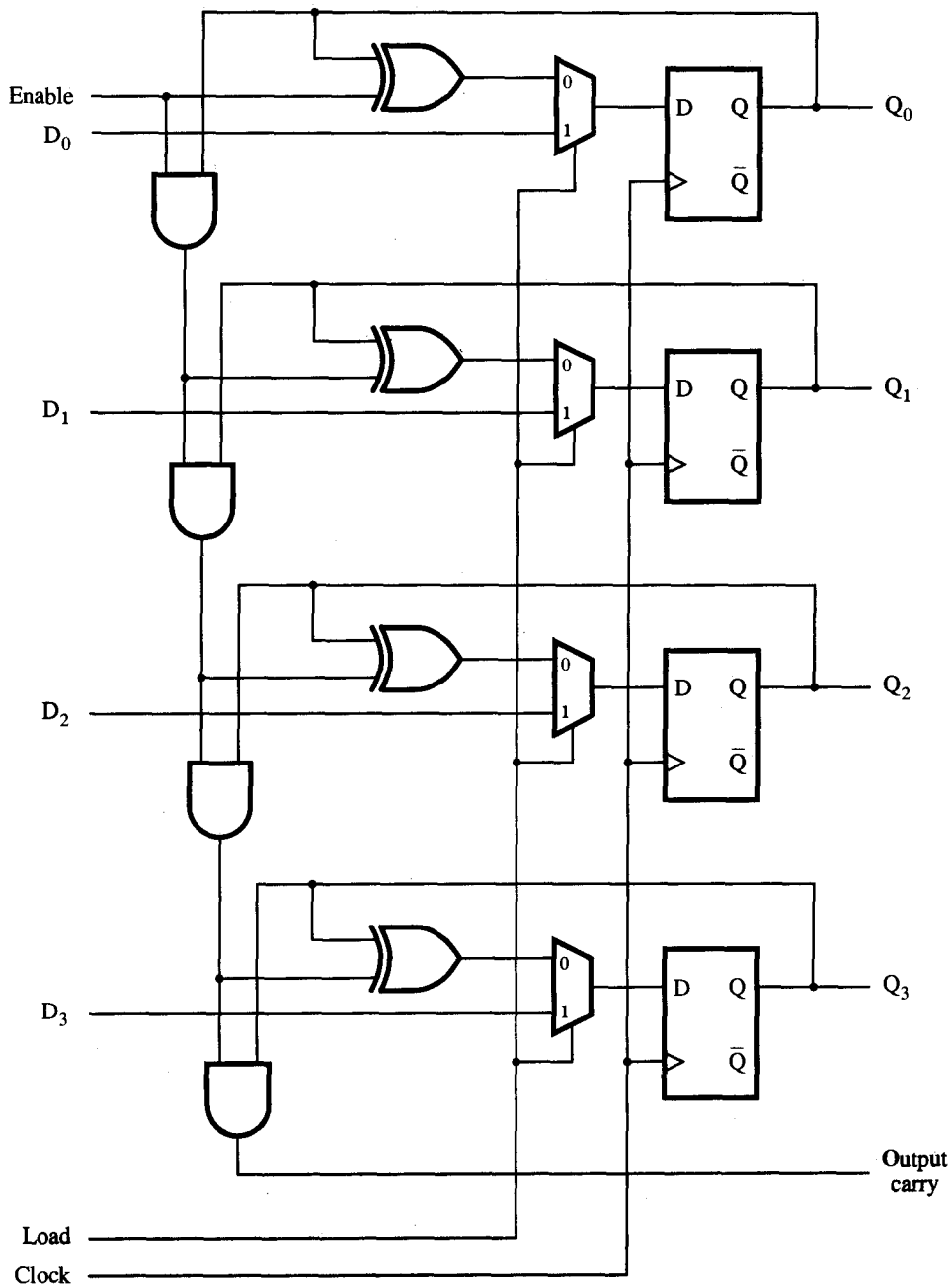
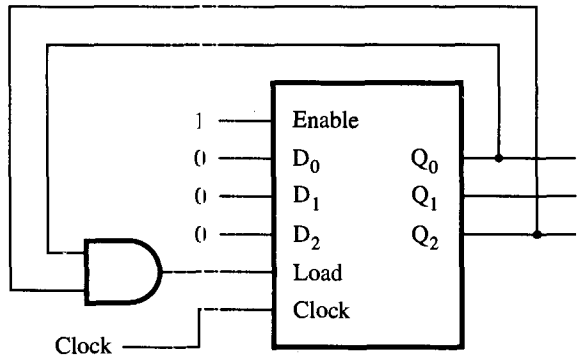
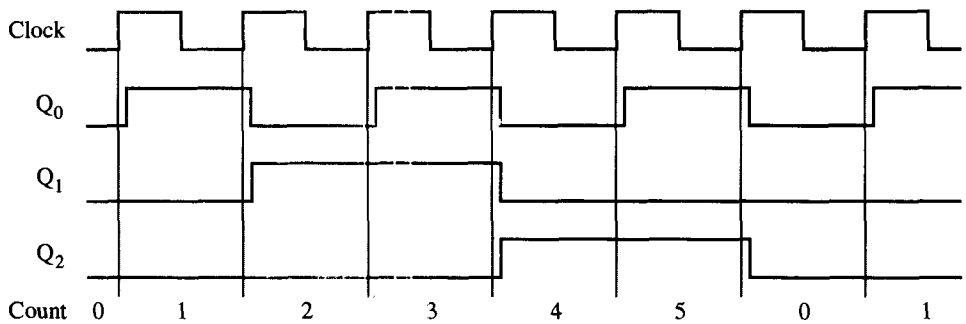


Figure 7.25 A counter with parallel-load capability.



(a) Circuit

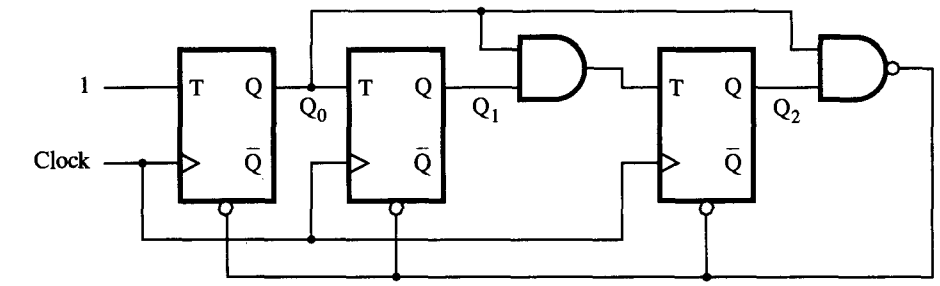


(b) Timing diagram

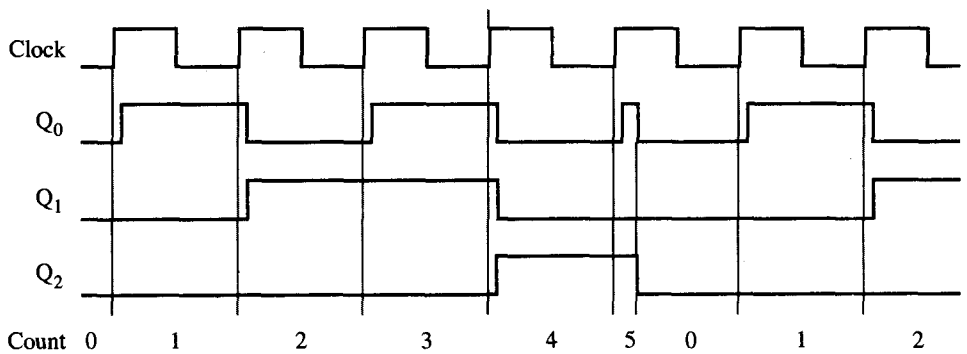
Figure 7.26 A modulo-6 counter with synchronous reset.

the desired counting sequence is achieved, with each value of the count being established for one full clock cycle. Because the counter is reset on the active edge of the clock, we say that this type of counter has a *synchronous reset.*

Consider now the possibility of using the clear feature of individual flip-flops, rather than the parallel-load approach. The circuit in Figure 7.27*a* illustrates one possibility. It uses the counter structure of Figure 7.22*a*. Since the clear inputs are active when low, a NAND gate is used to detect the occurrence of the count of 5 and cause the clearing of all three flip-flops. Conceptually, this seems to work fine, but closer examination reveals a potential problem. The timing diagram for this circuit is given in Figure 7.27*b*. It shows a difficulty that arises when the count is equal to 5. As soon as the count reaches this value, the NAND gate triggers the resetting action. The flip-flops are cleared to 0 a short time after the NAND gate has detected the count of 5. This time depends on the gate delays in the



(a) Circuit



(b) Timing diagram

Figure 7.27 A modulo-6 counter with asynchronous reset.

circuit, but not on the clock. Therefore, signal values $Q_2Q_1Q_0 = 101$ are maintained for a time that is much less than a clock cycle. Depending on a particular application of such a counter, this may be adequate, but it may also be completely unacceptable. For example, if the counter is used in a digital system where all operations in the system are synchronized by the same clock, then this narrow pulse denoting *Count* = 5 would not be seen by the rest of the system. To solve this problem, we could try to use a modulo-7 counter instead, assuming that the system would ignore the short pulse that denotes the count of 6. This is not a good way of designing circuits, because undesirable pulses often cause unforeseen difficulties in practice. The approach employed in Figure 7.27*a* is said to use *asynchronous reset.*

The timing diagrams in Figures 7.26*b* and 7.27*b* suggest that synchronous reset is a better choice than asynchronous reset. The same observation is true if the natural counting sequence has to be broken by loading some value other than zero. The new value of the count can be established cleanly using the parallel-load feature. The alternative of using the clear and preset capability of individual flip-flops to set their states to reflect the desired count has the same problems as discussed in conjunction with the asynchronous reset.

7.11 OTHER TYPES OF COUNTERS

In this section we discuss three other types of counters that can be found in practical applications. The first uses the decimal counting sequence, and the other two generate sequences of codes that do not represent binary numbers.

7.11.1 BCD COUNTER

Binary-coded-decimal (BCD) counters can be designed using the approach explained in section 7.10. A two-digit BCD counter is presented in Figure 7.28. It consists of two modulo-10 counters, one for each BCD digit, which we implemented using the parallel-load four-bit counter of Figure 7.25. Note that in a modulo-10 counter it is necessary to reset the four flip-flops after the count of 9 has been obtained. Thus the *Load* input to each stage is equal to 1 when $Q_3 = Q_0 = 1$, which causes 0s to be loaded into the flip-flops at the next positive edge of the clock signal. Whenever the count in stage 0, BCD_0, reaches 9 it is necessary to enable the second stage so that it will be incremented when the next clock

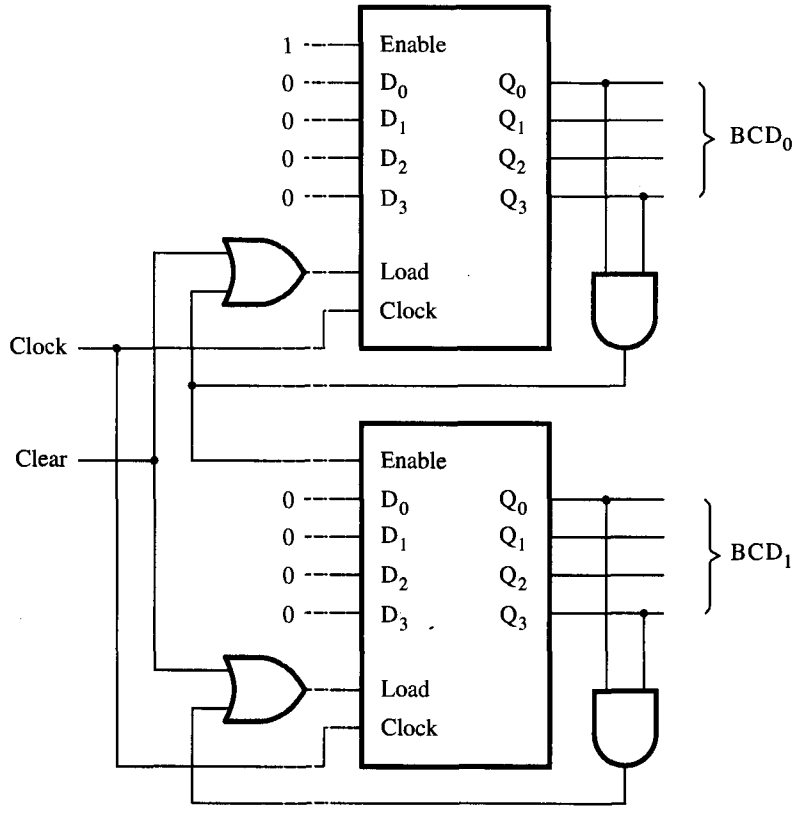


Figure 7.28 A two-digit BCD counter.

pulse arrives. This is accomplished by keeping the *Enable* signal for BCD_1 low at all times except when $BCD_0 = 9$.

In practice, it has to be possible to clear the contents of the counter by activating some control signal. Two OR gates are included in the circuit for this purpose. The control input *Clear* can be used to load 0s into the counter. Observe that in this case *Clear* is active when high. Verilog code for a two-digit BCD counter is given in Figure 7.81.

In any digital system there is usually one or more clock signals used to drive all synchronous circuitry. In the preceding counter, as well as in all counters presented in the previous figures, we have assumed that the objective is to count the number of clock pulses. Of course, these counters can be used to count the number of pulses in any signal that may be used in place of the clock signal.

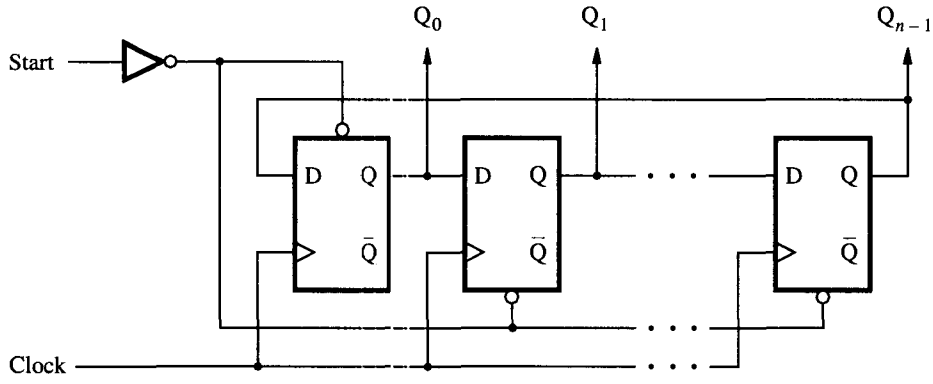
7.11.2 RING COUNTER

In the preceding counters the count is indicated by the state of the flip-flops in the counter. In all cases the count is a binary number. Using such counters, if an action is to be taken as a result of a particular count, then it is necessary to detect the occurrence of this count. This may be done using AND gates, as illustrated in Figures 7.26 through 7.28.

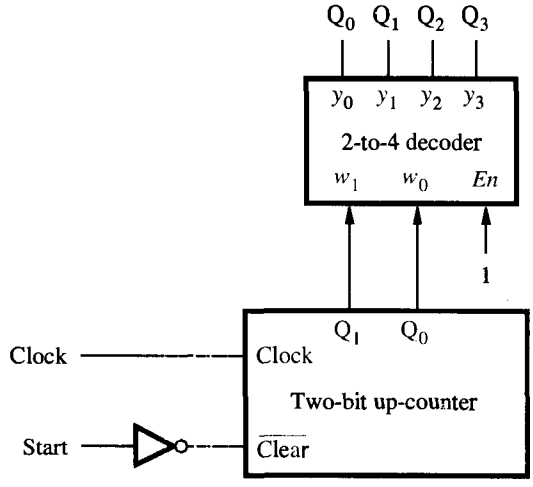
It is possible to devise a counterlike circuit in which each flip-flop reaches the state $Q_i = 1$ for exactly one count, while for all other counts $Q_i = 0$. Then Q_i indicates directly an occurrence of the corresponding count. Actually, since this does not represent binary numbers, it is better to say that the outputs of the flips-flops represent a code. Such a circuit can be constructed from a simple shift register, as indicated in Figure 7.29*a*. The Q output of the last stage in the shift register is fed back as the input to the first stage, which creates a ringlike structure. If a single 1 is injected into the ring, this 1 will be shifted through the ring at successive clock cycles. For example, in a four-bit structure, the possible codes $Q_0Q_1Q_2Q_3$ will be 1000, 0100, 0010, and 0001. As we said in section 6.2, such encoding, where there is a single 1 and the rest of the code variables are 0, is called a *one-hot code.*

The circuit in Figure 7.29*a* is referred to as a *ring counter.* Its operation has to be initialized by injecting a 1 into the first stage. This is achieved by using the *Start* control signal, which presets the left-most flip-flop to 1 and clears the others to 0. We assume that all changes in the value of the *Start* signal occur shortly after an active clock edge so that the flip-flop timing parameters are not violated.

The circuit in Figure 7.29*a* can be used to build a ring counter with any number of bits, n. For the specific case of $n = 4$, part (*b*) of the figure shows how a ring counter can be constructed using a two-bit up-counter and a decoder. When *Start* is set to 1, the counter is reset to 00. After *Start* changes back to 0, the counter increments its value in the normal way. The 2-to-4 decoder, described in section 6.2, changes the counter output into a one-hot code. For the count values 00, 01, 10, 11, 00, and so on, the decoder produces $Q_0Q_1Q_2Q_3 = 1000, 0100, 0010, 0001, 1000$, and so on. This circuit structure can be used for larger ring counters, as long as the number of bits is a power of two. We will give an example of a larger circuit that uses the ring counter in Figure 7.29*b* as a subcircuit in section 7.14.



(a) An *n*-bit ring counter



(b) A four-bit ring counter

Figure 7.29 Ring counter.

7.11.3 JOHNSON COUNTER

An interesting variation of the ring counter is obtained if, instead of the Q output, we take the $\overline{Q}$ output of the last stage and feed it back to the first stage, as shown in Figure 7.30. This circuit is known as a *Johnson counter*. An *n*-bit counter of this type generates a counting sequence of length $2n$. For example, a four-bit counter produces the sequence 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, and so on. Note that in this sequence, only a single bit has a different value for two consecutive codes.

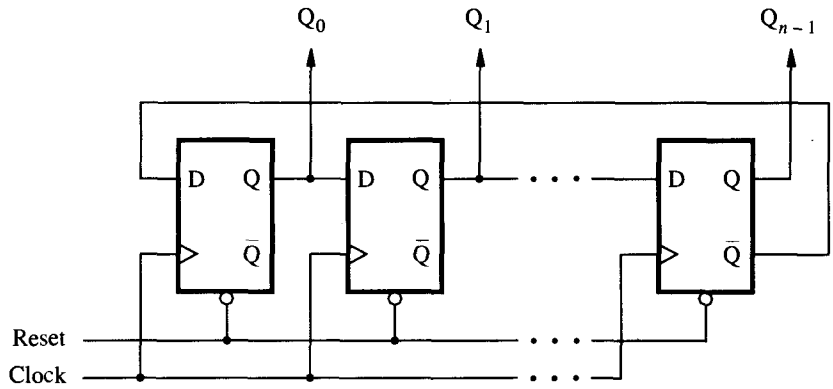


Figure 7.30 Johnson counter.

To initialize the operation of the Johnson counter, it is necessary to reset all flip-flops, as shown in the figure. Observe that neither the Johnson nor the ring counter will generate the desired counting sequence if not initialized properly.

7.11.4 REMARKS ON COUNTER DESIGN

The sequential circuits presented in this chapter, namely, registers and counters, have a regular structure that allows the circuits to be designed using an intuitive approach. In Chapter 8 we will present a more formal approach to design of sequential circuits and show how the circuits presented in this chapter can be derived using this approach.

7.12 USING STORAGE ELEMENTS WITH CAD TOOLS

This section shows how circuits with storage elements can be designed using either schematic capture or Verilog code.

7.12.1 INCLUDING STORAGE ELEMENTS IN SCHEMATICS

One way to create a circuit is to draw a schematic that builds latches and flip-flops from logic gates. Because these storage elements are used in many applications, most CAD systems provide them as prebuilt modules. Figure 7.31 shows a schematic created with a schematic capture tool, which includes three types of flip-flops that are imported from a library provided as part of the CAD system. The top element is a gated D latch, the middle element is a positive-edge-triggered D flip-flop, and the bottom one is a positive-edge-triggered T flip-flop. The D and T flip-flops have asynchronous, active-low clear and preset inputs. If these inputs are not connected in a schematic, then the CAD tool makes them inactive by assigning the default value of 1 to them.

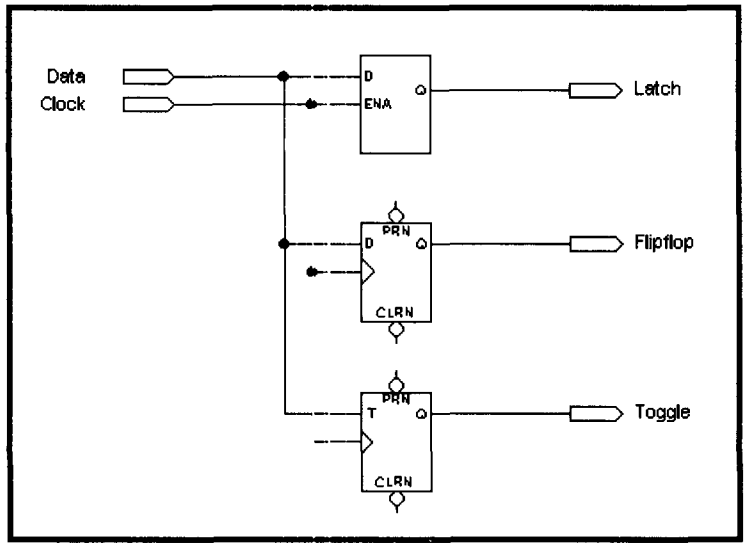


Figure 7.31 Three types of storage elements in a schematic.

When the gated D latch is synthesized for implementation in a chip, the CAD tool may not generate the cross-coupled NOR or NAND gates shown in section 7.2. In some chips, such as a CPLD, the AND-OR circuit depicted in Figure 7.32 may be preferable. This circuit is functionally equivalent to the cross-coupled version in section 7.2. The sum-of-products circuit is used because it is more suitable for implementation in a CPLD macrocell. One aspect of this circuit should be mentioned. From the functional point of view, it appears that the circuit can be simplified by removing the AND gate with the inputs *Data* and *Latch*. Without this gate, the top AND gate sets the value stored in the latch when the clock is 1, and the bottom AND gate maintains the stored value when the clock is 0. But without this gate, the circuit has a timing problem known as a *static hazard*. A detailed explanation of hazards will be given in section 9.6.

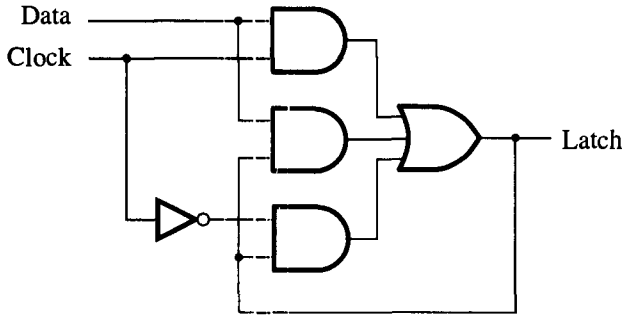


Figure 7.32 Gated D latch generated by CAD tools.

The circuit in Figure 7.31 can be implemented in a CPLD as shown in Figure 7.33. The D and T flip-flops are realized using the flip-flops on the chip that are configurable as either D or T types. The figure depicts in blue the gates and wires needed to implement the circuit in Figure 7.31.

The results of a timing simulation for the implementation in Figure 7.33 are given in Figure 7.34. The *Latch* signal, which is the output of the gated D latch, implemented as indicated in Figure 7.32, follows the *Data* input whenever the *Clock* signal is 1. Because

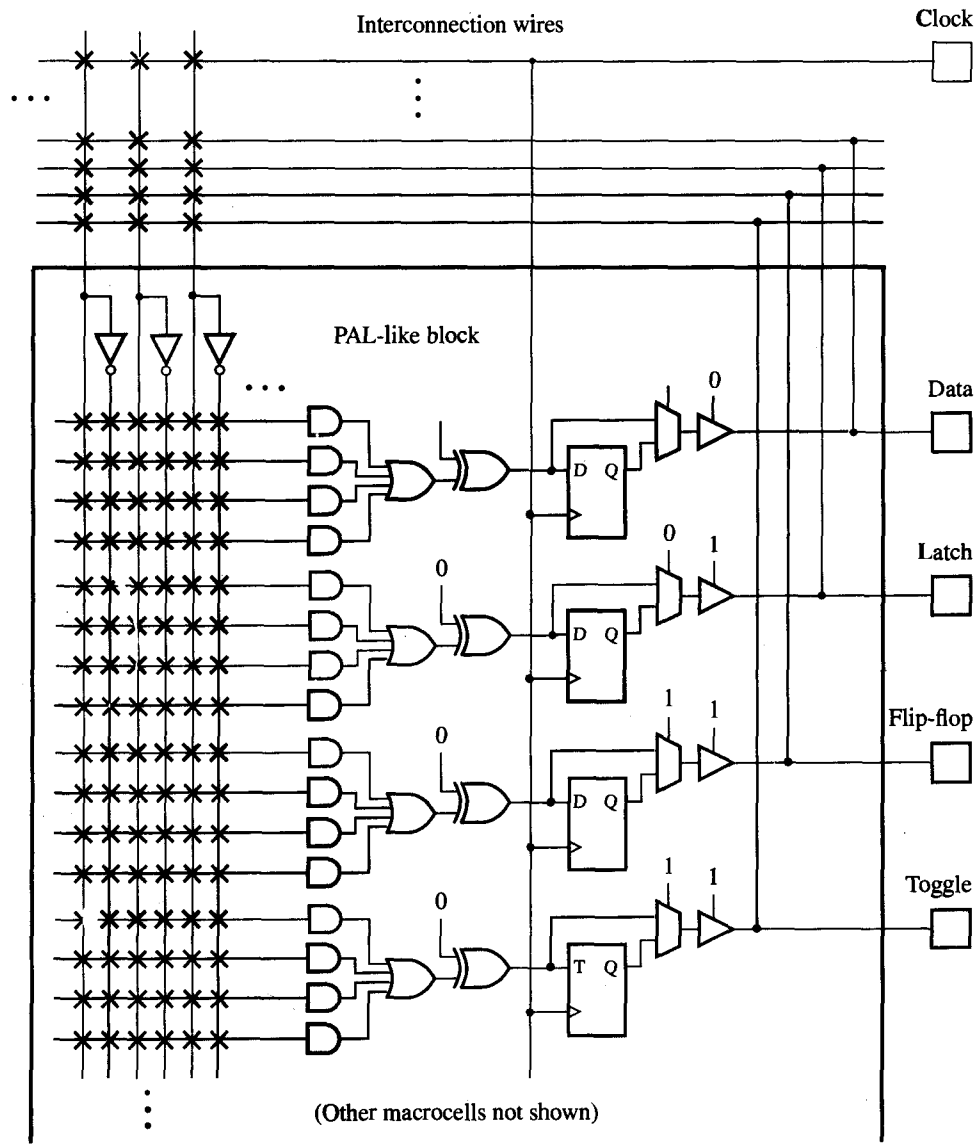


Figure 7.33 Implementation of the schematic in Figure 7.31 in a CPLD.

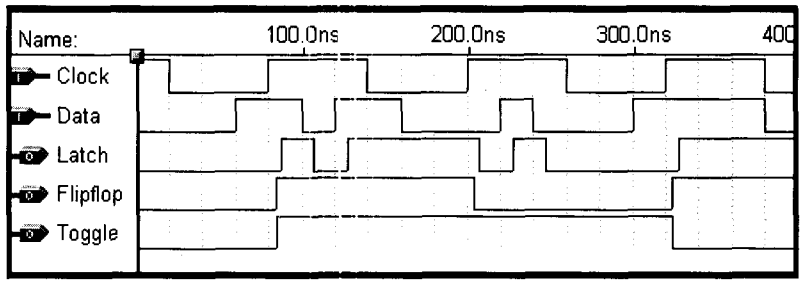


Figure 7.34 Timing simulation for the storage elements in Figure 7.31.

of propagation delays in the chip, the *Latch* signal is delayed in time with respect to the *Data* signal. Since the *Flipflop* signal is the output of the D flip-flop, it changes only after a positive clock edge. Similarly, the output of the T flip-flop, called *Toggle* in the figure, toggles when *Data* = 1 and a positive clock edge occurs. The timing diagram illustrates the delay from when the positive clock edge occurs at the input pin of the chip until a change in the flip-flop output appears at the output pin of the chip. This time is called the *clock-to-output time*, t_{co}.

7.12.2 USING VERILOG CONSTRUCTS FOR STORAGE ELEMENTS

In section 6.6 we described a number of Verilog constructs. We now show how these constructs can be used to describe storage elements.

A simple way of specifying a storage element is by using the **if-else** statement to describe the desired behavior responding to changes in the levels of data and clock inputs. Consider the **always** block

```
always @(Control or B)
  if (Control)
    A = B;
```

where *A* is a variable of **reg** type. This code specifies that the value of *A* should be made equal to the value of *B* when *Control* = 1. But the statement does not indicate an action that should occur when *Control* = 0. In the absence of an assigned value, the Verilog compiler assumes that the value of *A* caused by the **if** statement must be maintained until the next time this **if** statement is evaluated. This notion of *implied memory* is realized by instantiating a latch in the circuit.

Example 7.1 **CODE FOR A GATED D LATCH** The code in Figure 7.35 defines a module named *D_latch*, which has the inputs *D* and *Clk* and the output Q. The **if** clause defines that the Q output must take the value of *D* when *Clk* = 1. Since no **else** clause is given, a latch will be synthesized to maintain the value of Q when *Clk* = 0. Therefore, the code describes a gated

```
module D_latch (D, Clk, Q);
  input D, Clk;
  output Q;
  reg Q;

  always @(D or Clk)
    if (Clk)
      Q = D;

endmodule
```

Figure 7.35 Code for a gated D latch.

D latch. The sensitivity list includes *Clk* and *D* because both of these signals can cause a change in the value of the Q output.

An **always** construct is used to define a circuit that responds to changes in the signals that appear in the sensitivity list. While in the examples presented so far the **always** blocks are sensitive to the *levels* of signals, it is also possible to specify that a response should take place only at a particular edge of a signal. The desired edge is specified by using the Verilog keywords **posedge** and **negedge**, which are used to implement edge-triggered circuits.

CODE FOR A D FLIP-FLOP Figure 7.36 defines a module named *flipflop*, which is a positive-edge-triggered D flip-flop. The sensitivity list contains only the clock signal because it is the only signal that can cause a change in the Q output. The keyword **posedge** specifies that a change may occur only on the positive edge of *Clock*. At this time the output

Example 7.2

```
module flipflop (D, Clock, Q);
  input D, Clock;
  output Q;
  reg Q;

  always @(posedge Clock)
    Q = D;

endmodule
```

Figure 7.36 Code for a D flip-flop.

Q is set to the value of the input *D*. Since Q is of **reg** type it will maintain its value between the positive edges of the clock.

7.12.3 BLOCKING AND NON-BLOCKING ASSIGNMENTS

In all our Verilog examples presented so far we have used the equal sign for assignments, as in

```
f = x1 & x2;
```

or

```
C = A + B;
```

or

```
Q = D;
```

This notation is called a *blocking* assignment. A Verilog compiler evaluates the statements in an **always** block in the order in which they are written. If a variable is given a value by a blocking assignment statement, then this new value is used in evaluating all subsequent statements in the block.

Example 7.3 Consider the code in Figure 7.37. Since the **always** block is sensitive to the positive clock edge, both Q1 and Q2 will be implemented as the outputs of D flip-flops. However, because blocking assignments are involved, these two flip-flops will not be connected in cascade, as the reader might expect. The first statement

```
Q1 = D;
```

sets Q1 to the value of *D*. This new value is used in evaluating the subsequent statement

```
module example7_3 (D, Clock, Q1, Q2);
  input D, Clock;
  output Q1, Q2;
  reg Q1, Q2;

  always @(posedge Clock)
  begin
    Q1 = D;
    Q2 = Q1;
  end

endmodule
```

Figure 7.37 Incorrect code for two cascaded flip-flops.

```
Q2 = Q1;
```

which results in Q2 = Q1 = D. The synthesized circuit has two parallel flip-flops, as illustrated in Figure 7.38. A synthesis tool will likely delete one of these redundant flip-flops as an optimization step.

Verilog also provides a *non-blocking* assignment, denoted with <=. All non-blocking assignment statements in an **always** block are evaluated using the values that the variables have when the **always** block is entered. Thus, a given variable has the same value for all statements in the block. The meaning of non-blocking is that the result of each assignment is not seen until the end of the **always** block.

Example 7.4

Figure 7.39 gives the same code as in Figure 7.37, but using non-blocking assignments. In the two statements

```
Q1 <= D;
Q2 <= Q1;
```

the variables Q1 and Q2 have some value at the start of evaluating the **always** block, and then they change to a new value concurrently at the end of the **always** block. This code generates a cascaded connection between flip-flops, which implements the shift register depicted in Figure 7.40.

The differences between blocking and non-blocking assignments are illustrated further by the following two examples.

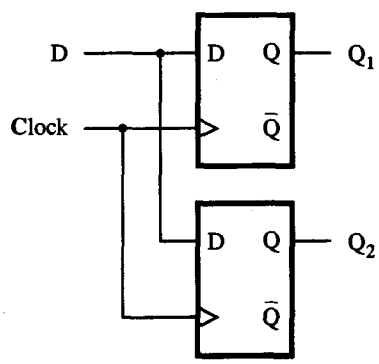


Figure 7.38 Circuit for Example 7.3.

```
module example7_4 (D, Clock, Q1, Q2);
  input D, Clock;
  output Q1, Q2;
  reg Q1, Q2;

  always @(posedge Clock)
  begin
    Q1 <= D;
    Q2 <= Q1;
  end

endmodule
```

Figure 7.39 Code for two cascaded flip-flops.

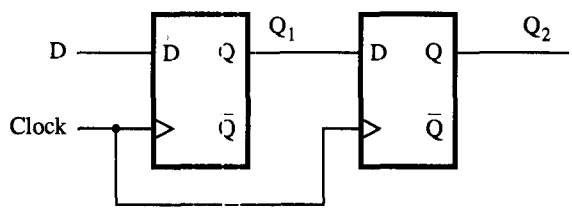


Figure 7.40 Circuit defined in Figure 7.39.

Example 7.5 Code that involves some gates in addition to flip-flops is defined in Figure 7.41 using blocking assignment statements. The resulting circuit is given in Figure 7.42. Both f and g are implemented as the outputs of D flip-flops, because the sensitivity list of the **always** block specifies the event **posedge** Clock. Since blocking assignments are used, the updated value of f generated by the statement f = x1 & x2 has to be seen immediately by the following statement g = f | x3. Thus, the AND gate that produces x1 & x2 is connected to the OR gate that feeds the g flip-flop, as shown in Figure 7.42.

Example 7.6 If non-blocking assignments are used, as given in Figure 7.43, then both f and g are updated simultaneously. Hence, the previous value of f is used in updating the value of g, which means that the output of the flip-flop that generates f is connected to the OR gate that feeds the g flip-flop. This gives rise to the circuit in Figure 7.44.

It is interesting to consider what circuit would be synthesized if the statements that specify f and g were reversed. For the code in Figure 7.41 the impact would be significant. If g is evaluated first, then the second statement does not depend on the first one, because f does not depend on g. The resulting circuit would be the same as the one in Figure 7.44.

```
module example7_5 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

   always @(posedge Clock)
   begin
      f = x1 & x2;
      g = f | x3;
   end

endmodule
```

Figure 7.41 Code for Example 7.5.

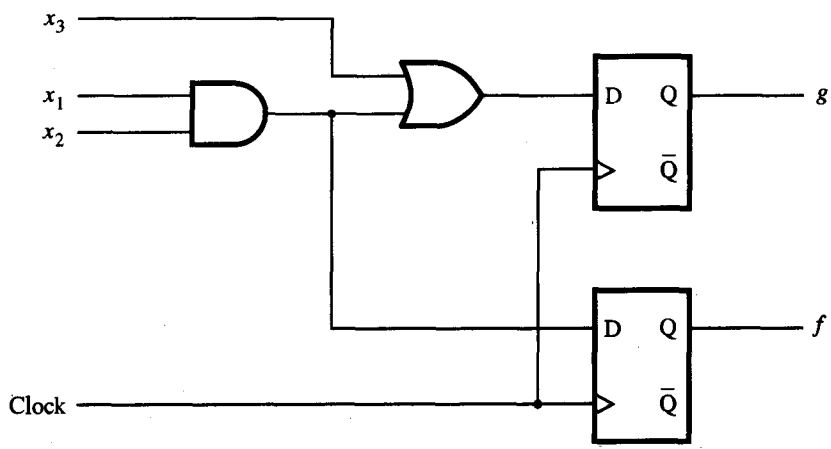


Figure 7.42 Circuit for Example 7.5.

```
module example7_6 (x1, x2, x3, Clock, f, g);
   input x1, x2, x3, Clock;
   output f, g;
   reg f, g;

   always @(posedge Clock)
   begin
      f <= x1 & x2;
      g <= f | x3;
   end

endmodule
```

Figure 7.43 Code for Example 7.6.

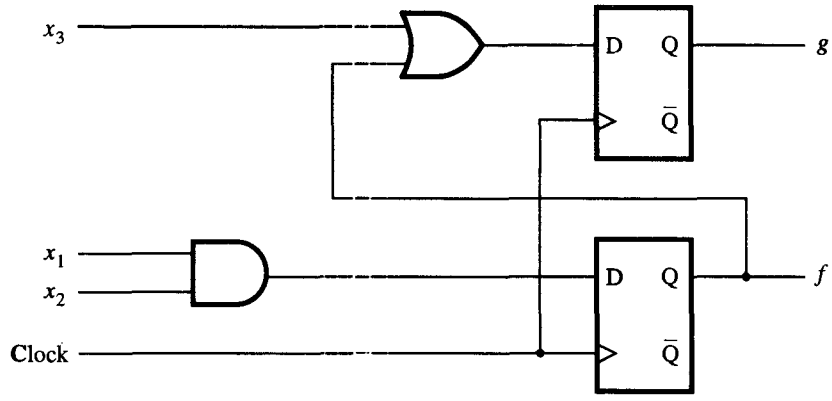


Figure 7.44 Circuit for Example 7.6.

Reversing the statement order would make no difference for the code in Figure 7.43, in which the non-blocking assignment is used.

The use of blocking assignments for sequential circuits can easily lead to wrong results, as demonstrated in Figure 7.38. The dependence on ordering of blocking assignments is dangerous, as shown in the previous example. It is better to use non-blocking assignments to describe sequential circuits.

7.12.4 NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

A natural question at this point is whether non-blocking assignments can be used for combinational circuits. The answer is that they can be used in most situations, but when subsequent assignments in an **always** block depend on the results of previous assignments, the non-blocking assignments can generate nonsensical circuits. As an example, assume that we have a three-bit vector $A = a_2a_1a_0$, and we wish to generate a combinational function f that is equal to 1 when there are two adjacent bits in A that have the value 1. One way to specify this function with blocking assignments is

```
always @(A)
begin
   f = A[1] & A[0];
   f = f | (A[2] & A[1]);
end
```

These statements produce the desired logic function, which is $f = a_1a_0 + a_2a_1$. Consider now changing the code to use the non-blocking assignments

```
f <= A[1] & A[0];
f <= f | (A[2] & A[1]);
```

There are two key aspects of the Verilog semantics relevant to this code:

1. The results of non-blocking assignments are visible only after all of the statements in the **always** block have been evaluated.
2. When there are multiple assignments to the same variable inside an **always** block, the result of the last assignment is maintained.

In this example, f has an unspecified initial value when we enter the **always** block. The first statement assigns $f = a_1 a_0$, but this result is not visible to the second statement. It still sees the original unspecified value of f. The second assignment overrides (deletes!) the first assignment and produces the logic function $f = f + a_2 a_1$. This expression does not correspond to a combinational circuit, because it represents an AND-OR circuit in which the OR-gate is fed back to itself. It is best to use blocking assignments when describing combinational circuits, so as to avoid accidentally creating a sequential circuit.

7.12.5 FLIP-FLOPS WITH CLEAR CAPABILITY

By using a particular sensitivity list and a specific style of **if-else** statement, it is possible to include clear (or preset) signals on flip-flops.

Example 7.7

ASYNCHRONOUS CLEAR Figure 7.45 gives a module that defines a D flip-flop with an asynchronous active-low reset (clear) input. When *Resetn*, the reset input, is equal to 0, the flip-flop's Q output is set to 0. Note that the sensitivity list specifies the negative edge of *Resetn* as an event trigger along with the positive edge of the clock. We cannot omit the keyword **negedge** because the sensitivity list cannot have both edge-triggered and level-sensitive signals.

```
module flipflop(D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;
   reg Q;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure 7.45 D flip-flop with asynchronous reset.

```
module flipflop(D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;
   reg Q;

   always @(posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure 7.46 D flip-flop with synchronous reset.

Example 7.8 **SYNCHRONOUS CLEAR** Figure 7.46 shows how a D flip-flop with a synchronous reset input can be described. In this case the reset signal is acted upon only when a positive clock edge arrives. This code generates the circuit in Figure 7.15, which has an AND gate connected to the flip-flop's D input.

7.13 USING REGISTERS AND COUNTERS WITH CAD TOOLS

In this section we show how registers and counters can be included in circuits designed with the aid of CAD tools. Examples are given using both schematic capture and Verilog code.

7.13.1 INCLUDING REGISTERS AND COUNTERS IN SCHEMATICS

In section 5.5.1 we explained that a CAD system usually includes libraries of prebuilt subcircuits. We introduced the library of parameterized modules (LPM) and used the adder/subtractor module, *lpm_add_sub*, as an example. The LPM includes modules that constitute flip-flops, registers, counters, and many other useful circuits. Figure 7.47 shows a symbol that represents the *lpm_ff* module. This module is a register with one or more positive-edge-triggered flip-flops that can be of either D or T type. The module has parameters that allow the number of flip-flops and flip-flop type to be chosen. In this case we chose to have four D flip-flops. The tutorial in Appendix D explains how the configuration of the module is done.

The D inputs to the four flip-flops, called *data* on the graphical symbol, are connected to the four-bit input signal *Data*[3..0]. The module's asynchronous active-high reset (clear) input, *aclr*, is shown in the schematic. The flip-flop outputs, *q*, are attached to the output symbol labeled Q[3..0].

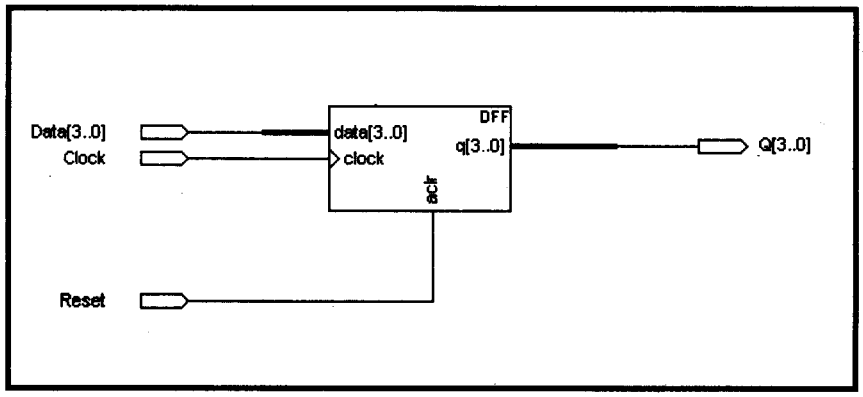


Figure 7.47 The *lpm_ff* parameterized flip-flop module.

In section 7.3 we said that a useful application of D flip-flops is to hold the results of an arithmetic computation, such as the output from an adder circuit. An example is given in Figure 7.48, which uses two LPM modules, *lpm_add_sub* and *lpm_ff*. The *lpm_add_sub* module was described in section 5.5.1. Its parameters, which are not shown in Figure 7.48, are set to configure the module as a four-bit adder circuit. The adder's four-bit data input *dataa* is driven by the *Data*[3..0] input signal. The sum bits, *result*, are connected to the *data* inputs of the *lpm_ff*, which is configured as a four-bit D register with asynchronous clear. The register generates the output of the circuit, Q[3..0], which appears on the left side of the schematic. This signal is fed back to the *datab* input of the adder. The sum bits

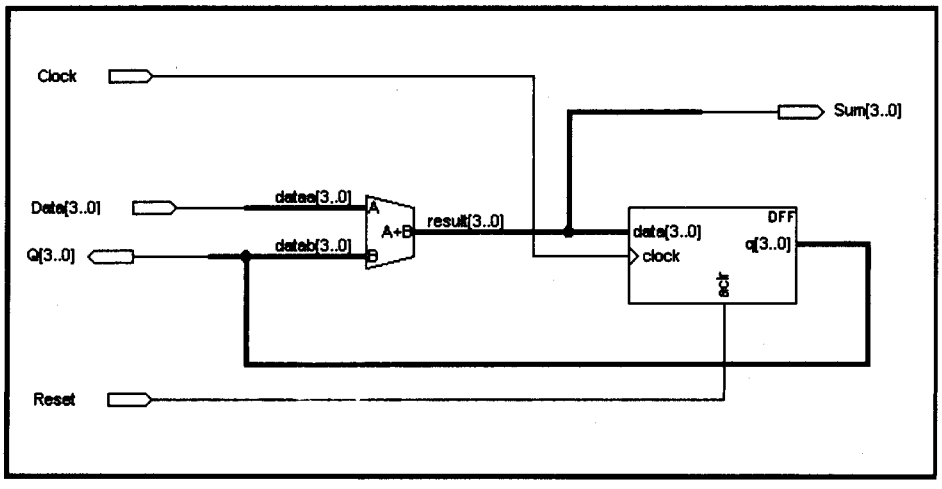


Figure 7.48 An adder with registered feedback.

from the adder are also provided as an output of the circuit, *Sum*[3..0], for ease of reference in the discussion that follows. If the register is first cleared to 0000, then the circuit can be used to add the binary numbers on the *Data*[3..0] input to a sum that is being accumulated in the register, if a new number is applied to the input during each clock cycle. A circuit that performs this function is referred to as an *accumulator* circuit.

We synthesized a circuit from the schematic and implemented the four-bit adder using the carry-lookahead structure. A timing simulation for the circuit appears in Figure 7.49. After resetting the circuit, the *Data* input is set to 0001. The adder produces the sum 0000 + 0001 = 0001, which is then clocked into the register at the 60 ns point in time. After the t_{co} delay, Q[3..0] becomes 0001, and this causes the adder to produce the new sum 0001 + 0001 = 0010. The time needed to generate the new sum is determined by the speed of the adder circuit, which produces the sum after 12.5 ns in this case. The new sum does not appear at the Q output until after the next positive clock edge, at 100 ns. The adder then produces 0011 as the next sum. When *Sum* changes from 0010 to 0011, some oscillations appear in the timing diagram, caused by the propagation of carry signals through the adder circuit. These oscillations are not seen at the Q output, because *Sum* is stable by the time the next positive clock edge occurs. Moving forward to the 180 ns point in time, *Sum* = 0100, and this value is clocked into the register. The adder produces the new sum 0101. Then at 200 ns *Data* is changed to 0010, which causes the sum to change to 0100 + 0010 = 0110. At the next positive clock edge, Q is set to 0110; the value *Sum* = 0101 that was present temporarily in the circuit is not observed at the Q output. The circuit continues to add 0010 to the Q output at each successive positive clock edge.

Having simulated the behavior of the circuit, we should consider whether or not we can conclude with some certainty that the circuit works properly. Ideally, it is prudent to test all possible combinations of a circuit's inputs before declaring that it works as desired. However, in practice, such testing is often not feasible because of the number of input combinations that exist. For the circuit in Figure 7.48, we could verify that a correct sum is produced by the adder, and we could also check that each of the four flip-flops in the register properly stores either 0 or 1. We will discuss issues associated with the testing of circuits in Chapter 11.

For the circuit in Figure 7.48 to work properly, the following timing constraints must be met. When the register is clocked by a positive clock edge, a change of signal value

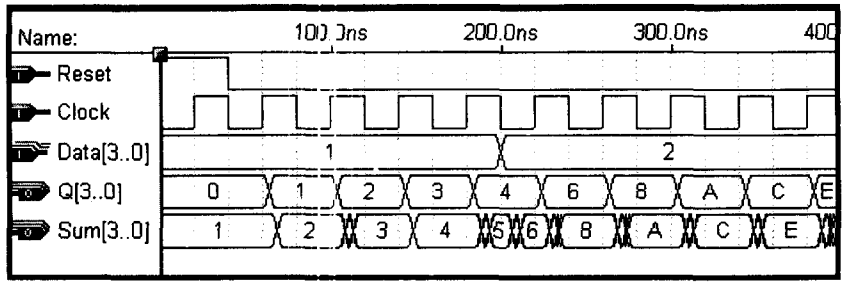


Figure 7.49 Timing simulation of the circuit from Figure 7.48.

at the register's output must propagate through the feedback path to the *datab* input of the adder. The adder then produces a new sum, which must propagate to the *data* input of the register. For the chip used to implement the circuit, the total delay incurred is 14 ns. The delay can be broken down as follows: It takes 2 ns from when the register is clocked until a change in its output reaches the *datab* input of the adder. The adder produces a new sum in 8 ns, and it takes 4 ns for the sum to propagate to the register's *data* input. In Figure 7.49 the clock period is 40 ns. Hence, after the new sum arrives at the *data* input of the register, there remain $40 - 14 = 26$ ns until the next positive clock edge occurs. The *data* input must be stable for the amount of the setup time, $t_{su} = 3$ ns, before the clock edge. Hence we have $26 - 3 = 23$ ns to spare. The clock period can be decreased by as much as 23 ns and the circuit will still work. But if the clock period is less than $40 - 23 = 17$ ns, then the circuit will not function properly. Of course, if a different chip were used to implement the circuit, then different timing results would be produced. CAD systems provide tools that can automatically determine the minimum allowable clock period for which a circuit will work correctly. The tutorial in Appendix D shows how this is done using the tools that accompany the book.

7.13.2 USING LIBRARY MODULES IN VERILOG CODE

The predefined subcircuits in a library of modules such as the LPM library can be instantiated in Verilog code. Figure 7.50 instantiates the *lpm_shiftreg* module, which is an *n*-bit shift register. The module's parameters are set using **defparam** statements. The number of flip-flops in the shift register is set to 4 using the parameter lpm_width = 4. The module can be configured to shift either left or right. The parameter lpm_direction = "RIGHT" sets the shift direction to be from the left to the right. The code uses the module's asynchronous active-high clear input, *aclr*, and the active-high parallel-load input, *load*, which allows the shift register to be loaded with the parallel data on the module's *data* input. When shifting takes place, the value on the *shiftin* input is shifted into the left-most flip-flop and the bit shifted out appears on the right-most bit of the *q* parallel output. The code uses *named*

```
module shift (Clock, Reset, w, Load, R, Q);
   input Clock, Reset, w, Load;
   input [3:0] R;
   output [3:0] Q ;

   lpm_shiftreg shift_right (.data(R), .aclr(Reset), .clock(Clock),
      .load(Load), .shiftin(w), .q(Q)) ;
      defparam shift_right.lpm_width = 4;
      defparam shift_right.lpm_direction = "RIGHT";

endmodule
```

Figure 7.50 Instantiation of the *lpm_shiftreg* module.

ports to connect the input and output signals of the *shift* module to the ports of the module. For example, the *R* input signal is connected to the module's *data* port. This is specified by writing .data(R) in the instantiation statement. Similarly, .aclr(Reset) specifies that the *Reset* input signal is connected to the *aclr* port on the module, and so on. When translated into a circuit, the *lpm_shiftreg* has the structure shown in Figure 7.19.

Predefined modules also exist for the various types of counters, which are commonly needed in logic circuits. An example is the *lpm_counter* module, which is a variable-width counter with parallel-load inputs.

7.13.3 USING VERILOG CONSTRUCTS FOR REGISTERS AND COUNTERS

Rather than instantiating predefined subcircuits for registers, shift registers, counters, and the like, the circuits can be described in Verilog code. Figure 7.45 gives code for a D flip-flop. One way to describe an *n*-bit register is to write hierarchical code that includes *n* instances of the D flip-flop subcircuit. A simpler approach is to use the same code as in Figure 7.45 and define the *D* input and Q output as multibit signals.

Example 7.9 **AN N-BIT REGISTER** Since registers of different sizes are often needed in logic circuits, it is advantageous to define a register module for which the number of flip-flops can be easily changed. The code for an *n*-bit register is given in Figure 7.51. The parameter *n* specifies the number of flip-flops in the register. By changing this parameter, the code can represent a register of any size.

```
module regn (D, Clock, Resetn, Q);
   parameter n = 16;
   input [n-1:0] D;
   input Clock, Resetn;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure 7.51 Code for an *n*-bit register with asynchronous clear.

A FOUR-BIT SHIFT REGISTER Assume that we wish to write Verilog code that represents the four-bit parallel-access shift register in Figure 7.19. One approach is to write hierarchical code that uses four subcircuits. Each subcircuit consists of a D flip-flop with a 2-to-1 multiplexer connected to the *D* input. Figure 7.52 defines the module named *muxdff*, which represents this subcircuit. The two data inputs are named D_0 and D_1, and they are selected using the *Sel* input. The **if-else** statement specifies that on the positive clock edge if *Sel* = 0, then Q is assigned the value of D_0; otherwise, Q is assigned the value of D_1.

Example 7.10

Figure 7.53 defines the four-bit shift register. The module *Stage3* instantiates the left-most flip-flop, which has the output Q_3, and the module *Stage0* instantiates the right-most flip-flop, Q_0. When $L = 1$, the register is loaded in parallel from the *R* input; and when $L = 0$, shifting takes place in the left to right direction. Serial data is shifted into the most-significant bit, Q_3, from the *w* input.

```
module muxdff (D0, D1, Sel, Clock, Q);
   input D0, D1, Sel, Clock;
   output Q;
   reg Q;

   always @(posedge Clock)
      if (!Sel)
         Q <= D0;
      else
         Q <= D1;

endmodule
```

Figure 7.52 Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output [3:0] Q;
   wire [3:0] Q;

   muxdff Stage3 (w, R[3], L, Clock, Q[3]);
   muxdff Stage2 (Q[3], R[2], L, Clock, Q[2]);
   muxdff Stage1 (Q[2], R[1], L, Clock, Q[1]);
   muxdff Stage0 (Q[1], R[0], L, Clock, Q[0]);

endmodule
```

Figure 7.53 Hierarchical code for a four-bit shift register.

Example 7.11 **ALTERNATIVE CODE FOR A FOUR-BIT SHIFT REGISTER** A different style of code for the four-bit shift register is given in Figure 7.54. Instead of using subcircuits, the shift register is defined using the approach presented in Example 7.4. All actions take place at the positive edge of the clock. If $L = 1$, the register is loaded in parallel with the four bits of input R. If $L = 0$, the contents of the register are shifted to the right and the value of the input w is loaded into the most-significant bit Q_3.

Example 7.12 **AN N-BIT SHIFT REGISTER** Figure 7.55 shows the code that can be used to represent shift registers of any size. The parameter n, which has the default value 16 in the figure, sets the number of flip-flops. The code is identical to that in Figure 7.54 with two exceptions. First, R and Q are defined in terms of n. Second, the **else** clause that describes the shifting operation is generalized to work for any number of flip-flops by using a **for** loop.

Example 7.13 **UP-COUNTER** Figure 7.56 represents a four-bit up-counter with a reset input, *Resetn*, and an enable input, E. The outputs of the flip-flops in the counter are represented by the vector named Q. The **if** statement specifies an asynchronous reset of the counter if *Resetn* = 0. The **else if** clause specifies that if $E = 1$ the count is incremented on the positive clock edge.

```
module shift4 (R, L, w, Clock, Q);
   input [3:0] R;
   input L, w, Clock;
   output [3:0] Q;
   reg [3:0] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] <= Q[1];
         Q[1] <= Q[2];
         Q[2] <= Q[3];
         Q[3] <= w;
      end

endmodule
```

Figure 7.54 Alternative code for a four-bit shift register.

```
module shiftn (R, L, w, Clock, Q);
   parameter n = 16;
   input [n-1:0] R;
   input L, w, Clock;
   output [n-1:0] Q;
   reg [n-1:0] Q;
   integer k;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         for (k = 0; k < n-1; k = k+1)
            Q[k] <= Q[k+1];
         Q[n-1] <= w;
      end

endmodule
```

Figure 7.55 An n-bit shift register.

```
module upcount (Resetn, Clock, E, Q);
   input Resetn, Clock, E;
   output [3:0] Q;
   reg [3:0] Q;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else if (E)
         Q <= Q + 1;

endmodule
```

Figure 7.56 Code for a four-bit up-counter.

Example 7.14

UP-COUNTER WITH PARALLEL LOAD The code in Figure 7.57 defines an up-counter that has a parallel-load input in addition to a reset input. The parallel data is provided as the input vector R. The first **if** statement provides the same asynchronous reset as in Figure 7.56. The **else if** clause specifies that if $L = 1$ the flip-flops in the counter are loaded in

```
module upcount (R, Resetn, Clock, E, L, Q);
  input [3:0] R;
  input Resetn, Clock, E, L;
  output [3:0] Q;
  reg [3:0] Q;

  always @(negedge Resetn or posedge Clock)
    if (!Resetn)
      Q <= 0;
    else if (L)
      Q <= R;
    else if (E)
      Q <= Q + 1;

endmodule
```

Figure 7.57 A four-bit up-counter with a parallel load.

parallel from the R inputs on the positive clock edge. If $L = 0$, the count is incremented, under control of the enable input E.

Example 7.15 **DOWN-COUNTER WITH PARALLEL LOAD** Figure 7.58 shows the code for a down-counter named *downcount*. A down-counter is normally used by loading it with some starting count and then decrementing its contents. The starting count is represented in the code by the vector R. On the positive clock edge, if $L = 1$ the counter is loaded with the input R, and if $L = 0$ the count is decremented. The counter also includes an enable input, E. Setting

```
module downcount (R, Clock, E, L, Q);
  parameter n = 8;
  input [n-1:0] R;
  input Clock, L, E;
  output [n-1:0] Q;
  reg [n-1:0] Q;

  always @(posedge Clock)
    if (L)
      Q <= R;
    else if (E)
      Q <= Q - 1;

endmodule
```

Figure 7.58 A down-counter with a parallel load.

```
module updowncount (R, Clock, L, E, up_down, Q);
   parameter n = 8;
   input [n−1:0] R;
   input Clock, L, E, up_down;
   output [n−1:0] Q;
   reg [n−1:0] Q;
   integer direction;

   always @(posedge Clock)
   begin
      if (up_down)
         direction = 1;
      else
         direction = −1;
      if (L)
         Q <= R;
      else if (E)
         Q <= Q + direction;
   end

endmodule
```

Figure 7.59 Code for an up/down counter.

$E = 0$ prevents the contents of the flip-flops from changing when an active clock edge occurs.

Example 7.16

UP/DOWN COUNTER Verilog code for an up/down counter is given in Figure 7.59. This module combines the capabilities of the counters defined in Figures 7.57 and 7.58. It includes a control signal *up_down* that governs the direction of counting. It also includes an **integer** variable named *direction*, which is equal to 1 for up-count and equal to −1 for down-count.

7.14 DESIGN EXAMPLES

This section presents examples of digital systems that make use of some of the building blocks described in this chapter and in Chapter 6.

7.14.1 BUS STRUCTURE

Digital systems often contain a set of registers used to store data. Figure 7.60 gives an example of a system that has k n-bit registers, $R1$ to Rk. Each register is connected to a

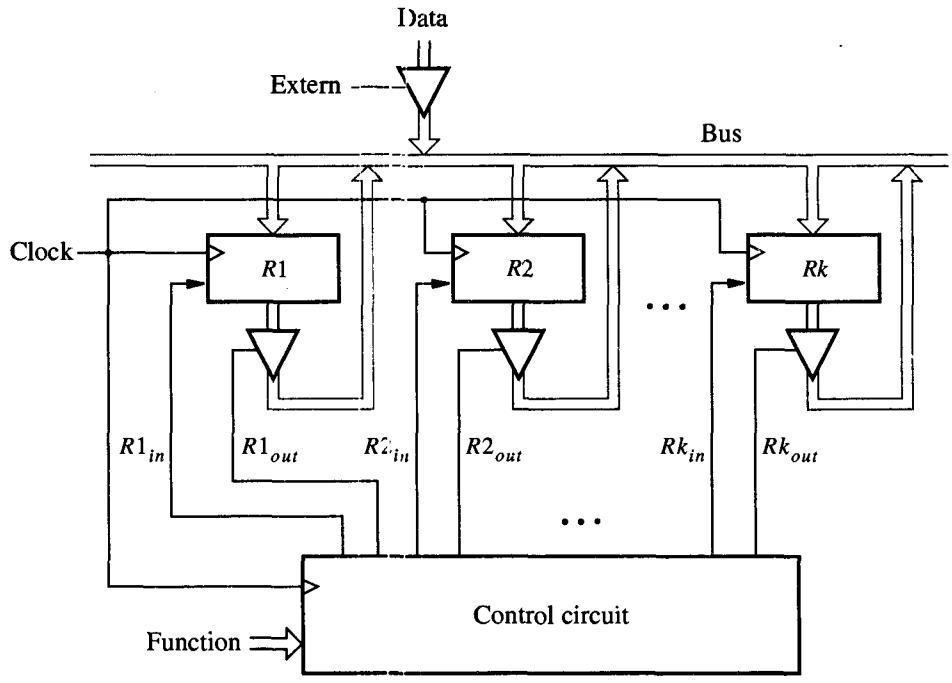


Figure 7.60 A digital system with k registers.

common set of n wires, which are used to transfer data into and out of the registers. This common set of wires is usually called a *bus*. In addition to registers, in a real system other types of circuit blocks would be connected to the bus. The figure shows how n bits of data can be placed on the bus from another circuit block, using the control input *Extern*. The data stored in any of the registers can be transferred via the bus to a different register or to another circuit block that is connected to the bus.

It is essential to ensure that only one circuit block attempts to place data onto the bus wires at any given time. In Figure 7.60 each register is connected to the bus through an n-bit tri-state buffer. A control circuit is used to ensure that only one of the tri-state buffer enable inputs, $R1_{out}, \ldots, Rk_{out}$, is asserted at a given time. The control circuit also produces the signals $R1_{in}, \ldots, Rk_{in}$, which control when data is loaded into each register. In general, the control circuit could perform a number of functions, such as transferring the data stored in one register into another register and the like. Figure 7.60 shows an input signal named *Function* that instructs the control circuit to perform a particular task. The control circuit is synchronized by a clock input, which is the same clock signal that controls the k registers.

Figure 7.61 provides a more detailed view of how the registers from Figure 7.60 can be connected to a bus. To keep the picture simple, 2 two-bit registers are shown, but the same scheme can be used for larger registers. For register $R1$, two tri-state buffers enabled by $R1_{out}$ are used to connect each flip-flop output to a wire in the bus. The D input on

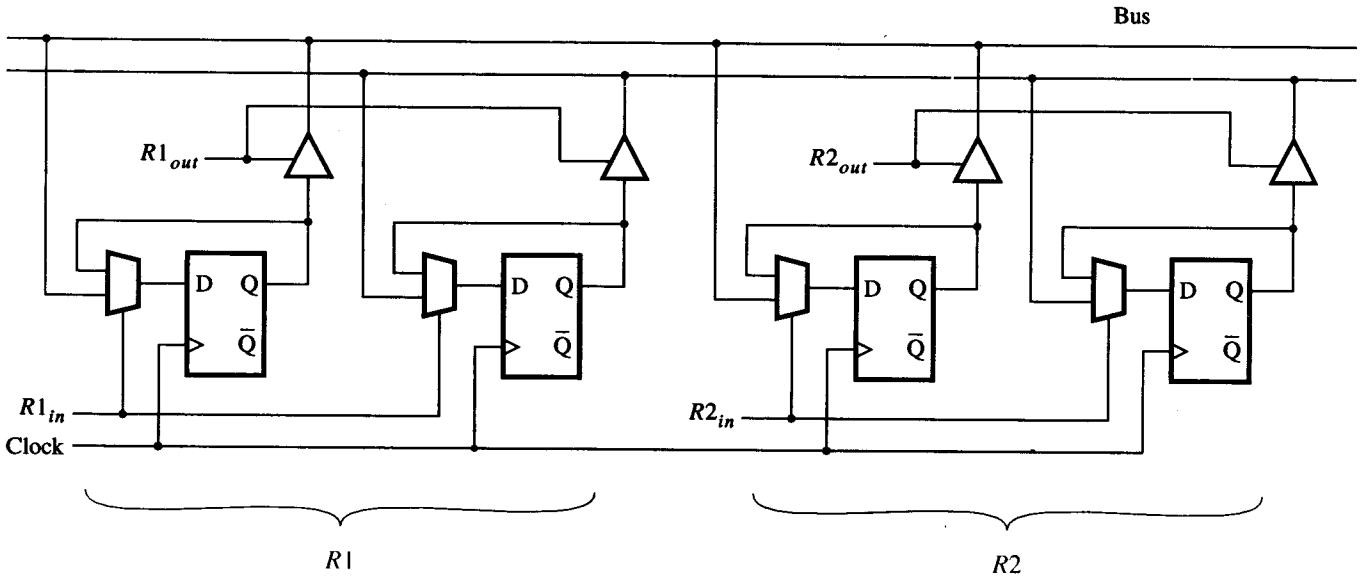


Figure 7.61 Details for connecting registers to a bus.

each flip-flop is connected to a 2-to-1 multiplexer, whose select input is controlled by $R1_{in}$. If $R1_{in} = 0$, the flip-flops are loaded from their Q outputs; hence the stored data does not change. But if $R1_{in} = 1$, data is loaded into the flip-flops from the bus. Instead of using multiplexers on the flip-flop inputs, one could attempt to connect the D inputs on the flip-flops directly to the bus. Then it is necessary to control the clock inputs on all flip-flops to ensure that they are clocked only when new data should be loaded into the register. This approach is not good because it may happen that different flip-flops will be clocked at slightly different times, leading to a problem known as *clock skew*. A detailed discussion of the issues related to the clocking of flip-flops is provided in section 10.3.

The system in Figure 7.60 can be used in many different ways, depending on the design of the control circuit and on how many registers and other circuit blocks are connected to the bus. As a simple example, consider a system that has three registers, $R1$, $R2$, and $R3$. Each register is connected to the bus as indicated in Figure 7.61. We will design a control circuit that performs a single function—it swaps the contents of registers $R1$ and $R2$, using $R3$ for temporary storage.

The required swapping is done in three steps, each needing one clock cycle. In the first step the contents of $R2$ are transferred into $R3$. Then the contents of $R1$ are transferred into $R2$. Finally, the contents of $R3$, which are the original contents of $R2$, are transferred into $R1$. Note that we say that the contents of one register, R_i, are "transferred" into another register, R_j. This jargon is commonly used to indicate that the new contents of R_j will be a copy of the contents of R_i. The contents of R_i are not changed as a result of the transfer. Therefore, it would be more precise to say that the contents of R_i are "copied" into R_j.

Using a Shift Register for Control

There are many ways to design a suitable control circuit for the swap operation. One possibility is to use the left-to-right shift register shown in Figure 7.62. Assume that the reset input is used to clear the flip-flops to 0. Hence the control signals $R1_{in}$, $R1_{out}$, and so on are not asserted, because the shift register outputs have the value 0. The serial input w normally has the value 0. We assume that changes in the value of w are synchronized to occur shortly after the active clock edge. This assumption is reasonable because w would normally be generated as the output of some circuit that is controlled by the same clock signal. When the desired swap should be performed, w is set to 1 for one clock cycle, and then w returns to 0. After the next active clock edge, the output of the left-most flip-flop

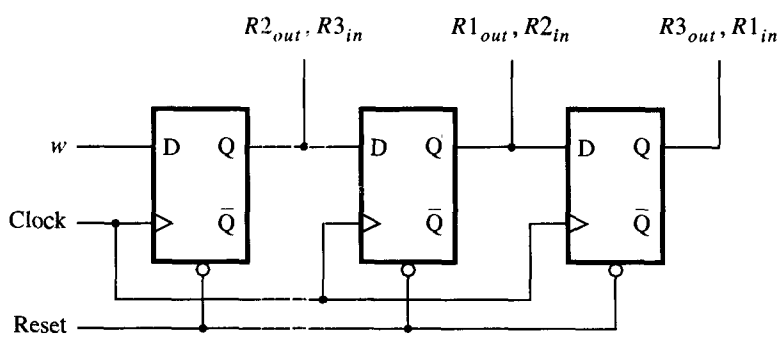


Figure 7.62 A shift-register control circuit.

becomes equal to 1, which asserts both $R2_{out}$ and $R3_{in}$. The contents of register $R2$ are placed onto the bus wires and are loaded into register $R3$ on the next active clock edge. This clock edge also shifts the contents of the shift register, resulting in $R1_{out} = R2_{in} = 1$. Note that since w is now 0, the first flip-flop is cleared, causing $R2_{out} = R3_{in} = 0$. The contents of $R1$ are now on the bus and are loaded into $R2$ on the next clock edge. After this clock edge the shift register contains 001 and thus asserts $R3_{out}$ and $R1_{in}$. The contents of $R3$ are now on the bus and are loaded into $R1$ on the next clock edge.

Using the control circuit in Figure 7.62, when w changes to 1 the swap operation does not begin until after the next active clock edge. We can modify the control circuit so that it starts the swap operation in the same clock cycle in which w changes to 1. One possible approach is illustrated in Figure 7.63. The reset signal is used to set the shift-register contents to 100, by presetting the left-most flip-flop to 1 and clearing the other two flip-flops. As long as $w = 0$, the output control signals are not asserted. When w changes to 1, the signals $R2_{out}$ and $R3_{in}$ are immediately asserted and the contents of $R2$ are placed onto the bus. The next active clock edge loads this data into $R3$ and also shifts the shift register contents to 010. Since the signal $R1_{out}$ is now asserted, the contents of $R1$ appear on the bus. The next clock edge loads this data into $R2$ and changes the shift register contents to 001. The contents of $R3$ are now on the bus; this data is loaded into $R1$ at the next clock edge, which also changes the shift register contents to 100. We assume that w had the value 1 for only one clock cycle; hence the output control signals are not asserted at this point. It may not be obvious to the reader how to design a circuit such as the one in Figure 7.63, because we have presented the design in an ad hoc fashion. In section 8.3 we will show how this circuit can be designed using a more formal approach.

The circuit in Figure 7.63 assumes that a preset input is available on the left-most flip-flop. If the flip-flop has only a clear input, then we can use the equivalent circuit shown in Figure 7.64. In this circuit we use the $\overline{Q}$ output of the left-most flip-flop and also complement the input to this flip-flop by using a NOR gate instead of an OR gate.

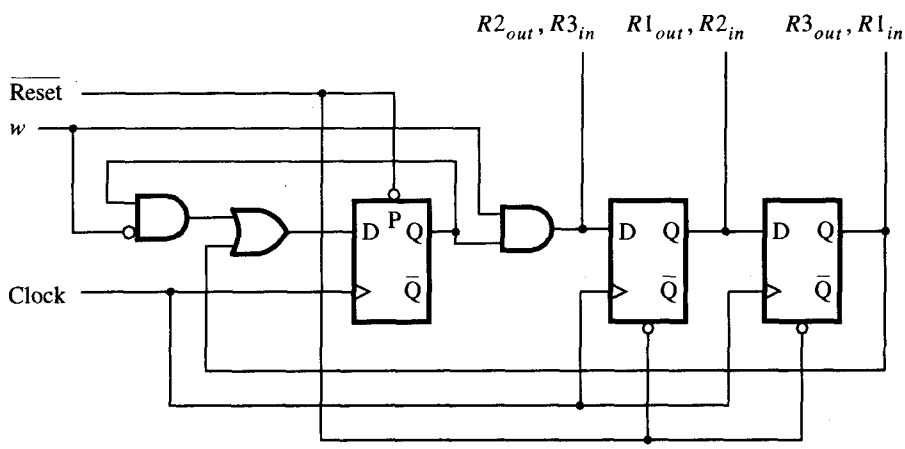


Figure 7.63 A modified control circuit.

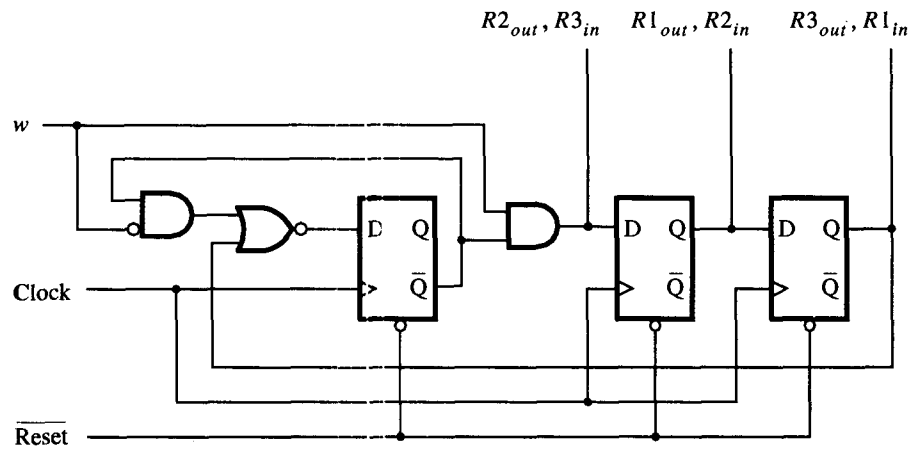


Figure 7.64 A modified version of the circuit in Figure 7.63.

Using Multiplexers to Implement a Bus

In Figure 7.60 we used tri-state buffers to control access to the bus. An alternative approach is to use multiplexers, as depicted in Figure 7.65. The outputs of each register are connected to a multiplexer. This multiplexer's output is connected to the inputs of the registers, thus realizing the bus. The multiplexer select inputs determine which register's contents appear on the bus. Although the figure shows just one multiplexer symbol, we actually need one multiplexer for each bit in the registers. For example, assume that there are 4 eight-bit registers, $R1$ to $R4$, plus the externally-supplied eight-bit *Data*. To

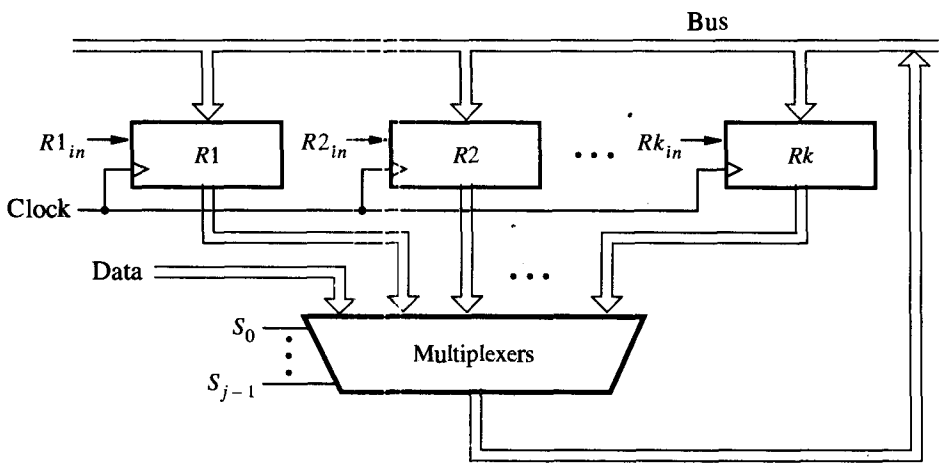


Figure 7.65 Using multiplexers to implement a bus.

interconnect them, we need eight 5-to-1 multiplexers. In Figure 7.62 we used a shift register to implement the control circuit. A similar approach can be used with multiplexers. The signals that control when data is loaded into a register, like $R1_{in}$, can still be connected directly to the shift-register outputs. However, instead of using control signals like $R1_{out}$ to place the contents of a register onto the bus, we have to generate the select inputs for the multiplexers. One way to do so is to connect the shift-register outputs to an encoder circuit that produces the select inputs for the multiplexer. We discussed encoder circuits in section 6.3.

The tri-state buffer and multiplexer approaches for implementing a bus are both equally valid. However, some types of chips, such as most PLDs, do not contain a sufficient number of tri-state buffers to realize even moderately large buses. In such chips the multiplexer-based approach is the only practical alternative. In practice, circuits are designed with CAD tools. If the designer describes the circuit using tri-state buffers, but there are not enough such buffers in the target device, then the CAD tools automatically produce an equivalent circuit that uses multiplexers.

Verilog Code

This section presents Verilog code for our circuit example that swaps the contents of two registers. We first give the code for the style of circuit in Figure 7.60 that uses tri-state buffers to implement the bus and then give the code for the style of circuit in Figure 7.65 that uses multiplexers. The code is written in a hierarchical fashion, using subcircuits for the registers, tri-state buffers, and the shift register. Figure 7.66 gives the code for an n-bit register of the type in Figure 7.61. The number of bits in the register is set by the parameter n, which has the default value of 8. The register is specified such that if the input $Rin = 1$, then the flip-flops are loaded from the n-bit input R. Otherwise, the flip-flops retain their presently stored values.

Figure 7.67 gives the code for a subcircuit that represents n tri-state buffers, each enabled by the input E. The inputs to the buffers are the n-bit signal Y, and the outputs are the n-bit signal F. The conditional assignment statement specifies that the output of

```
module regn (R, Rin, Clock, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Rin, Clock;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock)
      if (Rin)
         Q <= R;

endmodule
```

Figure 7.66 Code for an n-bit register of the type in Figure 7.61.

```
module trin (Y, E, F);
   parameter n = 8;
   input [n-1:0] Y;
   input E;
   output [n-1:0] F;
   wire [n-1:0] F;

   assign F = E ? Y : 'bz;

endmodule
```

Figure 7.67 Code for an n-bit tri-state module.

each buffer is set to $F = Y$ if $E = 1$, otherwise, the output is set to the high impedance value z. The conditional assignment statement uses an unsized number to define the high impedance case. The Verilog compiler will make the size of this number the same as the size of vector Y, namely n. We cannot define the number as n'bz because the size of a sized number cannot be given as a parameter.

Figure 7.68 defines a shift register that can be used to implement the control circuit in Figure 7.62. The number of flip-flops is set by the generic parameter m, which has the default value of 4. The shift register has an active-low asynchronous reset input. The shift operation is defined with a **for** loop in the style used in Example 7.12.

```
module shiftr (Resetn, w, Clock, Q);
   parameter m = 4;
   input Resetn, w, Clock;
   output [1:m] Q;
   reg [1:m] Q;
   integer k;

   always @(negedge Resetn or posedge Clock)
      if (!Resetn)
         Q <= 0;
      else
      begin
         for (k = m; k > 1 ; k = k-1)
            Q[k] <= Q[k-1];
         Q[1] <= w;
      end

endmodule
```

Figure 7.68 Code for the shift register in Figure 7.62.

The code in Figure 7.69 represents a digital system like the one in Figure 7.60, with 3 eight-bit registers, *R1*, *R2*, and *R3*. The circuit in Figure 7.60 includes tri-state buffers that are used to place *n* bits of externally supplied data on the bus. In Figure 7.69, these buffers are instantiated in the module *tri_ext*. Each of the eight buffers is enabled by the input signal *Extern*, and the data inputs on the buffers are attached to the eight-bit signal *Data*. When *Extern* = 1, the value of *Data* is placed on the bus, which is represented by the signal *BusWires*. The *BusWires* vector represents the circuit's output as well as the internal bus wiring. We declared this vector to be of **tri** type rather than of **wire** type. The keyword **tri** is treated in the same way as the keyword **wire** by the Verilog compiler. The designation **tri** makes it obvious to a reader that the synthesized connections will have tri-state capability.

We assume that a three-bit control signal named *RinExt* exists, which allows the externally supplied data to be loaded from the bus into register *R1*, *R2*, or *R3*. The *RinExt*

```
module swap (Data, Resetn, w, Clock, Extern, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock, Extern;
   input [1:3] RinExt;
   output [7:0] BusWires;
   tri [7:0] BusWires;
   wire [1:3] Rin, Rout, Q;
   wire [7:0] R1, R2, R3;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;

   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   assign Rout[1] = Q[2];
   assign Rout[2] = Q[1];
   assign Rout[3] = Q[3];

   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   trin tri_ext (Data, Extern, BusWires);
   trin tri_1 (R1, Rout[1], BusWires);
   trin tri_2 (R2, Rout[2], BusWires);
   trin tri_3 (R3, Rout[3], BusWires);

endmodule
```

Figure 7.69 A digital system like the one in Figure 7.60.

input is not shown in Figure 7.60, to keep the figure simple, but it would be generated by the same external circuit block that produces *Extern* and *Data*. When *RinExt*[1] = 1, the data on the bus is loaded into register *R*1; when *RinExt*[2] = 1, the data is loaded into *R*2; and when *RinExt*[3] = 1, the data is loaded into *R*3.

In Figure 7.69 the three-bit shift register is instantiated using the *shiftr* module under the instance name *control*. The outputs of the shift register are the three-bit signal Q. The parameter that defines the number of flip-flops in the *shiftr* module, *m*, has the default value of 4. Since we need to instantiate only a three-bit shift register, we have to change the value of parameter *m*. The parameter is set with the statement

```
defparam control.m = 3;
```

The **defparam** statement defines the values of the parameters indicated. The intended module instance is identified using the syntax *instance_name.parameter_name*. In our example, the instance name is *control* and the parameter name is *m*.

The next three statements in Figure 7.69 connect Q to the control signals that determine when data is loaded into each register, which are represented by the three-bit signal *Rin*. The signals *Rin*[1], *Rin*[2], and *Rin*[3] in the code correspond to the signals $R1_{in}$, $R2_{in}$, and $R3_{in}$ in Figure 7.60. As specified in Figure 7.62, the left-most shift-register output, Q[1], controls when data is loaded into register *R*3. Similarly, Q[2] controls register *R*2, and Q[3] controls *R*1. Each bit in *Rin* is ORed with the corresponding bit in *RinExt* so that externally supplied data can be stored in the registers as discussed above. The code also connects the shift-register outputs to the enable inputs, *Rout*, on the tri-state buffers. Figure 7.62 shows that Q[1] is used to put the contents of *R*2 onto the bus; hence *Rout*[2] is assigned the value of Q[1]. Similarly, *Rout*[1] is assigned the value of Q[2], and *Rout*[3] is assigned the value of Q[3]. The remaining statements in the code instantiate the registers and tri-state buffers in the system.

Verilog Code Using Multiplexers

Figure 7.70 shows how the code in Figure 7.69 can be modified to use multiplexers instead of tri-state buffers. Using the circuit structure shown in Figure 7.65, the bus is implemented with eight 4-to-1 multiplexers. Three of the data inputs on each 4-to-1 multiplexer are connected to one bit from registers *R*1, *R*2, and *R*3. The fourth data input is connected to one bit of the *Data* input signal to allow externally supplied data to be written into the registers. When the shift register's contents are 000, the multiplexers select *Data* to be placed on the bus. This data is loaded into the register selected by *RinExt*. It is loaded into *R*1 if *RinExt*[1] = 1, *R*2 if *RinExt*[2] = 1, and *R*3 if *RinExt*[3] = 1.

The *Rout* signal in Figure 7.69, which enables the tri-state buffers connected to the bus, is not needed for the multiplexer implementation. Instead, we have to provide the select inputs on the multiplexers. In Figure 7.70, the shift-register outputs are called Q. These signals generate the *Rin* control signals for the registers in the same way as shown in Figure 7.69. We said in the discussion concerning Figure 7.65 that an encoder is needed between the shift-register outputs and the multiplexer select inputs. A suitable encoder is described in the first **if-else** statement in Figure 7.70. It produces the multiplexer select inputs, which are named *S*. It sets $S = 00$ when the shift register contains 000, $S = 10$ when the shift register contains 100, and so on, as given in the code. The multiplexers are described by

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output [7:0] BusWires;
   reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;
   reg [1:0] S;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.m = 3;
   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];
   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q or Data or R1 or R2 or R3 or S)
   begin
      // Encoder
      if (Q == 3'b000)  S = 2'b00;
      else if (Q == 3'b100)  S = 2'b10;
      else if (Q == 3'b010)  S = 2'b01;
      else  S = 2'b11;

      // Multiplexers
      if (S == 2'b00)  BusWires = Data;
      else if (S == 2'b01)  BusWires = R1;
      else if (S == 2'b10)  BusWires = R2;
      else  BusWires = R3;
   end

endmodule
```

Figure 7.70 Using multiplexers to implement a bus.

the second **if-else** statement, which places the value of *Data* onto the bus (*BusWires*) if $S = 00$, the contents of register $R1$ if $S = 01$, and so on. Using this scheme, when the swap operation is not active, the multiplexers place the bits from the *Data* input on the bus.

As described above, Figure 7.70 uses two **if-else** statements, one to describe an encoder and the other to describe the bus multiplexers. A simpler approach is to write a single **if-else** statement as shown in Figure 7.71. Here, each clause specifies directly which signal should

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
   input [7:0] Data;
   input Resetn, w, Clock;
   input [1:3] RinExt;
   output [7:0] BusWires;
   reg [7:0] BusWires;
   wire [1:3] Rin, Q;
   wire [7:0] R1, R2, R3;

   shiftr control (Resetn, w, Clock, Q);
      defparam control.n = 3;

   assign Rin[1] = RinExt[1] | Q[3];
   assign Rin[2] = RinExt[2] | Q[2];
   assign Rin[3] = RinExt[3] | Q[1];

   regn reg_1 (BusWires, Rin[1], Clock, R1);
   regn reg_2 (BusWires, Rin[2], Clock, R2);
   regn reg_3 (BusWires, Rin[3], Clock, R3);

   always @(Q or Data or R1 or R2 or R3)
   begin
      if (Q == 3'b000)  BusWires = Data;
      else if (Q == 3'b100)  BusWires = R2;
      else if (Q == 3'b010)  BusWires = R1;
      else  BusWires = R3;
   end

endmodule
```

Figure 7.71 A simplified version of the specification in Figure 7.70.

appear on *BusWires* for each pattern of the shift-register outputs. The circuit generated from the code in Figure 7.71 is equivalent to the one generated from the code in Figure 7.70.

Figure 7.72 gives an example of a timing simulation for a circuit synthesized from the code in Figure 7.71. In the first half of the simulation, the circuit is reset, and the contents of registers $R1$ and $R2$ are initialized. The hex value 55 is loaded into $R1$, and the value AA is loaded into $R2$. The clock edge at 275 ns, marked by the vertical reference line in Figure 7.72, loads the value $w = 1$ into the shift register. The contents of $R2$ (AA) then appear on the bus and are loaded into $R3$ by the clock edge at 325 ns. Following this clock edge, the contents of the shift register are 010, and the data stored in $R1$ (55) is on the bus. The clock edge at 375 ns loads this data into $R2$ and changes the shift register to 001. The contents of $R3$ (AA) now appear on the bus and are loaded into $R1$ by the clock edge at 425 ns. The shift register is now in state 000, and the swap is completed.

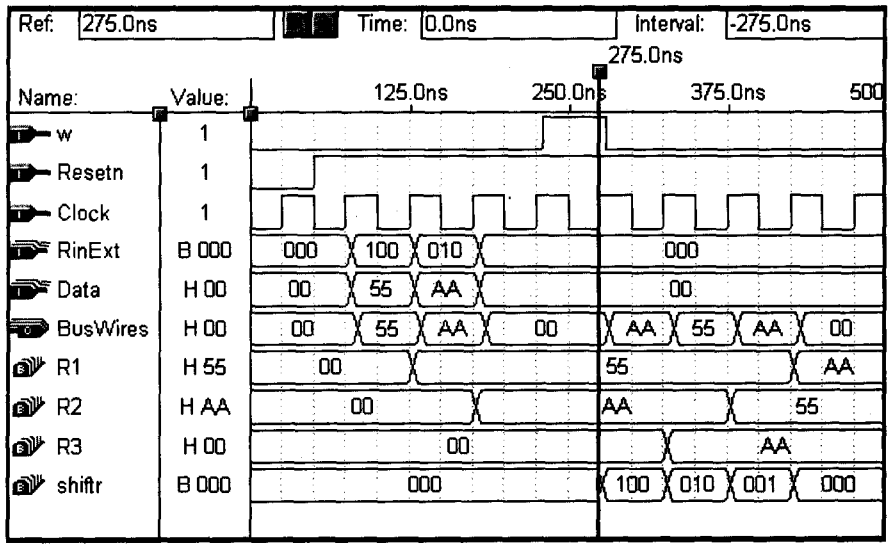


Figure 7.72 Timing simulation for the Verilog code in Figure 7.71.

7.14.2 SIMPLE PROCESSOR

A second example of a digital system like the one in Figure 7.60 is shown in Figure 7.73. It has four n-bit registers, $R0, \ldots, R3$, that are connected to the bus with tri-state buffers. External data can be loaded into the registers from the n-bit *Data* input, which is connected to the bus using tri-state buffers enabled by the *Extern* control signal. The system also includes an adder/subtractor module. One of its data inputs is provided by an n-bit register, A, that is attached to the bus, while the other data input, B, is directly connected to the bus. If the *AddSub* signal has the value 0, the module generates the sum $A + B$; if $AddSub = 1$, the module generates the difference $A - B$. To perform the subtraction, we assume that the adder/subtractor includes the required XOR gates to form the 2's complement of B, as discussed in section 5.3. The register G stores the output produced by the adder/subtractor. The A and G registers are controlled by the signals A_{in}, G_{in}, and G_{out}.

The system in Figure 7.73 can perform various functions, depending on the design of the control circuit. As an example, we will design a control circuit that can perform the four operations listed in Table 7.2. The left column in the table shows the name of an operation and its operands; the right column indicates the function performed in the operation. For the *Load* operation the meaning of $Rx \leftarrow Data$ is that the data on the external *Data* input is transferred across the bus into any register, Rx, where Rx can be $R0$ to $R3$. The *Move* operation copies the data stored in register Ry into register Rx. In the table the square brackets, as in $[Rx]$, refer to the *contents* of a register. Since only a single transfer across the bus is needed, both the *Load* and *Move* operations require only one step (clock cycle) to be completed. The *Add* and *Sub* operations require three steps, as follows: In the first step

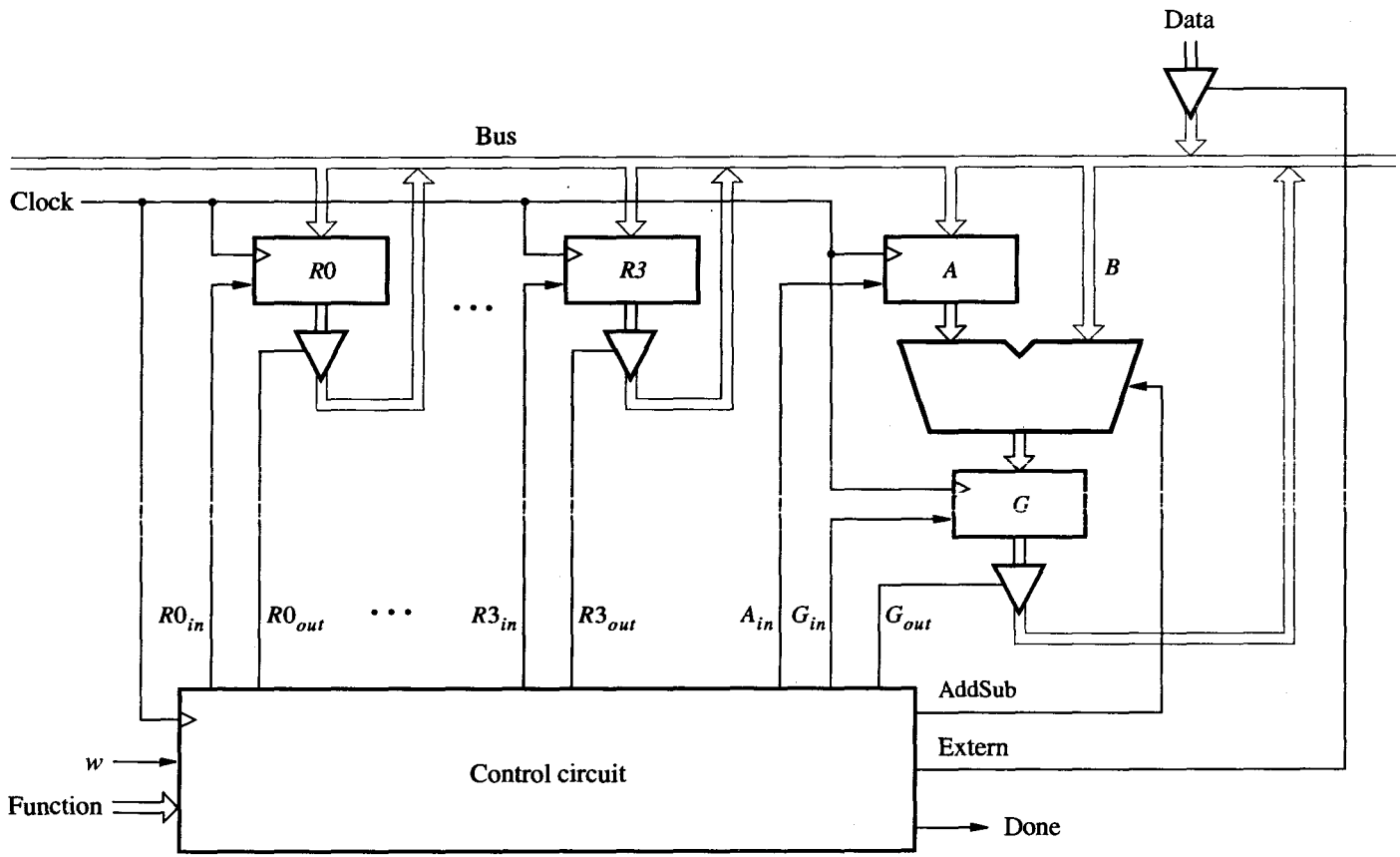


Figure 7.73 A digital system that implements a simple processor.

Table 7.2 Operations performed in the processor.

Operation	Function performed
Load *Rx*, *Data*	$Rx \leftarrow Data$
Move *Rx*, *Ry*	$Rx \leftarrow [Ry]$
Add *Rx*, *Ry*	$Rx \leftarrow [Rx] + [Ry]$
Sub *Rx*, *Ry*	$Rx \leftarrow [Rx] - [Ry]$

the contents of *Rx* are transferred across the bus into register *A*. Then in the next step, the contents of *Ry* are placed onto the bus. The adder/subtractor module performs the required function, and the results are stored in register *G*. Finally, in the third step the contents of *G* are transferred into *Rx*.

A digital system that performs the types of operations listed in Table 7.2 is usually called a *processor*. The specific operation to be performed at any given time is indicated using the control circuit input named *Function*. The operation is initiated by setting the *w* input to 1, and the control circuit asserts the *Done* output when the operation is completed.

In Figure 7.60 we used a shift register to implement the control circuit. It is possible to use a similar design for the system in Figure 7.73. To illustrate a different approach, we will base the design of the control circuit on a counter. This circuit has to generate the required control signals in each step of each operation. Since the longest operations (*Add* and *Sub*) need three steps (clock cycles), a two-bit counter can be used. Figure 7.74 shows a two-bit up-counter connected to a 2-to-4 decoder. Decoders are discussed in section 6.2. The decoder is enabled at all times by setting its enable (*En*) input permanently to the value 1. Each of the decoder outputs represents a step in an operation. When no operation is currently being performed, the count value is 00; hence the T_0 output of the decoder is

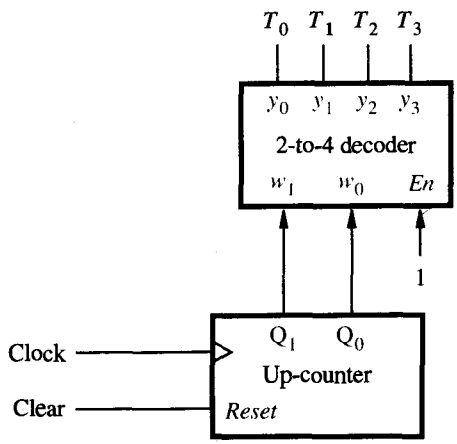


Figure 7.74 A part of the control circuit for the processor.

asserted. In the first step of an operation, the count value is 01, and T_1 is asserted. During the second and third steps of the *Add* and *Sub* operations, T_2 and T_3 are asserted, respectively.

In each of steps T_0 to T_3, various control signal values have to be generated by the control circuit, depending on the operation being performed. Figure 7.75 shows that the operation is specified with six bits, which form the *Function* input. The two left-most bits, $F = f_1 f_0$, are used as a two-bit number that identifies the operation. To represent *Load*, *Move*, *Add*, and *Sub*, we use the codes $f_1 f_0 = 00, 01, 10$, and 11, respectively. The inputs $Rx_1 Rx_0$ are a binary number that identifies the *Rx* operand, while $Ry_1 Ry_0$ identifies the *Ry* operand. The *Function* inputs are stored in a six-bit Function Register when the FR_{in} signal is asserted.

Figure 7.75 also shows three 2-to-4 decoders that are used to decode the information encoded in the *F*, *Rx*, and *Ry* inputs. We will see shortly that these decoders are included as a convenience because their outputs provide simple-looking logic expressions for the various control signals.

The circuits in Figures 7.74 and 7.75 form a part of the control circuit. Using the input *w* and the signals $T_0, \ldots, T_3$, $I_0, \ldots, I_3$, $X_0, \ldots, X_3$, and $Y_0, \ldots, Y_3$, we will show how to derive the rest of the control circuit. It has to generate the outputs *Extern*, *Done*, A_{in}, G_{in}, G_{out}, *AddSub*, $R0_{in}, \ldots, R3_{in}$, and $R0_{out}, \ldots, R3_{out}$. The control circuit also has to generate the *Clear* and FR_{in} signals used in Figures 7.74 and 7.75.

Clear and FR_{in} are defined in the same way for all operations. *Clear* is used to ensure that the count value remains at 00 as long as $w = 0$ and no operation is being executed. Also, it is used to clear the count value to 00 at the end of each operation. Hence an appropriate

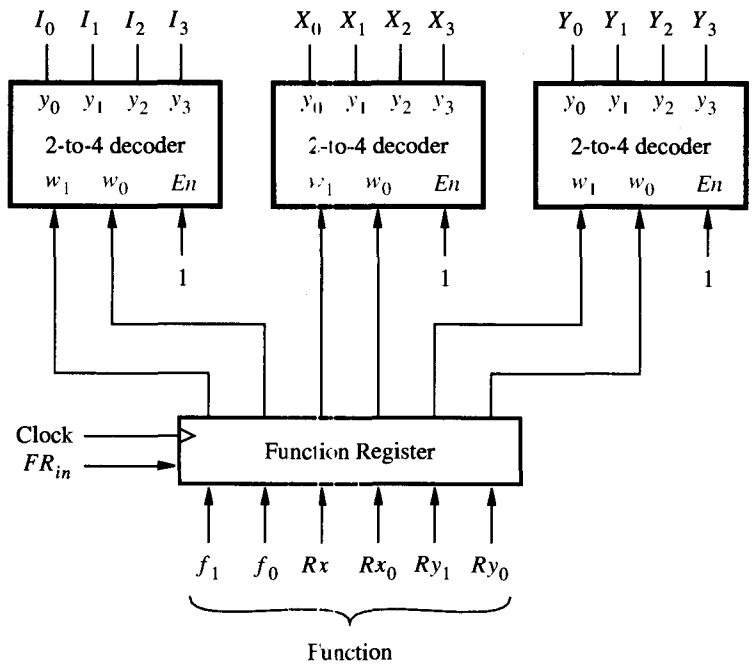


Figure 7.75 The function register and decoders.

logic expression is

$$Clear = \overline{w}T_0 + Done$$

The FR_{in} signal is used to load the values on the *Function* inputs into the Function Register when w changes to 1. Hence

$$FR_{in} = wT_0$$

The rest of the outputs from the control circuit depend on the specific step being performed in each operation. The values that have to be generated for each signal are shown in Table 7.3. Each row in the table corresponds to a specific operation, and each column represents one time step. The *Extern* signal is asserted only in the first step of the *Load* operation. Therefore, the logic expression that implements this signal is

$$Extern = I_0T_1$$

Done is asserted in the first step of *Load* and *Move*, as well as in the third step of *Add* and *Sub*. Hence

$$Done = (I_0 + I_1)T_1 + (I_2 + I_3)T_3$$

The A_{in}, G_{in}, and G_{out} signals are asserted in the *Add* and *Sub* operations. A_{in} is asserted in step T_1, G_{in} is asserted in T_2, and G_{out} is asserted in T_3. The *AddSub* signal has to be set to 0 in the *Add* operation and to 1 in the *Sub* operation. This is achieved with the following logic expressions

$$\begin{aligned} A_{in} &= (I_2 + I_3)T_1 \\ G_{in} &= (I_2 + I_3)T_2 \\ G_{out} &= (I_2 + I_3)T_3 \\ AddSub &= I_3 \end{aligned}$$

The values of $R0_{in}, \ldots, R3_{in}$ are determined using either the $X_0, \ldots, X_3$ signals or the $Y_0, \ldots, Y_3$ signals. In Table 7.3 these actions are indicated by writing either $R_{in} = X$ or $R_{in} = Y$. The meaning of $R_{in} = X$ is that $R0_{in} = X_0$, $R1_{in} = X_1$, and so on. Similarly, the values of $R0_{out}, \ldots, R3_{out}$ are specified using either $R_{out} = X$ or $R_{out} = Y$.

Table 7.3 Control signals asserted in each operation/time step.

	T_1	T_2	T_3
(Load): I_0	Extern, $R_{in} = X$, Done		
(Move): I_1	$R_{in} = X$, $R_{out} = Y$, Done		
(Add): I_2	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, AddSub = 0	G_{out}, $R_{in} = X$, Done
(Sub): I_3	$R_{out} = X$, A_{in}	$R_{out} = Y$, G_{in}, AddSub = 1	G_{out}, $R_{in} = X$, Done

We will develop the expressions for $R0_{in}$ and $R0_{out}$ by examining Table 7.3 and then show how to derive the expressions for the other register control signals. The table shows that $R0_{in}$ is set to the value of X_0 in the first step of both the *Load* and *Move* operations and in the third step of both the *Add* and *Sub* operations, which leads to the expression

$$R0_{in} = (I_0 + I_1)T_1X_0 + (I_2 + I_3)T_3X_0$$

Similarly, $R0_{out}$ is set to the value of Y_0 in the first step of *Move*. It is set to X_0 in the first step of *Add* and *Sub* and to Y_0 in the second step of these operations, which gives

$$R0_{out} = I_1T_1Y_0 + (I_2 + I_3)(T_1X_0 + T_2Y_0)$$

The expressions for $R1_{in}$ and $R1_{out}$ are the same as those for $R0_{in}$ and $R0_{out}$ except that X_1 and Y_1 are used in place of X_0 and Y_0. The expressions for $R2_{in}$, $R2_{out}$, $R3_{in}$, and $R3_{out}$ are derived in the same way.

The circuits shown in Figures 7.74 and 7.75, combined with the circuits represented by the above expressions, implement the control circuit in Figure 7.73.

Processors are extremely useful circuits that are widely used. We have presented only the most basic aspects of processor design. However, the techniques presented can be extended to design realistic processors, such as modern microprocessors. The interested reader can refer to books on computer organization for more details on processor design [1–2].

Verilog Code

In this section we give two different styles of Verilog code for describing the system in Figure 7.73. The first style uses tri-state buffers to represent the bus, and it gives the logic expressions shown above for the outputs of the control circuit. The second style of code uses multiplexers to represent the bus, and it uses **case** statements that correspond to Table 7.3 to describe the outputs of the control circuit.

Verilog code for an up-counter is shown in Figure 7.56. A modified version of this counter, named *upcount*, is shown in the code in Figure 7.76. It has a synchronous reset input, which is active high. Other subcircuits that we use in the Verilog code for the processor are the *dec2to4*, *regn*, and *trin* modules in Figures 6.35, 7.66, and 7.67.

```
module upcount (Clear, Clock, Q);
  input Clear, Clock;
  output [1:0] Q;
  reg [1:0] Q;

  always @(posedge Clock)
    if (Clear)
      Q <= 0;
    else
      Q <= Q + 1;

endmodule
```

Figure 7.76 A two-bit up-counter with synchronous reset.

Complete code for the processor is given in Figure 7.77. The instantiated modules *counter* and *decT* represent the subcircuits in Figure 7.74. Note that we have assumed that the circuit has an active-high reset input, *Reset*, which is used to initialize the counter to 00. The statement **assign** Func = {F, Rx, Ry} uses the concatenate operator to create the six-bit signal *Func*, which represents the inputs to the Function Register in Figure 7.75. The *functionreg* module represents the Function Register with the data inputs *Func* and the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
   input [7:0] Data;
   input Reset, w, Clock;
   input [1:0] F, Rx, Ry;
   output [7:0] BusWires;
   output Done;
   wire [7:0] BusWires;
   reg [0:3] Rin, Rout;
   reg [7:0] Sum;
   wire Clear, AddSub, Extern, Ain, Gin, Gout, FRin;
   wire [1:0] Count;
   wire [0:3] T, I, Xreg, Y;
   wire [7:0] R0, R1, R2, R3, A, G;
   wire [1:6] Func, FuncReg;
   integer k;

   upcount counter (Clear, Clock, Count);
   dec2to4 decT (Count, 1, T);

   assign Clear = Reset | Done | (~w & T[0]);
   assign Func = {F, Rx, Ry};
   assign FRin = w & T[0];

   regn functionreg (Func, FRin, Clock, FuncReg);
      defparam functionreg.n = 6;
   dec2to4 decI (FuncReg[1:2], 1, I);
   dec2to4 decX (FuncReg[3:4], 1, Xreg);
   dec2to4 decY (FuncReg[5:6], 1, Y);

   assign Extern = I[0] & T[1];
   assign Done = ((I[0] | I[1]) & T[1]) | ((I[2] | I[3]) & T[3]);
   assign Ain = (I[2] | I[3]) & T[1];
   assign Gin = (I[2] | I[3]) & T[2];
   assign Gout = (I[2] | I[3]) & T[3];
   assign AddSub = I[3];
```

... continued in Part *b*.

Figure 7.77 Code for the prcoessor (Part a).

```
    // RegCntl
    always @(I or T or Xreg or Y)
      for (k = 0; k < 4; k = k+1)
      begin
        Rin[k] = ((I[0] | I[1]) & T[1] & Xreg[k]) |
          ((I[2] | I[3]) & T[1] & Y[k]);
        Rout[k] = (I[1] & T[1] & Y[k]) | ((I[2] | I[3]) &
          ((T[1] & Xreg[k]) | (T[2] & Y[k])));
      end

    trin tri_ext (Data, Extern, BusWires);
    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    trin tri_0 (R0, Rout[0], BusWires);
    trin tri_1 (R1, Rout[1], BusWires);
    trin tri_2 (R2, Rout[2], BusWires);
    trin tri_3 (R3, Rout[3], BusWires);
    regn reg_A (BusWires, Ain, Clock, A);

    // alu
    always @(AddSub or A or BusWires)
      if (!AddSub)
        Sum = A + BusWires;
      else
        Sum = A - BusWires;

    regn reg_G (Sum, Gin, Clock, G);
    trin tri_G (G, Gout, BusWires);

endmodule
```

Figure 7.77 Code for the processor (Part *b*).

outputs *FuncReg*. The instantiated modules *decI*, *decX*, and *decY* represent the decoders in Figure 7.75. Following these statements the previously derived logic expressions for the outputs of the control circuit are given. For $R0_{in}, \ldots, R3_{in}$ and $R0_{out}, \ldots, R3_{out}$, a **for** loop is used to produce the expressions.

At the end of the code, the adder/subtractor module is defined and the tri-state buffers and registers in the processor are instantiated.

Using Multiplexers and Case Statements

We showed in Figure 7.65 that a bus can be implemented with multiplexers, rather than tri-state buffers. Verilog code that describes the processor using this approach is shown in Figure 7.78. The code illustrates a different way of describing the control circuit in the

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
   input [7:0] Data;
   input Reset, w, Clock;
   input [1:0] F, Rx, Ry;
   output [7:0] BusWires;
   output Done;
   reg [7:0] BusWires, Sum;
   reg [0:3] Rin, Rout;
   reg Extern, Done, Ain, Gin, Gout, AddSub;
   wire [1:0] Count, I;
   wire [0:3] Xreg, Y;
   wire [7:0] R0, R1, R2, R3, A, G;
   wire [1:6] Func, FuncReg, Sel;

   wire Clear = Reset | Done | (~w & ~Count[1] & ~Count[0]);
   upcount counter (Clear, Clock, Count);
   assign Func = {F, Rx, Ry};
   wire FRin = w & ~Count[1] & ~Count[0];
   regn functionreg (Func, FRin, Clock, FuncReg);
      defparam functionreg.n = 6;
   assign I = FuncReg[1:2];
   dec2to4 decX (FuncReg[3:4], 1, Xreg);
   dec2to4 decY (FuncReg[5:6], 1, Y);

   always @(Count or I or Xreg or Y)
   begin
      Extern = 1'b0; Done = 1'b0; Ain = 1'b0; Gin = 1'b0;
      Gout = 1'b0; AddSub = 1'b0; Rin = 4'b0; Rout = 4'b0;
      case (Count)
         2'b00: ; //no signals asserted in time step T0
         2'b01:   //define signals in time step T1
            case (I)
               2'b00: begin //Load
                        Extern = 1'b1; Rin = Xreg; Done = 1'b1;
                      end
               2'b01: begin //Move
                        Rout = Y; Rin = Xreg; Done = 1'b1;
                      end
               default: begin //Add, Sub
                          Rout = Xreg; Ain = 1'b1;
                        end
            endcase
```

... continued in Part *b*.

Figure 7.78 Alternative code for the processor (Part *a*).

```
            2'b10: //define signals in time step T2
               case(I)
                  2'b10: begin //Add
                              Rout = Y; Gin = 1'b1;
                         end
                  2'b11: begin //Sub
                              Rout = Y; AddSub = 1'b1; Gin = 1'b1;
                         end
                  default: ; //Add, Sub
               endcase
            2'b11:
               case (I)
                  2'b10, 2'b11: begin
                                   Gout = 1'b1; Rin = Xreg; Done = 1'b1;
                                end
                  default: ; //Add, Sub
               endcase
         endcase
      end

      regn reg_0 (BusWires, Rin[0], Clock, R0);
      regn reg_1 (BusWires, Rin[1], Clock, R1);
      regn reg_2 (BusWires, Rin[2], Clock, R2);
      regn reg_3 (BusWires, Rin[3], Clock, R3);
      regn reg_A (BusWires, Ain, Clock, A);
```

... continued in Part *c*.

Figure 7.78 Alternative code for the processor (Part *b*).

processor. It does not give logic expressions for the signals *Extern*, *Done*, and so on, as in Figure 7.77. Instead, **case** statements are used to represent the information shown in Table 7.3. Each control signal is first assigned the value 0 as a default. This is required because the **case** statements specify the values of the control signals only when they should be asserted, as we did in Table 7.3. As explained in section 7.12.2, when the value of a signal is not specified, the signal retains its current value. This implied memory results in a feedback connection in the synthesized circuit. We avoid this problem by providing the default value of 0 for each of the control signals involved in the **case** statements.

In Figure 7.77 the decoders *decT* and *decI* are used to decode the *Count* signal and the stored values of the F input, respectively. The *decT* decoder has the outputs $T_0, \ldots, T_3$, and *decI* produces $I_0, \ldots, I_3$. In Figure 7.78 these two decoders are not used, because they do not serve a useful purpose in this code. Instead, the signal I is defined as a two-bit signal, and the two-bit signal *Count* is used instead of T. These signals are used in the **case** statements. The code sets I to the value of the two left-most bits in the Function Register, which correspond to the stored values of the input F.

```
// alu
always @(AddSub or A or BusWires)
begin
   if (!AddSub)
      Sum = A + BusWires;
   else
      Sum = A – BusWires;
end

regn reg_G (Sum, Gin, Clock, G);
assign Sel = {Rout, Gout, Extern};

always @(Sel or R0 or R1 or R2 or R3 or G or Data)
begin
   if (Sel == 6'b100000)
      BusWires = R0;
   else if (Sel == 6'b010000)
      BusWires = R1;
   else if (Sel == 6'b001000)
      BusWires = R2;
   else if (Sel == 6'b000100)
      BusWires = R3;
   else if (Sel == 6'b000010)
      BusWires = G;
   else BusWires = Data;
end

endmodule
```

Figure 7.78 Alternative code for the processor (Part c).

There are two nested levels of **case** statements. The first one enumerates the possible values of *Count*. For each alternative in this **case** statement, which represents a column in Table 7.3, there is a nested **case** statement that enumerates the four values of I. As indicated by the comments in the code, the nested **case** statements correspond exactly to the information given in Table 7.3.

At the end of Figure 7.78, the bus is described with an **if-else** statement which represents multiplexers that place the appropriate data onto *BusWires*, depending on the values of R_{out}, G_{out}, and *Extern*.

The circuits synthesized from the code in Figures 7.77 and 7.78 are functionally equivalent. The style of code in Figure 7.78 has the advantage that it does not require the manual effort of analyzing Table 7.3 to generate the logic expressions for the control signals in Figure 7.77. By using the style of code in Figure 7.78, these expressions are produced automatically by the Verilog compiler as a result of analyzing the **case** statements. The style of code in Figure 7.78 is less prone to careless errors. Also, using this style of code it

would be straightforward to provide additional capabilities in the processor, such as adding other operations.

We synthesized a circuit to implement the code in Figure 7.78 in a chip. Figure 7.79 gives an example of the results of a timing simulation. Each clock cycle in which $w = 1$ in this timing diagram indicates the start of an operation. In the first such operation, at 250 ns in the simulation time, the values of both inputs F and Rx are 00. Hence the operation corresponds to "*Load R0, Data.*" The value of *Data* is 2A, which is loaded into $R0$ on the next positive clock edge. The next operation loads 55 into register $R1$, and the subsequent operation loads 22 into $R2$. At 850 ns the value of the input F is 10, while $Rx = 01$ and $Ry = 00$. This operation is "*Add R1, R0.*" In the following clock cycle, the contents of $R1$ (55) appear on the bus. This data is loaded into register A by the clock edge at 950 ns, which also results in the contents of $R0$ (2A) being placed on the bus. The adder/subtractor module generates the correct sum (7F), which is loaded into register G at 1050 ns. After this clock edge the new contents of G (7F) are placed on the bus and loaded into register $R1$ at 1150 ns. Two more operations are shown in the timing diagram. The one at 1250 ns ("*Move R3, R1*") copies the contents of $R1$ (7F) into $R3$. Finally, the operation starting at 1450 ns ("*Sub R3, R2*") subtracts the contents of $R2$ (22) from the contents of $R3$ (7F), producing the correct result, $7F - 22 = 5D$.

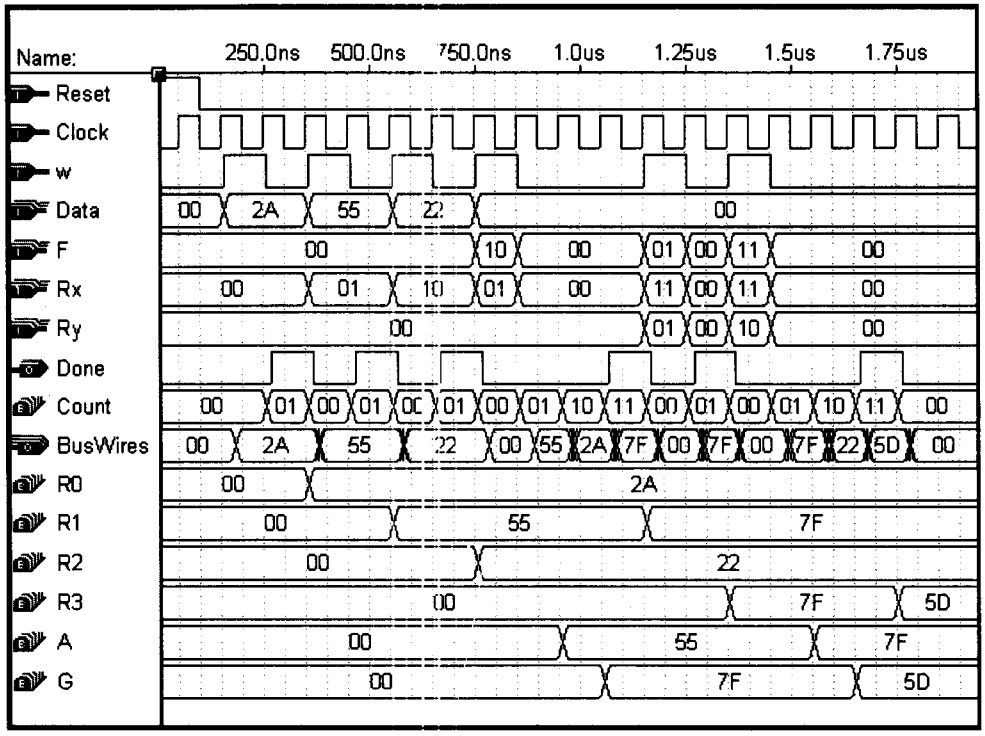


Figure 7.79 Timing simulation for the Verilog code in Figure 7.78.

7.14.3 REACTION TIMER

We showed in Chapter 3 that electronic devices operate at remarkably fast speeds, with the typical delay through a logic gate being less than 1 ns. In this example we use a logic circuit to measure the speed of a much slower type of device—a person.

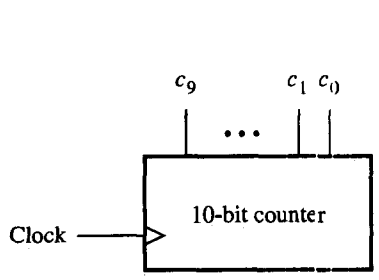
We will design a circuit that can be used to measure the reaction time of a person to a specific event. The circuit turns on a small light, called a *light-emitting diode (LED)*. In response to the LED being turned on, the person attempts to press a switch as quickly as possible. The circuit measures the elapsed time from when the LED is turned on until the switch is pressed.

To measure the reaction time, a clock signal with an appropriate frequency is needed. In this example we use a 100 Hz clock, which measures time at a resolution of 1/100 of a second. The reaction time can then be displayed using two digits that represent fractions of a second from 00/100 to 99/100.

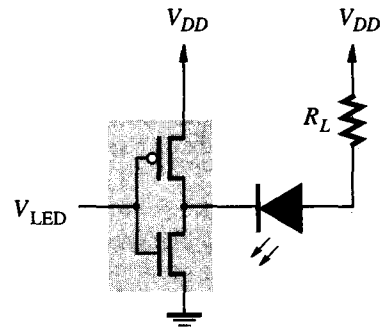
Digital systems often include high-frequency clock signals to control various subsystems. In this case assume the existence of an input clock signal with the frequency 102.4 kHz. From this signal we can derive the required 100 Hz signal by using a counter as a *clock divider*. A timing diagram for a four-bit counter is given in Figure 7.22. It shows that the least-significant bit output, Q_0, of the counter is a periodic signal with half the frequency of the clock input. Hence we can view Q_0 as dividing the clock frequency by two. Similarly, the Q_1 output divides the clock frequency by four. In general, output Q_i in an n-bit counter divides the clock frequency by 2^{i+1}. In the case of our 102.4 kHz clock signal, we can use a 10-bit counter, as shown in Figure 7.80*a*. The counter output c_9 has the required 100 Hz frequency because 102400 Hz/1024 = 100 Hz.

The reaction timer circuit has to be able to turn an LED on and off. The graphical symbol for an LED is shown in blue in Figure 7.80*b*. Small blue arrows in the symbol represent the light that is emitted when the LED is turned on. The LED has two terminals: the one on the left in the figure is the *cathode*, and the terminal on the right is the *anode*. To turn the LED on, the cathode has to be set to a lower voltage than the anode, which causes a current to flow through the LED. If the voltages on its two terminals are equal, the LED is off.

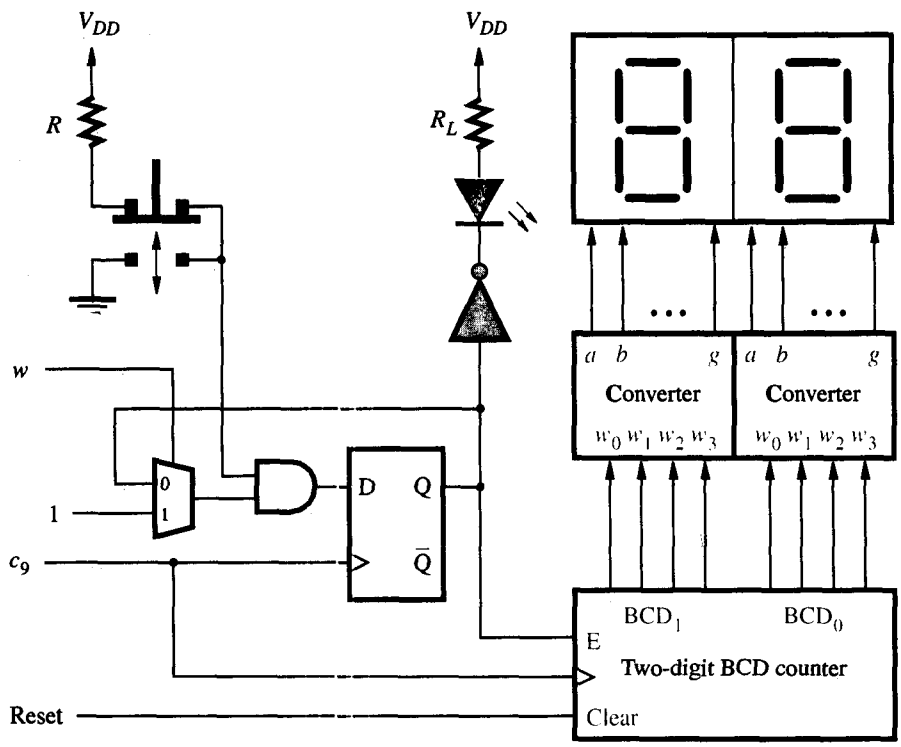
Figure 7.80*b* shows one way to control the LED, using an inverter. If the input voltage $V_{LED} = 0$, then the voltage at the cathode is equal to V_{DD}; hence the LED is off. But if $V_{LED} = V_{DD}$, the cathode voltage is 0 V and the LED is on. The amount of current that flows is limited by the value of the resistor R_L. This current flows through the LED and the NMOS transistor in the inverter. Since the current flows *into* the inverter, we say that the inverter *sinks* the current. The maximum current that a logic gate can sink without sustaining permanent damage is usually called I_{OL}, which stands for the "maximum current when the output is low." The value of R_L is chosen such that the current is less than I_{OL}. As an example assume that the inverter is implemented inside a PLD device. The typical value of I_{OL}, which would be specified in the data sheet for the PLD, is about 12 mA. For $V_{DD} = 5$ V, this leads to $R_L \approx 450\ \Omega$ because $5\ V/450\ \Omega = 11$ mA (there is actually a small voltage drop across the LED when it is turned on, but we ignore this for simplicity). The amount of light emitted by the LED is proportional to the current flow. If 11 mA is insufficient, then the inverter should be implemented in



(a) Clock divider



(b) LED circuit



(c) Push-button switch, LED, and 7-segment displays

Figure 7.80 A reaction-timer circuit.

a buffer chip, like those described in section 3.5, because buffers provide a higher value of I_{OL}.

The complete reaction-timer circuit is illustrated in Figure 7.80*c*, with the inverter from part (*b*) shaded in grey. The graphical symbol for a push-button switch is shown in the top left of the diagram. The switch normally makes contact with the top terminals, as depicted in the figure. When depressed, the switch makes contact with the bottom terminals; when released, it automatically springs back to the top position. In the figure the switch is connected such that it normally produces a logic value of 1, and it produces a 0 pulse when pressed.

The push-button switch is connected to the clear input on a D flip-flop. The output of this flip-flop determines whether the LED is on or off, and it also provides the count enable input to a two-digit BCD counter. As discussed in section 7.11, each digit in a BCD counter has four bits that take the values 0000 to 1001. Thus the counting sequence can be viewed as decimal numbers from 00 to 99. A circuit for the BCD counter is given in Figure 7.28. In Figure 7.80*c* both the flip-flop and the counter are clocked by the c_9 output of the clock divider in part (*a*) of the figure. The intended use of the reaction-timer circuit is to first depress the switch to turn off the LED and disable the counter. Then the *Reset* input is asserted to clear the contents of the counter to 00. The input w normally has the value 0, which keeps the flip-flop cleared and prevents the count value from changing. The reaction test is initiated by setting $w = 1$ for one c_9 clock cycle. After the next positive edge of c_9, the flip-flop output becomes a 1, which turns on the LED. We assume that w returns to 0 after one clock cycle, but the flip-flop output remains at 1 because of the 2-to-1 multiplexer connected to the D input. The counter is then incremented every 1/100 of a second. Each digit in the counter is connected through a code converter to a 7-segment display, which we described in the discussion for Figure 6.25. When the user depresses the switch, the flip-flop is cleared, which turns off the LED and stops the counter. The two-digit display shows the elapsed time to the nearest 1/100 of a second from when the LED was turned on until the user was able to respond by depressing the switch.

Verilog Code

To describe the circuit in Figure 7.80*c* using Verilog code, we can make use of subcircuits for the BCD counter and the 7-segment code converter. The code for the latter subcircuit is given in Figure 6.38 and is not repeated here. Code for the BCD counter, which represents the circuit in Figure 7.28, is shown in Figure 7.81. The two-digit BCD output is represented by the 2 four-bit signals *BCD*1 and *BCD*0. The *Clear* input provides a synchronous reset for both digits in the counter. If $E = 1$, the count value is incremented on the positive clock edge; and if $E = 0$, the count value is unchanged. Each digit can take the values from 0000 to 1001.

Figure 7.82 gives the code for the reaction timer. The input signal *Pushn* represents the value produced by the push-button switch. The output signal *LEDn* represents the output of the inverter that is used to control the LED. The two 7-segment displays are controlled by the seven-bit signals *Digit*1 and *Digit*0.

In Figure 7.61 we showed how a register, R, can be designed with a control signal R_{in}. If $R_{in} = 1$ data is loaded into the register on the active clock edge and if $R_{in} = 0$, the stored contents of the register are not changed. The flip-flop in Figure 7.80 is used in the same

```
module BCDcount (Clock, Clear, E, BCD1, BCD0);
   input Clock, Clear, E;
   output [3:0] BCD1, BCD0;
   reg [3:0] BCD1, BCD0;

   always @(posedge Clock)
   begin
      if (Clear)
      begin
         BCD1 <= 0;
         BCD0 <= 0;
      end
      else if (E)
         if (BCD0 == 4'b1001)
         begin
            BCD0 <= 0;
            if (BCD1 == 4'b1001)
               BCD1 <= 0;
            else
               BCD1 <= BCD1 + 1;
         end
         else
            BCD0 <= BCD0 + 1;
   end

endmodule
```

Figure 7.81 Code for the two-digit BCD counter in Figure 7.28.

way. If $w = 1$, the flip-flop is loaded with the value 1, but if $w = 0$ the stored value in the flip-flop is not changed. This circuit is described by the **always** block in Figure 7.82, which also includes a synchronous reset input. We have chosen to use a synchronous reset because the flip-flop output is connected to the enable input E on the BCD counter. As we know from the discussion in section 7.3, it is important that all signals connected to flip-flops meet the required setup and hold times. The push-button switch can be pressed at any time and is not synchronized to the c_9 clock signal. By using a synchronous reset for the flip-flop in Figure 7.80, we avoid possible timing problems in the counter.

The flip-flop output is named *LED*, which is inverted to produce the *LEDn* signal that controls the LED. In the device used to implement the circuit, *LEDn* would be generated by a buffer that is connected to an output pin on the chip package. If a PLD is used, this buffer has the associated value of $I_{OL} = 12$ mA that we mentioned earlier. At the end of Figure 7.82, the BCD counter and 7-segment code converters are instantiated as subcircuits.

A simulation of the reaction-timer circuit implemented in a chip is shown in Figure 7.83. Initially, *Pushn* is set to 0 to simulate depressing the switch to turn off the LED, and

```
module reaction (c9, Reset, w, Pushn, LEDn, Digit1, Digit0);
   input c9, Reset, w, Pushn;
   output LEDn;
   output [1:7] Digit1, Digit0;
   wire LEDn;
   wire [1:7] Digit1, Digit0;
   reg LED;
   wire [3:0] BCD1, BCD0;

   always @(posedge c9)
   begin
      if (Pushn == 0)
         LED <= 0;
      else if (w)
         LED <= 1;
   end

   assign LEDn = ~LED;
   BCDcount counter (c9, Reset, LED, BCD1, BCD0);
   seg7 seg1 (BCD1, Digit1);
   seg7 seg0 (BCD0, Digit0);

endmodule
```

Figure 7.82 Code for the reaction timer.

then *Pushn* returns to 1. Also, *Reset* is asserted to clear the counter. When *w* changes to 1, the circuit sets *LEDn* to 0, which represents the LED being turned on. After some amount of time, the switch will be depressed. In the simulation we arbitrarily set *Pushn* to 0 after 18 c_9 clock cycles. Thus this choice represents the case when the person's reaction time is about 0.18 seconds. In human terms this duration is a very short time; for electronic circuits it is a very long time. An inexpensive personal computer can perform tens of millions of operations in 0.18 seconds!

7.14.4 REGISTER TRANSFER LEVEL (RTL) CODE

At this point, we have introduced most of the Verilog constructs that are needed for synthesis. Most of our examples give behavioral code, utilizing **if-else** statements, **case** statements, **for** loops, and other procedural statements. It is possible to write behavioral code in a style that resembles a computer program, in which there is a complex flow of control with many loops and branches. With such code, sometimes called *high-level* behavioral code, it is difficult to relate the code to the final hardware implementation; it may even be difficult to predict what circuit a high-level synthesis tool will produce. In this book we do not use the high-level

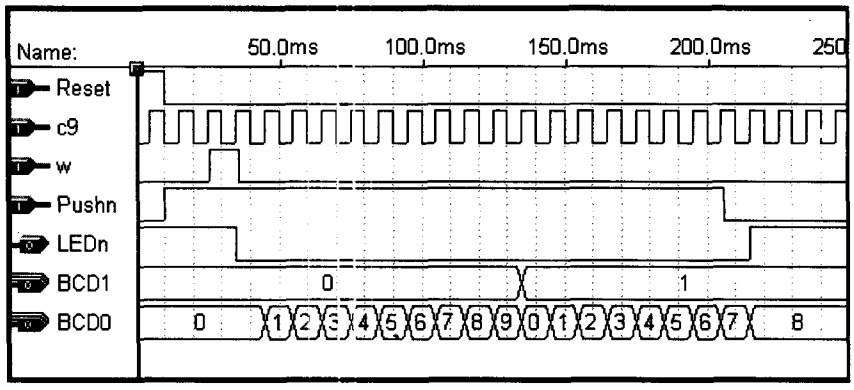


Figure 7.83 Simulation of the reaction timer circuit.

style of code. Instead, we present Verilog code in such a way that the code can be easily related to the circuit that is being described. Most design modules presented are fairly small, to facilitate simple descriptions. Larger designs are built by interconnecting the smaller modules. This approach is usually referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. RTL code is characterized by a straightforward flow of control through the code; it comprises well-understood subcircuits that are connected together in a simple way.

7.15 CONCLUDING REMARKS

In this chapter we have presented circuits that serve as basic storage elements in digital systems. These elements are used to build larger units such as registers, shift registers, and counters. Many other texts that deal with this material are available [3–11]. We have illustrated how circuits with flip-flops can be described using Verilog code. More information on Verilog can be found in [12–19]. In the next chapter a more formal method for designing circuits with flip-flops will be presented.

PROBLEMS

7.1 Consider the timing diagram in Figure P7.1. Assuming that the *D* and *Clock* inputs shown are applied to the circuit in Figure 7.12, draw waveforms for the Q_a, Q_b, and Q_c signals.

7.2 Can the circuit in Figure 7.3 be modified to implement an SR latch? Explain your answer.

7.3 Figure 7.5 shows a latch built with NOR gates. Draw a similar latch using NAND gates. Derive its truth table and show its timing diagram.

7.4 Show a circuit that implements the gated SR latch using NAND gates only.

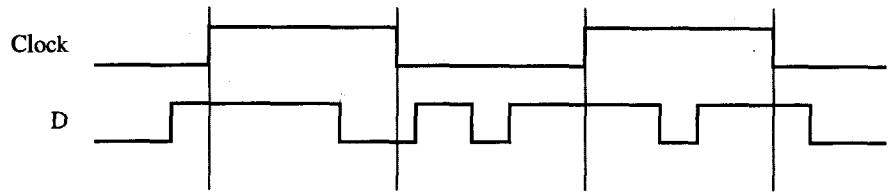


Figure P7.1 Timing diagram for problem 7.1.

7.5 Given a 100-MHz clock signal, derive a circuit using D flip-flops to generate 50-MHz and 25-MHz clock signals. Draw a timing diagram for all three clock signals, assuming reasonable delays.

7.6 An SR flip-flop is a flip-flop that has set and reset inputs like a gated SR latch. Show how an SR flip-flop can be constructed using a D flip-flop and other logic gates.

7.7 The gated SR latch in Figure 7.6*a* has unpredictable behavior if the S and R inputs are both equal to 1 when the *Clk* changes to 0. One way to solve this problem is to create a *set-dominant* gated SR latch in which the condition $S = R = 1$ cause the latch to be set to 1. Design a set-dominant gated SR latch and show the circuit.

7.8 Show how a JK flip-flop can be constructed using a T flip-flop and other logic gates.

7.9 Consider the circuit in Figure P7.2. Assume that the two NAND gates have much longer (about four times) propagation delay than the other gates in the circuit. How does this circuit compare with the circuits that we discussed in this chapter?

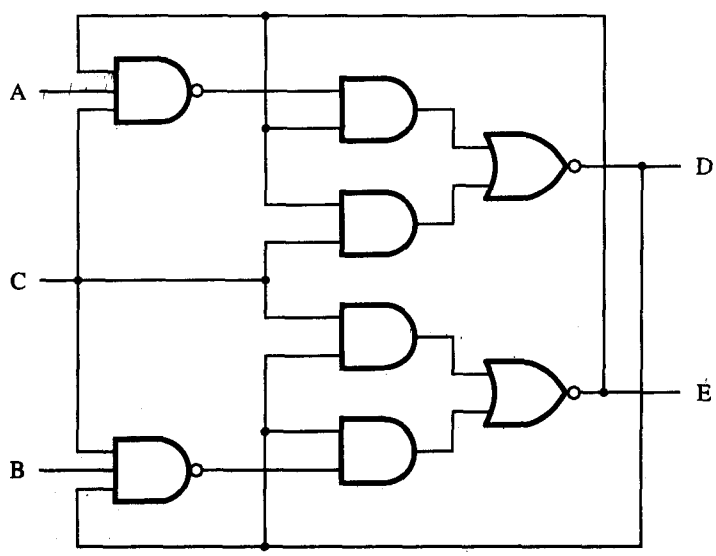


Figure P7.2 Circuit for problem 7.9.

7.10 Write Verilog code that represents a T flip-flop with an asynchronous clear input. Use behavioral code, rather than structural code.

7.11 Write Verilog code that represents a JK flip-flop. Use behavioral code, rather than structural code.

7.12 Synthesize a circuit for the code written for problem 7.11 by using your CAD tools. Simulate the circuit and show a timing diagram that verifies the desired functionality.

7.13 A four-bit barrel shifter is a combinational circuit with four data inputs, two control inputs, and two data outputs. It allows the data inputs to be shifted onto the outputs by 0, 1, 2, or 3 bit positions, with the rightmost bits wrapping around (rotating) to the leftmost bits. For example, if the data inputs are 1100 and the control input specifies a two-bit shift, then the output would be 0011. If the data input is 1110, a two-bit rotation produces 1011. Design a four-bit shift register using a barrel shifter that can shift to the right by 0, 1, 2, or 3 positions.

7.14 Write Verilog code for the shift register described in problem 7.13.

7.15 Design a four-bit synchronous counter with parallel load. Use T flip-flops, instead of the D flip-flops used in section 7.9.3.

7.16 Design a three-bit up/down counter using T flip-flops. It should include a control input called $\overline{\text{Up}}$/Down. If $\overline{\text{Up}}$/Down = 0, then the circuit should behave as an up-counter. If $\overline{\text{Up}}$/Down = 1, then the circuit should behave as a down-counter.

7.17 Repeat problem 7.16 using D flip-flops.

7.18 The circuit in Figure P7.3 looks like a counter. What is the sequence that this circuit counts in?

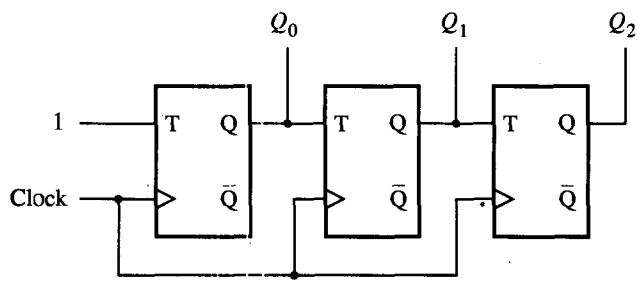


Figure P7.3 The circuit for problem 7.18.

7.19 Consider the circuit in Figure P7.4. How does this circuit compare with the circuit in Figure 7.17? Can the circuits be used for the same purposes? If not, what is the key difference between them?

7.20 Construct a NOR-gate circuit, similar to the one in Figure 7.11*a*, which implements a negative-edge-triggered D flip-flop.

7.21 Write Verilog code that represents a modulo-12 up-counter with synchronous reset.

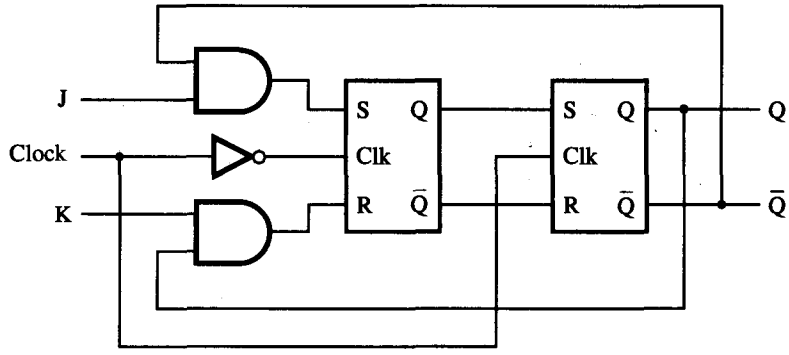


Figure P7.4 Circuit for problem 7.19.

7.22 For the flip-flops in the counter in Figure 7.25, assume that $t_{su} = 3$ ns, $t_h = 1$ ns, and the propagation delay through a flip-flop is 1 ns. Assume that each AND gate, XOR gate, and 2-to-1 multiplexer has the propagation delay equal to 1 ns. What is the maximum clock frequency for which the circuit will operate correctly?

7.23 Write Verilog code that represents an eight-bit Johnson counter. Synthesize the code with your CAD tools and give a timing simulation that shows the counting sequence.

7.24 Write Verilog code in the style shown in Figure 7.55 that represents a ring counter. Your code should have a parameter n that sets the number of flip-flops in the counter.

7.25 Write Verilog code that describes the functionality of the circuit shown in Figure 7.48.

7.26 Write Verilog code that instantiates the *lpm_counter* module from the LPM library. Configure the module as a 32-bit up-counter. For the counter circuit in Figure 7.24, we said that the AND-gate chain can be thought of as the carry-chain. The FLEX 10K FPGA contains special-purpose logic to implement this carry-chain such that it has minimal propagation delay. Use the MAX+plusII synthesis options to implement the *lpm_counter* in two ways: with the dedicated carry-chain used and with the dedicated carry-chain not used. Use the Timing Analyzer in MAX+plusII to determine the maximum speed of operation of the counter in both cases. See the tutorials in Appendices B, C, and D for instructions on using the appropriate features of the CAD tools.

7.27 Figure 7.69 gives Verilog code for a digital system that swaps the contents of two registers, $R1$ and $R2$, using register $R3$ for temporary storage. Create an equivalent schematic using your CAD tools for this system. Synthesize a circuit for this schematic and perform a timing simulation.

7.28 Repeat problem 7.27 using the control circuit in Figure 7.63.

7.29 Modify the code in Figure 7.71 to use the control circuit in Figure 7.63. Synthesize the code for implementation in a chip and perform a timing simulation.

7.30 In section 7.14.2 we designed a processor that performs the operations listed in Table 7.3. Design a modified circuit that performs an additional operation Swap *Rx*, *Ry*. This operation

swaps the contents of registers *Rx* and *Ry*. Use three bits $f_2 f_1 f_0$ to represent the input *F* shown in Figure 7.75 because there are now five operations, rather than four. Add a new register, named *Tmp*, into the system, to be used for temporary storage during the swap operation. Show logic expressions for the outputs of the control circuit, as was done in section 7.14.2.

7.31 A ring oscillator is a circuit that has an odd number, *n*, of inverters connected in a ringlike structure, as shown in Figure P7.5. The output of each inverter is a periodic signal with a certain period.

(a) Assume that all the inverters are identical; hence they all have the same delay, t_p. Let the output of one of the inverters be named *f*. Give an equation that expresses the period of the signal *f* in terms of *n* and t_p.

(b) For this part you are to design a circuit that can be used to experimentally measure the delay t_p through one of the inverters in the ring oscillator. Assume the existence of an input called *Reset* and another called *Interval*. The timing of these two signals is shown in Figure P7.6. The length of time for which *Interval* has the value 1 is known. Assume that this length of time is 100 ns. Design a circuit that uses the *Reset* and *Interval* signals and the signal *f* from part (*a*) to experimentally measure t_p. In your design you may use logic gates and subcircuits such as adders, flip-flops, counters, registers, and so on.

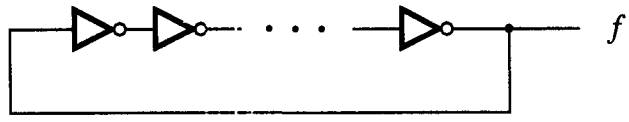


Figure P7.5 A ring oscillator.

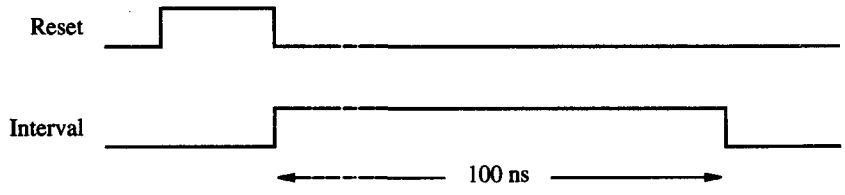
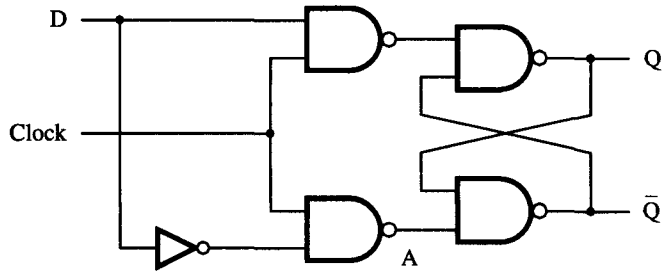


Figure P7.6 Timing of signals for problem 7.31.

7.32 A circuit for a gated D latch is shown in Figure P7.7. Assume that the propagation delay through either a NAND gate or an inverter is 1 ns. Complete the timing diagram given in the figure, which shows the signal values with 1 ns resolution.

7.33 A logic circuit has two inputs, *Clock* and *Start*, and two outputs, *f* and *g*. The behavior of the circuit is described by the timing diagram in Figure P7.8. When a pulse is received on the *Start* input, the circuit produces pulses on the *f* and *g* outputs as shown in the



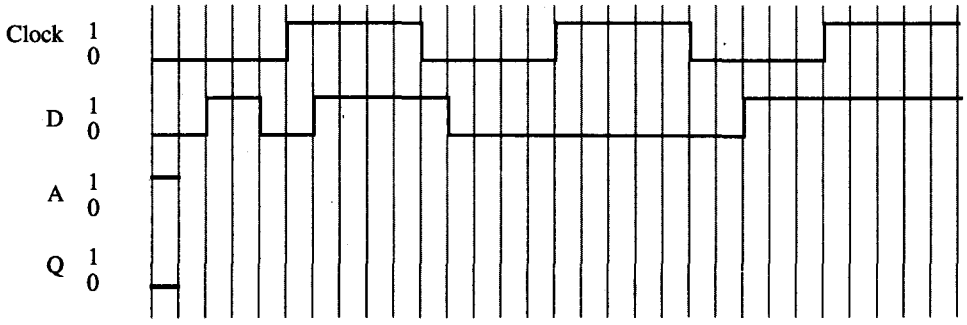


Figure P7.7 Circuit and timing diagram for problem 7.32.

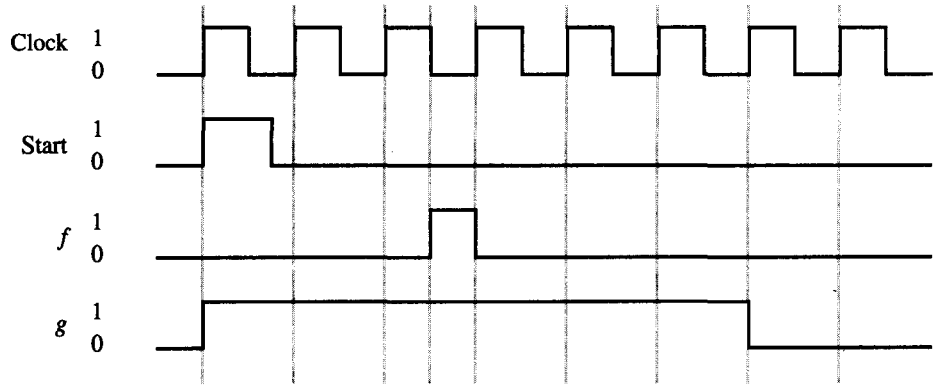


Figure P7.8 Timing diagram for problem 7.33.

timing diagram. Design a suitable circuit using only the following components: a three-bit resettable positive-edge-triggered synchronous counter and basic logic gates. For your answer assume that the delays through all logic gates and the counter are negligible.

7.34 The following code checks for adjacent ones in an n-bit vector.

```
always @(A)
begin
   f = A[1] & A[0];
   for (k = 2; k < n; k = k+1)
      f = f | (A[k] & A[k−1]);
end
```

With blocking assignments this code produces the desired logic function, which is $f = a_1a_0 + \cdots + a_{n-1}a_{n-2}$. What logic function is produced if we change the code to use non-blocking assignments?

7.35 The Verilog code in Figure P7.9 represents a 3-bit *linear-feedback shift register* (LFSR). This type of circuit generates a counting sequence of pseudo-random numbers that repeats after $2^n - 1$ clock cycles, where n is the number of flip-flops in the LFSR. Synthesize a circuit to implement the LFSR in a chip. Draw a diagram of the circuit. Simulate the circuit's behavior by loading the pattern 001 into the LFSR and then enabling the register to count. What is the counting sequence?

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
         Q <= {Q[2], Q[0] ^ Q[2], Q[1]};

endmodule
```

Figure P7.9 Code for a linear-feedback shift register.

7.36 Repeat problem 7.35 for the Verilog code in Figure P7.10.

7.37 The Verilog code in Figure P7.11 is equivalent to the code in Figure P7.9, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

7.38 The Verilog code in Figure P7.12 is equivalent to the code in Figure P7.10, except that blocking assignments are used. Draw the circuit represented by this code. What is its counting sequence?

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
         Q <= {Q[2], Q[0], Q[1] ^ Q[2]};

endmodule
```

Figure P7.10 Code for a linear-feedback shift register.

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] = Q[2];
         Q[1] = Q[0] ^ Q[2];
         Q[2] = Q[1];
      end

endmodule
```

Figure P7.11 Code for problem 7.37.

7.39 A universal shift register can shift in both the left-to-right and right-to-left directions, and it has parallel-load capability. Draw a circuit for such a shift register.

7.40 Write Verilog code for a universal shift register with n bits.

```
module lfsr (R, L, Clock, Q);
   input [0:2] R;
   input L, Clock;
   output [0:2] Q;
   reg [0:2] Q;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else
      begin
         Q[0] = Q[2];
         Q[1] = Q[0];
         Q[2] = Q[1] ^ Q[2];
      end

endmodule
```

Figure P7.12 Code for problem 7.38.

REFERENCES

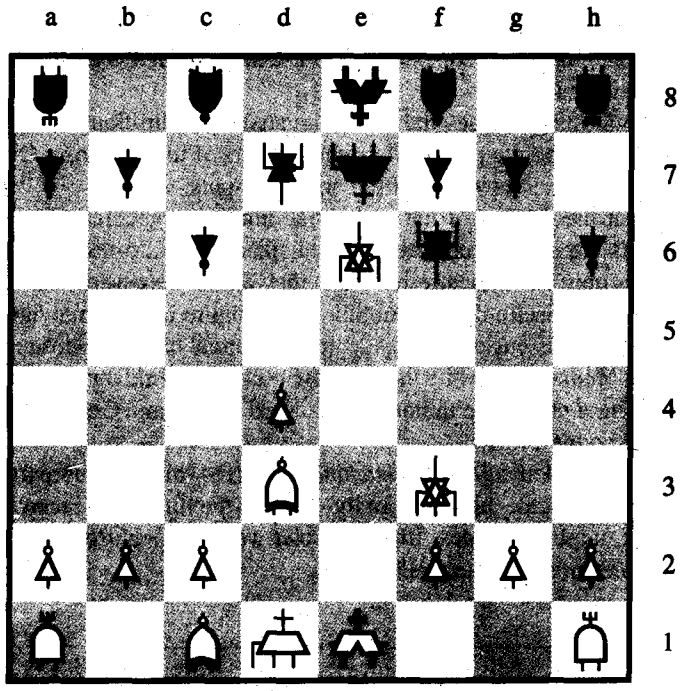
1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed., (McGraw-Hill: New York, 2002).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2nd ed., (Morgan Kaufmann: San Francisco, CA, 1998).
3. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
7. R. H. Katz, *Contemporary Logic Design*, (Benjamin/Cummings: Redwood City, CA, 1994).
8. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed., (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, (Prentice-Hall: Englewood Cliffs, NJ, 1990).

11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 1995).
13. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
14. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice-Hall: Upper Saddle River, NJ, 1996).
15. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
16. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
17. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
18. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
19. S. Sutherland, *Verilog 2001—A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

chapter

8

SYNCHRONOUS SEQUENTIAL CIRCUITS



8. Ng5xe6, Qd8–e7

In preceding chapters we considered combinational logic circuits in which outputs are determined fully by the present values of inputs. We also discussed how simple storage elements can be implemented in the form of flip-flops. The output of a flip-flop depends on the state of the flip-flop rather than the value of its inputs at any given time; the inputs cause changes in the state.

In this chapter we deal with a general class of circuits in which the outputs depend on the past behavior of the circuit, as well as on the present values of inputs. They are called *sequential circuits*. In most cases a clock signal is used to control the operation of a sequential circuit; such a circuit is called a *synchronous sequential circuit*. The alternative, in which no clock signal is used, is called an *asynchronous sequential circuit*. Synchronous circuits are easier to design and are used in a vast majority of practical applications; they are the topic of this chapter. Asynchronous circuits will be discussed in Chapter 9.

Synchronous sequential circuits are realized using combinational logic and one or more flip-flops. The general structure of such a circuit is shown in Figure 8.1. The circuit has a set of primary inputs, W, and produces a set of outputs, Z. The values of the outputs of the flip-flops are referred to as the *state*, Q, of the circuit. Under control of the clock signal, the flip-flop outputs change their state as determined by the combinational logic that feeds the inputs of these flip-flops. Thus the circuit moves from one state to another. To ensure that only one transition from one state to another takes place during one clock cycle, the flip-flops have to be of the edge-triggered type. They can be triggered either by the positive (0 to 1 transition) or by the negative (1 to 0 transition) edge of the clock. We will use the term *active clock edge* to refer to the clock edge that causes the change in state.

The combinational logic that provides the input signals to the flip-flops derives its inputs from two sources: the primary inputs, W, and the present (current) outputs of the flip-flops, Q. Thus changes in state depend on both the present state and the values of the primary inputs.

Figure 8.1 indicates that the outputs of the sequential circuit are generated by another combinational circuit, such that the outputs are a function of the present state of the flip-flops and of the primary inputs. Although the outputs always depend on the present state, they do not necessarily have to depend directly on the primary inputs. Thus the connection shown in blue in the figure may or may not exist. To distinguish between these two possibilities, it is customary to say that sequential circuits whose outputs depend only on the state of the circuit are of *Moore* type, while those whose outputs depend on both the state and the primary inputs are of *Mealy* type. These names are in honor of Edward Moore and George Mealy, who investigated the behavior of such circuits in the 1950s.

Sequential circuits are also called *finite state machines (FSMs)*, which is a more formal name that is often found in technical literature. The name derives from the fact that the functional behavior of these circuits can be represented using a finite number of states. In this chapter we will often use the term *finite state machine*, or simply *machine*, when referring to sequential circuits.

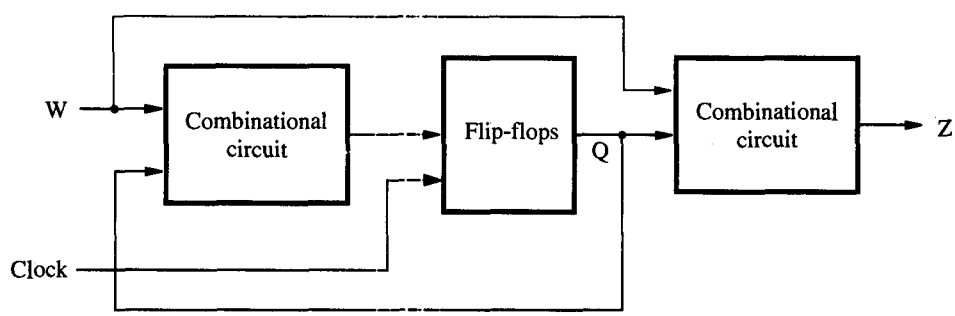


Figure 8.1 The general form of a sequential circuit.

8.1 BASIC DESIGN STEPS

We will introduce the techniques for designing sequential circuits by means of a simple example. Suppose that we wish to design a circuit that meets the following specification:

1. The circuit has one input, w, and one output, z.
2. All changes in the circuit occur on the positive edge of a clock signal.
3. The output z is equal to 1 if during two immediately preceding clock cycles the input w was equal to 1. Otherwise, the value of z is equal to 0.

From this specification it is apparent that the output z cannot depend solely on the present value of w. To illustrate this, consider the sequence of values of the w and z signals during 11 clock cycles, as shown in Figure 8.2. The values of w are assumed arbitrarily; the values of z correspond to our specification. These sequences of input and output values indicate that for a given input value the output may be either 0 or 1. For example, $w = 0$ during clock cycles t_2 and t_5, but $z = 0$ during t_2 and $z = 1$ during t_5. Similarly, $w = 1$ during t_1 and t_8, but $z = 0$ during t_1 and $z = 1$ during t_8. This means that z is not determined only by the present value of w, so there must exist different states in the circuit that determine the value of z.

8.1.1 STATE DIAGRAM

The first step in designing a finite state machine is to determine how many states are needed and which transitions are possible from one state to another. There is no set procedure for this task. The designer must think carefully about what the machine has to accomplish. A good way to begin is to select one particular state as a *starting* state; this is the state that the circuit should enter when power is first turned on or when a *reset* signal is applied. For our example let us assume that the starting state is called state A. As long as the input w is 0, the circuit need not do anything, and so each active clock edge should result in the circuit remaining in state A. When w becomes equal to 1, the machine should recognize this, and move to a different state, which we will call state B. This transition takes place on the next active clock edge after w has become equal to 1. In state B, as in state A, the circuit should keep the value of output z at 0, because it has not yet seen $w = 1$ for two consecutive clock cycles. When in state B, if w is 0 at the next active clock edge, the circuit should move back to state A. However, if $w = 1$ when in state B, the circuit should change to a third state, called C, and it should then generate an output $z = 1$. The circuit should remain in state C as long as $w = 1$ and should continue to maintain $z = 1$. When w becomes 0, the machine should move back to state A. Since the preceding description handles all possible values

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	0	1	0	0	1	1	0

Figure 8.2 Sequences of input and output signals.

of input w that the machine can encounter in its various states, we can conclude that three states are needed to implement the desired machine.

Now that we have determined in an informal way the possible transitions between states, we will describe a more formal procedure that can be used to design the corresponding sequential circuit. Behavior of a sequential circuit can be described in several different ways. The conceptually simplest method is to use a pictorial representation in the form of a *state diagram*, which is a graph that depicts states of the circuit as nodes (circles) and transitions between states as directed arcs. The state diagram in Figure 8.3 defines the behavior that corresponds to our specification. States A, B, and C appear as nodes in the diagram. Node A represents the starting state, and it is also the state that the circuit will reach *after* an input $w = 0$ is applied. In this state the output z should be 0, which is indicated as $A/z=0$ in the node. The circuit should remain in state A as long as $w = 0$, which is indicated by an arc with a label $w = 0$ that originates and terminates at this node. The first occurrence of $w = 1$ (following the condition $w = 0$) is recorded by moving from state A to state B. This transition is indicated on the graph by an arc originating at A and terminating at B. The label $w = 1$ on this arc denotes the input value that causes the transition. In state B the output remains at 0, which is indicated as $B/z=0$ in the node.

When the circuit is in state B, it will change to state C if w is still equal to 1 at the next active clock edge. In state C the output z becomes equal to 1. If w stays at 1 during subsequent clock cycles, the circuit will remain in state C maintaining $z = 1$. However, if w becomes 0 when the circuit is either in state B or in state C, the next active clock edge will cause a transition to state A to take place.

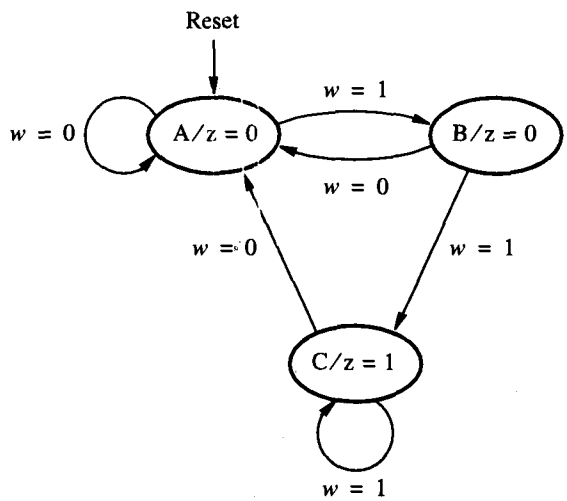


Figure 8.3 State diagram of a simple sequential circuit.

8.1.2 STATE TABLE

Although the state diagram provides a description of the behavior of a sequential circuit that is easy to understand, to proceed with the implementation of the circuit, it is convenient to translate the information contained in the state diagram into a tabular form. Figure 8.4 shows the *state table* for our sequential circuit. The table indicates all transitions from each *present state* to the *next state* for different values of the input signal. Note that the output z is specified with respect to the present state, namely, the state that the circuit is in at present time.

We now show the design steps that will produce the final circuit. To explain the basic design concepts, we first go through a traditional process of manually performing each design step. This is followed by a discussion of automated design techniques that use modern computer aided design (CAD) tools.

8.1.3 STATE ASSIGNMENT

The state table in Figure 8.4 defines the three states in terms of letters A, B, and C. When implemented in a logic circuit, each state is represented by a particular valuation (combination of values) of *state variables*. Each state variable may be implemented in the form of a flip-flop. Since three states have to be realized, it is sufficient to use two state variables. Let these variables be y_1 and y_2.

Now we can adapt the general block diagram in Figure 8.1 to our example as shown in Figure 8.5, to indicate the structure of the circuit that implements the required finite state machine. Two flip-flops represent the state variables. In the figure we have not specified the type of flip-flops to be used; this issue is addressed in the next subsection. From the specification in Figures 8.3 and 8.4, the output z is determined only by the present state of the circuit. Thus the block diagram in Figure 8.5 shows that z is a function of only y_1 and y_2; our design is of Moore type. We need to design a combinational circuit that uses y_1 and y_2 as input signals and generates a correct output signal z for all possible valuations of these inputs.

The signals y_1 and y_2 are also fed back to the combinational circuit that determines the next state of the FSM. This circuit also uses the primary input signal w. Its outputs are two signals, Y_1 and Y_2, which are used to set the state of the flip-flops. Each active edge of the clock will cause the flip-flops to change their state to the values of Y_1 and Y_2 at that

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	A	B	0
B	A	C	0
C	A	C	1

Figure 8.4 State table for the sequential circuit in Figure 8.3.

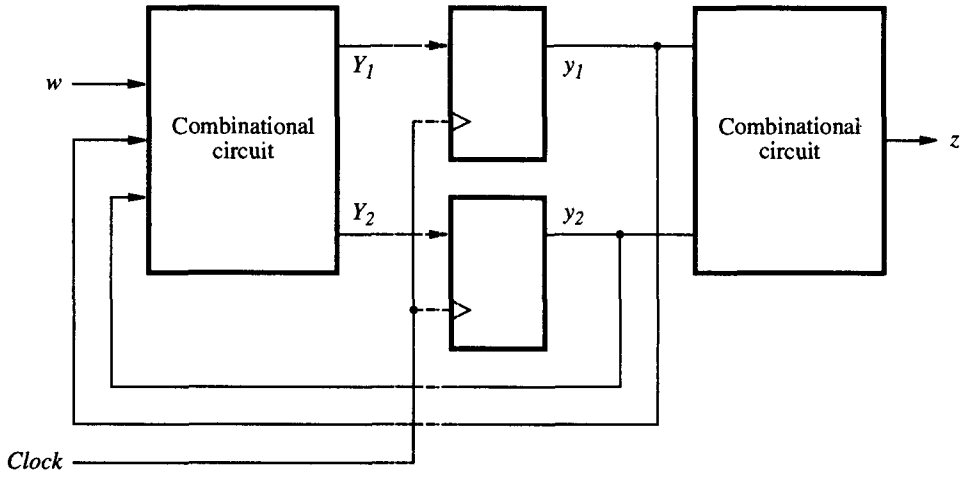


Figure 8.5 A general sequential circuit with input w, output z, and two state flip-flops.

time. Therefore, Y_1 and Y_2 are called the *next-state variables*, and y_1 and y_2 are called the *present-state variables*. We need to design a combinational circuit with inputs w, y_1, and y_2, such that for all valuations of these inputs the outputs Y_1 and Y_2 will cause the machine to move to the next state that satisfies our specification. The next step in the design process is to create a truth table that defines this circuit, as well as the circuit that generates z.

To produce the desired truth table, we assign a specific valuation of variables y_1 and y_2 to each state. One possible assignment is given in Figure 8.6, where the states A, B, and C are represented by $y_2y_1 = 00$, 01, and 10, respectively. The fourth valuation, $y_2y_1 = 11$, is not needed in this case.

The type of table given in Figure 8.6 is usually called a *state-assigned table*. This table can serve directly as a truth table for the output z with the inputs y_1 and y_2. Although for the next-state functions Y_1 and Y_2 the table does not have the appearance of a normal truth

	Present state	Next state		Output
		$w = 0$	$w = 1$	z
	y_2y_1	Y_2Y_1	Y_2Y_1	
A	00	00	01	0
B	01	00	10	0
C	10	00	10	1
	11	*dd*	*dd*	*d*

Figure 8.6 State-assigned table for the sequential circuit in Figure 8.4.

table, because there are two separate columns in the table for each value of w, it is obvious that the table includes all of the information that defines the next-state functions in terms of valuations of inputs w, y_1, and y_2.

8.1.4 CHOICE OF FLIP-FLOPS AND DERIVATION OF NEXT-STATE AND OUTPUT EXPRESSIONS

From the state-assigned table in Figure 8.6, we can derive the logic expressions for the next-state and output functions. But first we have to decide on the type of flip-flops that will be used in the circuit. The most straightforward choice is to use D-type flip-flops, because in this case the values of Y_1 and Y_2 are simply clocked into the flip-flops to become the new values of y_1 and y_2. In other words, if the inputs to the flip-flops are called D_1 and D_2, then these signals are the same as Y_1 and Y_2. Note that the diagram in Figure 8.5 corresponds exactly to this use of D-type flip-flops. For other types of flip-flops, such as JK type, the relationship between the next-state variable and inputs to a flip-flop is not as straightforward; we will consider this situation in section 8.7.

The required logic expressions can be derived as shown in Figure 8.7. We use Karnaugh maps to make it easy for the reader to verify the validity of the expressions. Recall that

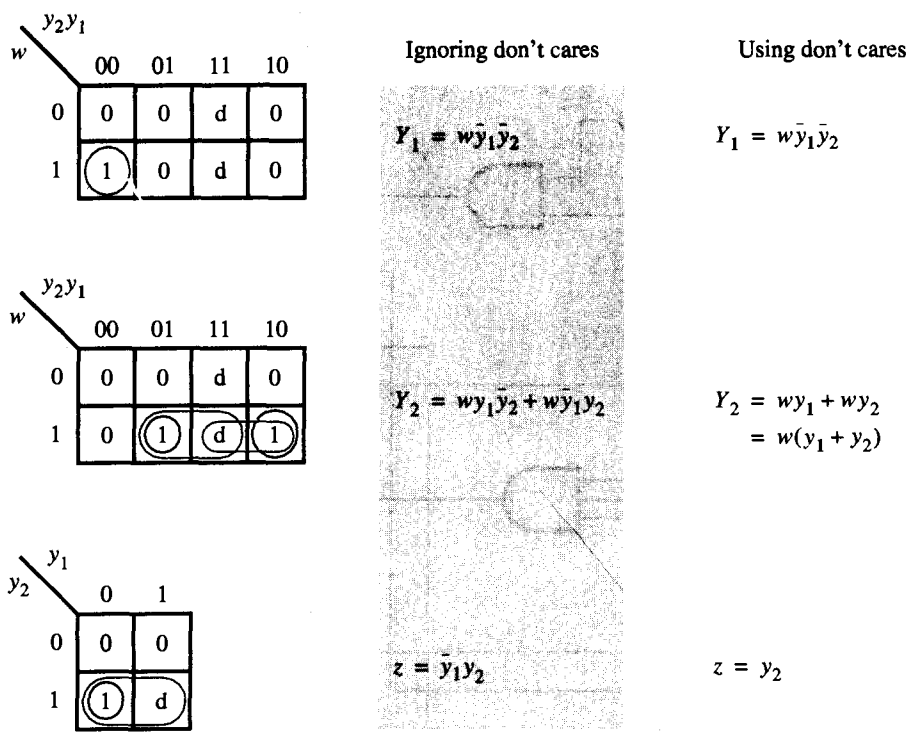


Figure 8.7 Derivation of logic expressions for the sequential circuit in Figure 8.6.

in Figure 8.6 we needed only three of the four possible binary valuations to represent the states. The fourth valuation, $y_2y_1 = 11$, should never occur in the circuit because the circuit is constrained to move only within states A, B, and C; therefore, we may choose to treat this valuation as a don't-care condition. The resulting don't-care squares in the Karnaugh maps are denoted by d's. Using the don't cares to simplify the expressions, we obtain

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = w(y_1 + y_2)$$
$$z = y_2$$

If we do not use don't cares, then the resulting expressions are slightly more complex; they are shown in the gray-shaded area of Figure 8.7.

Since $D_1 = Y_1$ and $D_2 = Y_2$, the logic circuit that corresponds to the preceding expressions is implemented as shown in Figure 8.8. Observe that a clock signal is included, and the circuit is provided with an active-low reset capability. Connecting the clear input on the

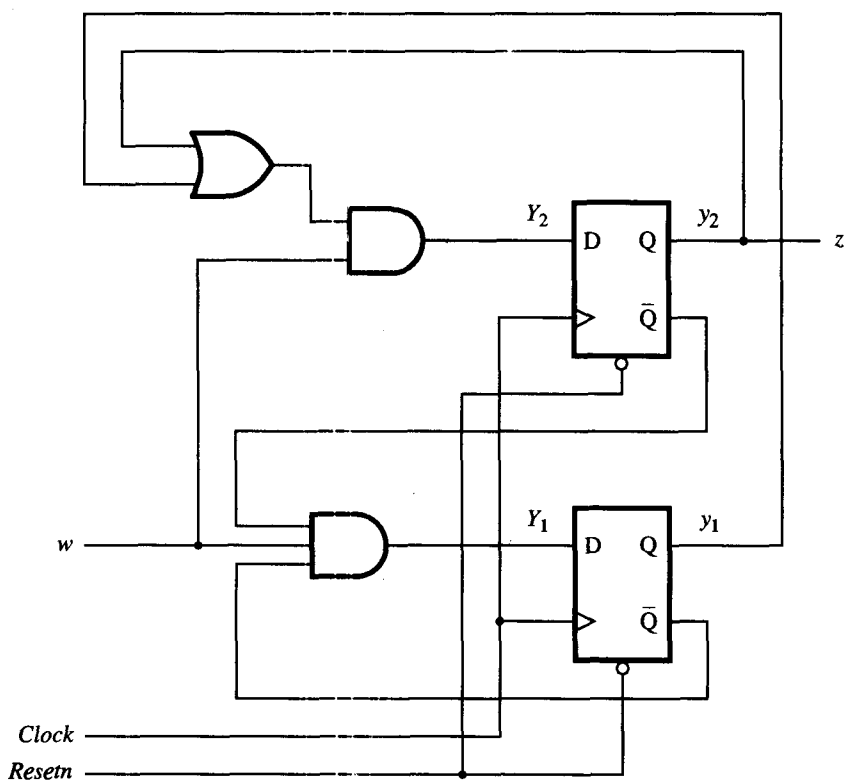


Figure 8.8 Final implementation of the sequential circuit in Figure 8.7.

flip-flops to an external *Resetn* signal, as shown in the figure, provides a simple means for forcing the circuit into a known state. If we apply the signal *Resetn* = *0* to the circuit, then both flip-flops will be cleared to 0, placing the FSM into the state $y_2y_1 = 00$.

8.1.5 TIMING DIAGRAM

To understand fully the operation of the circuit in Figure 8.8, let us consider its timing diagram presented in Figure 8.9. The diagram depicts the signal waveforms that correspond to the sequences of values in Figure 8.2.

Because we are using positive-edge-triggered flip-flops, all changes in the signals occur shortly after the positive edge of the clock. The amount of delay from the clock edge depends on the propagation delays through the flip-flops. Note that the input signal *w* is also shown to change slightly after the active edge of the clock. This is a good assumption because in a typical digital system an input such as *w* would be just an output of another circuit that is synchronized by the same clock. We discuss the synchronization of input signals with the clock signal in section 10.3.

A key point to observe is that even though *w* changes slightly after the active clock edge, and thus the value of *w* is equal to 1 (or 0) for almost the entire clock cycle, no change in the circuit will occur until the beginning of the next clock cycle when the positive edge causes the flip-flops to change their state. Thus the value of *w* must be equal to 1 for two clock cycles if the circuit is to reach state *C* and generate the output $z = 1$.

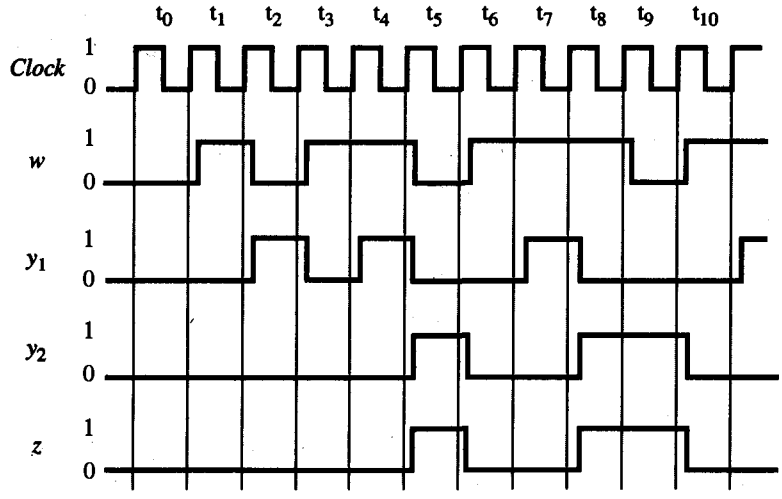


Figure 8.9 Timing diagram for the circuit in Figure 8.8.

8.1.6 SUMMARY OF DESIGN STEPS

We can summarize the steps involved in designing a synchronous sequential circuit as follows:

1. Obtain the specification of the desired circuit.
2. Derive the states for the machine by first selecting a starting state. Then, given the specification of the circuit, consider all valuations of the inputs to the circuit and create new states as needed for the machine to respond to these inputs. To keep track of the states as they are visited, create a state diagram. When completed, the state diagram shows all states in the machine and gives the conditions under which the circuit moves from one state to another.
3. Create a state table from the state diagram. Alternatively, it may be convenient to directly create the state table in step 2, rather than first creating a state diagram.
4. In our sequential circuit example, there were only three states; hence it was a simple matter to create the state table that does not contain more states than necessary. However, in practice it is common to deal with circuits that have a large number of states. In such cases it is unlikely that the first attempt at deriving a state table will produce optimal results. Almost certainly we will have more states than is really necessary. This can be corrected by a procedure that minimizes the number of states. We will discuss the process of state minimization in section 8.6.
5. Decide on the number of state variables needed to represent all states and perform the state assignment. There are many different state assignments possible for a given sequential circuit. Some assignments may be better than others. In the preceding example we used what seemed to be a natural state assignment. We will return to this example in section 8.2 and show that a different assignment may lead to a simpler circuit.
6. Choose the type of flip-flops to be used in the circuit. Derive the next-state logic expressions to control the inputs to all flip-flops and then derive logic expressions for the outputs of the circuit. So far we have used only D-type flip-flops. We will consider other types of flip-flops in section 8.7.
7. Implement the circuit as indicated by the logic expressions.

Example 8.1 We have illustrated the design steps using a very simple sequential circuit. From the reader's point of view, a circuit that detects that an input signal was high for two consecutive clock pulses may not have much practical significance. We will now consider an example that is closely tied to practical application.

Section 7.14 introduced the concept of a bus and showed the connections that have to be made to allow the contents of a register to be transferred into another register. The circuit in Figure 7.60 shows how tri-state buffers can be used to place the contents of a selected register onto the bus and how the data on the bus can be loaded into a register. Figure 7.62 shows how a control mechanism that swaps the contents of registers $R1$ and $R2$ can be realized using a shift register. We will now design the desired control mechanism, using the finite state machine approach.

The contents of registers $R1$ and $R2$ can be swapped using register $R3$ as a temporary storage location as follows: The contents of $R2$ are first loaded into $R3$, using the control signals $R2_{out} = 1$ and $R3_{in} = 1$. Then the contents of $R1$ are transferred into $R2$, using $R1_{out} = 1$ and $R2_{in} = 1$. Finally, the contents of $R3$ (which are the previous contents of $R2$) are transferred into $R1$, using $R3_{out} = 1$ and $R1_{in} = 1$. Since this step completes the required swap, we will indicate that the task is completed by setting the signal $Done = 1$. Assume that the swapping is performed in response to a pulse on an input signal called w, which has a duration of one clock cycle. Figure 8.10 indicates the external signals involved in the desired control circuit. Figure 8.11 gives a state diagram for a sequential circuit that generates the output control signals in the required sequence. Note that to keep the diagram simple, we have indicated the output signals only when they are equal to 1. In all other cases the output signals are equal to 0.

In the starting state, A, no transfer is indicated, and all output signals are 0. The circuit remains in this state until a request to swap arrives in the form of w changing to 1. In state B the signals required to transfer the contents of $R2$ into $R3$ are asserted. The next active clock edge places these contents into $R3$. It also causes the circuit to change to state C, regardless of whether w is equal to 0 or 1. In this state the signals for transferring $R1$ into $R2$ are asserted. The transfer takes place at the next active clock edge, and the circuit changes to state D regardless of the value of w. The final transfer, from $R3$ to $R1$, is performed on the clock edge that leaves state D, which also causes the circuit to return to state A.

Figure 8.12 presents the same information in a state table. Since there are four states, it is necessary to use two state variables, y_2 and y_1. A straightforward state assignment where the states A, B, C, and D are assigned the valuations $y_2y_1 = 00$, 01, 10, and 11, respectively, leads to the state-assigned table in Figure 8.13. Using this assignment and D-type flip-flops, the next-state expressions can be derived as shown in Figure 8.14. They are

$$Y_1 = w\bar{y}_1 + \bar{y}_1y_2$$
$$Y_2 = y_1\bar{y}_2 + \bar{y}_1y_2$$

The output control signals are derived as

$$R1_{out} = R2_{in} = \bar{y}_1y_2$$
$$R1_{in} = R3_{out} = Done = y_1y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

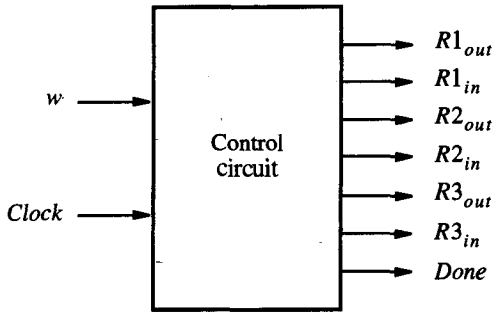


Figure 8.10 Signals needed in Example 8.1.

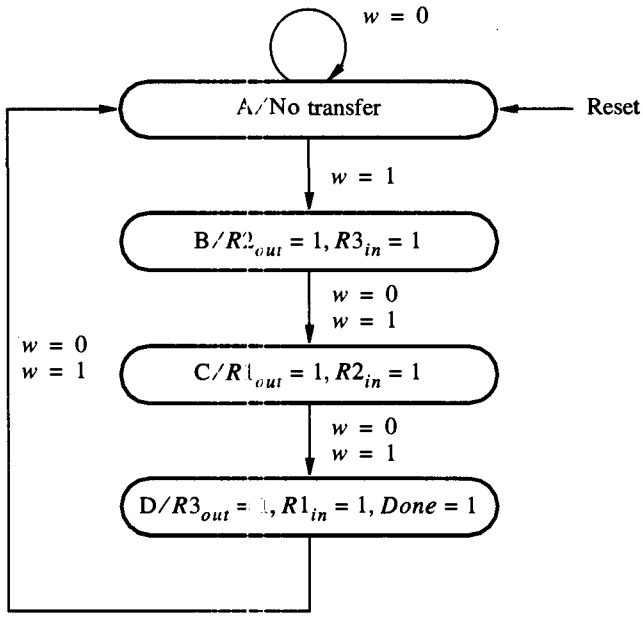


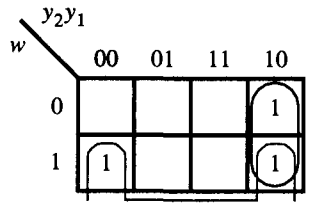
Figure 8.11 State diagram for Example 8.1.

Present state	Next state		Outputs						
	$w = 0$	$w = 1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	*Done*
A	A	B	0	0	0	0	0	0	0
B	C	C	0	0	1	0	0	1	0
C	D	D	1	0	0	1	0	0	0
D	A	A	0	1	0	0	1	0	1

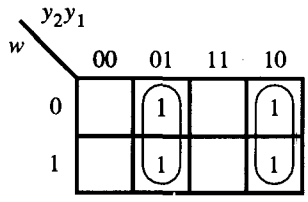
Figure 8.12 State table for Example 8.1.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	*Done*
A	0 0	0 0	0 1	0	0	0	0	0	0	0
B	0 1	1 0	1 0	0	0	1	0	0	1	0
C	1 0	1 1	1 1	1	0	0	1	0	0	0
D	1 1	0 0	0 0	0	1	0	0	1	0	1

Figure 8.13 State-assigned table for the sequential circuit in Figure 8.12.



$$Y_1 = w\bar{y}_1 + \bar{y}_1 y_2$$



$$Y_2 = y_1\bar{y}_2 + \bar{y}_1 y_2$$

Figure 8.14 Derivation of next-state expressions for the sequential circuit in Figure 8.13.

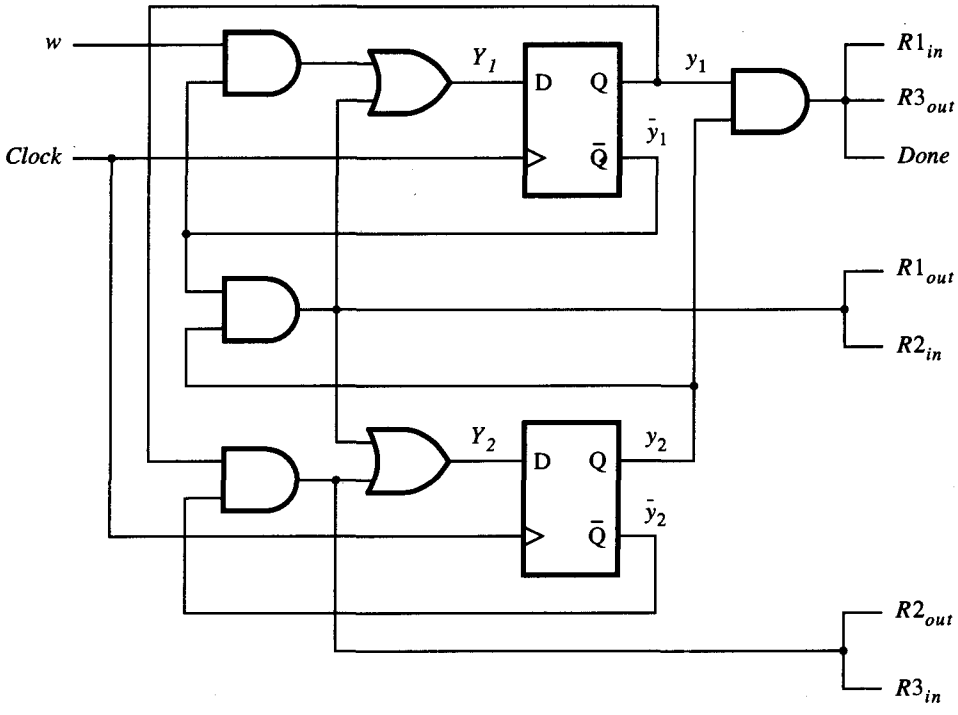


Figure 8.15 Final implementation of the sequential circuit in Figure 8.13.

These expressions lead to the circuit in Figure 8.15. This circuit appears more complex than the shift register in Figure 7.62, but it has only two flip-flops, rather than three.

8.2 STATE-ASSIGNMENT PROBLEM

Having introduced the basic concepts involved in the design of sequential circuits, we should revisit some details where alternative choices are possible. In section 8.1.6 we suggested that some state assignments may be better than others. To illustrate this we can reconsider the example in Figure 8.4. We already know that the state assignment in Figure 8.6 leads to a simple-looking circuit in Figure 8.8. But can the FSM of Figure 8.4 be implemented with an even simpler circuit by using a different state assignment?

Figure 8.16 gives one possible alternative. In this case we represent the states A, B, and C with the valuations $y_2y_1 = 00$, 01, and 11, respectively. The remaining valuation, $y_2y_1 = 10$, is not needed, and we will treat it as a don't-care condition. If we again choose to implement the circuit using D-type flip-flops, the next-state and output expressions derived from the figure will be

$$Y_1 = D_1 = w$$
$$Y_2 = D_2 = wy_1$$
$$z = y_2$$

These expressions define the circuit shown in Figure 8.17. Comparing this circuit with the one in Figure 8.8, we see that the cost of the new circuit is lower because it requires fewer gates.

In general, circuits are much larger than our example, and different state assignments can have a substantial effect on the cost of the final implementation. While highly desirable, it is often impossible to find the best state assignment for a large circuit. The exhaustive approach of trying all possible state assignments is not practical because the number of available state assignments is huge. CAD tools usually perform the state assignment using heuristic techniques. These techniques are usually proprietary, and their details are seldom published.

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	y_2y_1	Y_2Y_1	Y_2Y_1	z
A	00	00	01	0
B	01	00	11	0
C	11	00	11	1
	10	*dd*	*dd*	*d*

Figure 8.16 Improved state assignment for the sequential circuit in Figure 8.4.

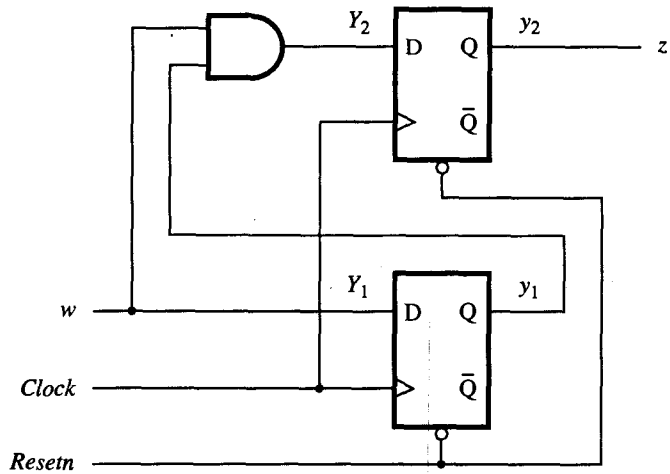


Figure 8.17 Final circuit for the improved state assignment in Figure 8.16.

Example 8.2

In Figure 8.13 we used a straightforward state assignment for the sequential circuit in Figure 8.12. Consider now the effect of interchanging the valuations assigned to states C and D, as shown in Figure 8.18. Then the next-state expressions are

$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$
$$Y_2 = y_1$$

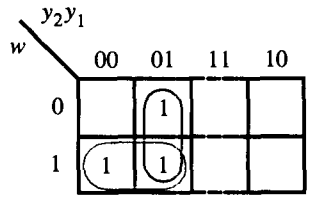
as derived in Figure 8.19. The output expressions are

$$R1_{out} = R2_{in} = y_1y_2$$
$$R1_{in} = R3_{out} = Done = \bar{y}_1y_2$$
$$R2_{out} = R3_{in} = y_1\bar{y}_2$$

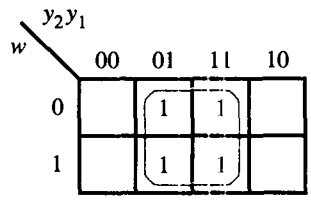
These expressions lead to a slightly simpler circuit than the one given in Figure 8.15.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	y_2y_1	Y_2Y_1	Y_2Y_1	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	00	00	01	0	0	0	0	0	0	0
B	01	11	11	0	0	1	0	0	1	0
C	11	10	10	1	0	0	1	0	0	0
D	10	00	00	0	1	0	0	1	0	1

Figure 8.18 Improved state assignment for the sequential circuit in Figure 8.12.



$$Y_1 = w\bar{y}_2 + y_1\bar{y}_2$$



$$Y_2 = y_1$$

Figure 8.19 Derivation of next-state expressions for the sequential circuit in Figure 8.18.

8.2.1 ONE-HOT ENCODING

Another interesting possibility is to use as many state variables as there are states in a sequential circuit. In this method, for each state all but one of the state variables are equal to 0. The variable whose value is 1 is deemed to be "hot." The approach is known as the *one-hot encoding* method.

Figure 8.20 shows how one-hot state assignment can be applied to the sequential circuit of Figure 8.4. Because there are three states, it is necessary to use three state variables. The chosen assignment is to represent the states A, B, and C using the valuations $y_3y_2y_1 = 001$, 010, and 100, respectively. The remaining five valuations of the state variables are not used. They can be treated as don't cares in the derivation of the next-state and output expressions. Using this assignment, the resulting expressions are

$$Y_1 = \overline{w}$$
$$Y_2 = wy_1$$
$$Y_3 = w\bar{y}_1$$
$$z = y_3$$

These expressions are not simpler than those obtained using the state assignment in Figure 8.16. Although in this case the one-hot assignment is not advantageous, there are many cases where this approach is attractive.

Example 8.3 The one-hot state assignment can be applied to the sequential circuit of Figure 8.12 as indicated in Figure 8.21. Four state variables are needed, and the states A, B, C, and D are

	Present state	Next state		Output
		$w = 0$	$w = 1$	
	$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	z
A	001	001	010	0
B	010	001	100	0
C	100	001	100	1

Figure 8.20 One-hot state assignment for the sequential circuit in Figure 8.4.

encoded as $y_4y_3y_2y_1 = 0001$, 0010, 0100, and 1000, respectively. Treating the remaining 12 valuations of the state variables as don't cares, the next-state expressions are

$$Y_1 = \overline{w}y_1 + y_4$$
$$Y_2 = wy_1$$
$$Y_3 = y_2$$
$$Y_4 = y_3$$

It is instructive to note that we can derive these expressions simply by inspecting the state diagram in Figure 8.11. Flip-flop y_1 should be set to 1 if the FSM is in state A and $w = 0$, or if the FSM is in state D; hence $Y_1 = \overline{w}y_1 + y_4$. Flip-flop y_2 should be set to 1 if the present state is A and $w = 1$; hence $Y_2 = wy_1$. Flip-flops y_3 and y_4 should be set to 1 if the FSM is presently in state B or C, respectively; hence $Y_3 = y_2$ and $Y_4 = y_3$.

The output expressions are just the outputs of the flip-flops, such that

$$R1_{out} = R2_{in} = y_3$$
$$R1_{in} = R3_{out} = Done = y_4$$
$$R2_{out} = R3_{in} = y_2$$

These expressions are simpler than those derived in Example 8.2, but four flip-flops are needed, rather than two.

	Present state	Next state		Outputs						
		$w = 0$	$w = 1$							
	$y_4y_3y_2y_1$	$Y_4Y_3Y_2Y_1$	$Y_4Y_3Y_2Y_1$	$R1_{out}$	$R1_{in}$	$R2_{out}$	$R2_{in}$	$R3_{out}$	$R3_{in}$	$Done$
A	0001	0001	0010	0	0	0	0	0	0	0
B	0010	0100	0100	0	0	1	0	0	1	0
C	0100	1000	1000	1	0	0	1	0	0	0
D	1000	0001	0001	0	1	0	0	1	0	1

Figure 8.21 One-hot state assignment for the sequential circuit in Figure 8.12.

An important feature of the one-hot state assignment is that it often leads to simpler output expressions than do assignments with the minimal number of state variables. Simpler output expressions may lead to a faster circuit. For instance, if the outputs of the sequential circuit are just the outputs of the flip-flops, as is the case in our example, then these output signals are valid as soon as the flip-flops change their states. If more complex output expressions are involved, then the propagation delay through the gates that implement these expressions must be taken into account. We will consider this issue in section 8.8.2.

The examples considered to this point show that there are many ways to implement a given finite state machine as a sequential circuit. Each implementation is likely to have a different cost and different timing characteristics. In the next section we introduce another way of modeling FSMs that leads to even more possibilities.

8.3 MEALY STATE MODEL

Our introductory examples were sequential circuits in which each state had specific values of the output signals associated with it. As we explained at the beginning of the chapter, such finite state machines are said to be of Moore type. We will now explore the concept of Mealy-type machines in which the output values are generated based on both the state of the circuit and the present values of its inputs. This provides additional flexibility in the design of sequential circuits. We will introduce the Mealy-type machines, using a slightly altered version of a previous example.

The essence of the first sequential circuit in section 8.1 is to generate an output $z = 1$ whenever a second occurrence of the input $w = 1$ is detected in consecutive clock cycles. The specification requires that the output z be equal to 1 in the clock cycle that follows the detection of the second occurrence of $w = 1$. Suppose now that we eliminate this latter requirement and specify instead that the output z should be equal to 1 in the same clock cycle when the second occurrence of $w = 1$ is detected. Then a suitable input-output sequence may be as shown in Figure 8.22. To see how we can realize the behavior given in this table, we begin by selecting a starting state, A. As long as $w = 0$, the machine should remain in state A, producing an output $z = 0$. When $w = 1$, the machine has to move to a new state, B, to record the fact that an input of 1 has occurred. If w remains equal to 1 when the machine is in state B, which happens if $w = 1$ for at least two consecutive clock cycles, the machine should remain in state B and produce an output $z = 1$. As soon as w becomes 0, z should immediately become 0 and the machine should move back to state A at the next

Clock cycle:	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
w:	0	1	0	1	1	0	1	1	1	0	1
z:	0	0	0	0	1	0	0	1	1	0	0

Figure 8.22 Sequences of input and output signals.

active edge of the clock. Thus the behavior specified in Figure 8.22 can be achieved with a two-state machine, which has a state diagram shown in Figure 8.23. Only two states are needed because we have allowed the output value to depend on the present value of the input as well as the present state of the machine. The diagram indicates that if the machine is in state A, it will remain in state A if $w = 0$ and the output will be 0. This is indicated by an arc with the label $w = 0/z = 0$. When w becomes 1, the output stays at 0 until the machine moves to state B at the next active clock edge. This is denoted by the arc from A to B with the label $w = 1/z = 0$. In state B the output will be 1 if $w = 1$, and the machine will remain in state B, as indicated by the label $w=1/z=1$ on the corresponding arc. However, if $w = 0$ in state B, then the output will be 0 and a transition to state A will take place at the next active clock edge. A key point to understand is that during the present clock cycle the output value corresponds to the label on the arc emanating from the present-state node.

We can implement the FSM in Figure 8.23, using the same design steps as in section 8.1. The state table is shown in Figure 8.24. The table shows that the output z depends on the present value of input w and not just on the present state. Figure 8.25 gives the state-assigned table. Because there are only two states, it is sufficient to use a single state variable, y. Assuming that y is realized as a D-type flip-flop, the required next-state and output expressions are

$$Y = D = w$$
$$z = wy$$

The resulting circuit is presented in Figure 8.26 along with a timing diagram. The timing diagram corresponds to the input-output sequences in Figure 8.22.

The greater flexibility of Mealy-type FSMs often leads to simpler circuit realizations. This certainly seems to be the case in our examples that produced the circuits in Figures 8.8, 8.17, and 8.26, assuming that the design requirement is only to detect two consecutive occurrences of input w being equal to 1. We should note, however, that the circuit in Figure 8.26 is not the same in terms of output behavior as the circuits in Figures 8.8 and 8.17. The difference is a shift of one clock cycle in the output signal in Figure 8.26*b*. If we wanted to produce exactly the same output behavior using the Mealy approach, we could modify the circuit in Figure 8.26*a* by adding another flip-flop as shown in Figure 8.27. This flip-flop merely delays the output signal, Z, by one clock cycle with respect to z, as indicated in the timing diagram. By making this change, we effectively turn the Mealy-type circuit into

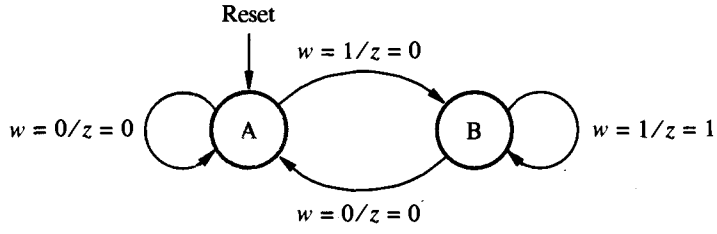


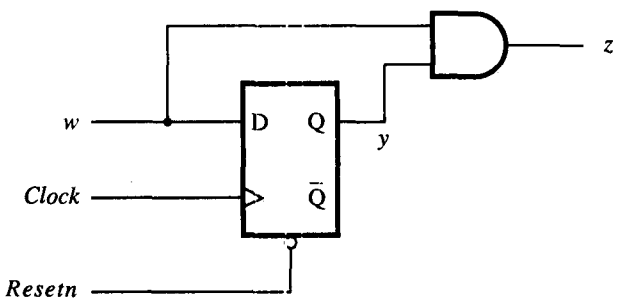
Figure 8.23 State diagram of an FSM that realizes the task in Figure 8.22.

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	A	B	0	1

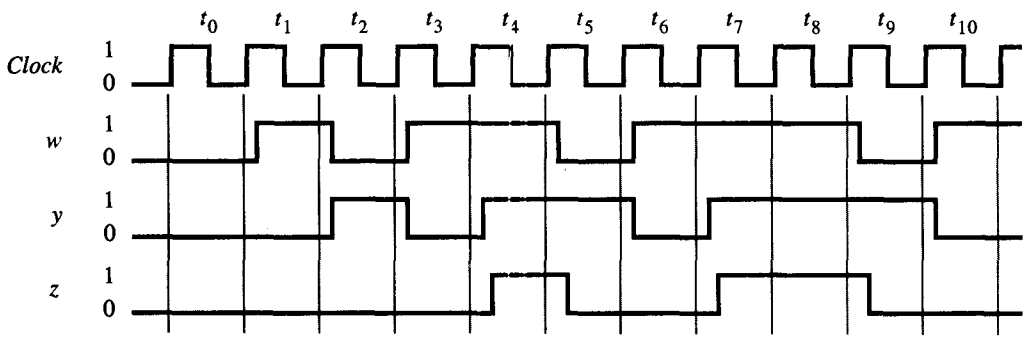
Figure 8.24 State table for the FSM in Figure 8.23.

	Present state	Next state		Output	
		$w = 0$	$w = 1$	$w = 0$	$w = 1$
	y	Y	Y	z	z
A	0	0	1	0	0
B	1	0	1	0	1

Figure 8.25 State-assigned table for the FSM in Figure 8.24.

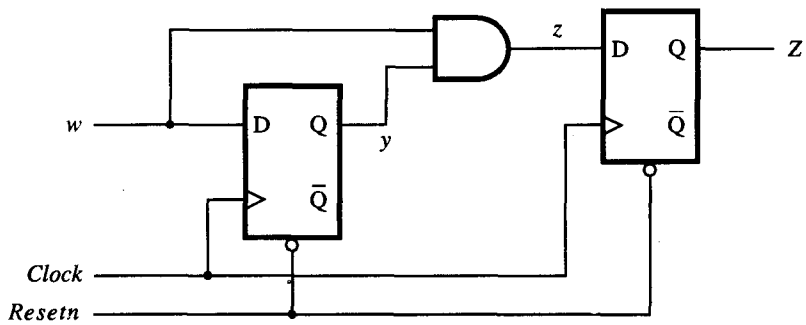


(a) Circuit

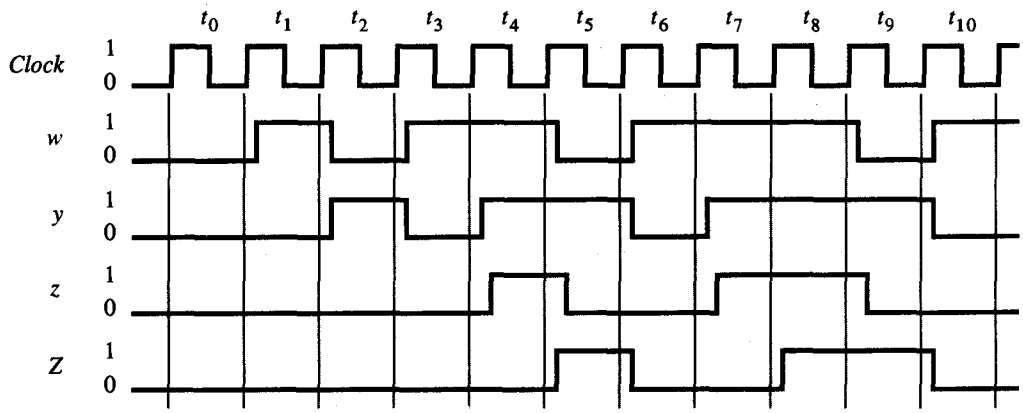


(b) Timing diagram

Figure 8.26 Implementation of FSM in Figure 8.25.



(a) Circuit



(b) Timing diagram

Figure 8.27 Circuit that implements the specification in Figure 8.2.

a Moore-type circuit with output Z. Note that the circuit in Figure 8.27 is essentially the same as the circuit in Figure 8.17.

Example 8.4

In Example 8.1 we considered the control circuit needed to swap the contents of two registers, implemented as a Moore-type finite state machine. The same task can be achieved using a Mealy-type FSM, as indicated in Figure 8.28. State A still serves as the reset state. But as soon as w changes from 0 to 1, the output control signals $R2_{out}$ and $R3_{in}$ are asserted. They remain asserted until the beginning of the next clock cycle, when the circuit will leave state A and change to B.. In state B the outputs $R1_{out}$ and $R2_{in}$ are asserted for both $w = 0$ and $w = 1$. Finally, in state C the swap is completed by asserting $R3_{out}$ and $R1_{in}$.

The Mealy-type realization of the control circuit requires three states. This does not necessarily imply a simpler circuit because two flip-flops are still needed to implement

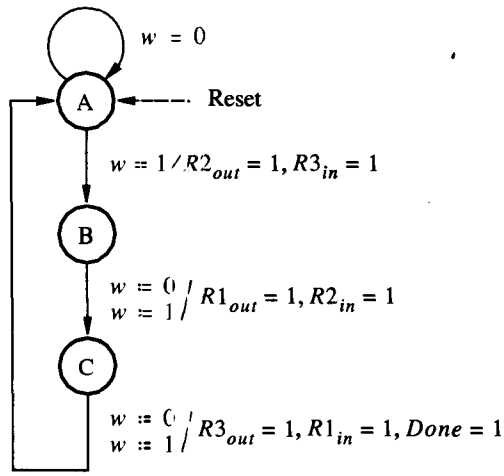


Figure 8.28 State diagram for Example 8.4.

the state variables. The most important difference in comparison with the Moore-type realization is the timing of output signals. A circuit that implements the FSM in Figure 8.28 generates the output control signals one clock cycle sooner than the circuits derived in Examples 8.1 and 8.2.

Note also that using the FSM in Figure 8.28, the entire process of swapping the contents of $R1$ and $R2$ takes three clock cycles, starting and finishing in state A. Using the Moore-type FSM in Example 8.1, the swapping process involves four clock cycles before the circuit returns to state A.

Suppose that we wish to implement this FSM using one-hot encoding. Then three flip-flops are needed, and the states A, B, and C may be assigned the valuations $y_3y_2y_1 = 001$, 010, and 100, respectively. Examining the state diagram in Figure 8.28, we can derive the next-state equations by inspection. The input to flip-flop y_1 should have the value 1 if the FSM is in state A and $w = 0$ or if the FSM is in state C; hence $Y_1 = \overline{w}y_1 + y_3$. Flip-flop y_2 should be set to 1 if the FSM is in state A and $w = 1$; hence $Y_2 = wy_1$. Flip-flop y_3 should be set to 1 if the present state is B; hence $Y_3 = y_2$. The derivation of the output expressions, which we leave as an exercise for the reader, can also be done by inspection. The corresponding circuit is shown in Figure 7.63, in section 7.14, where it was derived using an ad hoc approach.

The preceding discussion deals with the basic principles involved in the design of sequential circuits. Although it is essential to understand these principles, the manual approach used in the examples is difficult and tedious when large circuits are involved. We will now show how CAD tools are used to greatly simplify the design task.

8.4 DESIGN OF FINITE STATE MACHINES USING CAD TOOLS

Sophisticated CAD tools are available for finite state machine design, and we introduce them in this section. A rudimentary way of using CAD tools for FSM design could be as follows: The designer employs the manual techniques described previously to derive a circuit that contains flip-flops and logic gates from a state diagram. This circuit is entered into the CAD system by drawing a schematic diagram or by writing structural hardware description language (HDL) code. The designer then uses the CAD system to simulate the behavior of the circuit and uses the CAD tools to automatically implement the circuit in a chip, such as a PLD.

It is tedious to manually synthesize a circuit from a state diagram. Since CAD tools are meant to obviate this type of task, more attractive ways of utilizing CAD tools for FSM design have been developed. A better approach is to directly enter the state diagram into the CAD system and perform the entire synthesis process automatically. CAD tools support this approach in two main ways. One method is to allow the designer to draw the state diagram using a graphical tool similar to the schematic capture tool. The designer draws circles to represent states and arcs to represent state transitions and indicates the outputs that the machine should generate. Another and more popular approach is to use an HDL to write code that represents the state diagram, as described below.

Many HDLs provide constructs that allow the designer to represent a state diagram. To show how this is done, we will provide Verilog code that represents the simple machine designed manually as the first example in section 8.1. Then we will use the CAD tools to synthesize a circuit that implements the machine in a chip.

8.4.1 VERILOG CODE FOR MOORE-TYPE FSMS

Verilog does not define a standard way of describing a finite state machine. Hence while adhering to the required Verilog syntax, there is more than one way to describe a given FSM. An example of Verilog code for the FSM of Figure 8.3 is given in Figure 8.29. The code reflects directly the FSM structure in Figure 8.5. The module *simple* has inputs *Clock*, *Resetn*, and w, and output z. Two-bit vectors y and Y represent the present and the next state of the machine, respectively. The state values are assigned in the **parameter** statement to match those in Figure 8.6.

The state transitions are specified by two separate **always** blocks. The first block describes the required combinational circuit. The values of the next state vector Y change in response to the changes in w and y signals, which are given in the sensitivity list. The circuit is defined in a **case** statement to give the value of Y for each value of y and w. Each case alternative corresponds to a present state of the machine, and the associated **if-else** statement specifies the next state to be reached. Since there are only three states specified as possible alternatives, we have included a **default** clause which indicates to the Verilog compiler that the unassigned fourth state can be treated as a don't care condition as depicted in Figure 8.6. This portion of the code corresponds to the combinational circuit on the left side of Figure 8.5.

```
module simple (Clock, Resetn, w, z);
  input Clock, Resetn, w;
  output z;
  reg [2:1] y, Y;
  parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

  // Define the next state combinational circuit
  always @(w or y)
    case (y)
      A: if (w)   Y = B;
         else     Y = A;
      B: if (w)   Y = C;
         else     Y = A;
      C: if (w)   Y = C;
         else     Y = A;
      default:    Y = 2'bxx;
    endcase

  // Define the sequential block
  always @(negedge Resetn or posedge Clock)
    if (Resetn == 0)  y <= A;
    else  y <= Y;

  // Define output
  assign  z = (y == C);

endmodule
```

Figure 8.29 Verilog code for the FSM in Figure 8.3.

The second **always** block introduces flip-flops into the circuit. Its sensitivity list comprises the reset and clock signals. Asynchronous reset is performed when the *Resetn* input goes to 0, placing the FSM into state *A*. The **else** clause stipulates that after each positive clock edge the *y* signal should take the value of the *Y* signal, thus implementing the state changes.

This is a Moore-type FSM in which the output z is equal to 1 only in state C. The output is conveniently defined in a conditional assignment statement that sets $z = 1$ if $y = C$. This realizes the combinational circuit on the right side of Figure 8.5.

8.4.2 SYNTHESIS OF VERILOG CODE

To give an example of the circuit produced by a synthesis tool, we synthesised the code in Figure 8.29 for implementation in a CPLD. The synthesis resulted in two flip-flops, with inputs Y_1 and Y_2, and outputs y_1 and y_2. The next-state expressions generated by the

synthesis tool are

$$Y_1 = w\bar{y}_1\bar{y}_2$$
$$Y_2 = wy_1 + wy_2$$

The output expression is

$$z = y_2$$

These expressions correspond to the case in Figure 8.7 when the unused state pattern $y_2y_1 = 11$ is treated as don't-cares in the Karnaugh maps for Y_1, Y_2, and z.

Figure 8.30 depicts a part of the FSM circuit implemented in a CPLD. To keep the figure simple, only the logic resources used for the two macrocells that implement y_1, y_2, and z are shown. The parts of the macrocells used for the circuit are highlighted in blue.

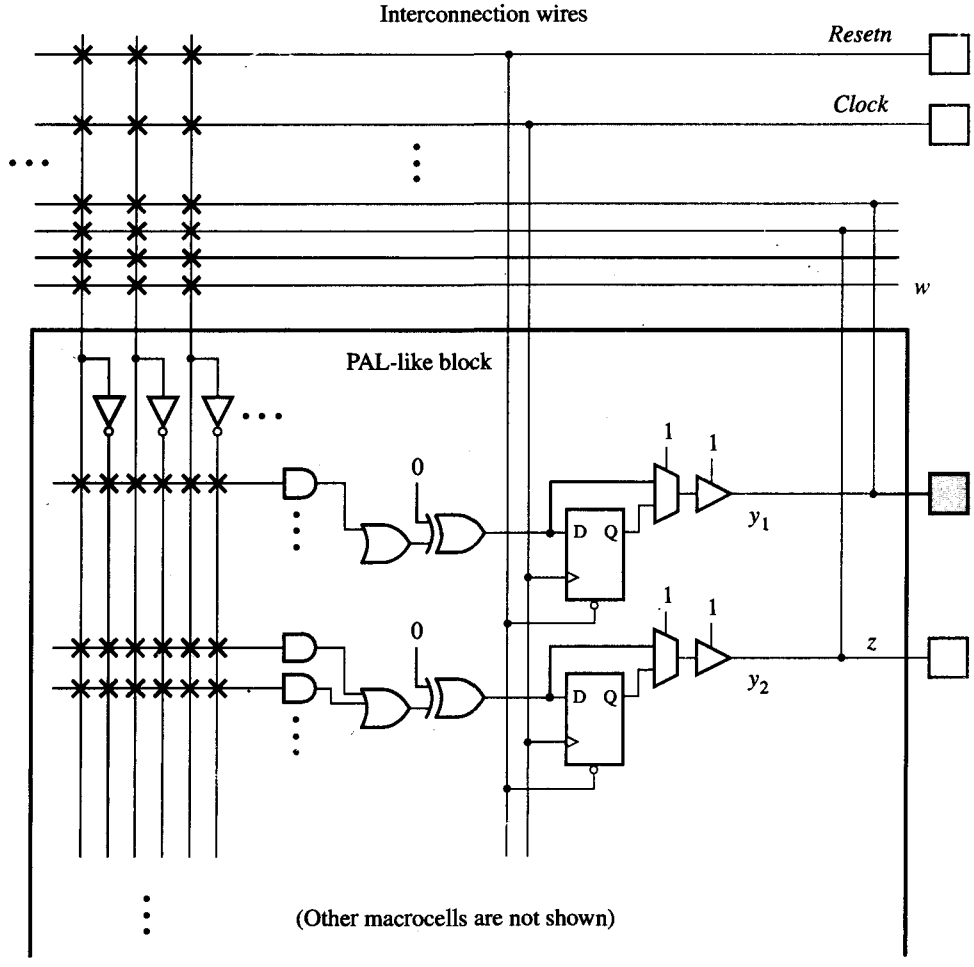


Figure 8.30 Implementation of the FSM of Figure 8.3 in a CPLD.

The w input to the circuit is shown connected to one of the interconnection wires in the CPLD. The source node in the chip that generates w is not shown. It could be either an input pin, or else w might be the output of another macrocell, assuming that the CPLD may contain other circuitry that is connected to our FSM. The *Clock* signal is assigned to a pin on the chip that is dedicated for use by clock signals. From this dedicated pin a *global wire* distributes the clock signal to all of the flip-flops in the chip. The global wire distributes the clock signal to the flip-flops such that the difference in the arrival time, or *clock skew*, of the clock signal at each flip-flop is minimized. The concept of clock skew is discussed in section 10.3. A global wire is also used for the reset signal.

The top macrocell in Figure 8.30 produces the state variable y_1. The other macrocell generates y_2. For signal y_1 the top macrocell produces the required product term, as shown. The other product-term wires in the macrocell are not shown in the figure, but each is set to 0 so that it does not affect the OR gate. The output of the OR gate passes through the XOR gate whose other input is 0. Although the XOR gate has no impact on this circuit's behavior, except to cause a small propagation delay, it is a part of the macrocell and cannot be avoided when implementing our circuit. The output of the XOR gate drives the flip-flop that represents y_1. The multiplexer select input is set to 1 so that the signal y_1 is passed through to the tri-state buffer. Similar to the XOR gate, this buffer is not needed in our circuit, but since it is present in the macrocell it must be used; hence its output enable control signal is set to 1. The signal y_1 is connected to the interconnection wires in the CPLD and fed back to the macrocells. Observe that although y_1 is not an output of the circuit, it uses a signal path that is attached to one of the chip's pins. Therefore, this pin cannot be used for any other purpose. The implementation of y_2 is similar to that for y_1, except that two product terms are involved. The signal y_2 is connected to the pin labeled z, which realizes the required output signal.

Figure 8.31 illustrates how the circuit might be assigned to the pins on a small CPLD in a 44-pin PLCC package. The figure is drawn with a part of the top of the chip package cut away, revealing a conceptual view of the two macrocells from Figure 8.30, which are indicated in blue. Our simple circuit uses only a small portion of the device.

8.4.3 SIMULATING AND TESTING THE CIRCUIT

The behavior of the circuit implemented in the CPLD chip can be tested using timing simulation, as depicted in Figure 8.32. The figure gives the waveforms that correspond to the timing diagram in Figure 8.9, assuming that a 100 ns clock period is used. The *Resetn* signal is set to 0 at the beginning of the simulation and then set to 1. The circuit produces the output $z = 1$ for one clock cycle after w has been equal to 1 for two successive clock cycles. When w is 1 for three clock cycles, z becomes 1 for two clock cycles, as it should be. We show the changes in state by using the letters A, B, and C for readability purposes. (The simulator included with the book actually shows the corresponding binary codes for the states.)

Having examined the simulation output, we should consider the question of whether we can conclude that the circuit functions correctly and satisfies all requirements. For our simple example it is not difficult to answer this question because the circuit has only one input and its behavior is straightforward. It is easy to see that the circuit works properly. However, in general it is difficult to ascertain with a high degree of confidence whether a

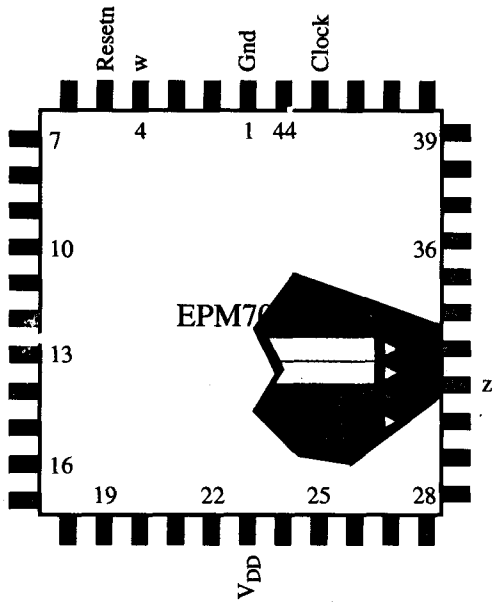


Figure 8.31 The circuit from Figure 8.30 in a small CPLD.

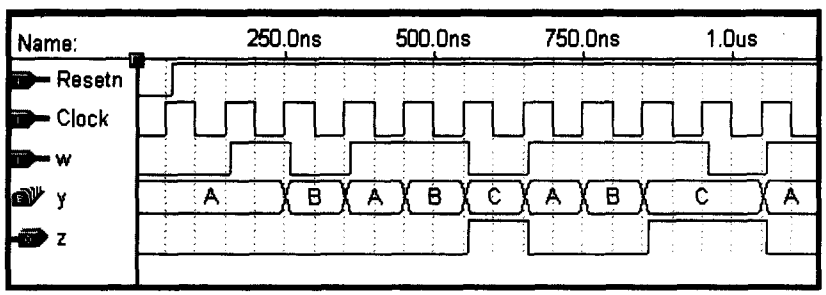


Figure 8.32 Simulation results for the circuit in Figure 8.30.

sequential circuit will work properly for all possible input sequences, because a very large number of input patterns may be possible. For large finite state machines, the designer must think carefully about patterns of inputs that may be used in simulation for testing purposes.

8.4.4 ALTERNATIVE STYLES OF VERILOG CODE

We mentioned earlier in this section that Verilog does not specify a standard way for writing code that represents a finite state machine. The code given in Figure 8.29 is only one possibility. A slightly different version of code for our simple machine is given in Figure 8.33. In this case, we specified the output z inside the **always** block that defines the required combinational circuit. The effect is the same as in Figure 8.29.

```
module simple (Clock, Resetn, w, z);
  input Clock, Resetn, w;
  output z;
  reg z;
  reg [2:1] y, Y;
  parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

  // Define the next state and output combinational circuits
  always @(w or y)
  begin
    case (y)
      A: if (w)    Y = B;
         else      Y = A;
      B: if (w)    Y = C;
         else      Y = A;
      C: if (w)    Y = C;
         else      Y = A;
      default:     Y = 2'bxx;
    endcase
    z = (y == C);  //Define output
  end

  // Define the sequential block
  always @(negedge Resetn or posedge Clock)
    if (Resetn == 0)  y <= A;
    else  y <= Y;

endmodule
```

Figure 8.33 Second version of code for the FSM in Figure 8.3.

A different approach is taken in Figure 8.34. A single **always** block is used. The states are represented by a two-bit vector y. The required state transitions are given in the **always** block with the sensitivity list that comprises the reset and clock signals. Asynchronous reset is specified when *Resetn* goes to 0. Other transitions are defined in the **case** statement to correspond directly to those in Figure 8.3. The **default** clause indicates that the valuation $y = y_2y_1 = 11$ can be treated as a don't-care condition.

The assignment statement that defines z is placed outside the **always** block. This leads to the correct operation as follows. The **always** block implements all changes on the positive edge of the clock. This infers flip-flops for the state variables y_1 and y_2. The **assign** statement makes z equal to 1 as soon as the circuit reaches state C, which is when $y_2y_1 = 10$. Using the don't care condition, this statement is equivalent to saying that $z = y_2$. Observe that this assignment cannot be made inside the **always** block as done in Figure 8.33. Doing so would infer a separate flip-flop for z, which would delay the changes in z with respect to y_2 by one clock cycle.

We have shown three styles of Verilog code for our FSM example. The circuit produced by the Verilog compiler for each version of the code may be somewhat different because,

```
module simple (Clock, Resetn, w, z);
  input Clock, Resetn, w;
  output z;
  reg [2:1] y;
  parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

  // Define the sequential block
  always @(negedge Resetn or posedge Clock)
    if (Resetn == 0)  y <= A;
    else
      case (y)
        A:  if (w)    y <= B;
            else      y <= A;
        B:  if (w)    y <= C;
            else      y <= A;
        C:  if (w)    y <= C;
            else      y <= A;
        default:      y <= 2'bxx;
      endcase

  // Define output
  assign z = (y == C);

endmodule
```

Figure 8.34 Third version of code for the FSM in Figure 8.3.

as the reader is well aware by this point, there are many ways to implement a given logic function. However, the circuits produced from the three versions of the code provide identical functionality.

Example 8.5

Figure 8.35 shows how the FSM in Figure 8.11 can be specified in Verilog using the style illustrated in Figure 8.29. This FSM has four states, which are encoded using all four possible valuations of the state variables, hence there is no need for a **default** clause in the **case** statement.

8.4.5 SUMMARY OF DESIGN STEPS WHEN USING CAD TOOLS

In section 8.1.6 we summarized the design steps needed to derive sequential circuits manually. We have now seen that CAD tools can automatically perform much of the work. However, it is important to realize that the CAD tools have not replaced *all* manual steps. With reference to the list given in section 8.1.6, the first two steps, in which the machine specification is obtained and a state diagram is derived, still have to be done manually. Given the state diagram information as input, the CAD tools then automatically perform

```
module control (Clock, Resetn, w, R1in, R1out, R2in, R2out, R3in, R3out, Done);
  input Clock, Resetn, w;
  output R1in, R1out, R2in, R2out, R3in, R3out, Done;
  reg [2:1] y, Y;
  parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

  // Define the next state combinational circuit
  always @(w or y)
    case (y)
      A: if (w)   Y = B;
         else     Y = A;
      B:          Y = C;
      C:          Y = D;
      D:          Y = A;
    endcase

  // Define the sequential block
  always @(negedge Resetn or posedge Clock)
    if (Resetn == 0)  y <= A;
    else  y <= Y;

  // Define outputs
  assign R2out = (y == B);
  assign R3in = (y == B);
  assign R1out = (y == C);
  assign R2in = (y == C);
  assign R3out = (y == D);
  assign R1in = (y == D);
  assign Done = (y == D);

endmodule
```

Figure 8.35 Verilog code for the FSM in Figure 8.11.

the tasks needed to generate a circuit with logic gates and flip-flops. In addition to the design steps given in section 8.1.6, we should add the testing and simulation stage. We will defer detailed discussion of this issue until Chapter 11.

8.4.6 SPECIFYING THE STATE ASSIGNMENT IN VERILOG CODE

In section 8.2 we saw that the state assignment may have an impact on the complexity of the designed circuit. An obvious objective of the state-assignment process is to minimize the cost of implementation. The cost function that should be optimized may be simply the

number of gates and flip-flops. But it could also be based on other considerations that may be representative of the structure of PLD chips used to implement the design. For example, the CAD software may try to find state encodings that minimize the total number of AND terms needed in the resulting circuit when the target chip is a CPLD.

A particular state assignment has to be specified in Verilog code by means of a **parameter** statement as done in Figures 8.29 through 8.35. However, Verilog compilers usually have a capability to search for different assignments that may give better results. Most compilers are able to recognize a finite state machine when they encounter its specification in the code that conforms to typical styles, such as those used in this chapter. When the compiler detects an FSM, it can try to optimize its implementation by applying certain strategies such as looking for a better state assignment, attempting to use the one-hot encoding, and exploiting specific features of the target device. The user can either allow the compiler to use its FSM-handling capability, or surpress it in which case the compiler simply deals with the Verilog statements in the usual way.

8.4.7 SPECIFICATION OF MEALY FSMS USING VERILOG

A Mealy-type FSM can be specified in a similar manner as a Moore-type FSM. Figure 8.36 gives Verilog code for the FSM in Figure 8.23. The state transitions are described in the same way as in our first Verilog example in Figure 8.29. The variables y and Y represent the present and next states, which can have values A and B. Compared to the code in Figure 8.29, the main difference is the way in which the code for the output is written. In Figure 8.36 the output z is defined within the **case** statement that also defines the state transitions. When the FSM is in state A, z should be 0, but when in state B, z should take the value of w. Since the sensitivity list for the **always** block includes w, a change in w will immediately reflect itself in the value of z if the machine is in state B, which meets the requirements of the Mealy-type FSM.

Implementing the FSM specified in Figure 8.36 in a CPLD chip yields the same equations as we derived manually in section 8.3. Simulation results for the synthesized circuit appear in Figure 8.37. The input waveform for w is the same as the one we used for the Moore-type machine in Figure 8.32. Our Mealy-type machine behaves correctly, with z becoming 1 just after the start of the second consecutive clock cycle in which w is 1.

In the simulation results we have given in this section, all changes in the input w occur immediately following a positive clock edge. This is based on the assumption, stated in section 8.1.5, that in a real circuit w would be synchronized with respect to the clock that controls the FSM. In Figure 8.38 we illustrate a problem that may arise if w does not meet this specification. In this case we have assumed that the changes in w take place at the negative edge of the clock, rather than at the positive edge when the FSM changes its state. The first pulse on the w input is 100 ns long. This should not cause the output z to become equal to 1. But the circuit does not behave in this manner. After the signal w becomes equal to 1, the first positive edge of the clock causes the FSM to change from state A to state B. As soon as the circuit reaches the state B, the w input is still equal to 1 for another 50 ns, which causes z to go to 1. When w returns to 0, the z signal does likewise. Thus an erroneous 50-ns pulse is generated on the output z.

```
module mealy (Clock, Resetn, w, z);
  input Clock, Resetn, w;
  output z;
  reg y, Y, z;
  parameter A = 0, B = 1;

  // Define the next state and output combinational circuits
  always @(w or y)
    case (y)
      A: if (w)
           begin
             z = 0;
             Y = B;
           end
           else
           begin
             z = 0;
             Y = A;
           end
      B: if (w)
           begin
             z = 1;
             Y = B;
           end
           else
           begin
             z = 0;
             Y = A;
           end
    endcase

  // Define the sequential block
  always @(negedge Resetn or posedge Clock)
    if (Resetn == 0)  y <= A;
    else  y <= Y;

endmodule
```

Figure 8.36 Verilog code for the Mealy machine of Figure 8.23.

We should pursue the consequences of this problem a little further. If z is used to drive another circuit that is not controlled by the same clock, then the extraneous pulse is likely to cause big problems. But if z is used as an input to a circuit (perhaps another FSM) that is controlled by the same clock, then the 50-ns pulse will be ignored by this circuit if $z = 0$ before the next positive edge of the clock (accounting for the setup time).

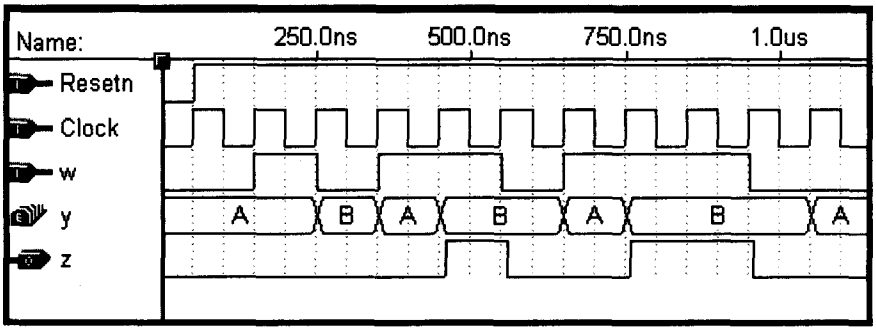


Figure 8.37 Simulation results for the Mealy machine.

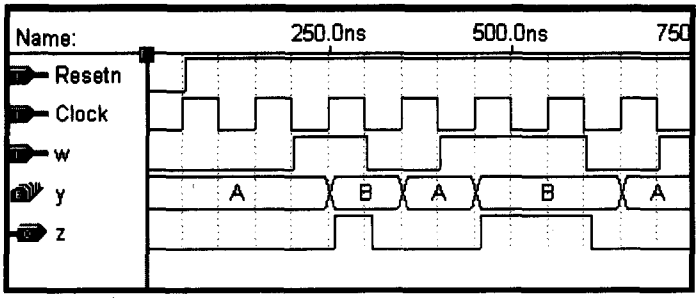


Figure 8.38 Potential problem with asynchronous inputs to a Mealy FSM.

8.5 SERIAL ADDER EXAMPLE

We will now present another simple example that illustrates the complete design process. In Chapter 5 we discussed the addition of binary numbers in detail. We explained several schemes that can be used to add two n-bit numbers in parallel, ranging from carry-ripple to carry-lookahead adders. In these schemes the speed of the adder unit is an important design parameter. Fast adders are more complex and thus more expensive. If speed is not of great importance, then a cost-effective option is to use a *serial adder*, in which bits are added a pair at a time.

8.5.1 MEALY-TYPE FSM FOR SERIAL ADDER

Let $A = a_{n-1}a_{n-2} \cdots a_0$ and $B = b_{n-1}b_{n-2} \cdots b_0$ be two unsigned numbers that have to be added to produce $Sum = s_{n-1}s_{n-2} \cdots s_0$. Our task is to design a circuit that will perform serial addition, dealing with a pair of bits in one clock cycle. The process starts by adding bits a_0 and b_0. In the next clock cycle, bits a_1 and b_1 are added, including a possible

carry from the bit-position 0, and so on. Figure 8.39 shows a block diagram of a possible implementation. It includes three shift registers that are used to hold A, B, and Sum as the computation proceeds. Assuming that the input shift registers have parallel-load capability, as depicted in Figure 7.19, the addition task begins by loading the values of A and B into these registers. Then in each clock cycle, a pair of bits is added by the adder FSM, and at the end of the cycle the resulting sum bit is shifted into the Sum register. We will use positive-edge-triggered flip-flops in which case all changes take place soon after the positive edge of the clock, depending on the propagation delays within the various flip-flops. At this time the contents of all three shift registers are shifted to the right; this shifts the existing sum bit into Sum, and it presents the next pair of input bits a_i and b_i to the adder FSM.

Now we are ready to design the required FSM. This cannot be a combinational circuit because different actions will have to be taken, depending on the value of the carry from the previous bit position. Hence two states are needed: let G and H denote the states where the carry-in values are 0 and 1, respectively. Figure 8.40 gives a suitable state diagram, defined as a Mealy model. The output value, s, depends on both the state and the present value of the inputs a and b. Each transition is labeled using the notation ab/s, which indicates the value of s for a given valuation ab. In state G the input valuation 00 will produce $s = 0$, and the FSM will remain in the same state. For input valuations 01 and 10, the output will

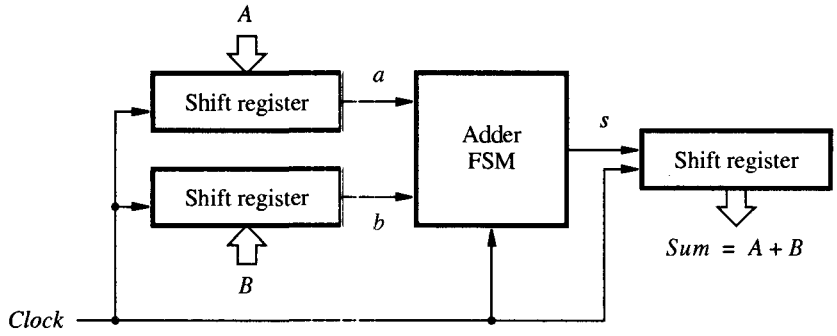


Figure 8.39 Block diagram for the serial adder.

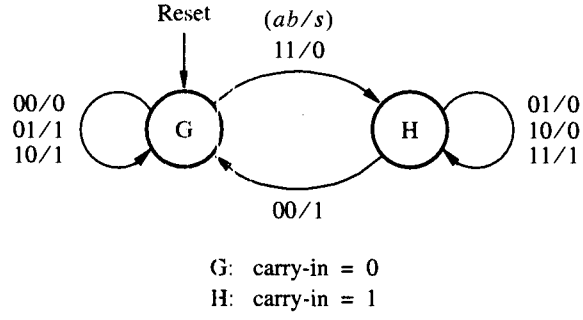


Figure 8.40 State diagram for the serial adder FSM.

be $s = 1$, and the FSM will remain in G. But for 11, $s = 0$ is generated, and the machine moves to state H. In state H valuations 01 and 10 cause $s = 0$, while 11 causes $s = 1$. In all three of these cases, the machine remains in H. However, when the valuation 00 occurs, the output of 1 is produced and a change into state G takes place.

The corresponding state table is presented in Figure 8.41. A single flip-flop is needed to represent the two states. The state assignment can be done as indicated in Figure 8.42. This assignment leads to the following next-state and output equations

$$Y = ab + ay + by$$
$$s = a \oplus b \oplus y$$

Comparing these expressions with those for the full-adder in section 5.2, it is obvious that y is the carry-in, Y is the carry-out, and s is the sum of the full-adder. Therefore, the adder FSM box in Figure 8.39 consists of the circuit shown in Figure 8.43. The flip-flop can be cleared by the *Reset* signal at the start of the addition operation.

The serial adder is a simple circuit that can be used to add numbers of any length. The structure in Figure 8.39 is limited in length only by the size of the shift registers.

8.5.2 MOORE-TYPE FSM FOR SERIAL ADDER

In the preceding example we saw that a Mealy-type FSM nicely meets the requirement for implementing the serial adder. Now we will try to achieve the same objective using a Moore-type FSM. A good starting point is the state diagram in Figure 8.40. In a Moore-type FSM, the output must depend only on the state of the machine. Since in both states, G and H, it is possible to produce two different outputs depending on the valuations of the inputs

Present state	Next state				Output s			
	$ab = 00$	01	10	11	00	01	10	11
G	G	G	G	H	0	1	1	0
H	G	H	H	H	1	0	0	1

Figure 8.41 State table for the serial adder FSM.

Present state	Next state				Output			
	$ab = 00$	01	10	11	00	01	10	11
y	Y				s			
0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	0	0	1

Figure 8.42 State-assigned table for Figure 8.41.

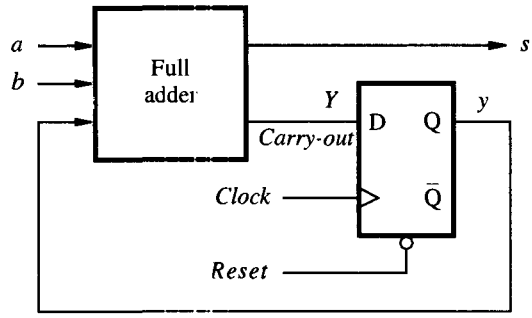


Figure 8.43 Circuit for the adder FSM in Figure 8.39.

a and b, a Moore-type FSM will need more than two states. We can derive a suitable state diagram by splitting both G and H into two states. Instead of G, we will use G_0 and G_1 to denote the fact that the carry is 0 and that the sum is either 0 or 1, respectively. Similarly, instead of H, we will use H_0 and H_1. Then the information in Figure 8.40 can be mapped into the Moore-type state diagram in Figure 8.44 in a straightforward manner.

The corresponding state table is given in Figure 8.45 and the state-assigned table in Figure 8.46. The next-state and output expressions are

$$Y_1 = a \oplus b \oplus y_2$$
$$Y_2 = ab + ay_2 + by_2$$
$$s = y_1$$

The expressions for Y_1 and Y_2 correspond to the sum and carry-out expressions in the full-adder circuit. The FSM is implemented as shown in Figure 8.47. It is interesting to observe that this circuit is very similar to the circuit in Figure 8.43. The only difference is that in the Moore-type circuit, the output signal, s, is passed through an extra flip-flop and thus delayed by one clock cycle with respect to the Mealy-type sequential circuit. Recall that we observed the same difference in our previous example, as depicted in Figures 8.26 and 8.27.

A key difference between the Mealy and Moore types of FSMs is that in the former a change in inputs reflects itself immediately in the outputs, while in the latter the outputs do not change until the change in inputs forces the machine into a new state, which takes place one clock cycle later. We encourage the reader to draw the timing diagrams for the circuits in Figures 8.43 and 8.47, which will exemplify further this key difference between the two types of FSMs.

8.5.3 VERILOG CODE FOR THE SERIAL ADDER

The serial adder can be described in Verilog by writing code for the shift registers and the adder FSM. We will first design the shift register and then use it as a subcircuit in the serial adder.

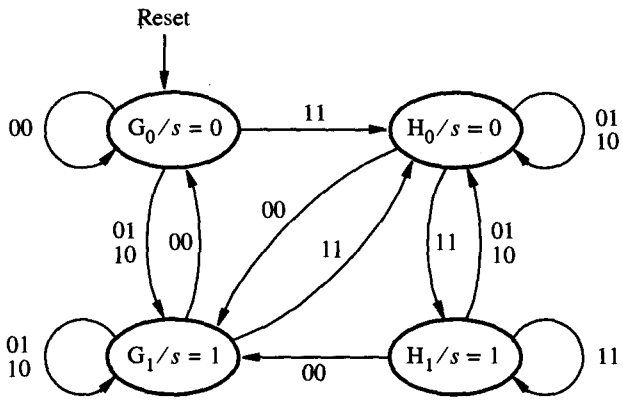


Figure 8.44 State diagram for the Moore-type serial adder FSM.

Present state	Next state				Output
	$ab = 00$	01	10	11	s
G_0	G_0	G_1	G_1	H_0	0
G_1	G_0	G_1	G_1	H_0	1
H_0	G_1	H_0	H_0	H_1	0
H_1	G_1	H_0	H_0	H_1	1

Figure 8.45 State table for the Moore-type serial adder FSM.

Present state y_2y_1	Next state				Output
	$ab = 00$	01	10	11	s
	Y_2Y_1				
0 0	0 0	0 1	0 1	1 0	0
0 1	0 0	0 1	0 1	1 0	1
1 0	0 1	1 0	1 0	1 1	0
1 1	0 1	1 0	1 0	1 1	1

Figure 8.46 State-assigned table for Figure 8.45.

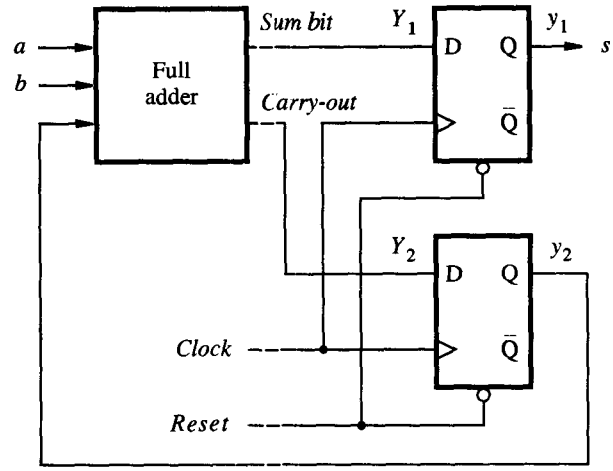


Figure 8.47 Circuit for the Moore-type serial adder FSM.

Shift Register Subcircuit

Figure 7.55 gives Verilog code for an n-bit shift register. In the serial adder it is beneficial to have the ability to prevent the shift register contents from changing when an active clock edge occurs. Figure 8.48 gives the code for a shift register named *shiftrne*, which has an enable input, E. When $E = 1$, the shift register behaves in the same way as the one in Figure 7.55. Setting $E = 0$ prevents the contents of the shift register from changing. The E input is usually called the *enable* input. It is useful for many types of circuits, as we will see in Chapter 10.

Complete Code

The code for the serial adder is shown in Figure 8.49. It instantiates three shift registers for the inputs A and B and the output *Sum*. The shift registers are loaded with parallel data when the circuit is reset. The state diagram for the adder FSM is described by two **always** blocks, using the style of code in Figure 8.36. In addition to the components of the serial adder shown in Figure 8.39, the Verilog code includes a down-counter to determine when the adder should be halted because all n bits of the required sum are present in the output shift register. When the circuit is reset, the counter is loaded with the number of bits in the serial adder, n. The counter counts down to 0, and then stops and disables further changes in the output shift register.

The code in Figure 8.49 implements a serial adder for eight-bit numbers. The wires QA and QB correspond to the parallel outputs of the shift registers with inputs A and B in Figure 8.39. The variable s represents the output of the adder FSM.

In Figure 8.39 the shift registers for inputs A and B do not use a serial input or an enable input. However, the *shiftrne* component, which is used for all three shift registers, includes these ports and so signals must be connected to them. The enable input for the two shift registers can be connected to logic value 1. The value shifted into the serial input does not

```
module shiftrne (R, L, E, w, Clock, Q);
   parameter n = 8;
   input [n-1:0] R;
   input L, E, w, Clock;
   output [n-1:0] Q;
   reg [n-1:0] Q;
   integer k;

   always @(posedge Clock)
      if (L)
         Q <= R;
      else if (E)
      begin
         for (k = n-1; k > 0; k = k-1)
            Q[k-1] <= Q[k];
         Q[n-1] <= w;
      end

endmodule
```

Figure 8.48 Code for a left-to-right shift register with an enable input.

matter, so it can be connected to either 1 or 0; we have chosen to connect it to 0. The shift registers are loaded in parallel by the *Reset* signal. We have chosen to use an active-high reset signal for the circuit. The output shift register does not need a parallel data input, so all 0s are connected to this input.

The first **always** block describes the state transitions and the output of the adder FSM in Figure 8.40. The output definition follows from observing in Figure 8.40 that when the FSM is in state G, the sum is $s = a \oplus b$, and when in state H, the sum is $s = \overline{a \oplus b}$. The second **always** block implements the flip-flop y and provides synchronous reset when *Reset* = 1.

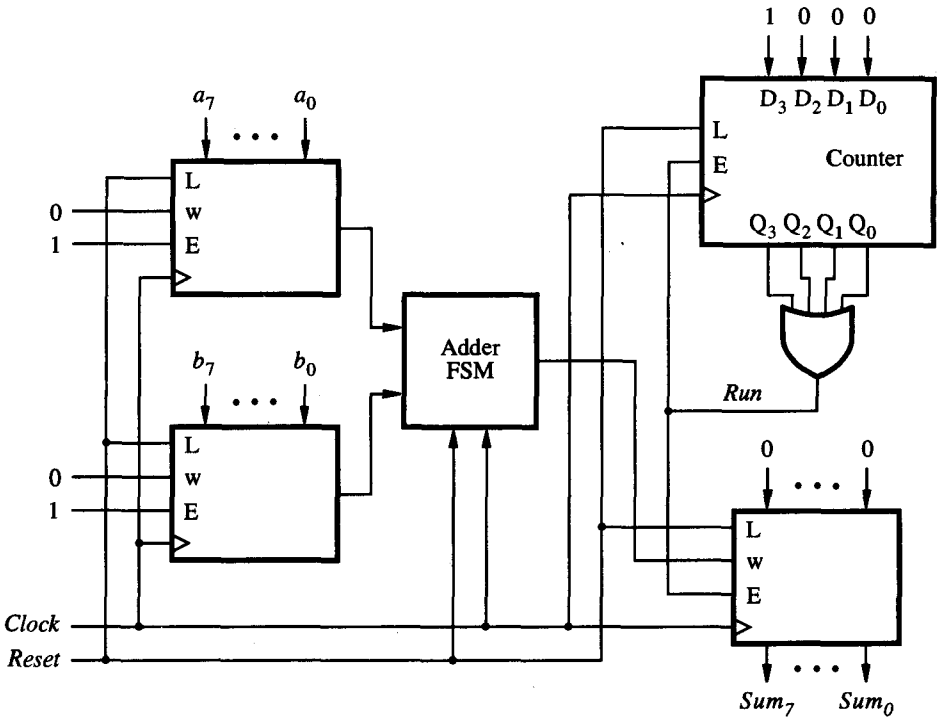
The enable input for the output shift register is named *Run*. It is derived from the outputs of the down-counter specified in the third **always** block. When *Reset* = 1, *Count* is initialized to the value 8. Then as long as *Run* = 1, *Count* is decremented in each clock cycle. *Run* is set to 0 when *Count* is equal to 0, which is detected by using the reduction OR operator.

Synthesis and Simulation of the Verilog Code

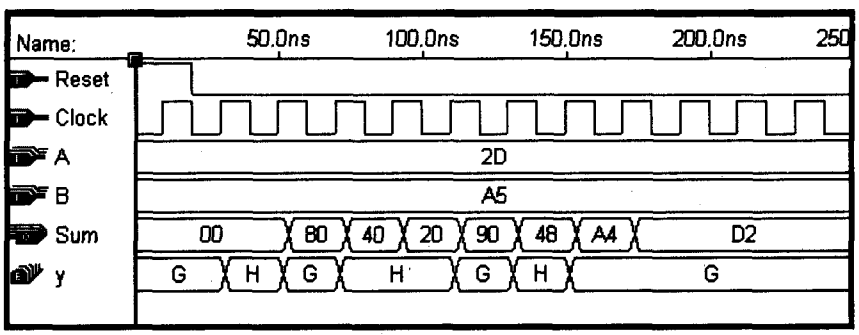
The results of synthesizing a circuit from the code in Figure 8.49 are illustrated in Figure 8.50*a*. The outputs of the counter are ORed to provide the *Run* signal, which enables clocking of both the output shift register and the counter. A sample of a timing simulation for the circuit is shown in Figure 8.50*b*. The circuit is first reset, resulting in the values of A and B being loaded into the input shift registers, and the value 8 loaded into the

```
module serial_adder (A, B, Reset, Clock, Sum);
   input [7:0] A, B;
   input Reset, Clock;
   output [7:0] Sum;
   reg [3:0] Count;
   reg s, y, Y;
   wire [7:0] QA, QB, Sum;
   wire Run;
   parameter G = 0, H = 1;

   shiftrne shift_A (A, Reset, 1, 0, Clock, QA);
   shiftrne shift_B (B, Reset, 1, 0, Clock, QB);
   shiftrne shift_Sum (0, Reset, Run, s, Clock, Sum);

   // Adder FSM
   // Output and next state combinational circuit
   always @(QA or QB or y)
      case (y)
         G: begin
               s = QA[0] ^ QB[0];
               if (QA[0] & QB[0])  Y = H;
               else  Y = G;
            end
         H: begin
               s = QA[0] ~^ QB[0];
               if (~QA[0] & ~QB[0])  Y = G;
               else  Y = H;
            end
         default:  Y = G;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Reset)  y <= G;
      else  y <= Y;

   // Control the shifting process
   always @(posedge Clock)
      if (Reset)  Count = 8;
      else if (Run)  Count = Count - 1;
   assign Run = |Count;

endmodule
```

Figure 8.49 Verilog code for the serial adder.



(a) Circuit



(b) Simulation Results

Figure 8.50 Synthesized serial adder.

down-counter. After each clock cycle one pair of bits of the input numbers is added by the adder FSM, and the sum bit is shifted into the output shift register. After eight clock cycles the output shift register contains the correct sum, and shifting is halted by the *Run* signal becoming equal to 0.

8.6 STATE MINIMIZATION

Our introductory examples of finite state machines were so simple that it was easy to see that the number of states that we used was the minimum possible to perform the required function. When a designer has to design a more complex FSM, it is likely that the initial attempt will result in a machine that has more states than is actually required. Minimizing the number of states is of interest because fewer flips-flops may be needed to represent the states and the complexity of the combinational circuit needed in the FSM may be reduced.

If the number of states in an FSM can be reduced, then some states in the original design must be equivalent to other states in their contribution to the overall behavior of the FSM. We can express this more formally in the following definition.

Definition 8.1 – *Two states S_i and S_j are said to be equivalent if and only if for every possible input sequence, the same output sequence will be produced regardless of whether S_i or S_j is the initial state.*

It is possible to define a minimization procedure that searches for any states that are equivalent. Such a procedure is very tedious to perform manually, but it can be automated for use in CAD tools. We will not pursue it here, because of its tediousness. However, to provide some appreciation of the impact of state minimization, we will present an alternative approach, which is much more efficient but not quite as broad in scope.

Instead of trying to show that some states in a given FSM are equivalent, it is often easier to show that some states are definitely **not** equivalent. This idea can be exploited to define a simple minimization procedure.

8.6.1 PARTITIONING MINIMIZATION PROCEDURE

Suppose that a state machine has a single input w. Then if the input signal $w = 0$ is applied to this machine in state S_i and the result is that the machine moves to state S_u, we will say that S_u is a 0-*successor* of S_i. Similarly, if $w = 1$ is applied in the state S_i and it causes the machine to move to state S_v, we will say that S_v is a 1-*successor* of S_i. In general, we will refer to the successors of S_i as its k-successors. When the FSM has only one input, k can be either 0 or 1. But if there are multiple inputs to the FSM, then k represents the set of all possible combinations (valuations) of the inputs.

From Definition 8.1 it follows that if the states S_i and S_j are equivalent, then their corresponding k-successors (for all k) are also equivalent. Using this fact, we can formulate a minimization procedure that involves considering the states of the machine as a set and then breaking the set into *partitions* that comprise subsets that are definitely not equivalent.

Definition 8.2 – *A partition consists of one or more blocks, where each block comprises a subset of states that may be equivalent, but the states in a given block are definitely not equivalent to the states in other blocks.*

Let us assume initially that all states are equivalent; this forms the initial partition, P_1, in which all states are in the same block. As the next step, we will form the partition P_2 in which the set of states is partitioned into blocks such that the states in each block generate the same output values. Obviously, the states that generate different outputs cannot possibly be equivalent. Then we will continue to form new partitions by testing whether the k-successors of the states in each block are contained in one block. Those states whose k-successors are in different blocks cannot be in one block. Thus new blocks are formed in each new partition. The process ends when a new partition is the same as the previous partition. Then all states in any one block are equivalent. To illustrate the procedure, consider Example 8.6.

Example 8.6

Figure 8.51 shows a state table for a particular FSM. In an attempt to minimize the number of states, let us apply the partitioning procedure. The initial partition contains all states in a single block

$$P_1 = (ABCDEFG)$$

The next partition separates the states that have different outputs (note that this FSM is of Moore type), which means that the states A, B, and D must be different from the states C, E, F, and G. Thus the new partition has two blocks

$$P_2 = (ABD)(CEFG)$$

Now we must consider all 0- and 1-successors of the states in each block. For the block (ABD), the 0-successors are (BDB), respectively. Since all of these successor states are in the same block in P_2, we should still assume that the states A, B, and D may be equivalent. The 1-successors for these states are (CFG). Since these successors are also in the same

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	1
B	D	F	1
C	F	E	0
D	B	G	1
E	F	C	0
F	E	D	0
G	F	G	0

Figure 8.51 State table for Example 8.6.

block in P_2, we conclude that (ABD) should remain in one block of P_3. Next consider the block $(CEFG)$. Its 0-successors are $(FFEF)$, respectively. They are in the same block in P_2. The 1-successors are $(ECDG)$. Since these states are not in the same block in P_2, it means that at least one of the states in the block $(CEFG)$ is not equivalent to the others. In particular, the state F must be different from the states C, E, and G because its 1-successor is D, which is in a different block than C, E, and G. Hence

$$P_3 = (ABD)(CEG)(F)$$

Repeating the process yields the following. The 0-successors of (ABD) are (BDB), which are in the same block of P_3. The 1-successors are (CFG), which are not in the same block. Since F is in a different block than C and G, it follows that the state B cannot be equivalent to states A and D. The 0- and 1-successors of (CEG) are (FFF) and (ECG), respectively. Both of these subsets are accommodated in the blocks of P_3. Therefore

$$P_4 = (AD)(B)(CEG)(F)$$

If we follow the same approach to check the 0- and 1-successors of the blocks (AD) and (CEG), we find that

$$P_5 = (AD)(B)(CEG)(F)$$

Since $P_5 = P_4$ and no new blocks are generated, it follows that states in each block are equivalent. If the states in some block were not equivalent, then their k-successors would have to be in different blocks. Therefore, states A and D are equivalent, and C, E, and G are equivalent. Since each block can be represented by a single state, only four states are needed to implement the FSM defined by the state table in Figure 8.51. If we let the symbol A represent both the states A and D in the figure and the symbol C represent the states C, E, and G, then the state table reduces to the state table in Figure 8.52.

The effect of the minimization is that we have found a solution that requires only two flip-flops to realize the four states of the minimized state table, instead of needing three flip-flops for the original design. The expectation is that the FSM with fewer states will be simpler to implement, although this is not always the case.

The state minimization concept is based on the fact that two different FSMs may exhibit identical behavior in terms of the outputs produced in response to all possible inputs. Such machines are functionally equivalent, even though they are implemented with circuits that may be vastly different. In general, it is not easy to determine whether or not

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	1
B	A	F	1
C	F	C	0
F	C	A	0

Figure 8.52 Minimized state table for Example 8.6.

two arbitrary FSMs are equivalent. Our minimization procedure ensures that a simplified FSM is functionally equivalent to the original one. We encourage the reader to get an intuitive feeling that the FSMs in Figures 8.51 and 8.52 are indeed functionally equivalent by implementing both machines and simulating their behavior using the CAD tools.

Example 8.7

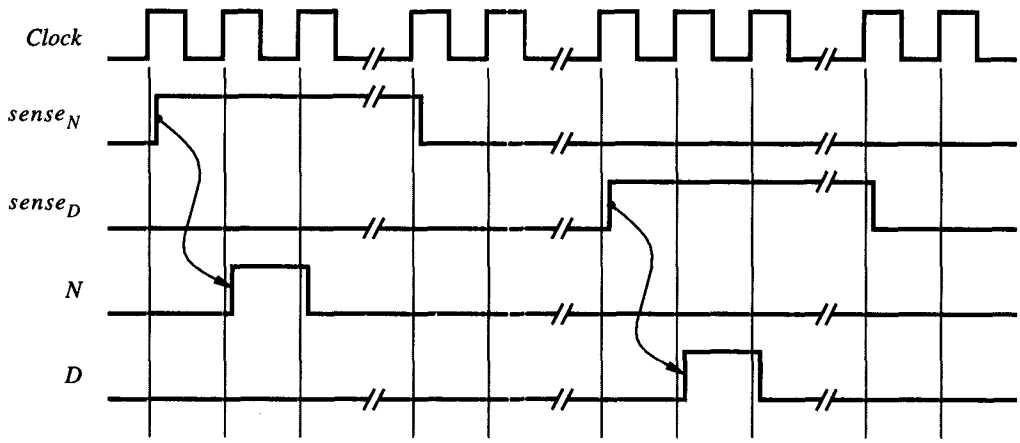
As another example of minimization, we will consider the design of a practical sequential circuit that could be used in a vending machine. Suppose that a coin-operated vending machine dispenses candy under the following conditions:

- The machine accepts nickels and dimes.
- It takes 15 cents for a piece of candy to be released from the machine.
- If 20 cents is deposited, the machine will not return the change, but it will credit the buyer with 5 cents and wait for the buyer to make a second purchase.

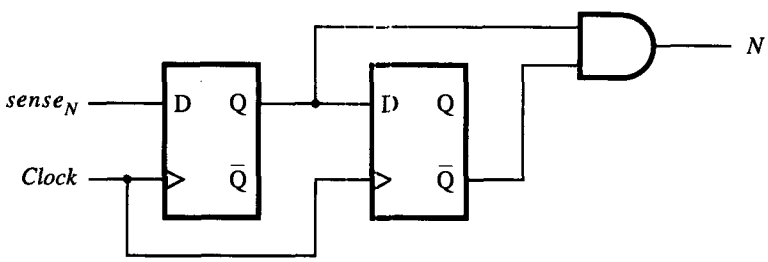
All electronic signals in the vending machine are synchronized to the positive edge of a clock signal, named *Clock*. The exact frequency of the clock signal is not important for our example, but we will assume a clock period of 100 ns. The vending machine's coin-receptor mechanism generates two signals, $sense_D$ and $sense_N$, which are asserted when a dime or a nickel is detected. Because the coin receptor is a mechanical device and thus very slow compared to an electronic circuit, inserting a coin causes $sense_D$ or $sense_N$ to be set to 1 for a large number of clock cycles. We will assume that the coin receptor also generates two other signals, named D and N. The D signal is set to 1 for one clock cycle after $sense_D$ becomes 1, and N is set to 1 for one clock cycle after $sense_N$ becomes 1. The timing relationships between *Clock*, $sense_D$, $sense_N$, D, and N are illustrated in Figure 8.53*a*. The hash marks on the waveforms indicate that $sense_D$ or $sense_N$ may be 1 for many clock cycles. Also, there may be an arbitrarily long time between the insertion of two consecutive coins. Note that since the coin receptor can accept only one coin at a time, it is not possible to have both D and N set to 1 at once. Figure 8.53*b* illustrates how the N signal may be generated from the $sense_N$ signal.

Based on these assumptions, we can develop an initial state diagram in a fairly straightforward manner, as indicated in Figure 8.54. The inputs to the FSM are D and N, and the starting state is $S1$. As long as $D = N = 0$, the machine remains in state $S1$, which is indicated by the arc labeled $\overline{D} \cdot \overline{N} = 1$. Inserting a dime leads to state $S2$, while inserting a nickel leads to state $S3$. In both cases the deposited amount is less than 15 cents, which is not sufficient to release the candy. This is indicated by the output, z, being equal to 0, as in $S2/0$ and $S3/0$. The machine will remain in state $S2$ or $S3$ until another coin is deposited because $D = N = 0$. In state $S2$ a nickel will cause a transition to $S4$ and a dime to $S5$. In both of these states, sufficient money is deposited to activate the output mechanism that releases the candy; hence the state nodes have the labels $S4/1$ and $S5/1$. In $S4$ the deposited amount is 15 cents, which means that on the next active clock edge the machine should return to the reset state $S1$. The condition $\overline{D} \cdot \overline{N}$ on the arc leaving $S4$ is guaranteed to be true because the machine remains in state $S4$ for only 100 ns, which is far too short a time for a new coin to have been deposited.

The state $S5$ denotes that an amount of 20 cents has been deposited. The candy is released, and on the next clock edge the FSM makes a transition to state $S3$, which



(a) Timing diagram



(b) Circuit that generates *N*

Figure 8.53 Signals for the vending machine.

represents a credit of 5 cents. A similar reasoning when the machine is in state $S3$ leads to states $S6$ through $S9$. This completes the state diagram for the desired FSM. A state table version of the same information is given in Figure 8.55.

Note that the condition $D = N = 1$ is denoted as don't care in the table. Note also other don't cares in states $S4$, $S5$, $S7$, $S8$, and $S9$. They correspond to cases where there is no need to check the D and N signals because the machine changes to another state in an amount of time that is too short for a new coin to have been inserted.

Using the minimization procedure, we obtain the following partitions

$$P_1 = (S1, S2, S3, S4, S5, S6, S7, S8, S9)$$
$$P_2 = (S1, S2, S3, S6)(S4, S5, S7, S8, S9)$$
$$P_3 = (S1)(S3)(S2, S6)(S4, S5, S7, S8, S9)$$
$$P_4 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$
$$P_5 = (S1)(S3)(S2, S6)(S4, S7, S8)(S5, S9)$$

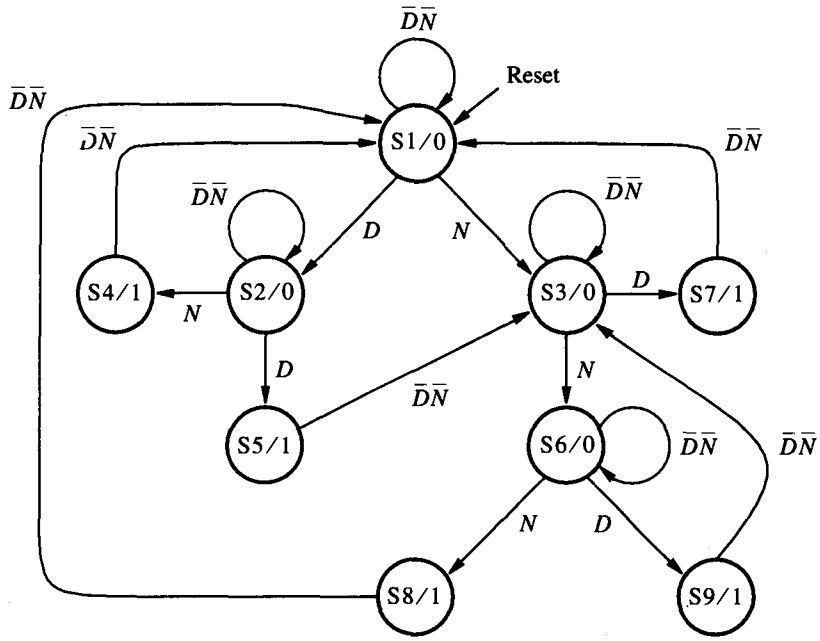


Figure 8.54 State diagram for Example 8.7.

Present state	Next state				Output
	$DN = 00$	01	10	11	z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S6	S7	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1
S6	S6	S8	S9	–	0
S7	S1	–	–	–	1
S8	S1	–	–	–	1
S9	S3	–	–	–	1

Figure 8.55 State table for Example 8.7.

The final partition has five blocks. Let $S2$ denote its equivalence to $S6$, let $S4$ denote the same with respect to $S7$ and $S8$, and let $S5$ represent $S9$. This leads to the minimized state table in Figure 8.56. The actual circuit that implements this table can be designed as explained in the previous sections.

Present state	Next state				Output
	$DN = 00$	01	10	11	z
S1	S1	S3	S2	–	0
S2	S2	S4	S5	–	0
S3	S3	S2	S4	–	0
S4	S1	–	–	–	1
S5	S3	–	–	–	1

Figure 8.56 Minimized state table for Example 8.7.

In this example we used a straightforward approach to derive the original state diagram, which we then minimized using the partitioning procedure. Figure 8.57 presents the information in the state table of Figure 8.56 in the form of a state diagram. Looking at this diagram, the reader can probably see that it may have been quite feasible to derive the optimized diagram directly, using the following reasoning. Suppose that the states correspond to the various amounts of money deposited. In particular, the states, $S1$, $S3$, $S2$, $S4$, and $S5$ correspond to the amounts of 0, 5, 10, 15, and 20 cents, respectively. With this interpretation of the states, it is not difficult to derive the transition arcs that define the

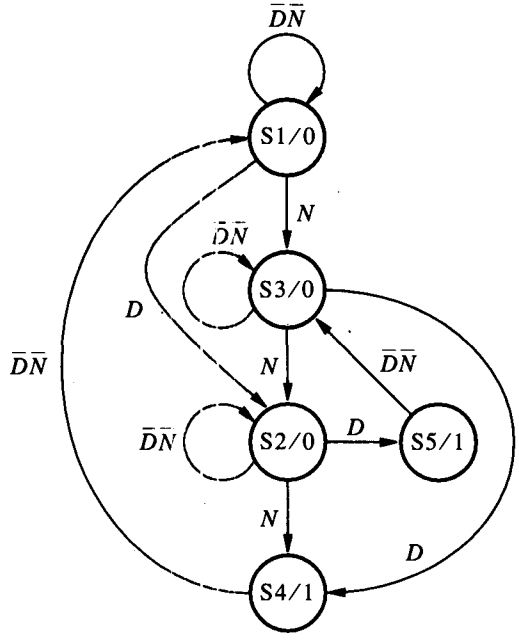


Figure 8.57 Minimized state diagram for Example 8.7.

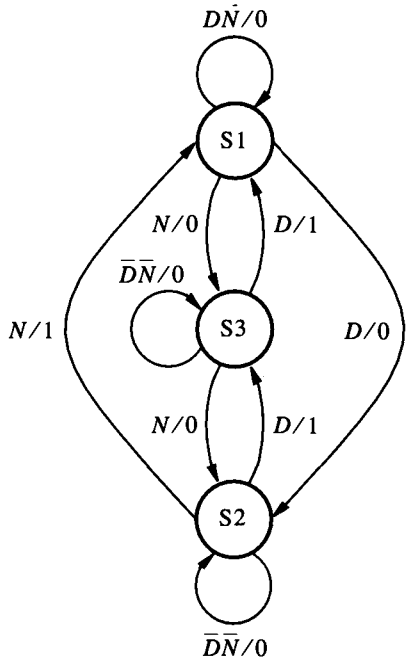


Figure 8.58 Mealy-type FSM for Example 8.7.

desired FSM. In practice, the designer can often produce initial designs that do not have a large number of superfluous states.

We have found a solution that requires five states, which is the minimum number of states for a Moore-type FSM that realizes the desired vending control task. From section 8.3 we know that Mealy-type FSMs may need fewer states than Moore-type machines, although they do not necessarily lead to simpler overall implementations. If we use the Mealy model, we can eliminate states $S4$ and $S5$ in Figure 8.57. The result is shown in Figure 8.58. This version requires only three states, but the output functions become more complicated. The reader is encouraged to compare the complexity of implementations by completing the design steps for the FSMs in Figures 8.57 and 8.58.

8.6.2 INCOMPLETELY SPECIFIED FSMS

The partitioning scheme for minimization of states works well when all entries in the state table are specified. Such is the case for the FSM defined in Figure 8.51. FSMs of this type are said to be *completely specified.* If one or more entries in the state table are not specified, corresponding to don't-care conditions, then the FSM is said to be *incompletely specified.* An example of such an FSM is given in Figure 8.55. As seen in Example 8.7,

the partitioning scheme works well for this FSM also. But in general, the partitioning scheme is less useful when incompletely specified FSMs are involved, as illustrated by Example 8.8.

Example 8.8 Consider the FSM in Figure 8.59 which has four unspecified entries, because we have assumed that the input $w = 1$ will not occur when the machine is in states B or G. Accordingly, neither a state transition nor an output value is specified for these two cases. An important difference between this FSM and the one in Figure 8.55 is that some outputs in this FSM are unspecified, whereas in the other FSM all outputs are specified.

The partitioning minimization procedure can be applied to Mealy-type FSMs in the same way as for Moore-type FSMs illustrated in Examples 8.6 and 8.7. Two states are considered equivalent, and are thus placed in the same block of a partition, if their outputs are equal for all corresponding input valuations. To perform the partitioning process, we can assume that the unspecified outputs have a specific value. Not knowing whether these values should be 0 or 1, let us first assume that both unspecified outputs have a value of 0. Then the first two partitions are

$$P_1 = (ABCDEFG)$$
$$P_2 = (ABDG)(CEF)$$

Note that the states A, B, D, and G are in the same block because their outputs are equal to 0 for both $w = 0$ and $w = 1$. Also, the states C, E, and F are in one block because they have the same output behavior; they all generate $z = 0$ if $w = 0$, and $z = 1$ if $w = 1$. Continuing the partitioning procedure gives the remaining partitions

$$P_3 = (AB)(D)(G)(CE)(F)$$
$$P_4 = (A)(B)(D)(G)(CE)(F)$$
$$P_5 = P_4$$

The result is an FSM that is specified by six states.

Next consider the alternative of assuming that both unspecified outputs in Figure 8.59 have a value of 1. This would lead to the partitions

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	B	C	0	0
B	D	–	0	–
C	F	E	0	1
D	B	G	0	0
E	F	C	0	1
F	E	D	0	1
G	F	–	0	–

Figure 8.59 Incompletely specified state table for Example 8.8.

$$P_1 = (ABCDEFG)$$
$$P_2 = (AD)(BCEFG)$$
$$P_3 = (AD)(B)(CEFG)$$
$$P_4 = (AD)(B)(CEG)(F)$$
$$P_5 = P_4$$

This solution involves four states. Evidently, the choice of values assigned to unspecified outputs is of considerable importance.

We will not pursue the issue of state minimization of incompletely specified FSMs any further. As we already mentioned, it is possible to develop a minimization technique that searches for equivalent states based directly on Definition 8.1. This approach is described in many books on logic design [2, 5–8, 12–14].

Finally, it is important to mention that reducing the number of states in a given FSM will not necessarily lead to a simpler implementation. Interestingly, the effect of state assignment, discussed in section 8.2, may have a greater influence on the simplicity of implementation than does the state minimization. In a modern design environment, the designer relies on the CAD tools to implement state machines efficiently.

8.7 DESIGN OF A COUNTER USING THE SEQUENTIAL CIRCUIT APPROACH

In this section we discuss the design of a counter circuit using the general approach for designing sequential circuits. From Chapter 7 we already know that counters can be realized as cascaded stages of flip-flops and some gating logic, where each stage divides the number of incoming pulses by two. To keep our example simple, we choose a counter of small size but also show how the design can be extended to larger sizes. The specification for the counter is

- The counting sequence is 0, 1, 2, ..., 6, 7, 0, 1, ...
- There exists an input signal w. The value of this signal is considered during each clock cycle. If $w = 0$, the present count remains the same; if $w = 1$, the count is incremented.

The counter can be designed as a synchronous sequential circuit using the design techniques introduced in the previous sections. We show first the classical manual approach to designing the counter, which illustrates the basic concepts involved in the design process. After that we show how the design task is accomplished using CAD tools, which is much easier to do and indicates how the task would be tackled in practice.

8.7.1 STATE DIAGRAM AND STATE TABLE FOR A MODULO-8 COUNTER

Figure 8.60 gives a state diagram for the desired counter. There is a state associated with each count. In the diagram state A corresponds to count 0, state B to count 1, and so on. We show the transitions between the states needed to implement the counting sequence. Note

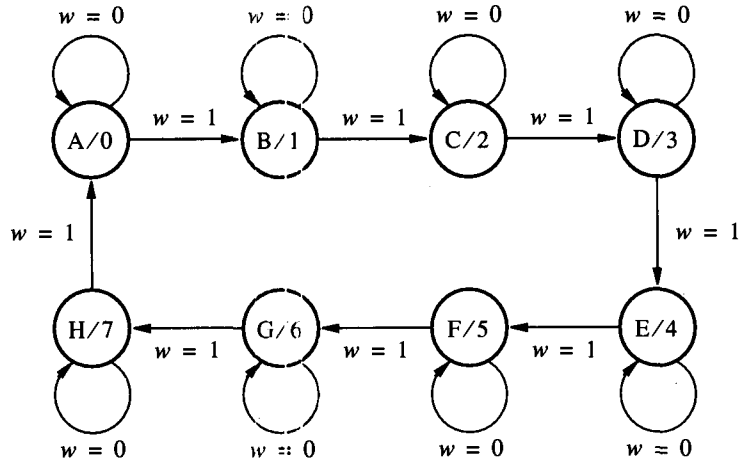


Figure 8.60 State diagram for the counter.

Present state	Next state		Output
	$w = 0$	$w = 1$	
A	A	B	0
B	B	C	1
C	C	D	2
D	D	E	3
E	E	F	4
F	F	G	5
G	G	H	6
H	H	A	7

Figure 8.61 State table for the counter.

that the output signals are specified as depending only on the state of the counter at a given time, which is the Moore model of sequential circuits.

The state diagram may be represented in the state-table form as shown in Figure 8.61.

8.7.2 STATE ASSIGNMENT

Three state variables are needed to represent the eight states. Let these variables, denoting the present state, be called y_2, y_1, and y_0. Let Y_2, Y_1, and Y_0 denote the corresponding next-state functions. The most convenient (and simplest) state assignment is to encode each state with the binary number that the counter should give as output in that state. Then the required output signals will be the same as the signals that represent the state variables. This leads to the state-assigned table in Figure 8.62.

	Present state $y_2y_1y_0$	Next state $w=0$ $Y_2Y_1Y_0$	Next state $w=1$ $Y_2Y_1Y_0$	Count $z_2z_1z_0$
A	000	000	001	000
B	001	001	010	001
C	010	010	011	010
D	011	011	100	011
E	100	100	101	100
F	101	101	110	101
G	110	110	111	110
H	111	111	000	111

Figure 8.62 State-assigned table for the counter.

The final step in the design is to choose the type of flip-flops and derive the expressions that control the flip-flop inputs. The most straightforward choice is to use D-type flip-flops. We pursue this approach first. Then we show the alternative of using JK-type flip-flops. In either case the flip-flops must be edge triggered to ensure that only one transition takes place during a single clock cycle.

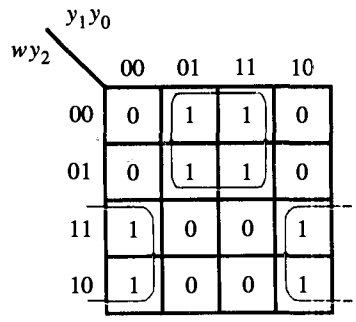
8.7.3 IMPLEMENTATION USING D-TYPE FLIP-FLOPS

When using D-type flip-flops to realize the finite state machine, each next-state function, Y_i, is connected to the D input of the flip-flop that implements the state variable y_i. The next-state functions are derived from the information in Figure 8.62. Using Karnaugh maps in Figure 8.63, we obtain the following implementation

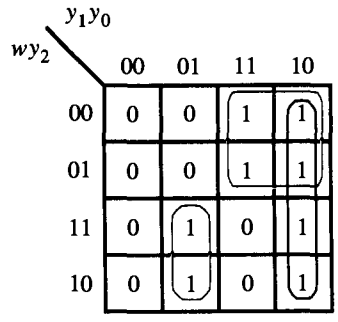
$$D_0 = Y_0 = \overline{w}y_0 + w\overline{y}_0$$
$$D_1 = Y_1 = \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1$$
$$D_2 = Y_2 = \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2$$

The resulting circuit is given in Figure 8.64. It is not obvious how to extend this circuit to implement a larger counter, because no clear pattern is discernible in the expressions for D_0, D_1, and D_2. However, we can rewrite these expressions as follows

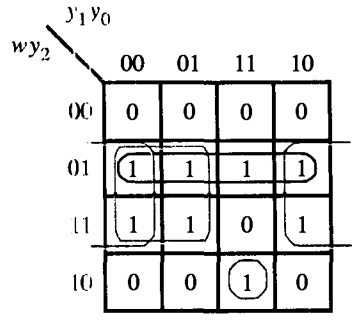
$$\begin{aligned} D_0 &= \overline{w}y_0 + w\overline{y}_0 \\ &= w \oplus y_0 \\ D_1 &= \overline{w}y_1 + y_1\overline{y}_0 + wy_0\overline{y}_1 \\ &= (\overline{w} + \overline{y}_0)y_1 + wy_0\overline{y}_1 \\ &= \overline{w\overline{y}_0}y_1 + wy_0\overline{y}_1 \\ &= wy_0 \oplus y_1 \\ D_2 &= \overline{w}y_2 + \overline{y}_0y_2 + \overline{y}_1y_2 + wy_0y_1\overline{y}_2 \end{aligned}$$



$Y_0 = \bar{w}y_0 + w\bar{y}_0$



$Y_1 = \bar{w}y_1 + y_1\bar{y}_0 + wy_0\bar{y}_1$



$Y_2 = \bar{w}y_2 + \bar{y}_0y_2 + \bar{y}_1y_2 + wy_0y_1\bar{y}_2$

Figure 8.63 Karnaugh maps for D flip-flops for the counter.

$$= (\bar{w} + \bar{y}_0 + \bar{y}_1)y_2 + wy_0y_1\bar{y}_2$$
$$= \overline{wy_0y_1}y_2 + wy_0y_1\bar{y}_2$$
$$= wy_0y_1 \oplus y_2$$

Then an obvious pattern emerges, which leads to the circuit in Figure 7.24.

8.7.4 IMPLEMENTATION USING JK-TYPE FLIP-FLOPS

JK-type flip-flops provide an attractive alternative. Using these flip-flops to implement the sequential circuit specified in Figure 8.62 requires derivation of J and K inputs for each flip-flop. The following control is needed:

- If a flip-flop in state 0 is to remain in state 0, then $J = 0$ and $K = d$ (where d means that K can be equal to either 0 or 1).
- If a flip-flop in state 0 is to change to state 1, then $J = 1$ and $K = d$.

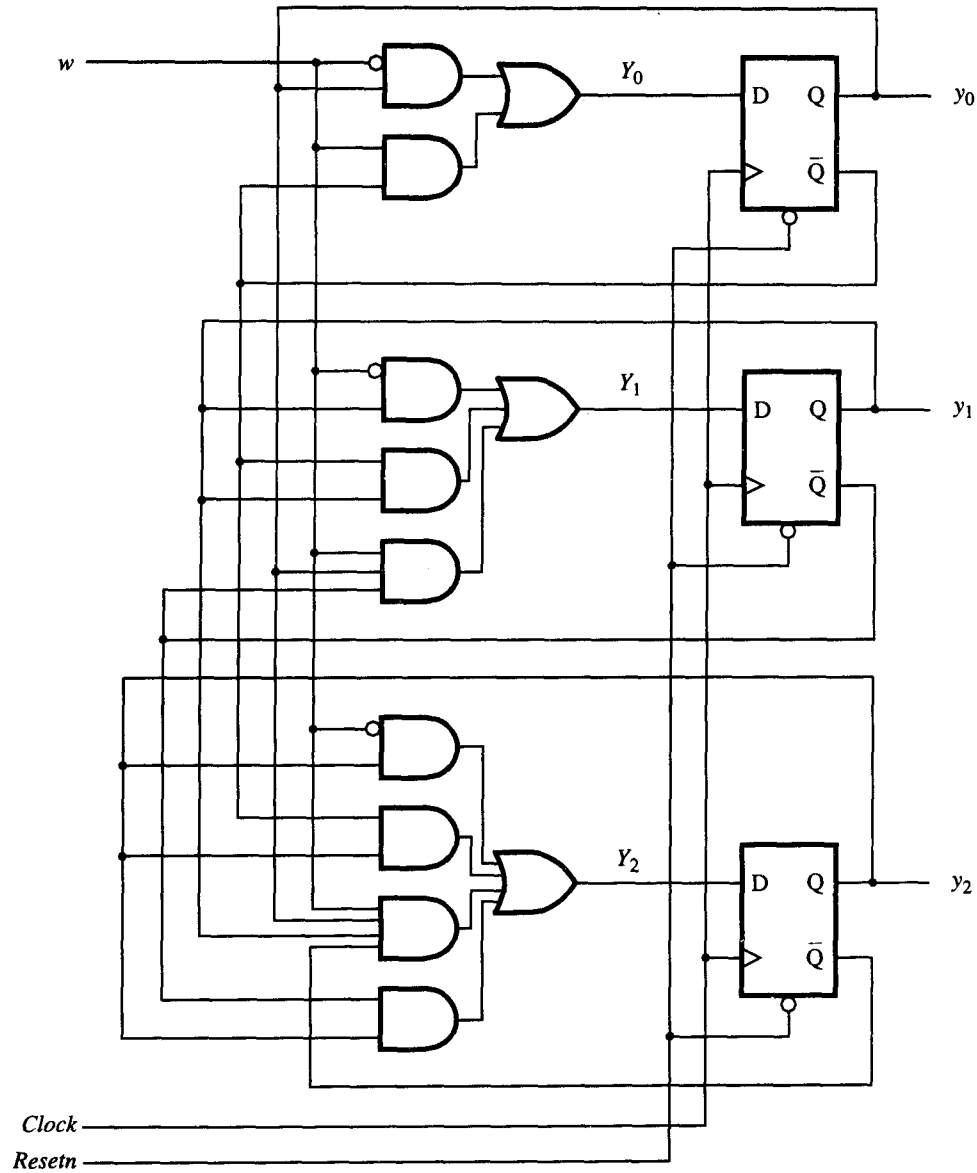


Figure 8.64 Circuit diagram for the counter implemented with D flip-flops.

- If a flip-flop in state 1 is to remain in state 1, then $J = d$ and $K = 0$.
- If a flip-flop in state 1 is to change to state 0, then $J = d$ and $K = 1$.

Following these guidelines, we can create a truth table that specifies the required values of the J and K inputs for the three flip-flops in our design. Figure 8.65 shows a modified

	Present state	Flip-flop inputs							Count
		$w = 0$				$w = 1$			
	$y_2y_1y_0$	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$Y_2Y_1Y_0$	J_2K_2	J_1K_1	J_0K_0	$z_2z_1z_0$
A	000	000	0*d*	0*d*	0*d*	001	0*d*	0*d*	1*d*	000
B	001	001	0*d*	0*d*	*d*0	010	0*d*	1*d*	*d*1	001
C	010	010	0*d*	*d*0	0*d*	011	0*d*	*d*0	1*d*	010
D	011	011	0*d*	*d*0	*d*0	100	1*d*	*d*1	*d*1	011
E	100	100	*d*0	0*d*	0*d*	101	*d*0	0*d*	1*d*	100
F	101	101	*d*0	0*d*	*d*0	110	*d*0	1*d*	*d*1	101
G	110	110	*d*0	*d*0	0*d*	111	*d*0	*d*0	1*d*	110
H	111	111	*d*0	*d*0	*d*0	000	*d*1	*d*1	*d*1	111

Figure 8.65 Excitation table for the counter with JK flip-flops.

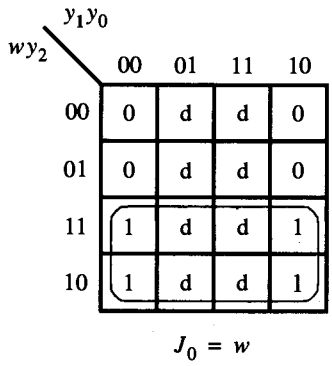
version of the state-assigned table in Figure 8.62, with the J and K input functions included. To see how this table is derived, consider the first row in which the present state is $y_2y_1y_0 = 000$. If $w = 0$, then the next state is also $Y_2Y_1Y_0 = 000$. Thus the present value of each flip-flop is 0, and it should remain 0. This implies the control $J = 0$ and $K = d$ for all three flip-flops. Continuing with the first row, if $w = 1$, the next state will be $Y_2Y_1Y_0 = 001$. Thus flip-flops y_2 and y_1 still remain at 0 and have the control $J = 0$ and $K = d$. However, flip-flop y_0 must change from 0 to 1, which is accomplished with $J = 1$ and $K = d$. The rest of the table is derived in the same manner by considering each present state $y_2y_1y_0$ and providing the necessary control signals to reach the new state $Y_2Y_1Y_0$.

A state-assigned table is essentially the state table in which each state is encoded using the state variables. When D flip-flops are used to implement an FSM, the next-state entries in the state-assigned table correspond directly to the signals that must be applied to the D inputs. This is not the case if some other type of flip-flops is used. A table that gives the state information in the form of the flip-flop inputs that must be "excited" to cause the transitions to the next states is usually called an *excitation table*. The excitation table in Figure 8.65 indicates how JK flip-flops can be used. In many books the term excitation table is used even when D flip-flops are involved, in which case it is synonymous with the state-assigned table.

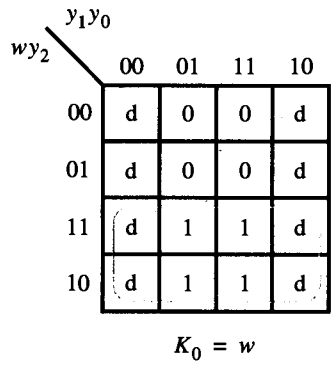
Once the table in Figure 8.65 has been derived, it provides a truth table with inputs y_2, y_1, y_0, and w, and outputs J_2, K_2, J_1, K_1, J_0, and K_0. We can then derive expressions for these outputs as shown in Figure 8.66. The resulting expressions are

$$J_0 = K_0 = w$$
$$J_1 = K_1 = wy_0$$
$$J_2 = K_2 = wy_0y_1$$

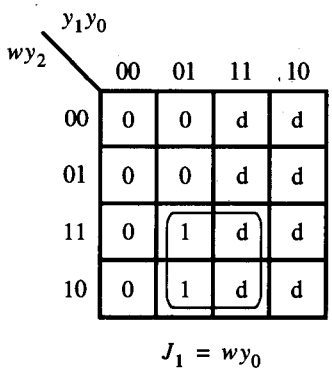
This leads to the circuit shown in Figure 8.67. It is apparent that this design can be extended easily to larger counters. The pattern $J_n = K_n = wy_0y_1 \cdots y_{n-1}$ defines the circuit for each stage in the counter. Note that the size of the AND gate that implements the product term $y_0y_1 \cdots y_{n-1}$ grows with successive stages. A circuit with a more regular structure can be



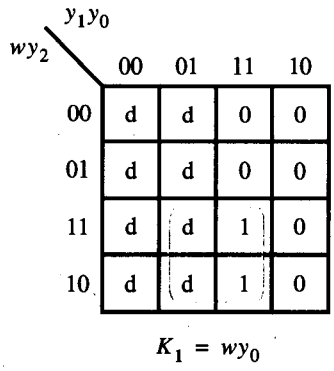
$J_0 = w$



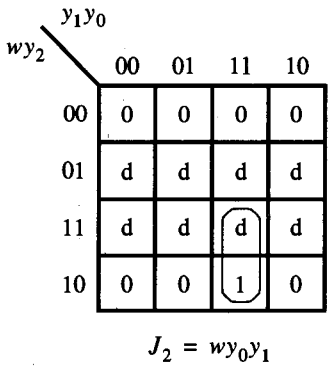
$K_0 = w$



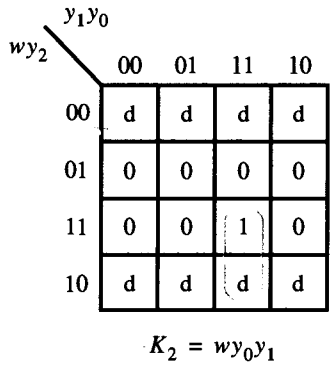
$J_1 = wy_0$



$K_1 = wy_0$



$J_2 = wy_0y_1$



$K_2 = wy_0y_1$

Figure 8.66 Karnaugh maps for JK flip-flops in the counter.

obtained by factoring out the previously needed terms as we progress through the stages of the counter. This gives

$$J_2 = K_2 = (wy_0)y_1 \qquad = J_1y_1$$
$$J_n = K_n = (wy_0 \cdots y_{n-2})y_{n-1} = J_{n-1}y_{n-1}$$

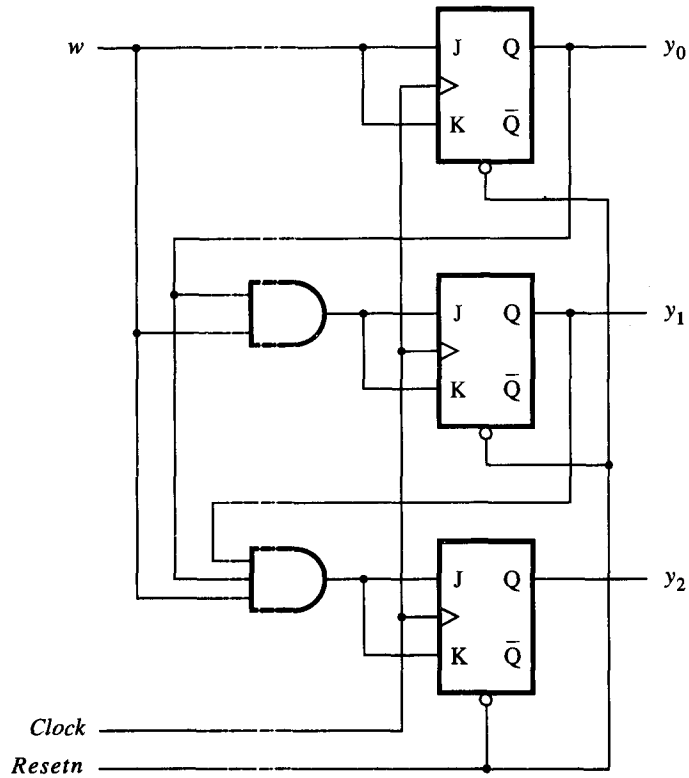


Figure 8.67 Circuit diagram using JK flip-flops.

Using the factored form, the counter circuit can be realized as indicated in Figure 8.68. In this circuit all stages (except the first) look the same. Note that this circuit has the same structure as the circuit in Figure 7.23 because connecting the J and K inputs of a flip-flop together turns the flip-flop into a T flip-flop.

8.7.5 EXAMPLE—A DIFFERENT COUNTER

Having considered the design of an ordinary counter, we will now apply this knowledge to design a slightly different counterlike circuit. Suppose that we wish to derive a three-bit counter that counts the pulses on an input line, w. But instead of displaying the count as 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, . . ., this counter must display the count in the sequence 0, 4, 2, 6, 1, 5, 3, 7, 0, 4, and so on. The count is to be represented directly by the flip-flop values themselves, without using any extra gates. Namely, Count = $Q_2Q_1Q_0$.

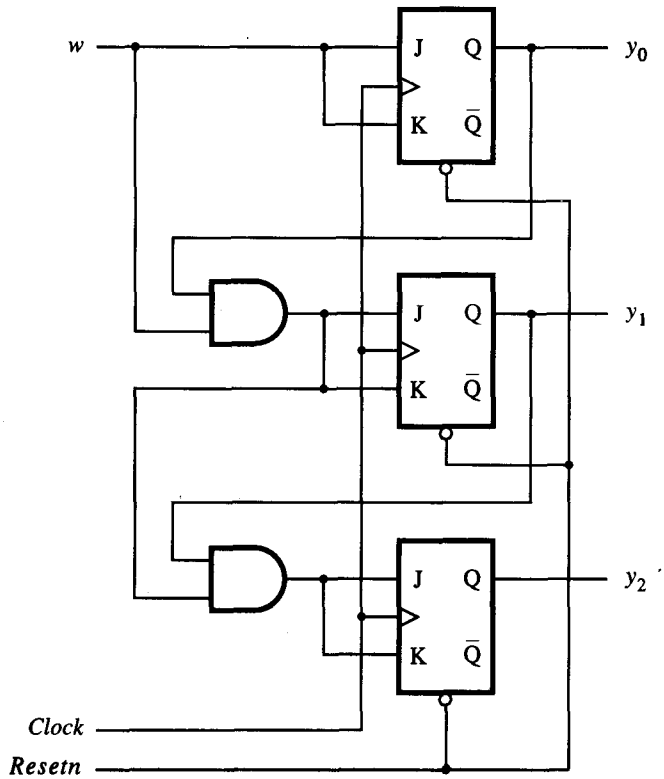


Figure 8.68 Factored-form implementation of the counter.

Since we wish to count the pulses on the input line w, it makes sense to use w as the clock input to the flip-flops. Thus the counter circuit should always be enabled, and it should change its state whenever the next pulse on the w line appears. The desired counter can be designed in a straightforward manner using the FSM approach. Figures 8.69 and 8.70 give the required state table and a suitable state assignment. Using D flip-flops, we obtain the next-state equations

$$\begin{aligned} D_2 &= Y_2 = \bar{y}_2 \\ D_1 &= Y_1 = y_1 \oplus y_2 \\ D_0 &= Y_0 = y_0\bar{y}_1 + y_0\bar{y}_2 + \bar{y}_0 y_1 y_2 \\ &= y_0(\bar{y}_1 + \bar{y}_2) + \bar{y}_0 y_1 y_2 \\ &= y_0 \oplus y_1 y_2 \end{aligned}$$

This leads to the circuit in Figure 8.71.

Present state	Next state	Output $z_2z_1z_0$
A	B	0 0 0
B	C	1 0 0
C	D	0 1 0
D	E	1 1 0
E	F	0 0 1
F	G	1 0 1
G	H	0 1 1
H	A	1 1 1

Figure 8.69 State table for counterlike example.

Present state $y_2y_1y_0$	Next state $Y_2Y_1Y_0$	Output $z_2z_1z_0$
0 0 0	1 0 0	0 0 0
1 0 0	0 1 0	1 0 0
0 1 0	1 1 0	0 1 0
1 1 0	0 0 1	1 1 0
0 0 1	1 0 1	0 0 1
1 0 1	0 1 1	1 0 1
0 1 1	1 1 1	0 1 1
1 1 1	0 0 0	1 1 1

Figure 8.70 State-assigned table for Figure 8.69.

The reader should compare this circuit with the normal up-counter in Figure 7.24. Take the first three stages of that counter, set the *Enable* input to 1, and let *Clock* = w. Then the two circuits are essentially the same with one small difference in the order of bits in the count. In Figure 7.24 the top flip-flop corresponds to the least-significant bit of the count, whereas in Figure 8.71 the top flip-flop corresponds to the most-significant bit of the count. This is not just a coincidence. In Figure 8.70 the required count is defined as $Count = y_2y_1y_0$. However, if the bit patterns that define the states are viewed in the reverse order and interpreted as binary numbers, such that $Count = y_0y_1y_2$, then the states $A, B, C, \ldots, H$ have the values $0, 1, 2, \ldots, 7$. These values are the same as the values that are associated with the normal three-bit up-counter.

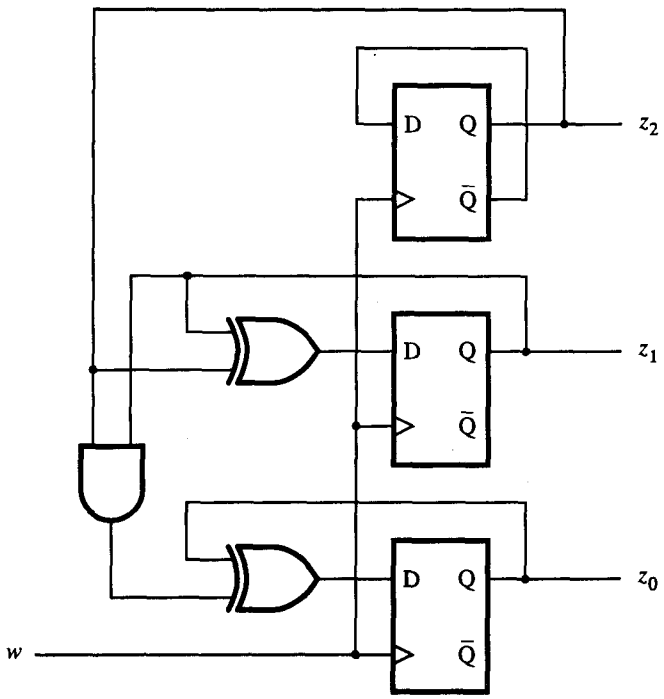


Figure 8.71 Circuit for Figure 8.70.

8.8 FSM AS AN ARBITER CIRCUIT

In this section we present the design of an FSM that is slightly more complex than the previous examples. The purpose of the machine is to control access by various devices to a shared resource in a given system. Only one device can use the resource at a time. Assume that all signals in the system can change values only on the positive edge of the clock signal. Each device provides one input to the FSM, called a *request*, and the FSM produces a separate output for each device, called a *grant*. A device indicates its need to use the resource by asserting its request signal. Whenever the shared resource is not already in use, the FSM considers all requests that are active. Based on a priority scheme, it selects one of the requesting devices and asserts its grant signal. When the device is finished using the resource, it deasserts its request signal.

We will assume that there are three devices in the system, called device 1, device 2, and device 3. It is easy to see how the FSM can be extended to handle more devices. The request signals are named r_1, r_2, and r_3, and the grant signals are called g_1, g_2, and g_3. The devices are assigned a priority level such that device 1 has the highest priority, device 2 has the next highest, and device 3 has the lowest priority. Thus if more than one request signal

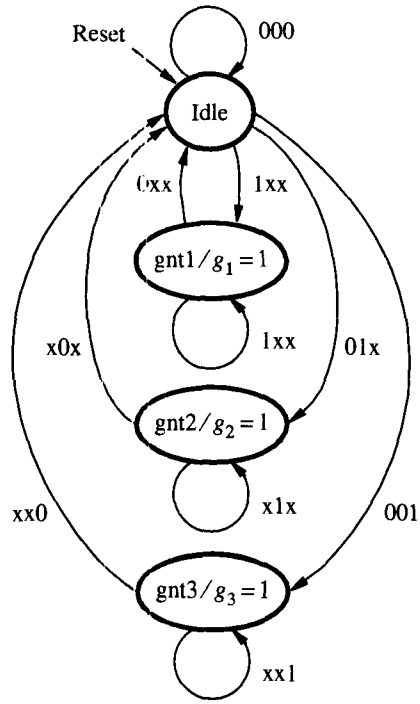


Figure 8.72 State diagram for the arbiter.

is asserted when the FSM assigns a grant, the grant is given to the requesting device that has the highest priority.

A state diagram for the desired FSM, designed as a Moore-type machine, is depicted in Figure 8.72. Initially, on reset the machine is in the state called *Idle*. No grant signals are asserted, and the shared resource is not in use. There are three other states, called *gnt*1, *gnt*2, and *gnt*3. Each of these states asserts the grant signal for one of the devices.

The FSM remains in the *Idle* state as long as all of the request signals are 0. In the state diagram the condition $r_1r_2r_3 = 000$ is indicated by the arc labeled 000. When one or more request signals become 1, the machine moves to one of the grant states, according to the priority scheme. If r_1 is asserted, then device 1 will receive the grant because it has the highest priority. This is indicated by the arc labeled 1xx that leads to state *gnt*1, which sets $g_1 = 1$. The meaning of 1xx is that the request signal r_1 is 1, and the values of signals r_2 and r_3 are irrelevant because of the priority scheme. As before, we use the symbol x to indicate that the value of the corresponding variable can be either 0 or 1. The machine stays in state *gnt*1 as long as r_1 is 1. When $r_1 = 0$, the arc labeled 0xx causes a change on the next positive clock edge back to state *Idle*, and g_1 is deasserted. If other requests are active at this time, then the FSM will change to a new grant state after the next clock edge.

The arc that causes a change to state *gnt*2 is labeled 01x. This label adheres to the priority scheme because it represents the condition that $r_2 = 1$, but $r_1 = 0$. Similarly, the condition for entering state *gnt*3 is given as 001, which indicates that the only request signal asserted is r_3.

The state diagram is repeated in Figure 8.73. The only difference between this diagram and Figure 8.72 is the way in which the arcs are labeled. Figure 8.73 uses a simpler labeling scheme that is more intuitive. For the condition that leads from state *Idle* to state *gnt*1, the arc is labeled as r_1, instead of 1xx. This label means that if $r_1 = 1$, the FSM changes to state *gnt*1, regardless of the other inputs. The arc with the label $\bar{r}_1 r_2$ that leads from state *Idle* to *gnt*2 represents the condition $r_1 r_2 = 01$, while the value of r_3 is irrelevant. There is no standardized scheme for labeling the arcs in state diagrams. Some designers prefer the style of Figure 8.72, while others prefer a style more similar to Figure 8.73.

Figure 8.74 gives the Verilog code for the machine. The three request and grant signals are specified as three-bit vectors *r* and *g*. The FSM transitions are described using a **case** statement in the style used for Figure 8.29. As shown in the state *Idle*, it is easy to describe the required priority scheme by using another nested **case** statement. If $r_1 = 1$, then the next state for the machine is *gnt*1. If r_1 is not asserted, then the alternative is evaluated, which stipulates that if $r_2 = 1$, then the next state will be *gnt*2. Each successive alternative considers a lower-priority request signal only if all of the higher-priority request signals are not asserted.

The transitions for each grant state are straightforward. The FSM stays in state *gnt*1 as long as $r_1 = 1$; when $r_1 = 0$, the next state is *Idle*. The other grant states have the same structure.

The grant signals, g_1, g_2, and g_3 are defined at the end. The value of g_1 is set to 1 when the machine is in state *gnt*1, and otherwise g_1 is set to 0. Similarly, each of the other grant signals is 1 only in the appropriate grant state.

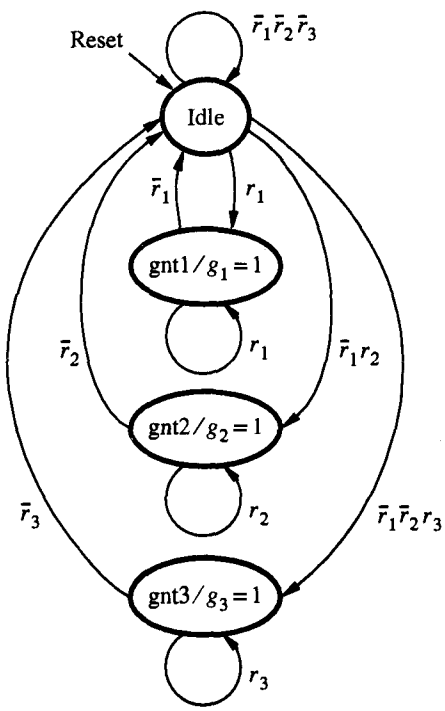


Figure 8.73 Alternative style of state diagram for the arbiter.

```
module arbiter (r, Resetn, Clock, g);
   input [1:3] r;
   input Resetn, Clock;
   output [1:3] g;
   wire [1:3] g;
   reg [2:1] y, Y;
   parameter Idle = 2'b00, gnt1 = 2'b01, gnt2 = 2'b10, gnt3 = 2'b11;

   // Next state combinational circuit
   always @(r or y)
      case (y)
         Idle:   case (r)
                    3'b000    Y = Idle;
                    3'b1xx    Y = gnt1;
                    3'b01x    Y = gnt2;
                    3'b001    Y = gnt3;
                    default:  Y = Idle;
                 endcase
         gnt1:   if (r[1])    Y = gnt1;
                 else         Y = Idle;
         gnt2:   if (r[2])    Y = gnt2;
                 else         Y = Idle;
         gnt3:   if (r[3])    Y = gnt3;
                 else         Y = Idle;
         default:             Y = Idle;
      endcase

   // Sequential block
   always @(posedge Clock)
      if (Resetn == 0)  y <= Idle;
      else  y <= Y;

   // Define output
   assign g[1] = (y == gnt1);
   assign g[2] = (y == gnt2);
   assign g[3] = (y == gnt3);

endmodule
```

Figure 8.74 Verilog code for the arbiter.

8.8.1 IMPLEMENTATION OF THE ARBITER CIRCUIT

We will now consider the effects of implementing the arbiter in both a CPLD and an FPGA. Any differences between the two implementations are likely to be more pronounced if the complexity of the FSM is greater. Hence instead of directly using the code in Figure 8.74, we will implement a larger version of the arbiter that controls eight devices. The request

signals are called $r_1, r_2, \ldots, r_8$, and the grant signals are $g_1, g_2, \ldots, g_8$. It is easy to see how the code in Figure 8.74 is extended to allow eight requesting devices, so we will not show it here.

Implementation in a CPLD

We first consider implementation of the arbiter in a CPLD. To represent the nine states in the FSM, the synthesis tool uses four flip-flops, called y_4, y_3, y_2, and y_1. The reset state, *Idle*, is assigned the code $y_4y_3y_2y_1 = 0000$. The other states are encoded as *gnt*1 = 0001, *gnt*2 = 0010, *gnt*3 = 0100, *gnt*4 = 1000, *gnt*5 = 0011, *gnt*6 = 0101, *gnt*7 = 0110, and *gnt*8 = 1001. This state assignment was chosen by the synthesis tool.

It is not obvious why the tool made this choice. The tool considers many different state assignments and selects one that minimizes the cost of the final circuit. For the CPLD implementation the synthesis tool attempts to choose the state assignment that results in the fewest product terms in the final circuit.

To see the complexity of the circuit, we need to examine the logic expressions generated for both the grant signals and the inputs to the state flip-flops. The expression for each grant signal is a direct result of the encoding used for the state that produces the grant. For instance, state *gnt*8 is encoded as 1001, resulting in $g_8 = y_4\bar{y}_3\bar{y}_2y_1$.

The logic feeding the state flip-flops is more complex. For example, the expression derived by the tool for the input, Y_4, to flip-flop y_4 is

$$Y_4 = \bar{r}_1\bar{r}_2\bar{r}_3\bar{r}_5\bar{r}_6\bar{r}_7r_8\bar{y}_1\bar{y}_2\bar{y}_3\bar{y}_4 + \bar{r}_1\bar{r}_2\bar{r}_3r_4\bar{y}_1\bar{y}_2\bar{y}_3 + r_8\bar{y}_1\bar{y}_2\bar{y}_3y_4 + r_4\bar{y}_1\bar{y}_2\bar{y}_3y_4$$

Figure 8.75 gives a timing simulation for the CPLD implementation. For simplicity only the request signals r_1, r_2, and r_8 are displayed, along with the grant signals g_1, g_2, and g_8. After the machine is reset at the beginning of the simulation, all three requests r_1, r_2, and r_8 are asserted. Although not shown in the timing diagram, all of the other request signals are set to 0. The machine first changes to state *gnt*1 and asserts g_1. After r_1 becomes 0 the machine changes back to state *Idle*. On the next clock cycle a transition to state *gnt*2 takes place and g_2 is asserted. After r_2 becomes 0 the machine changes back to state *Idle*,

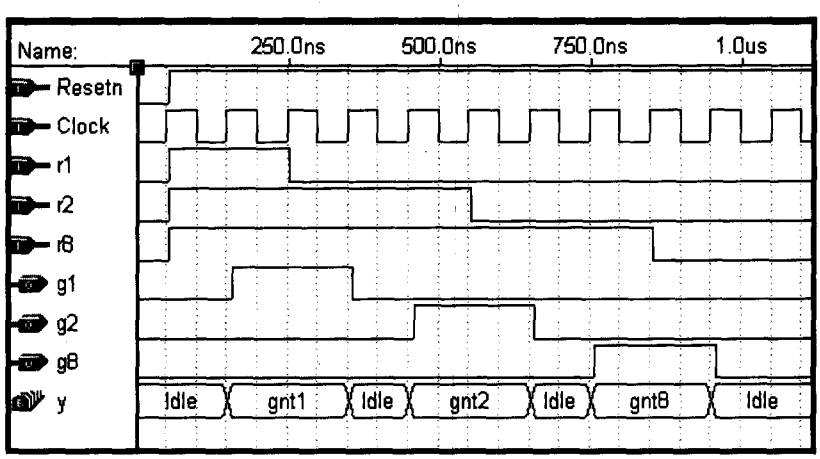


Figure 8.75 Simulation results for the arbiter circuit.

and then to state *gnt*8 to assert g_8. The simulation results indicate that the required priority scheme is properly implemented by our Verilog code.

A more detailed display of a part of the simulation results appears in Figure 8.76. The waveforms are arranged such that only the signals *Clock*, g_8, and y are visible during the time period when g_8 is asserted. The simulation results show that a propagation delay (about 7 ns) is needed for the g_8 signal to be produced after the machine changes to the *gnt*8 state. This delay corresponds to the time needed to generate the function $g_8 = y_1\bar{y}_2\bar{y}_3y_4$. We will show in section 8.8.2 that it is possible to optimize the timing of the implemented circuit such that a grant signal is produced immediately when the machine enters the grant state.

Implementation in an FPGA

Next we consider implementing the arbiter FSM in an FPGA chip. Instead of using four flip-flops to represent the nine states in the FSM, the FPGA implementation generated by the synthesis tool has nine state flip-flops, called $y_9, y_8, \ldots, y_1$. The state assignment is *Idle* = 000000000, *gnt*1 = 110000000, *gnt*2 = 101000000, *gnt*3 = 100100000, *gnt*4 = 100010000, *gnt*5 = 100001000, *gnt*5 = 100000100, *gnt*7 = 100000010, and *gnt*8 = 100000001. This assignment is very similar to the one-hot encoding. The only difference is that the left-most flip-flop output, y_9, is complemented. This is done to provide a simple reset mechanism. When all flip-flops are reset, they define the state represented by all state variables being 0, which is the *Idle* state.

In section 4.6 we discussed the issue of the limited fan-in of the logic gates provided in certain types of chips. We said that in such chips logic functions with a large number of inputs must be decomposed into smaller functions. For an FSM, this means that if the logic circuit that feeds each state flip-flop has many inputs, then several levels of gates may be needed. This increases the propagation delays in the circuit and results in a slower speed of operation. For the preceding CPLD implementation of the arbiter FSM, we showed the logic expression for the input to flip-flop y_4. If that expression were implemented in an FPGA that has four-input lookup tables (LUTs) it would require a total of eight LUTs in a circuit that has three of the LUTs connected in series.

By contrast, the choice of nine state variables with the preceding state assignment results in a much simpler circuit. As an example, for the input to flip-flop y_8, the synthesis tool produces $Y_8 = r_1y_8 + r_1\bar{y}_9$. Since it has only four inputs, this expression can be realized in a single four-input lookup table. The other eight next-state expressions are also relatively simple. To see the effect that the state assignment has on the speed of operation of the FSM, we compared two versions of the circuit implemented in an FPGA chip: one

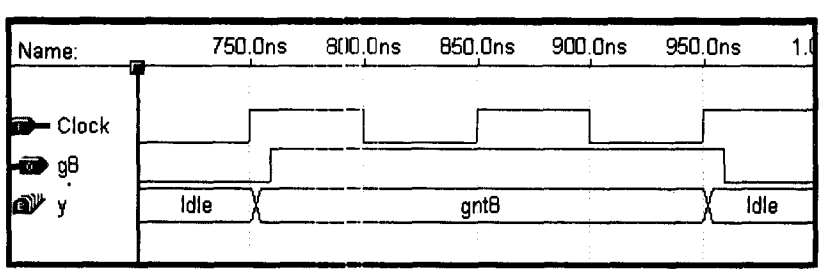


Figure 8.76 Output delays in the arbiter circuit.

that has nine state flip-flops as shown above and another that has four flip-flops with the state assignment given earlier for the CPLD implementation. The results showed that when nine state variables are used, the arbiter FSM works correctly up to a maximum clock rate of 88.5 MHz, whereas when four state variables are used, the maximum clock rate is only 54.1 MHz. Note that the speed of operation of the circuit depends on the specific target chip and can also vary based on the synthesis options selected in the CAD tools.

We should also consider the complexity of the logic needed for the grant signals. These signals are trivial to generate when nine flip-flops are used. Each grant signal is the output of one of the flip-flops. For example, $g_8 = y_1$.

8.8.2 MINIMIZING THE OUTPUT DELAYS FOR AN FSM

Figure 8.76 shows the propagation delay incurred to produce the grant signals when the arbiter circuit is implemented in a CPLD. Once the circuit changes to a grant state, the appropriate grant signal is asserted after a delay of about 7 ns. The delay is caused by the circuitry that generates the grant signal depending on the values of the state flip-flops. However, as we showed in the FPGA implementation, when one-hot encoding is used each grant signal is provided as the output of one of the state flip-flops. Hence no extra circuitry is needed to generate the output signals. Figure 8.77 shows a timing simulation when the arbiter circuit is implemented in a CPLD using one-hot encoding. There is very little delay from when the circuit enters a grant state until the grant signal is produced. A small delay is incurred because of the time needed to propagate through the buffer that exists between the flip-flop output and the pin on the CPLD chip package, but this delay is only about 2 ns. This type of timing optimization is done in practice by designers of sequential circuits, because design specifications often require that outputs be produced after the shortest possible delays.

8.8.3 SUMMARY

Our arbiter FSM is a practical circuit that is useful in many types of systems. An example is a computer system in which various devices in the system are connected by a bus. One aspect of the arbiter may have to be changed for use in such a system. Because of the priority scheme, it is possible that devices with high priority could prevent a lower-priority device from receiving a grant signal for an arbitrarily long time. This condition is often

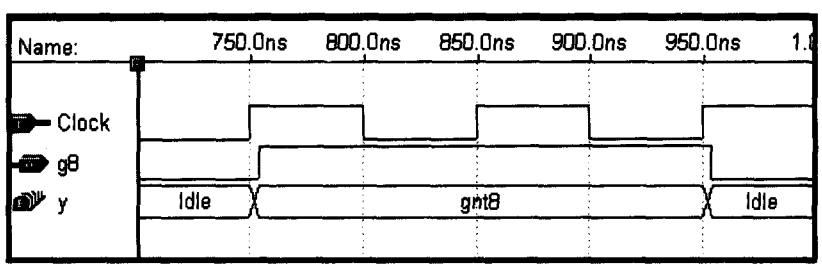


Figure 8.77 Output delays when using one-hot encoding.

called *starvation* of the low-priority device. It is not difficult to modify the arbiter FSM to account for this issue (see problem 8.37).

8.9 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUITS

In addition to knowing how to design a synchronous sequential circuit, the designer has to be able to analyze the behavior of an existing circuit. The analysis task is much simpler than the synthesis task. In this section we will show how analysis may be performed.

To analyze a circuit, we simply reverse the steps of the synthesis process. The outputs of the flip-flops represent the present-state variables. Their inputs determine the next state that the circuit will enter. From this information we can construct the state-assigned table for the circuit. This table leads to a state table and the corresponding state diagram by giving a name to each state. The type of flip-flops used in the circuit is a factor, as we will see in the examples that follow.

Example 8.9 **D-TYPE FLIP-FLOPS** Figure 8.78 gives an FSM that has two D flip-flops. Let y_1 and y_2 be the present-state variables and Y_1 and Y_2 the next-state variables. The next-state and output expressions are

$$Y_1 = w\bar{y}_1 + wy_2$$
$$Y_2 = wy_1 + wy_2$$
$$z = y_1y_2$$

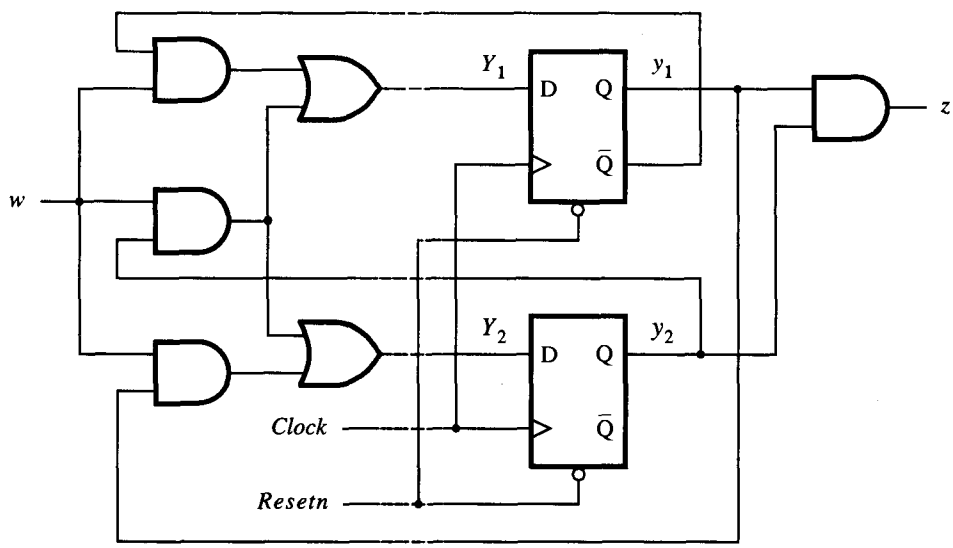


Figure 8.78 Circuit for Example 8.9.

Since there are two flip-flops, the FSM has four states. A good starting point in the analysis is to assume an initial state of the flip-flops such as $y_1 = y_2 = 0$. From the expressions for Y_1 and Y_2, we can derive the state-assigned table in Figure 8.79*a*. For example, in the first row of the table $y_1 = y_2 = 0$. Then $w = 0$ causes $Y_1 = Y_2 = 0$, and $w = 1$ causes $Y_1 = 1$ and $Y_2 = 0$. The output for this state is $z = 0$. The other rows are derived in the same manner. Labeling the states as A, B, C, and D yields the state table in Figure 8.79*b*. From this table it is apparent that following the reset condition the FSM produces the output $z = 1$ whenever three consecutive 1s occur on the input w. Therefore, the FSM acts as a sequence detector for this pattern.

JK-TYPE FLIP-FLOPS Now consider the circuit in Figure 8.80, which has two JK flip-flops. The expressions for the inputs to the flip-flops are **Example 8.10**

Present state y_2y_1	Next State $w = 0$ Y_2Y_1	Next State $w = 1$ Y_2Y_1	Output z
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	0 0	1 1	0
1 1	0 0	1 1	1

(a) State-assigned table

Present state	Next state $w = 0$	Next state $w = 1$	Output z
A	A	B	0
B	A	C	0
C	A	D	0
D	A	D	1

(b) State table

Figure 8.79 Tables for the circuit in Figure 8.78.

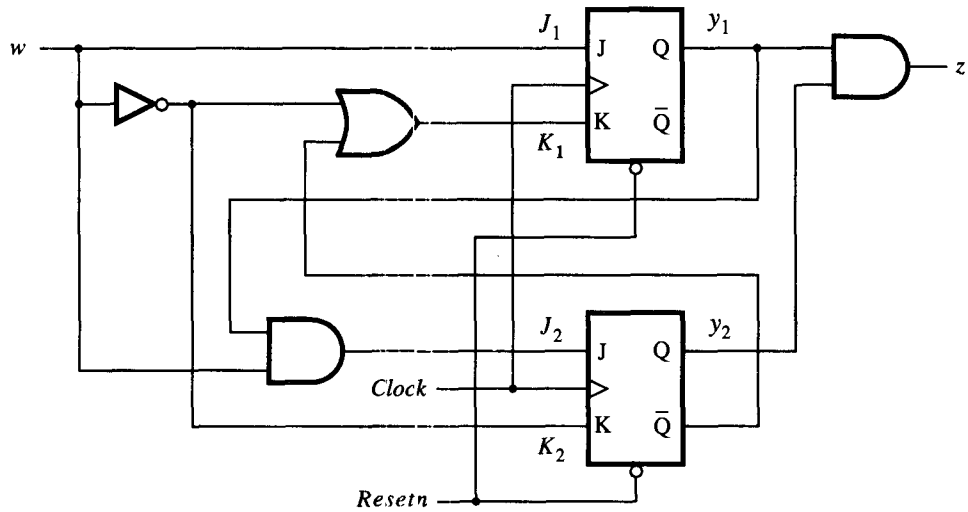


Figure 8.80 Circuit for Example 8.10.

$$J_1 = w$$
$$K_1 = \bar{w} + \bar{y}_2$$
$$J_2 = wy_1$$
$$K_2 = \bar{w}$$

The output is given by $z = y_1 y_2$.

From these expressions we can derive the excitation table in Figure 8.81. Interpreting the entries in this table, we can construct the state-assigned table. For example, consider $y_2y_1 = 00$ and $w = 0$. Then, since $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, both flip-flops will remain in the 0 state; hence $Y_2 = Y_1 = 0$. If $y_2y_1 = 00$ and $w = 1$, then $J_2 = K_2 = 0$ and $J_1 = K_1 = 1$, which leaves the y_2 flip-flop unchanged and sets the y_1 flip-flop to 1; hence

Present state y_2y_1	Flip-flop inputs				Output z
	$w = 0$		$w = 1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
0 0	0 1	0 1	0 0	1 1	0
0 1	0 1	0 1	1 0	1 1	0
1 0	0 1	0 1	0 0	1 0	0
1 1	0 1	0 1	1 0	1 0	1

Figure 8.81 The excitation table for the circuit in Figure 8.80.

$Y_2 = 0$ and $Y_1 = 1$. If $y_2y_1 = 01$ and $w = 0$, then $J_2 = J_1 = 0$ and $K_2 = K_1 = 1$, which resets the y_1 flip-flop and results in the state $y_2y_1 = 00$; hence $Y_2 = Y_1 = 0$. Similarly, if $y_2y_1 = 01$ and $w = 1$, then $J_2 = 1$ and $K_2 = 0$ sets y_2 to 1; hence $Y_2 = 1$, while $J_1 = K_1 = 1$ toggles y_1; hence $Y_1 = 0$. This leads to the state $y_2y_1 = 10$. Completing this process, we find that the resulting state-assigned table is the same as the one in Figure 8.79*a*. The conclusion is that the circuits in Figures 8.78 and 8.80 implement the same FSM.

Example 8.11

MIXED FLIP-FLOPS There is no reason why one cannot use a mixture of flip-flop types in one circuit. Figure 8.82 shows a circuit with one D and one T flip-flop. The expressions for this circuit are

$$D_1 = w(\overline{y}_1 + y_2)$$
$$T_2 = \overline{w}y_2 + wy_1\overline{y}_2$$
$$z = y_1y_2$$

From these expressions we derive the excitation table in Figure 8.83. Since it is a T flip-flop, y_2 changes its state only when $T_2 = 1$. Thus if $y_2y_1 = 00$ and $w = 0$, then because $T_2 = D_1 = 0$ the state of the circuit will not change. An example of where $T_2 = 1$ is when $y_2y_1 = 01$ and $w = 1$, which causes y_2 to change to 1; $D_1 = 0$ makes $y_1 = 0$, hence $Y_2 = 1$ and $Y_1 = 0$. The other cases where $T_2 = 1$ occur when $w = 0$ and $y_2y_1 = 10$ or 11. In both of these cases $D_1 = 0$. Hence the T flip-flop changes its state from 1 to 0, while the D

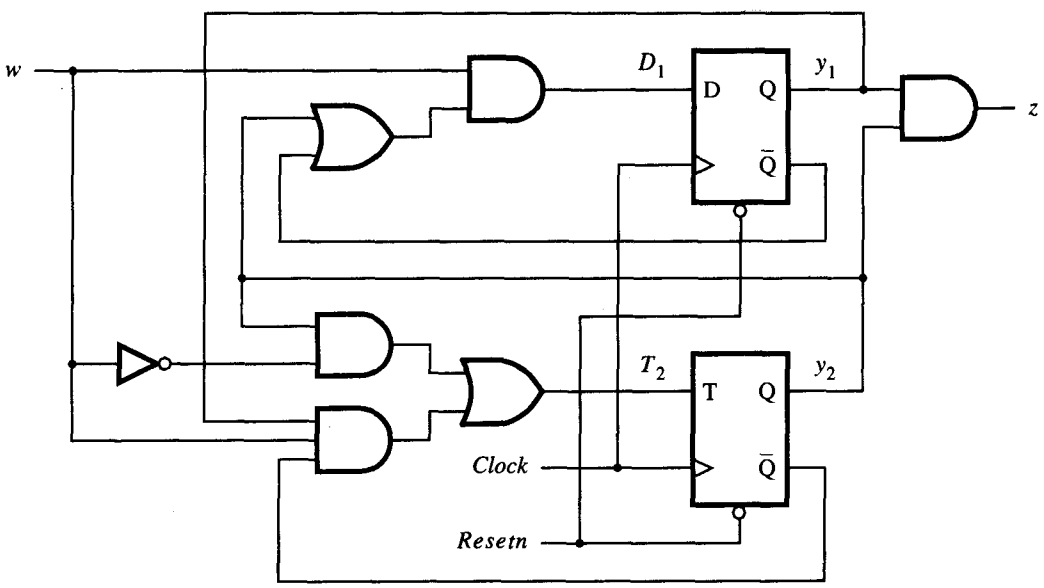


Figure 8.82 Circuit for Example 8.11.

Present state y_2y_1	Flip-flop inputs		Output z
	$w = 0$	$w = 1$	
	T_2D_1	T_2D_1	
0 0	0 0	0 1	0
0 1	0 0	1 0	0
1 0	1 0	0 1	0
1 1	1 0	0 1	1

Figure 8.83 The excitation table for the circuit in Figure 8.82.

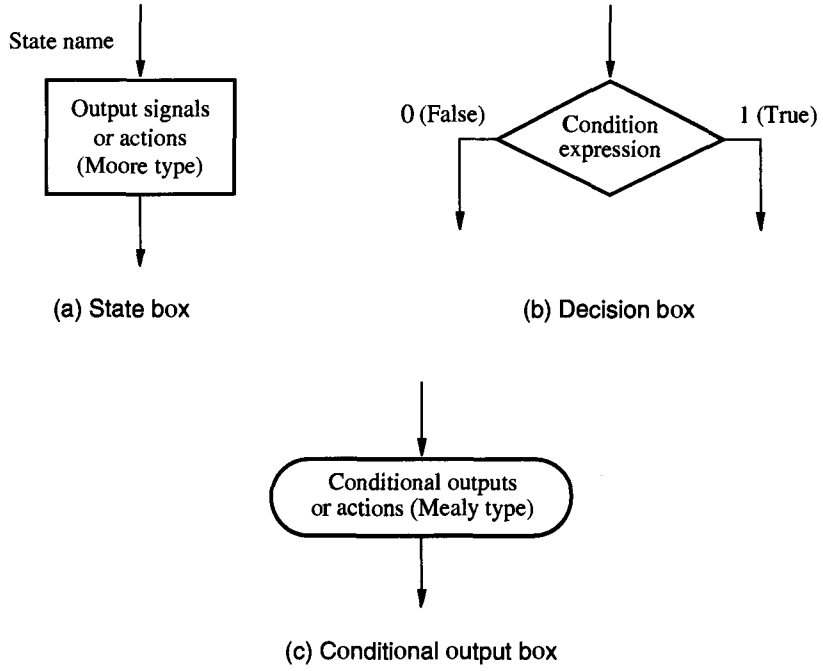
flip-flop is cleared, which means that the next state is $Y_2Y_1 = 00$. Completing this analysis we again obtain the state-assigned table in Figure 8.79*a*. Thus this circuit is yet another implementation of the FSM represented by the state table in Figure 8.79*b*.

8.10 ALGORITHMIC STATE MACHINE (ASM) CHARTS

The state diagrams and tables used in this chapter are convenient for describing the behavior of FSMs that have only a few inputs and outputs. For larger machines the designers often use a different form of representation, called the *algorithmic state machine (ASM) chart.*

An ASM chart is a type of flowchart that can be used to represent the state transitions and generated outputs for an FSM. The three types of elements used in ASM charts are depicted in Figure 8.84.

- **State box** – A rectangle represents a state of the FSM. It is equivalent to a node in the state diagram or a row in the state table. The name of the state is indicated outside the box in the top-left corner. The Moore-type outputs are listed inside the box. These are the outputs that depend only on the values of the state variables that define the state; we will refer to them simply as *Moore outputs*. It is customary to write only the name of the signal that has to be asserted. Thus it is sufficient to write z, rather than $z = 1$, to indicate that the output z must have the value 1. Also, it may be useful to indicate an action that must be taken; for example, $Count \leftarrow Count + 1$ specifies that the contents of a counter have to be incremented by 1. Of course, this is just a simple way of saying that the control signal that causes the counter to be incremented must be asserted. We will use this way of specifying actions in larger systems that are discussed in Chapter 10.
- **Decision box** – A diamond indicates that the stated condition expression is to be tested and the exit path is to be chosen accordingly. The condition expression consists of one or more inputs to the FSM. For example, w indicates that the decision is based on the value of the input w, whereas $w_1 \cdot w_2$ indicates that the true path is taken if $w_1 = w_2 = 1$ and the false path is taken otherwise.



(a) State box

(b) Decision box

(c) Conditional output box

Figure 8.84 Elements used in ASM charts.

- **Conditional output box** – An oval denotes the output signals that are of Mealy type. These outputs depend on the values of the state variables and the inputs of the FSM; we will refer to these outputs simply as *Mealy outputs*. The condition that determines whether such outputs are generated is specified in a decision box.

Figure 8.85 gives the ASM chart that represents the FSM in Figure 8.3. The transitions between state boxes depend on the decisions made by testing the value of the input variable w. In each case if $w = 0$, the exit path from a decision box leads to state A. If $w = 1$, then a transition from A to B or from B to C takes place. If $w = 1$ in state C, then the FSM stays in that state. The chart specifies a Moore output z, which is asserted only in state C, as indicated in the state box. In states A and B, the value of z is 0 (not asserted), which is implied by leaving the corresponding state boxes blank.

Figure 8.86 provides an example with Mealy outputs. This chart represents the FSM in Figure 8.23. The output, z, is equal to 1 when the machine is in state B and $w = 1$. This is indicated using the conditional output box. In all other cases the value of z is 0, which is implied by not specifying z as an output of state B for $w = 0$ and state A for w equal to 0 or 1.

Figure 8.87 gives the ASM chart for the arbiter FSM in Figure 8.73. The decision box drawn below the state box for *Idle* specifies that if $r_1 = 1$, then the FSM changes to state *gnt1*. In this state the FSM asserts the output signal g_1. The decision box to the right of the state box for *gnt1* specifies that as long as $r_1 = 1$, the machine stays in state *gnt1*, and

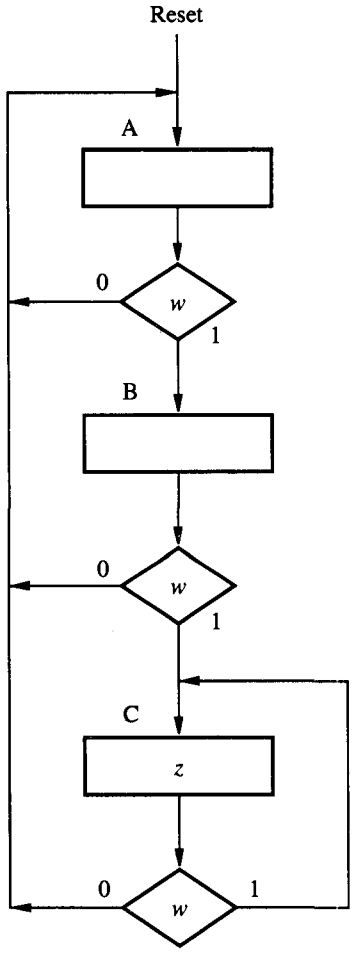


Figure 8.85 ASM chart for the FSM in Figure 8.3.

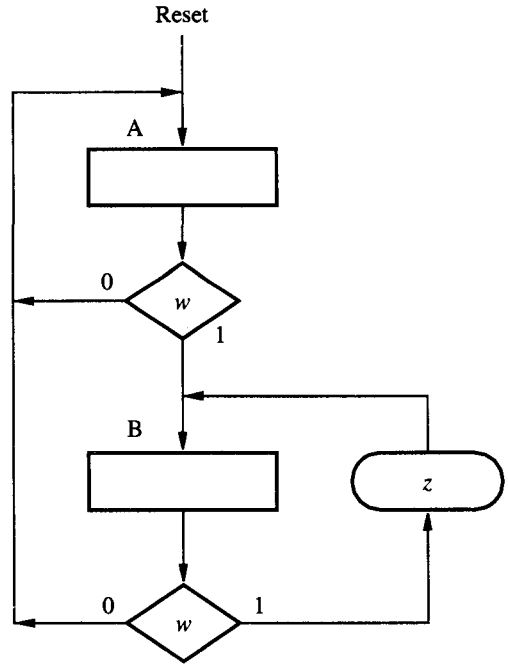


Figure 8.86 ASM chart for the FSM in Figure 8.23.

when $r_1 = 0$, it changes to state *Idle*. The decision box labeled r_2 that is drawn below the state box for *Idle* specifies that if $r_2 = 1$, then the FSM changes to state *gnt2*. This decision box can be reached only after first checking the value of r_1 and following the arrow that corresponds to $r_1 = 0$. Similarly, the decision box labeled r_3 can be reached only if both r_1 and r_2 have the value 0. Hence the ASM chart describes the required priority scheme for the arbiter.

ASM charts are similar to traditional flowcharts. Unlike a traditional flowchart, the ASM chart includes timing information because it implicitly specifies that the FSM changes (flows) from one state to another only after each active clock edge. The examples of ASM charts presented here are quite simple. We have used them to introduce the ASM chart

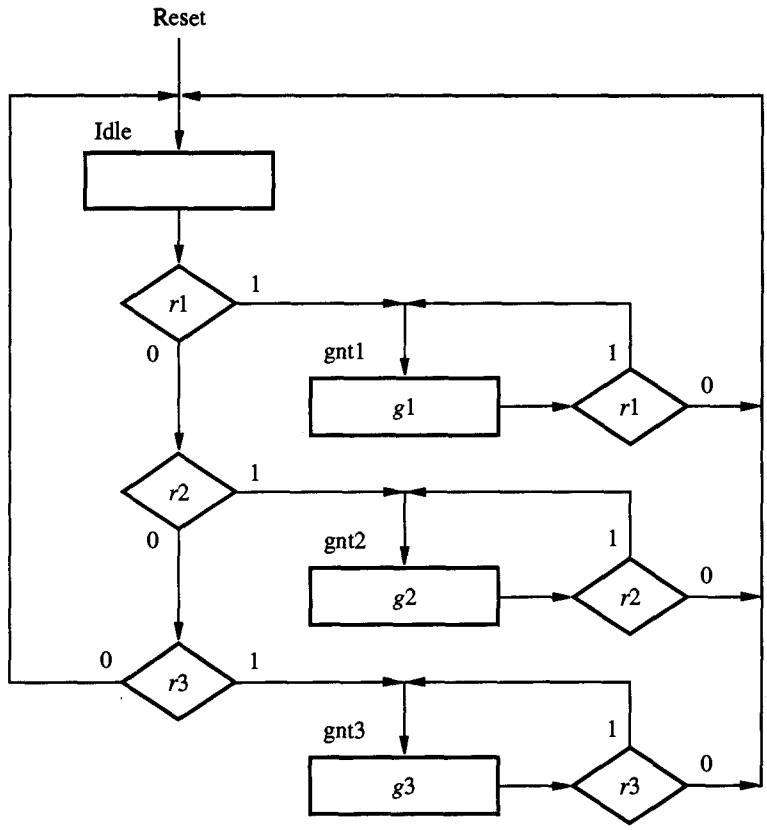


Figure 8.87 ASM chart for the arbiter FSM in Figure 8.73.

terminology by giving examples of state, decision, and conditional-output boxes. Another term sometimes applied to ASM charts is *ASM block*, which refers to a single state box and any decision and conditional-output boxes that the state box may be connected to. The ASM charts can be used to describe complex circuits that include one or more finite state machines and other circuitry such as registers, shift registers, counters, adders, and multipliers. We will use ASM charts as an aid for designing more complex circuits in Chapter 10.

8.11 FORMAL MODEL FOR SEQUENTIAL CIRCUITS

This chapter has presented the synchronous sequential circuits using a rather informal approach because this is the easiest way to grasp the concepts that are essential in designing such circuits. The same topics can also be presented in a more formal manner, which has been the style adopted in many books that emphasize the switching theory aspects rather

than the design using CAD tools. A formal model often gives a concise specification that is difficult to match in a more descriptive presentation. In this section we will describe a formal model that represents a general class of sequential circuits, including those of the synchronous type.

Figure 8.88 represents a general sequential circuit. The circuit has $W = \{w_1, w_2, \ldots, w_n\}$ inputs, $Z = \{z_1, z_2, \ldots, z_m\}$ outputs, $y = \{y_1, y_2, \ldots, y_k\}$ present-state variables, and $Y = \{Y_1, Y_2, \ldots, Y_k\}$ next-state variables. It can have up to 2^k states, $S = \{S_1, S_2, \ldots, S_{2^k}\}$. There are delay elements in the feedback paths for the state-variables which ensure that y will take the values of Y after a time delay Δ. In the case of synchronous sequential circuits, the delay elements are flip-flops, which change their state on the active edge of a clock signal. Thus the delay Δ is determined by the clock period. The clock period must be long enough to allow for the propagation delay in the combinational circuit, in addition to the setup and hold parameters of the flip-flops.

Using the model in Figure 8.88, a synchronous sequential circuit, M, can be defined formally as a quintuple

$$M = (W, Z, S, \varphi, \lambda)$$

where

- W, Z, and S are finite, nonempty sets of inputs, outputs, and states, respectively.
- φ is the state transition function, such that $S(t + 1) = \varphi[W(t), S(t)]$.
- λ is the output function, such that $\lambda(t) = \lambda[S(t)]$ for the Moore model and $\lambda(t) = \lambda[W(t), S(t)]$ for the Mealy model.

This definition assumes that the time between t and $t + 1$ is one clock cycle.

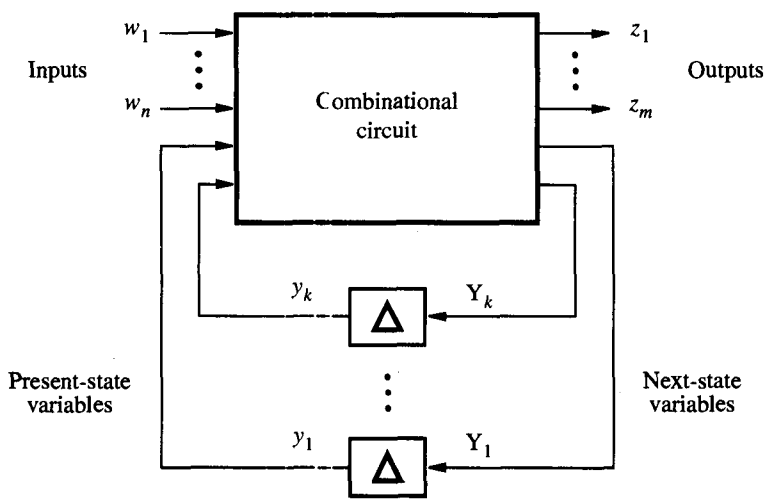


Figure 8.88 The general model for a sequential circuit.

We will see in the next chapter that the delay Δ need not be controlled by a clock. In asynchronous sequential circuits the delays are due solely to the propagation delays through various gates.

8.12 CONCLUDING REMARKS

The existence of closed loops and delays in a sequential circuit leads to a behavior that is characterized by the set of states that the circuit can reach. The present values of the inputs are not the sole determining factor in this behavior, because a given valuation of inputs may cause the circuit to behave differently in different states.

The propagation delays through a sequential circuit must be taken into account. The design techniques presented in this chapter are based on the assumption that all changes in the circuit are triggered by the active edge of a clock signal. Such circuits work correctly only if all internal signals are stable when the clock signal arrives. Thus the clock period must be longer than the longest propagation delay in the circuit.

Synchronous sequential circuits are used extensively in practical designs. They are supported by the commonly used CAD tools. All textbooks on the design of logic circuits devote considerable space to synchronous sequential circuits. Some of the more notable references are [1–14].

In the next chapter we will present a different class of sequential circuits, which do not use flip-flops to represent the states of the circuit and do not use clock pulses to trigger changes in the states.

PROBLEMS

8.1 An FSM is defined by the state-assigned table in Figure P8.1. Derive a circuit that realizes this FSM using D flip-flops.

Present state y_2y_1	Next state $w=0$ Y_2Y_1	Next state $w=1$ Y_2Y_1	Output z
00	10	11	0
01	01	00	0
10	11	00	0
11	10	01	1

Figure P8.1 State-assigned table for problems 8.1 and 8.2.

8.2 Derive a circuit that realizes the FSM defined by the state-assigned table in Figure P8.1 using JK flip-flops.

8.3 Derive the state diagram for an FSM that has an input w and an output z. The machine has to generate $z = 1$ when the previous four values of w were 1001 or 1111; otherwise, $z = 0$. Overlapping input patterns are allowed. An example of the desired behavior is

$$w : 0101111001100111111$$
$$z : 0000001001000100111$$

8.4 Write Verilog code for the FSM described in problem 8.3.

8.5 Derive a minimal state table for a single-input and single-output Moore-type FSM that produces an output of 1 if in the input sequence it detects either 110 or 101 patterns. Overlapping sequences should be detected.

8.6 Repeat problem 8.5 for a Mealy-type FSM.

8.7 Derive the circuits that implement the state tables in Figures 8.51 and 8.52. What is the effect of state minimization on the cost of implementation?

8.8 Derive the circuits that implement the state tables in Figures 8.55 and 8.56. Compare the costs of these circuits.

8.9 A sequential circuit has two inputs, w_1 and w_2, and an output, z. Its function is to compare the input sequences on the two inputs. If $w_1 = w_2$ during any four consecutive clock cycles, the circuit produces $z = 1$; otherwise, $z = 0$. For example

$$w_1 : 0110111000110$$
$$w_2 : 1110101000111$$
$$z : 0000100001110$$

Derive a suitable circuit.

8.10 Write Verilog code for the FSM described in problem 8.9.

8.11 A given FSM has an input, w, and an output, z. During four consecutive clock pulses, a sequence of four values of the w signal is applied. Derive a state table for the FSM that produces $z = 1$ when it detects that either the sequence $w : 0010$ or $w : 1110$ has been applied; otherwise, $z = 0$. After the fourth clock pulse, the machine has to be again in the reset state, ready for the next sequence. Minimize the number of states needed.

8.12 Derive a minimal state table for an FSM that acts as a three-bit parity generator. For every three bits that are observed on the input w during three consecutive clock cycles, the FSM generates the parity bit $p = 1$ if and only if the number of 1s in the three-bit sequence is odd.

8.13 Write Verilog code for the FSM described in problem 8.12.

8.14 Draw timing diagrams for the circuits in Figures 8.43 and 8.47, assuming the same changes in a and b signals for both circuits. Account for propagation delays.

8.15 Show a state table for the state-assigned table in Figure P8.1, using A, B, C, D for the four rows in the table. Give a new state-assigned table using a one-hot encoding. For A use the code $y_4y_3y_2y_1 = 0001$. For states B, C, D use the codes 0010, 0100, and 1000, respectively. Synthesize a circuit using D flip-flops.

8.16 Show how the circuit derived in problem 8.15 can be modified such that the code $y_4y_3y_2y_1 = 0000$ is used for the reset state, A, and the other codes for state B, C, D are changed as needed. (Hint: you do not have to resynthesize the circuit!)

8.17 In Figure 8.59 assume that the unspecified outputs in states B and G are 0 and 1, respectively. Derive the minimized state table for this FSM.

8.18 In Figure 8.59 assume that the unspecified outputs in states B and G are 1 and 0, respectively. Derive the minimized state table for this FSM.

8.19 Derive circuits that implement the FSMs defined in Figures 8.57 and 8.58. Can you draw any conclusions about the complexity of circuits that implement Moore and Mealy types of machines?

8.20 Design a counter that counts pulses on line w and displays the count in the sequence 0, 2, 1, 3, 0, 2, ... Use D flip-flops in your circuit.

8.21 Repeat problem 8.20 using JK flip-flops.

8.22 Repeat problem 8.20 using T flip-flops.

8.23 Design a modulo-6 counter, which counts in the sequence 0, 1, 2, 3, 4, 5, 0, 1, ... The counter counts the clock pulses if its enable input, w, is equal to 1. Use D flip-flops in your circuit.

8.24 Repeat problem 8.23 using JK flip-flops.

8.25 Repeat problem 8.23 using T flip-flops.

8.26 Design a three-bit counterlike circuit controlled by the input w. If $w = 1$, then the counter adds 2 to its contents, wrapping around if the count reaches 8 or 9. Thus if the present state is 8 or 9, then the next state becomes 0 or 1, respectively. If $w = 0$, then the counter subtracts 1 from its contents, acting as a normal down-counter. Use D flip-flops in your circuit.

8.27 Repeat problem 8.26 using JK flip-flops.

8.28 Repeat problem 8.26 using T flip-flops.

8.29 Derive the state table for the circuit in Figure P8.2. What sequence of input values on wire w is detected by this circuit?

8.30 Write Verilog code for the FSM shown in Figure 8.57, using the style of code in Figure 8.29.

8.31 Repeat problem 8.30, using the style of code in Figure 8.34.

8.32 Write Verilog code for the FSM shown in Figure 8.58, using the style of code in Figure 8.29.

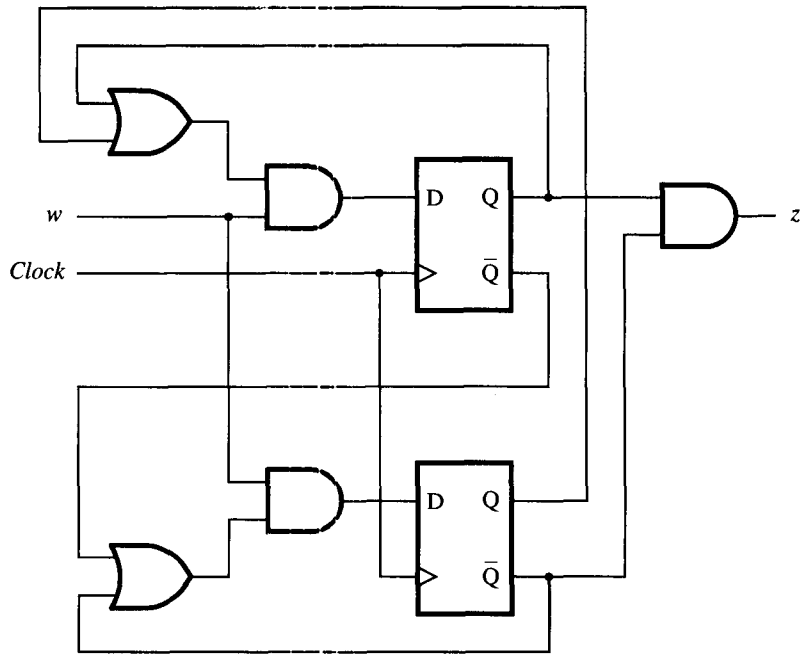


Figure P8.2 Circuit for problem 8.29.

8.33 Repeat problem 8.32, using the style of code in Figure 8.34.

8.34 Write Verilog code for the FSM shown in Figure P8.1.

8.35 Represent the FSM in Figure 8.57 in form of an ASM chart.

8.36 Represent the FSM in Figure 8.58 in form of an ASM chart.

8.37 The arbiter FSM defined in section 8.8 (Figure 8.72) may cause device 3 to never get serviced if devices 1 and 2 continuously keep raising requests, so that in the Idle state it always happens that either device 1 or device 2 has an outstanding request. Modify the proposed FSM to ensure that device 3 will get serviced, such that if it raises a request, the devices 1 and 2 will be serviced only once before the device 3 is granted its request.

8.38 Write Verilog code for the FSM designed in problem 8.37.

8.39 Consider a more general version of the task presented in Example 8.1. Assume that there are four n-bit registers connected to a bus in a processor. The contents of register R are placed on the bus by asserting the control signal R_{out}. The data on the bus are loaded into register R on the active edge of the clock signal if the control signal R_{in} is asserted. Assume that three of the registers, called $R1$, $R2$, and $R3$, are used as normal registers. The fourth register, called *TEMP*, is used for temporary storage in special cases.

We want to realize an operation SWAP Ri,Rj, which swaps the contents of registers *Ri* and *Rj*. This is accomplished by the following sequence of steps (each performed in one clock cycle)

$$\begin{aligned} \text{TEMP} &\leftarrow [\text{Rj}] \\ \text{Rj} &\leftarrow [\text{Ri}] \\ \text{Ri} &\leftarrow [\text{TEMP}] \end{aligned}$$

Two input signals, w_1 and w_2, are used to indicate that two registers have to be swapped as follows

$$\begin{aligned} &\text{If } w_2w_1 = 01, \text{ then swap } R1 \text{ and } R2. \\ &\text{If } w_2w_1 = 10, \text{ then swap } R1 \text{ and } R3. \\ &\text{If } w_2w_1 = 11, \text{ then swap } R2 \text{ and } R3. \end{aligned}$$

An input valuation that specifies a swap is present for three clock cycles. Design a circuit that generates the required control signals: $R1_{out}$, $R1_{in}$, $R2_{out}$, $R2_{in}$, $R3_{out}$, $R3_{in}$, $TEMP_{out}$, and $TEMP_{in}$. Derive the next-state and output expressions for this circuit, trying to minimize the cost.

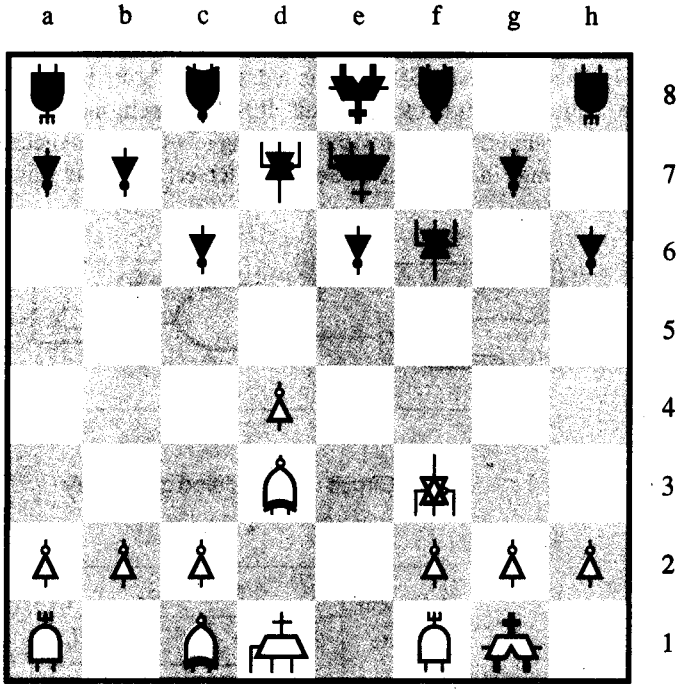
REFERENCES

1. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, (PWS Publishing Co.: 1997).
2. D. D. Gajski, *Principles of Digital Design*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
3. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. J. P. Daniels, *Digital Design from Zero to One*, (Wiley: New York, 1996).
5. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
6. R. H. Katz, *Contemporary Logic Design*, (Benjamin/Cummings: Redwood City, CA, 1994).
7. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., (Wiley: New York, 1993).
8. J. P. Hayes, *Introduction to Logic Design*, (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed., (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices*, (Prentice-Hall: Engelwood Cliffs, NJ, 1990).
11. E. J. McCluskey, *Logic Design Principles*, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
12. T. L. Booth, *Digital Networks and Computer Systems*, (Wiley: New York, 1971).
13. Z. Kohavi, *Switching and Finite Automata Theory*, (McGraw-Hill: New York, 1970).
14. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, (Prentice-Hall: Englewood Cliffs, NJ, 1966).

chapter

9

ASYNCHRONOUS SEQUENTIAL CIRCUITS



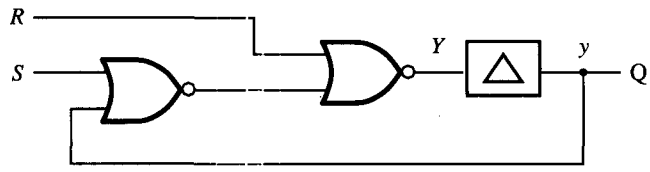
9. 0-0, f7xe6

In the previous chapter we covered the design of synchronous sequential circuits in which the state variables are represented by flip-flops that are controlled by a clock. The clock is a periodic signal that consists of pulses. Changes in state can occur on the positive or negative edge of each clock pulse. Since they are controlled by pulses, synchronous sequential circuits are said to operate in *pulse mode*. In this chapter we present sequential circuits that do not operate in pulse mode and do not use flip-flops to represent state variables. These circuits are called *asynchronous sequential circuits*.

In an asynchronous sequential circuit, changes in state are not triggered by clock pulses. Instead, changes in state are dependent on whether each of the inputs to the circuit has the logic level 0 or 1 at any given time. To achieve reliable operation, the inputs to the circuit must change one at a time. Moreover, there must be sufficient time between the changes in input signals to allow the circuit to reach a *stable state*, which is achieved when all internal signals stop changing. A circuit that adheres to these constraints is said to operate in the *fundamental mode*.

9.1 ASYNCHRONOUS BEHAVIOR

To introduce asynchronous sequential circuits, we will reconsider the basic latch circuit in Figure 7.4. This Set-Reset (SR) latch is redrawn in Figure 9.1*a*. The feedback loop gives rise to the sequential nature of the circuit. It is an asynchronous circuit because changes in the value of the output, Q, occur without having to wait for a synchronizing clock pulse. In response to a change in either the S (Set) or R (Reset) input, the value of Q will change after



(a) Circuit with modeled gate delay

Present state y	Next state $SR = 00$	01	10	11
	Y	Y	Y	Y
0	(0)	(0)	1	(0)
1	(1)	0	(1)	0

(b) State-assigned table

Figure 9.1 Analysis of the SR latch.

a short propagation time through the NOR gates. In Figure 9.1*a* the combined propagation delay through the two NOR gates is represented by the box labeled Δ. Then, the NOR gate symbols represent ideal gates with zero delay. Using the notation in Chapter 8, Q corresponds to the *present state* of the circuit, represented by the *present-state variable*, y. The value of y is fed back through the circuit to generate the value of the *next-state variable*, Y, which represents the *next state* of the circuit. After the Δ time delay, y takes the value of Y. Observe that we have drawn the circuit in a style that conforms to the general model for sequential circuits presented in Figure 8.88.

By analyzing the SR latch, we can derive a state-assigned table, as illustrated in Figure 9.1*b*. When the present state is $y = 0$ and the inputs are $S = R = 0$, the circuit produces $Y = 0$. Since $y = Y$, the state of the circuit will not change. We say that the circuit is *stable* under these input conditions. Now assume that R changes to 1 while S remains at 0. The circuit still generates $Y = 0$ and remains stable. Assume next that S changes to 1 and R remains at 1. The value of Y is unchanged, and the circuit is stable. Then let R change to 0 while S remains at 1. This input valuation, $SR = 10$, causes the circuit to generate $Y = 1$. Since $y \neq Y$, the circuit is not stable. After the Δ time delay, the circuit changes to the new present state $y = 1$. Once this new state is reached, the value of Y remains equal to 1 as long as $SR = 10$. Hence the circuit is again stable. The analysis for the present state $y = 1$ can be completed using similar reasoning.

The concept of stable states is very important in the context of asynchronous sequential circuits. For a given valuation of inputs, if a circuit reaches a particular state and remains in this state, then the state is said to be stable. To clearly indicate the conditions under which the circuit is stable, it is customary to encircle the stable states in the table, as illustrated in Figure 9.1*b*.

From the state-assigned table, we can derive the state table in Figure 9.2*a*. The state names A and B represent the present states $y = 0$ and $y = 1$, respectively. Since the output Q depends only on the present state, the circuit is a Moore-type FSM. The state diagram that represents the behavior of this FSM is shown in Figure 9.2*b*.

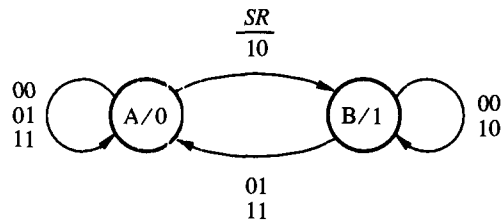
The preceding analysis shows that the behavior of an asynchronous sequential circuit can be represented as an FSM in a similar way as the synchronous sequential circuits in Chapter 8. Consider now performing the opposite task. That is, given the state table in Figure 9.2*a*, we can synthesize an asynchronous circuit as follows: After performing the state assignment, we have the state-assigned table in Figure 9.1*b*. This table represents a truth table for Y, with the inputs y, S, and R. Deriving a minimal product-of-sums expression yields

$$Y = \overline{R} \cdot (S + y)$$

If we were deriving a synchronous sequential circuit using the methods in Chapter 8, then Y would be connected to the D input of a flip-flop and a clock signal would be used to control the time when the changes in state take place. But since we are synthesizing an asynchronous circuit, we do not insert a flip-flop in the feedback path. Instead, we create a circuit that realizes the preceding expression using the necessary logic gates, and we feed back the output signal as the present-state input y. Implementation using NOR gates results in the circuit in Figure 9.1*a*. This simple example suggests that asynchronous circuits and synchronous circuits can be synthesized using similar techniques. However, we will see

Present state	Next state				Output Q
	$SR = 00$	01	10	11	
A	(A)	(A)	B	(A)	0
B	(B)	A	(B)	A	1

(a) State table



(b) State diagram

Figure 9.2 FSM model for the SR latch.

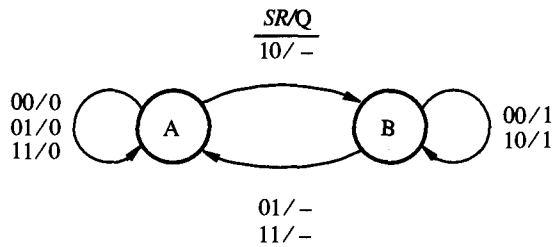
shortly that for more complex asynchronous circuits, the design task is considerably more difficult.

To further explore the nature of asynchronous circuits, it is interesting to consider how the behavior of the SR latch can be represented in the form of a Mealy model. As depicted in Figure 9.3, the outputs produced when the circuit is in a stable state are the same as in the Moore model, namely 0 in state A and 1 in state B. Consider now what happens when the state of the circuit changes. Suppose that the present state is A and that the input valuation SR changes from 00 to 10. As the state table specifies, the next state of the FSM is B. When the circuit reaches state B, the output Q will be 1. But in the Mealy model, the output is supposed to be affected immediately by a change in the input signals. Thus while still in state A, the change in SR to 10 should result in Q = 1. We could have written a 1 in the corresponding entry in the top row of the state table, but we have chosen to leave this entry unspecified instead. The reason is that since Q will change to 1 as soon as the circuit reaches state B, there is little to be gained in trying to make Q go to 1 a little sooner. Leaving the entry unspecified allows us to assign either 0 or 1 to it, which may make the circuit that implements the state table somewhat simpler. A similar reasoning leads to the conclusion that the two output entries where a change from B to A takes place can also be left unspecified.

Using the state assignment $y = 0$ for A and $y = 1$ for B, the state-assigned table represents a truth table for both Y and Q. The minimal expression for Y is the same as for the Moore model. To derive an expression for Q, we need to set the unspecified entries to

Present state	Next state				Output, Q			
	$SR = 00$	01	10	11	00	01	10	11
A	(A)	(A)	B	(A)	0	0	–	0
B	(B)	A	(B)	A	1	–	1	–

(a) State table



(b) State diagram

Figure 9.3 Mealy representation of the SR latch.

0 or 1. Assigning a 0 to the unspecified entry in the first row and 1 to the two unspecified entries in the second row produces $Q = y$ and results in the circuit in Figure 9.1*a*.

Terminology

In the preceding discussion we used the same terminology as in the previous chapter on synchronous sequential circuits. However, when dealing with asynchronous sequential circuits, it is customary to use two different terms. Instead of a "state table," it is more common to speak of a *flow table*, which indicates how the changes in state flow as a result of the changes in the input signals. Instead of a "state-assigned table," it is usual to refer to a *transition table* or an *excitation table*. We will use the terms *flow table* and *excitation table* in this chapter. A flow table will define the state changes and outputs that must be generated. An excitation table will depict the transitions in terms of the state variables. The term excitation table derives from the fact that a change from a stable state is performed by "exciting" the next-state variables to start changing towards a new state.

9.2 ANALYSIS OF ASYNCHRONOUS CIRCUITS

To gain familiarity with asynchronous circuits, it is useful to analyze a few examples. We will keep in mind the general model in Figure 8.88, assuming that the delays in the feedback

paths are a representation of the propagation delays in the circuit. Then each gate symbol will represent an ideal gate with zero delay.

Example 9.1 **GATED D LATCH** In Chapters 7 and 8, we used the gated D latch as a key component in circuits that are controlled by a synchronizing clock. It is instructive to analyze this latch as an asynchronous circuit, where the clock is just one of the inputs. It is reasonable to assume that the signals on the D and clock inputs do not change at the same time, thus meeting the basic requirement of asynchronous circuits.

Figure 9.4*a* shows the gated D latch drawn in the style of the model of Figure 8.88. This circuit was introduced in Figure 7.8 and discussed in section 7.3. The next-state expression for this circuit is

$$\begin{aligned} Y &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y) \\ &= CD + \overline{C}y + Dy \end{aligned}$$

The term Dy in this expression is redundant and could be deleted without changing the logic function of Y. Hence the minimal expression is

$$Y = CD + \overline{C}y$$

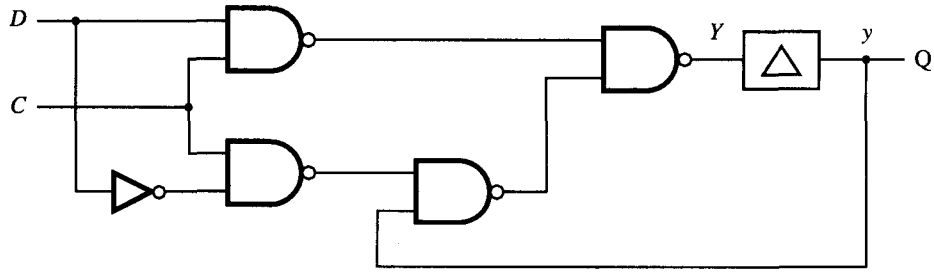
The reason that the circuit implements the redundant term Dy is that this term solves a race condition known as a *hazard*; we will discuss hazards in detail in section 9.6.

Evaluating the expression for Y for all valuations of C, D, and y leads to the excitation table in Figure 9.4*b*. Note that the circuit changes its state only when $C = 1$ and D is different from the present state, y. In all other cases the circuit is stable. Using the symbols A and B to represent the states $y = 0$ and $y = 1$, we obtain the flow table and the state diagram shown in parts (*c*) and (*d*).

Example 9.2 **MASTER-SLAVE D FLIP-FLOP** In Example 9.1 we analyzed the gated D latch as an asynchronous circuit. Actually, all practical circuits are asynchronous. However, if the circuit's behavior is tightly controlled by a clock signal, then simpler operating assumptions can be used, as we did in Chapter 8. Recall that in a synchronous sequential circuit all signals change values in synchronization with the clock signal. Now we will analyze another synchronous circuit as if it were an asynchronous circuit.

Two gated D latches are used to implement the master-slave D flip-flop, as illustrated in Figure 7.10. This circuit is reproduced in Figure 9.5. We can analyze the circuit by treating it as a series connection of two gated D latches. Using the results from Example 9.1, the simplified next-state expressions can be written as

$$\begin{aligned} Y_m &= CD + \overline{C}y_m \\ Y_s &= \overline{C}y_m + Cy_s \end{aligned}$$



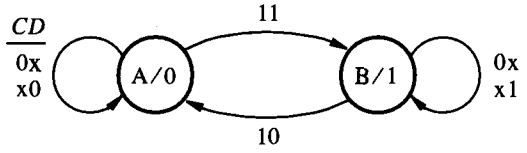
(a) State diagram

Present state y	Next state $CD = 00$	01	10	11	Q
	Y	Y	Y	Y	
0	(0)	(0)	(0)	1	0
1	(1)	(1)	0	(1)	1

(b) Excitation table

Present state	Next state $CD = 00$	01	10	11	Q
A	(A)	(A)	(A)	B	0
B	(B)	(B)	A	(B)	1

(c) Flow table



(d) State diagram

Figure 9.4 The gated D latch.

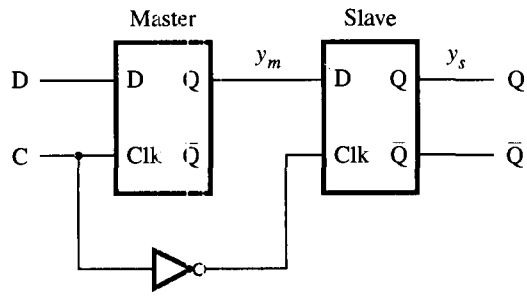


Figure 9.5 Circuit for the master-slave D flip-flop.

where the subscripts m and s refer to the master and slave stages of the flip-flop. These expressions lead to the excitation table in Figure 9.6*a*. Labeling the four states as $S1$ through $S4$, we derive the flow table in Figure 9.6*b*. A state-diagram form of this information is given in Figure 9.7.

Let us consider the behavior of this FSM in more detail. The state $S1$, where $y_m y_s = 00$, is stable for all input valuations except $CD = 11$. When $C = 1$, the value of D is stored in the master stage; hence $CD = 11$ causes the flip-flop to change to S3, where $y_m = 1$ and $y_s = 0$. If the D input now changes back to 0, while the clock remains at 1, the flip-flop moves back to the state $S1$. The transitions between $S1$ and $S3$ indicate that if $C = 1$, the output of the master stage, $Q_m = y_m$, tracks the changes in the D input signal without affecting the slave stage. From $S3$ the circuit changes to $S4$ when the clock goes to 0. In $S4$ both master and slave stages are set to 1 because the information from the master stage is transferred to the slave stage on the negative edge of the clock. Now the flip-flop remains in $S4$ until the clock goes to 1 and the D input changes to 0, which causes a change to $S2$. In $S2$ the master stage is cleared to 0, but the slave stage remains at 1. Again the flip-flop may change between $S2$ and $S4$ because the master stage will track the changes in the D input signal while $C = 1$. From $S2$ the circuit changes to $S1$ when the clock goes low.

In Figures 9.6 and 9.7, we indicated that the flip-flop has only one output Q, which one sees when the circuit is viewed as a negative-edge-triggered flip-flop. From the observer's point of view, the flip-flop has only two states, 0 and 1. But internally, the flip-flop consists of the master and slave parts, which gives rise to the four states described above.

We should also examine the basic assumption that the inputs must change one at a time. If the circuit is stable in state $S2$, for which $CD = 10$, it is impossible to go from this state to $S1$ under the influence of the input valuation $CD = 01$ because this simultaneous change in both inputs cannot occur. Thus in the second row of the flow table, instead of showing $S2$ changing to $S1$ under $CD = 01$, this entry can be labeled as unspecified. The change from $S2$ to $S1$ can be caused only by CD changing from 10 to 00. Similarly, if the circuit is in state $S3$, where $CD = 11$, it cannot change to $S4$ by having $CD = 00$. This entry can also be left unspecified in the table. The resulting flow table is shown in Figure 9.6*c*.

If we reverse the analysis procedure and, using the state assignment in Figure 9.6*a*, synthesize logic expressions for Y_m and Y_s, we get

Present state $y_m y_s$	Next state $Y_m Y_s$				Output Q
	$CD = 00$	01	10	11	
00	(00)	(00)	(00)	10	0
01	00	00	(01)	11	1
10	11	11	00	(10)	0
11	(11)	(11)	01	(11)	1

(a) Excitation table

Present state	Next state				Output Q
	$CD = 00$	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	S1	(S2)	S4	1
S3	S4	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(b) Flow table

Present state	Next state				Output Q
	$CD = 00$	01	10	11	
S1	(S1)	(S1)	(S1)	S3	0
S2	S1	–	(S2)	S4	1
S3	–	S4	S1	(S3)	0
S4	(S4)	(S4)	S2	(S4)	1

(c) Flow table with unspecified entries

Figure 9.6 Excitation and flow tables for Example 9.2.

$$Y_m = CD + \overline{C}y_m + y_m D$$
$$Y_s = \overline{C}y_m + Cy_s + y_m y_s$$

The terms $y_m D$ and $y_m y_s$ in these expressions are redundant. As mentioned earlier, they are included in the circuit to avoid race conditions, which are discussed in section 9.6.

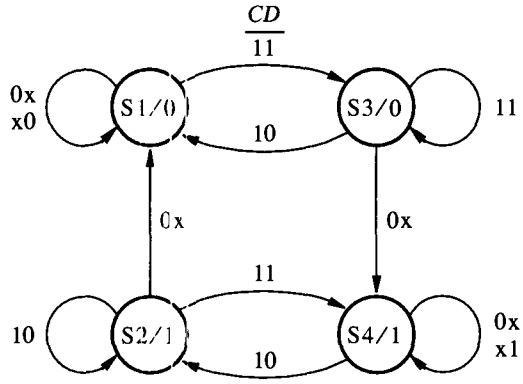


Figure 9.7 State diagram for the master-slave D flip-flop.

Example 9.3 Consider the circuit in Figure 9.8. It is represented by the following expressions

$$Y_1 = y_1\bar{y}_2 + w_1\bar{y}_2 + \bar{w}_1\bar{w}_2y_1$$
$$Y_2 = y_1y_2 + w_1y_2 + w_2 + \bar{w}_1\bar{w}_2y_1$$
$$z = \bar{y}_1y_2$$

The corresponding excitation and flow tables are given in Figure 9.9.

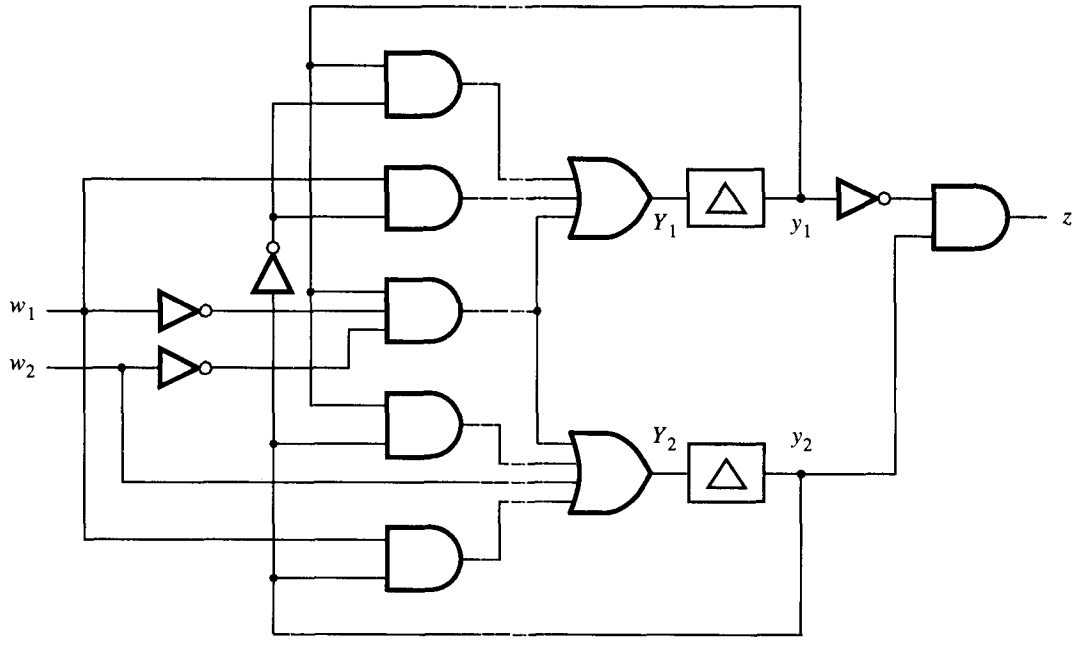


Figure 9.8 Circuit for Example 9.3.

Present state y_2y_1	Next state $w_2w_1 = 00$	01	10	11	Output z
	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	
00	(00)	01	10	11	0
01	11	(01)	11	11	0
10	00	(10)	(10)	(10)	1
11	(11)	10	10	10	0

(a) Excitation table

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z
A	(A)	B	C	D	0
B	D	(B)	D	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

(b) Flow table

Figure 9.9 Excitation and flow tables for the circuit in Figure 9.8

Some transitions in the flow table will not occur in practice because of the assumption that both w_1 and w_2 cannot change simultaneously. In state A the circuit is stable under the valuation $w_2w_1 = 00$. Its inputs cannot change to 11 without passing through the valuations 01 or 10, in which case the new state would be B or C, respectively. Thus the transition from A under $w_2w_1 = 11$ can be left unspecified. Similarly, if the circuit is stable in state B, in which case $w_2w_1 = 01$, it is impossible to force a change to state D by changing the inputs to $w_2w_1 = 10$. This entry should also be unspecified. If the circuit is stable in state C under $w_2w_1 = 11$, it is not possible to go to A by changing the inputs directly to $w_2w_1 = 00$. However, the transition to A is possible by changing the inputs one at a time because the circuit remains stable in C for both $w_2w_1 = 01$ and $w_2w_1 = 10$.

A different situation arises if the circuit is stable in state D under $w_2w_1 = 00$. It may seem that the entry under $w_2w_1 = 11$ should be unspecified because this input change cannot be made from the stable state D. But suppose that the circuit is stable in state B under $w_2w_1 = 01$. Now let the inputs change to $w_2w_1 = 11$. This causes a change to state D. The circuit indeed changes to D, but it is not stable in this state for this input condition. As soon as it arrives into state D, the circuit proceeds to change to state C as required by $w_2w_1 = 11$. It is then stable in state C as long as both inputs remain at 1. The conclusion is that the entry that specifies the change from D to C under $w_2w_1 = 11$ is meaningful and

should not be omitted. The transition from the stable state B to the stable state C, which passes through state D, illustrates that it is not imperative that all transitions be directly from one stable state to another. A state through which a circuit passes en route from one stable state to another is called an *unstable state*. Transitions that involve passing through an unstable state are not harmful as long as the unstable state does not generate an undesirable output signal. For example, if a transition is between two stable states for which the output signal should be 0, it would be unacceptable to pass through an unstable state that causes the output to be 1. Even though the circuit changes through the unstable state very quickly, the short glitch in the output signal is likely to be troublesome. This is not a problem in our example. When the circuit is stable in B, the output is $z = 0$. When the inputs change to $w_2w_1 = 11$, the transition to state D maintains the output at 0. It is only when the circuit finally changes into state C that z will change to 1. Therefore, the change from $z = 0$ to $z = 1$ occurs only once during the course of these transitions.

A modified flow table, showing the unspecified transitions, is presented in Figure 9.10. The table indicates the behavior of the circuit in Figure 9.8 in terms of state transitions. If we don't know what the circuit is supposed to do, it may be difficult to discover the practical application for a given circuit. Fortunately, in practice the purpose of the circuit is known, and the analysis is done by the designer to ascertain that the circuit performs as desired. In our example it is apparent that the circuit generates the output $z = 1$ in state C, which it reaches as a result of some input patterns that are detected using the other three states. The state diagram derived from Figure 9.10 is shown in Figure 9.11.

This diagram actually implements a control mechanism for a simple vending machine that accepts two types of coins, say, dimes and nickels, and dispenses merchandise such as candy. If w_1 represents a nickel and w_2 represents a dime, then a total of 10 cents must be deposited to get the FSM into state C where the candy is released. The coin mechanism accepts only one coin at a time, which means that $w_2w_1 = 11$ can never occur. Therefore, the transition discussed above, from B to C, through the unstable state D would not occur. Observe that both states B and D indicate that 5 cents has been deposited. State B indicates that a nickel is presently being sensed by the coin receptor, while D indicates that 5 cents has been deposited and the coin receptor is presently empty. In state D it is possible to deposit either a nickel or a dime, both leading to state C. No distinction is made between the two types of coins in state D; hence the machine would not give change if 15 cents is deposited. From state A a dime leads directly to state C. Knowing that the condition

Present state	Next state				Output
	w_2w_1 = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	D	0
C	A	(C)	(C)	(C)	1
D	(D)	C	C	C	0

Figure 9.10 Modified flow table for Example 9.3.

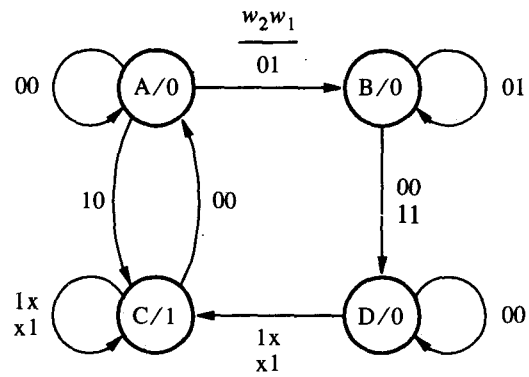


Figure 9.11 State table for Example 9.3.

$w_2w_1 = 11$ will not occur allows the flow table to be specified as shown in Figure 9.12. If we were to synthesize the sum-of-products logic expressions for Y_1 and Y_2, using the state assignment in Figure 9.9*a*, we would end up with the circuit in Figure 9.8.

Steps in the Analysis Process

We have demonstrated the analysis process using illustrative examples. The required steps can be stated as follows:

- A given circuit is interpreted in the form of the general model in Figure 8.88. That is, each feedback path is cut, and a delay element is inserted at the point where the cut is made. The input signal to the delay element represents a corresponding next-state variable, Y_i, while the output signal is the present-state variable, y_i. A cut can be made anywhere in a particular loop formed by the feedback connection, as long as there is

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	(C)	(C)	–	1
D	(D)	C	C	–	0

$w_2 \equiv$ dime $w_1 \equiv$ nickel

Figure 9.12 Flow table for a simple vending machine.

only one cut per (state variable) loop. Thus the number of cuts that should be made is the smallest number that results in there being no feedback anywhere in the circuit except from the output of a delay element. This minimal number of cuts is sometimes referred to as the *cut set*. Note that the analysis based on a cut made at one point in a given loop may not produce the same flow table as an analysis on a cut made at some other point in this loop. But both flow tables would reflect the same functional behavior in terms of the applied inputs and generated outputs.

- Next-state and output expressions are derived from the circuit.
- The excitation table corresponding to the next-state and output expressions is derived.
- A flow table is obtained, associating some (arbitrary) names with the particular encoded states.
- A corresponding state diagram is derived from the flow table if desired.

9.3 SYNTHESIS OF ASYNCHRONOUS CIRCUITS

Synthesis of asynchronous sequential circuits follows the same basic steps used to synthesize the synchronous circuits, which were discussed in Chapter 8. There are some differences due to the asynchronous nature, which make the asynchronous circuits more difficult to design. We will explain the differences by investigating a few design examples. The basic steps are

- Devise a state diagram for an FSM that realizes the required functional behavior.
- Derive the flow table and reduce the number of states if possible.
- Perform the state assignment and derive the excitation table
- Obtain the next-state and output expressions.
- Construct a circuit that implements these expressions.

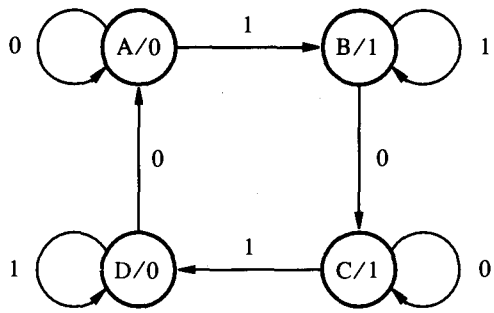
When devising a state diagram, or perhaps the flow table directly, it is essential to ensure that when the circuit is in a stable state, the correct output signals are generated. Should it be necessary to pass through an unstable state, this state must not produce an undesirable output signal.

Minimization of states is not straightforward. A minimization procedure is described in section 9.4.

State assignment is not done with the sole purpose of reducing the cost of the final circuit. In asynchronous circuits some state assignments may cause the circuit to be unreliable. We will explain this problem using the examples that follow.

Example 9.4 **SERIAL PARITY GENERATOR** Suppose that we want to design a circuit that has an input w and an output z, such that when pulses are applied to w, the output z is equal to 0 if the number of previously applied pulses is even and z is equal to 1 if the number of pulses is odd. Hence the circuit acts as a serial parity generator.

Let A be the state that indicates that an even number of pulses has been received. Using the Moore model, the output z will be equal to 0 when the circuit is in state A. As long as $w = 0$, the circuit should remain in A, which is specified by a transition arc that both originates and terminates in state A. Thus A is stable when $w = 0$. When the next pulse arrives, the input $w = 1$ should cause the FSM to move to a new state, say, B, which produces the output $z = 1$. When the FSM reaches B, it must remain stable in this state as long as $w = 1$. This is specified by a transition arc that originates and terminates in B. The next input change occurs when w goes to 0. In response the FSM must change to a state where $z = 1$ and which corresponds to the fact that a complete pulse has been observed, namely, that w has changed from 1 to 0. Let this state be C; it must be stable under the input condition $w = 0$. The arrival of the next pulse makes $w = 1$, and the FSM must change to a state, D, that indicates that an even number of pulses has been observed and that the last pulse is still present. The state D is stable under $w = 1$, and it causes the output to be $z = 0$. Finally, when w returns to 0 at the end of the pulse, the FSM returns to state A, which indicates an even number of pulses and w equal to 0 at the present time. The resulting state diagram is shown in Figure 9.13*a*.



(a) State diagram

Present State	Next state		Output
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	A	(D)	0

(b) Flow table

Figure 9.13 Parity-generating asynchronous FSM.

A key point to understand is why it is necessary to have four states rather than just two, considering that we are merely trying to distinguish between the even and odd number of input pulses. States B and C cannot be combined into a single state even though they both indicate that an odd number of pulses has been observed. Suppose we had simply tried to use state B alone for this purpose. Then it would have been necessary to add an arc with a label 0 that originates and terminates in state B, which is fine. The problem is that without state C, there would have to be a transition from state B directly to D if the input is $w = 1$ to respond to the next change in the input when a new pulse arrives. It would be impossible to have B both stable under $w = 1$ and have a change to D effected for the same input condition. Similarly, we can show that the states A and D cannot be combined into a single state.

Figure 9.13*b* gives the flow table that corresponds directly to the state diagram. In many cases the designer can derive a flow table directly. We are using the state diagram mostly because it provides a simpler visual picture of the effect of the transitions in an FSM.

The next step is to assign values to the states in terms of the state variables. Since there are four states in our FSM, there have to be at least two state variables. Let these variables be y_1 and y_2. As a first attempt at the state assignment, let the states A, B, C, and D be encoded as $y_2y_1 = 00$, 01, 10, and 11, respectively. This assignment leads to the excitation table in Figure 9.14*a*. Unfortunately, it has a major flaw. The circuit that implements this table is stable in state $D = 11$ under the input condition $w = 1$. But consider what happens next if the input changes to $w = 0$. According to the excitation table, the circuit should change to state $A = 00$ and remain stable in this state. The problem is that in going from $y_2y_1 = 11$ to $y_2y_1 = 00$ both state variables must change their values. This is unlikely to occur at exactly the same time. In an asynchronous circuit the values of the next-state variables are determined by networks of logic gates with varying propagation delays. Thus we should expect that one state variable will change slightly before the other, which could put the circuit into a state where it may react to the input in an undesirable way. Suppose that y_1 changes first. Then the circuit goes from $y_2y_1 = 11$ to $y_2y_1 = 10$. As soon as it reaches this state, C, it will attempt to remain there if $w = 0$, which is a wrong outcome. On the other hand, suppose that y_2 changes first. Then there will be a change from $y_2y_1 = 11$ to $y_2y_1 = 01$, which corresponds to state B. Since $w = 0$, the circuit will now try to change to $y_2y_1 = 10$. This again requires that both y_1 and y_2 change; assuming that y_1 changes first in the transition from $y_2y_1 = 01$, the circuit will find itself in the state $y_2y_1 = 00$, which is the correct destination state, A. This discussion indicates that the required transition from D to A will be performed correctly if y_2 changes before y_1, but it will not work if y_1 changes before y_2. The result depends on the outcome of the "race" to change between the signals y_1 and y_2.

The uncertainty caused by multiple changes in the state variables in response to an input that should lead to a predictable change from one stable state to another has to be eliminated. The term *race condition* is used to refer to such unpredictable behavior. We will discuss this issue in detail in section 9.5.

Race conditions can be eliminated by treating the present-state variables as if they were inputs to the circuit, meaning that only one state variable is allowed to change at a time. For our example the assignment $A = 00$, $B = 01$, $C = 11$, and $D = 10$ achieves this objective. The resulting excitation table is presented in Figure 9.14*b*. The reader should verify that all transitions involve changing a single state variable.

Present state y_2y_1	Next state Y_2Y_1 $w = 0$	Next state Y_2Y_1 $w = 1$	Output z
00	(00)	01	0
01	10	(01)	1
10	(10)	11	1
11	00	(11)	0

(a) Poor state assignment

Present state y_2y_1	Next state Y_2Y_1 $w = 0$	Next state Y_2Y_1 $w = 1$	Output z
00	(00)	01	0
01	11	(01)	1
11	(11)	10	1
10	00	(10)	0

(b) Good state assignment

Figure 9.14 State assignment for Figure 9.13*b*.

From Figure 9.14*b* the next-state and output expressions are

$$Y_1 = w\bar{y}_2 + \bar{w}y_1 + y_1\bar{y}_2$$
$$Y_2 = wy_2 + \bar{w}y_1 + y_1y_2$$
$$z = y_1$$

The last product term in the expressions for Y_1 and Y_2 is included to deal with possible hazards, which are discussed in section 9.6. The corresponding circuit is shown in Figure 9.15.

It is interesting to consider how the serial parity generator could be implemented using a synchronous approach. All that is needed is a single flip-flop that changes its state with the arrival of each input pulse. The positive-edge-triggered D flip-flop in Figure 9.16 accomplishes the task, assuming that the flip-flop is initially set to Q = 0. The logic complexity of the flip-flop is exactly the same as the circuit in Figure 9.15. Indeed, if we use the preceding expressions for Y_1 and Y_2 and substitute C for w, D for $\bar{y}_2$, y_m for y_1, and

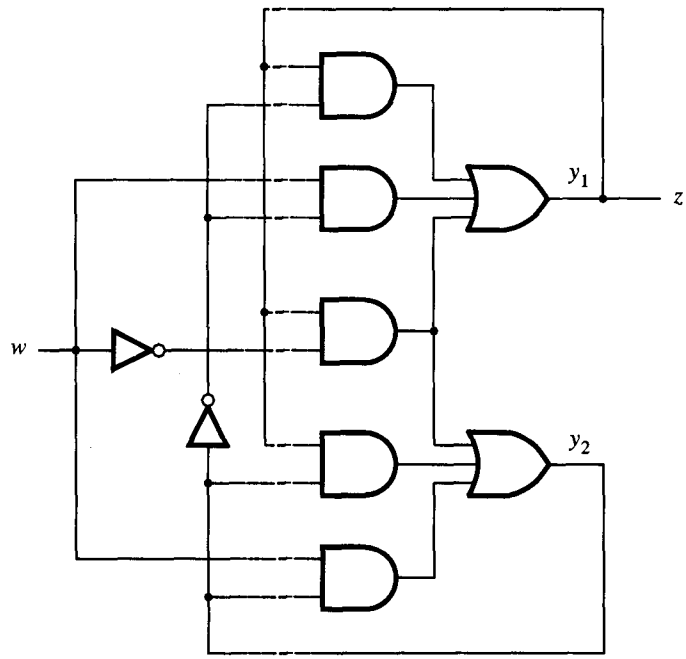


Figure 9.15 Circuit that implements the FSM in Figure 9.14*b*.

y_s for y_2, we end up with the excitation expressions shown for the master-slave D flip-flop in Example 9.2. The circuit in Figure 9.15 is actually a negative-edge-triggered master-slave flip-flop, with the complement of its Q output (y_2) connected to its D input. The output z is connected to the output of the master stage of the flip-flop.

Example 9.5 **MODULO-4 COUNTER** Chapters 7 and 8 described how counters can be implemented using flip-flops. Now we will synthesize a counter as an asynchronous sequential circuit. Figure 9.17 depicts a state diagram for a modulo-4 up-counter, which counts the number of pulses on an input line, w. The circuit must be able to react to all changes in the input

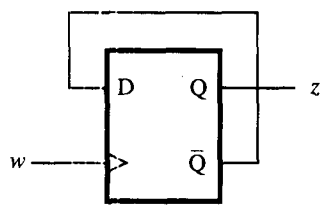


Figure 9.16 Synchronous solution for Example 9.4.

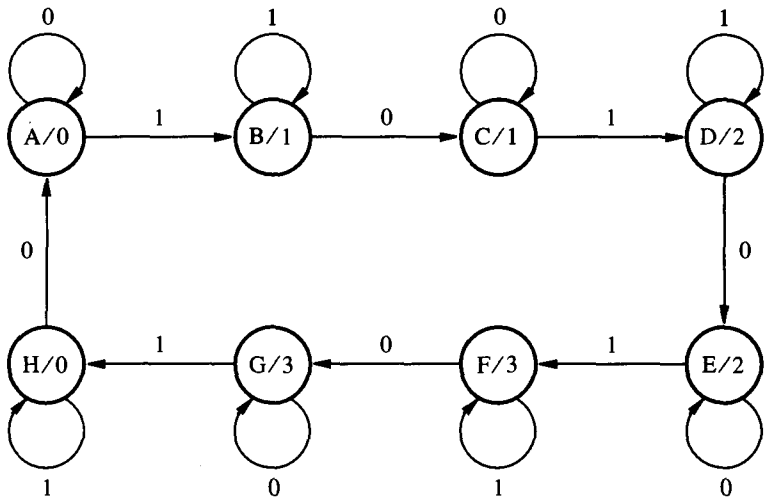


Figure 9.17 State diagram for a modulo-4 counter.

signal; thus it must take specific actions at both the positive and negative edges of each pulse. Therefore, eight states are needed to deal with the edges in four consecutive pulses.

The counter begins in state A and stays in this state as long as $w = 0$. When w changes to 1, a transition to state B is made and the circuit remains stable in this state as long as $w = 1$. When w goes back to 0, the circuit moves to state C and remains stable until w becomes 1 again, which causes a transition to state D, and so on. Using the Moore model, the states correspond to specific counts. There are two states for each particular count: the state that the FSM enters when w changes from 0 to 1 at the start of a pulse and the state that the FSM enters when w goes back to 0 at the end of the pulse. States B and C correspond to the count of 1, states D and E to 2, and states F and G to 3. States A and H represent the count of 0.

Figure 9.18 shows the flow and excitation tables for the counter. The state assignment is chosen such that all transitions between states require changing the value of only one state variable to eliminate the possibility of race conditions. The output is encoded as a binary number, using variables z_2 and z_1. From the excitation table the next-state and output expressions are

$$\begin{aligned}
Y_1 &= \overline{w}y_1 + wy_2y_3 + w\overline{y}_2\overline{y}_3 + y_1y_2y_3 + y_1\overline{y}_2\overline{y}_3 \\
&= \overline{w}y_1 + (w + y_1)(y_2y_3 + \overline{y}_2\overline{y}_3) \\
Y_2 &= wy_2 + \overline{w}y_1\overline{y}_3 + \overline{y}_1y_2 + y_2\overline{y}_3 \\
Y_3 &= wy_3 + y_1y_3 + \overline{y}_1y_2\overline{w} + y_2y_3 \\
z_1 &= y_1 \\
z_2 &= y_1y_3 + \overline{y}_1y_2
\end{aligned}$$

These expressions define the circuit that implements the required modulo-4 pulse counter.

Present state	Next state		Output
	$w = 0$	$w = 1$	z
A	(A)	B	0
B	C	(B)	1
C	(C)	D	1
D	E	(D)	2
E	(E)	F	2
F	G	(F)	3
G	(G)	H	3
H	A	(H)	0

(a) Flow table

Present state	Next state		Output
$y_3y_2y_1$	$w = 0$	$w = 1$	z_2z_1
	$Y_3Y_2Y_1$		
000	(000)	001	00
001	011	(001)	01
011	(011)	010	01
010	110	(010)	10
110	(110)	111	10
111	101	(111)	11
101	(101)	100	11
100	000	(100)	00

(b) Excitation table

Mod-8 output
$z_3z_2z_1$
000
001
010
011
100
101
110
111

(c) Output for counting the edges

Figure 9.18 Flow and excitation tables for a modulo-4 counter.

In the preceding derivation we designed a circuit that changes its state on every edge of the input signal w, requiring a total of eight states. Since the circuit is supposed to count the number of complete pulses, which contain a rising and a falling edge, the output count z_2z_1 changes its value only in every second state. This FSM behaves like a synchronous sequential circuit in which the output count changes only as a result of w changing from 0 to 1.

Suppose now that we want to count the number of times the signal w changes its value, that is, the number of its edges. The state transitions specified in Figures 9.17 and 9.18 define an FSM that can operate as a modulo-8 counter for this purpose. We only need to specify a distinct output in each state, which can be done as shown in Figure 9.18*c*. The values of $z_3z_2z_1$ indicate the counting sequence 0, 1, 2, . . . , 7, 0. Using this specification of the output and the state assignment in Figure 9.18*b*, the resulting output expressions are

$$z_1 = y_1 \oplus y_2 \oplus y_3$$
$$z_2 = y_2 \oplus y_3$$
$$z_3 = y_3$$

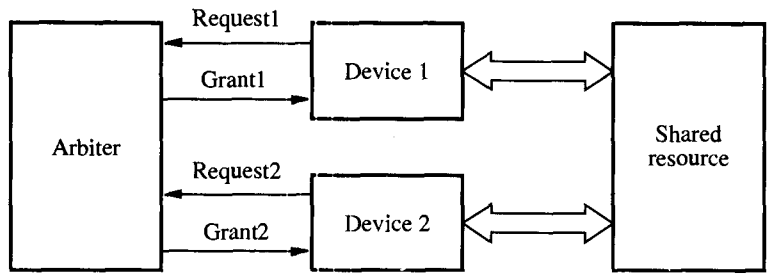
Example 9.6

A SIMPLE ARBITER In computer systems it is often useful to have some resource shared by a number of different devices. Usually, the resource can be used by only one device at a time. When various devices need to use the resource, they have to request to do so. These requests are handled by an arbiter circuit. When there are two or more outstanding requests, the arbiter may use some priority scheme to choose one of them, as already discussed in section 8.8.

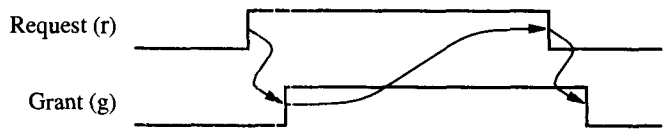
We will now consider an example of a simple arbiter implemented as an asynchronous sequential circuit. To keep the example small, suppose that two devices are competing for the shared resource, as indicated in Figure 9.19*a*. Each device communicates with the arbiter by means of two signals—*Request* and *Grant*. When a device needs to use the shared resource, it raises its Request signal to 1. Then it waits until the arbiter responds with the Grant signal.

Figure 9.19*b* illustrates a commonly used scheme for communication between two entities in the asynchronous environment, known as *handshake signaling*. Two signals are used to provide the handshake. A device initiates the activity by raising a request, $r = 1$. When the shared resource is available, the arbiter responds by issuing a grant, $g = 1$. When the device receives the grant signal, it proceeds to use the requested shared resource. When it completes its use of the resource, it drops its request by setting $r = 0$. When the arbiter sees that $r = 0$, it deactivates the grant signal, making $g = 0$. The arrows in the figure indicate the cause-effect relationships in this signaling scheme; a change in one signal causes a change in the other signal. The time elapsed between the changes in the cause-effect signals depends on the specific implementation of the circuit. A key point is that there is no need for a synchronizing clock.

A state diagram for our simple arbiter is given in Figure 9.20. There are two inputs, the request signals r_1 and r_2, and two outputs, the grant signals g_1 and g_2. The diagram depicts



(a) Arbitration structure



(b) Handshake signaling

Figure 9.19 Arbitration example.

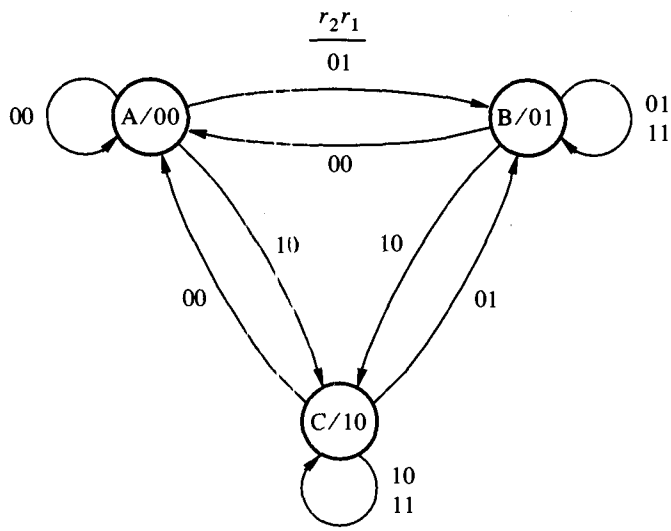


Figure 9.20 State diagram for the arbiter.

the Moore model of the required FSM, where the arcs are labeled as r_2r_1 and the state outputs as g_2g_1. The quiescent state is A, where there are no requests. State B represents the situation in which Device 1 is given permission to use the resource, and state C denotes the same for Device 2. Thus B is stable if $r_2r_1 = 01$, and C is stable if $r_2r_1 = 10$. To conform to the rules of asynchronous circuit design, we will assume that the inputs r_1 and r_2 become activated one at a time. Hence, in state A it is impossible to have a change from $r_2r_1 = 00$ to $r_2r_1 = 11$. The situation where $r_2r_1 = 11$ occurs only when a second request is raised before the device that has the grant signal completes its use of the shared resource, which can happen in states B and C. If the FSM is stable in either state B or C, it will remain in this state if both r_1 and r_2 go to 1.

The flow table is given in Figure 9.21*a*, and the excitation table is presented in Figure 9.21*b*. It is impossible to choose a state assignment such that all changes between states A, B, and C involve a change in a single state variable only. In the chosen assignment the transitions to or from state A are handled properly, but the transitions between states B and C involve changes in the values of both state variables y_1 and y_2. Suppose that the circuit is stable in state B under input valuation $r_2r_1 = 11$. Now let the inputs change to $r_2r_1 = 10$. This should cause a change to state C, which means that the state variables must change from $y_2y_1 = 01$ to 10. If y_1 changes faster than y_2, then the circuit will find itself

Present state	Next state $r_2r_1 = 00$	01	10	11	Output g_2g_1
A	(A)	B	C	–	00
B	A	(B)	C	(B)	01
C	A	B	(C)	(C)	10

(a) Flow table

	Present state y_2y_1	Next state $r_2r_1 = 00$	01	10	11	Output g_2g_1
		Y_2Y_1				
A	00	(00)	01	10	–	00
B	01	00	(01)	10	(01)	01
C	10	00	01	(10)	(10)	10
D	11	–	01	10	–	*dd*

(b) Excitation table

Figure 9.21 Implementation of the arbiter.

momentarily in state $y_2y_1 = 00$, which leads to the desired final state because from state A there is a specified transition to C under the input valuation 10. But if y_2 changes faster than y_1, the circuit will reach the state $y_2y_1 = 11$, which is not defined in the flow table. To make sure that even in this case the circuit will proceed to the required destination C, we can include the state $y_2y_1 = 11$, labeled D, in the excitation table and specify the required transition as shown in the figure. A similar situation arises when the circuit is stable in C under $r_2r_1 = 11$, and it has to change to B when r_2 changes from 1 to 0.

The output values for the extra state D are indicated as don't cares. Whenever a specific output is changing from 0 to 1 or from 1 to 0, exactly when this change takes place is not important if the correct value is produced when the circuit is in a stable state. The don't-care specification may lead to a simpler realization of the output functions. It is important to ensure that unspecified outputs will not result in a value that may cause erroneous behavior. From Figure 9.21*b* it is possible that during the short time when the circuit passes through the unstable state D the outputs become $g_2g_1 = 11$. This is harmless in our example because the device that has just finished using the shared resource will not try to use it again until its grant signal has returned to 0 to indicate the end of the handshake with the arbiter. Observe that if this condition occurs when changing from B to C, then g_1 remains 1 slightly longer and g_2 becomes 1 slightly earlier. Similarly, if the transition is from C to B, then the change in g_1 from 0 to 1 happens slightly earlier and g_2 changes to 0 slightly later. In both of these cases there is no glitch on either g_1 or g_2.

From the excitation table the following next-state and output expressions are derived

$$\begin{aligned} Y_1 &= \bar{r}_2 r_1 + r_1 \bar{y}_2 \\ Y_2 &= r_2 \bar{r}_1 + r_2 y_2 \\ g_1 &= y_1 \\ g_2 &= y_2 \end{aligned}$$

Rewriting the first two expressions as

$$\begin{aligned} Y_1 &= r_1(\bar{r}_2 + \bar{y}_2) \\ &= r_1 \overline{r_2 y_2} \\ Y_2 &= r_2(\bar{r}_1 + y_2) \end{aligned}$$

produces the circuit in Figure 9.22. Observe that this circuit responds very quickly to the changes in the input signals. This behavior is in sharp contrast to the arbiter discussed in section 8.8 in which the synchronizing clock determines the minimum response time.

The difficulty with the race condition that arises in state changes between B and C can be resolved in another way. We can simply prevent the circuit from reaching an unspecified state. Figure 9.23*a* shows a modified flow table in which transitions between states B and C are made via state A. If the circuit is stable in B and the input valuation changes from $r_2r_1 = 11$ to 10, a change to A will occur first. As soon as the circuit reaches A, which is not stable for the input valuation 10, it will proceed to the stable state C. The detour through the unstable state A is acceptable because in this state the output is $g_2g_1 = 00$, which is consistent with the desired operation of the arbiter. The change from C to B is handled using the same approach. From the modified excitation table in Figure 9.23*b*, the following

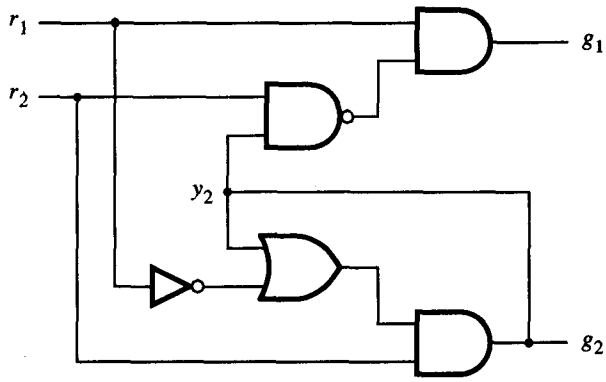


Figure 9.22 The arbiter circuit.

Present state	Next state $r_2r_1 = 00$	01	10	11	Output g_2g_1
A	(A)	B	C	–	00
B	A	(B)	A	(B)	01
C	A	A	(C)	(C)	10

(a) Modified flow table

Present state y_2y_1	Next state $r_2r_1 = 00$	01	10	11	Output g_2g_1
	Y_2Y_1				
00	(00)	01	10	–	00
01	00	(01)	00	(01)	01
10	00	00	(10)	(10)	10

(b) Modified excitation table

Figure 9.23 An alternative for avoiding a critical race in Figure 9.21*a*.

next-state expressions are derived

$$Y_1 = r_1\bar{y}_2$$
$$Y_2 = \bar{r}_1 r_2 \bar{y}_1 + r_2 y_2$$

These expressions give rise to a circuit different from the one in Figure 9.22. However, both circuits implement the functionality required in the arbiter.

Next we will attempt to design the same arbiter using the Mealy model specification. From Figure 9.20 it is apparent that the states B and C are fundamentally different because for the input $r_2r_1 = 11$ they must produce two different outputs. But state A is unique only to the extent that it generates the output $g_2g_1 = 00$ whenever $r_2r_1 = 00$. This condition could be specified in both B and C if the Mealy model is used. Figure 9.24 gives a suitable state diagram. The flow and excitation tables are presented in Figure 9.25, which lead to

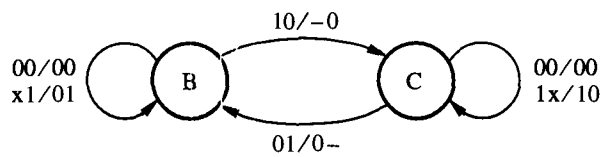


Figure 9.24 Mealy model for the arbiter FSM.

Present state	Next state				Output g_2g_1			
	$r_2r_1 = 00$	01	10	11	00	01	10	11
B	(B)	(B)	C	(B)	00	01	–0	01
C	(C)	B	(C)	(C)	00	0–	10	10

(a) Flow diagram

Present state y	Next state				Output			
	$r_2r_1 = 00$	01	10	11	00	01	10	11
	Y				g_2g_1			
0	0	0	1	0	00	01	$d0$	01
1	1	0	1	1	00	$0d$	10	10

(b) Excitation table

Figure 9.25 Mealy model implementation of the arbiter FSM.

the following expressions

$$Y = r_2\bar{r}_1 + \bar{r}_1 y + r_2 y$$
$$g_1 = r_1\bar{y}$$
$$g_2 = r_2 y$$

Despite needing a single state variable, this circuit requires more gates for implementation than does the Moore version in Figure 9.22.

An important notion in the above examples is that it is necessary to pay careful attention to the state assignment, to avoid races in changing of the values of the state variables. Section 9.5 deals with this issue in more detail.

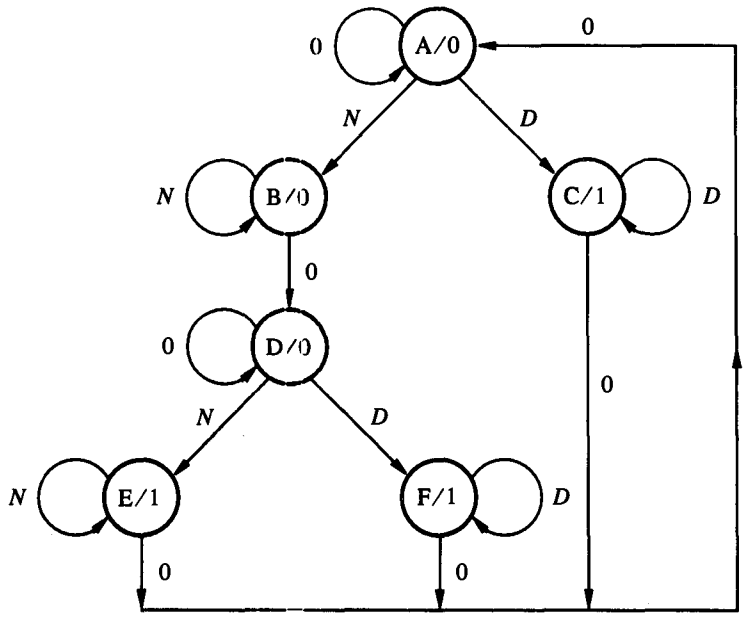
We made the basic assumption that the request inputs to the arbiter FSM change their values one at a time, which allows the circuit to reach a stable state before the next change takes place. If the devices are totally independent, they can raise their requests at any time. Suppose that each device raises a request every few seconds. Since the arbiter circuit needs only a few nanoseconds to change from one stable state to another, it quite unlikely that both devices will raise their requests so close to each other that the arbiter circuit will produce erroneous outputs. However, while the probability of an error caused by the simultaneous arrival of requests is extremely low, it is not zero. If this small possibility of an error cannot be tolerated, then it is possible to feed the request signals through a special circuit called the *mutual exclusion* (ME) element. This circuit has two inputs and two outputs. If both inputs are 0, then both outputs are 0. If only one input is 1, then the corresponding output is 1. If both inputs are 1, the circuit makes one output go to 1 and keeps the other at 0. Using the ME element would change the design of the arbiter slightly; because the valuation $r_2r_1 = 11$ would never occur, all entries in the corresponding column in Figure 9.21 would be don't cares. The ME element and the issue of simultaneous changes in input signals are discussed in detail in reference [6]. Finally, we should note that a similar problem arises in synchronous circuits in which one or more inputs are generated by a circuit that is not controlled by a common clock. We will deal with this issue in section 10.3.3 in Chapter 10.

9.4 STATE REDUCTION

In Chapter 8 we saw that reducing the number of states needed to realize the functionality of a given FSM usually leads to fewer state variables, which means that fewer flip-flops are required in the corresponding synchronous sequential circuit. In asynchronous sequential circuits it is also useful to try to reduce the number of states because this usually results in simpler implementations.

When designing an asynchronous FSM, the initial flow table is likely to have many unspecified (don't-care) entries, because the designer has to obey the restriction that only one input variable can change its value at a time. For example, suppose that we want to design the FSM for the simple vending machine considered in Example 9.3. Recall that the machine accepts nickels and dimes and dispenses candy when 10 cents is deposited;

the machine does not give change if 15 cents is deposited. An initial state diagram for this FSM can be derived in straightforward fashion by enumerating all possible sequences of depositing the coins to give a sum of at least 10 cents. Figure 9.26*a* shows a possible diagram, defined as a Moore model. Starting in a reset state, *A*, the FSM remains in this state as long as no coin is deposited. This is denoted by an arc labeled 0 to indicate that



(a) Initial state diagram

Present State	Next state				Output
	$DN =$ 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	A	–	(C)	–	1
D	(D)	E	F	–	0
E	A	(E)	–	–	1
F	A	–	(F)	–	1

(b) Initial flow table

Figure 9.26 Derivation of an FSM for the simple vending machine.

$N = D = 0$. Now let an arc with the label N denote that the coin-sensing mechanism has detected a nickel and has generated a signal $N = 1$. Similarly, let D denote that a dime has been deposited. If $N = 1$, then the FSM has to move to a new state, say, B, and it must remain stable in this state as long as N has the value of 1. Since B corresponds to 5 cents being deposited, the output in this state has to be 0. If a dime is deposited in state A, then the FSM must move to a different state, say, C. The machine should stay in C as long as $D = 1$, and it should release the candy by generating the output of 1. These are the only possible transitions from state A, because it is impossible to insert two coins at the same time, which means that $DN = 11$ can be treated as a don't-care condition. Next, in state B there must be a return to the condition $DN = 00$ because the coin-sensing mechanism will detect the second coin some time after the first coin has cleared the mechanism. This behavior is consistent with the requirement that only one input variable can change at a time; hence it is not allowed to go from $DN = 01$ to $DN = 10$. The input $DN = 10$ cannot occur in state B and should be treated as a don't care. The input $DN = 00$ takes the FSM to a new state, D, which indicates that 5 cents has been deposited and that there is no coin in the sensing mechanism. In state D it is possible to deposit either a nickel or a dime. If $DN = 01$, the machine moves to state E, which denotes that 10 cents has been deposited and generates the output of 1. If $DN = 10$, the machine moves to state F, which also generates the output of 1. Finally, when the FSM is in any of the states C, E, or F, the only possible input is $DN = 00$, which returns the machine to state A.

The flow table for this FSM is given in Figure 9.26*b*. It shows explicitly all don't-care entries. Such unspecified entries provide a certain amount of flexibility that can be exploited in reducing the number of states. Note that in each row of this table there is only one stable state. Such tables, where there is only one stable state for each row, are often referred to as *primitive flow tables*.

Several techniques have been developed for state reduction. In this section we will describe a two-step process. In the first step we will apply the partitioning procedure from section 8.6.1, assuming that the potentially equivalent rows in a flow table must produce the same outputs. As an additional constraint, for two rows to be potentially equivalent any unspecified entries must be in the same next-state columns. Thus combining the equivalent states into a single state will not remove the don't cares and the flexibility that they provide. In the second step, the rows are *merged* exploiting the unspecified entries. Two rows can be merged if they have no conflicting next-state entries. This means that their next-state entries for any given valuation of inputs are either the same, or one of them is unspecified, or both rows indicate a stable state. If the Moore model is used, then the two rows (states) must produce the same outputs. If the Mealy model is used, then the two states must produce the same outputs for any input valuations for which both states are stable.

Example 9.7

We will now show how the flow diagram in Figure 9.26*b* can be reduced to the optimized form in Figure 9.12. The first step in the state-reduction process is the partitioning procedure from section 8.6.1. States A and D are stable under the input valuation $DN = 00$, producing the output of 0; they also have the unspecified entries in the same position. States C and F are stable under $DN = 10$, generating $z = 1$, and they have the same unspecified entries. States B and E have the same unspecified entries, but when they are stable under $DN = 01$

the state B produces $z = 0$ while E generates $z = 1$; they are not equivalent. Therefore, the initial partition is

$$P_1 = (AD)(B)(CF)(E)$$

The successors of A and D are (A, D) for $DN = 00$, (B, E) for 01, and (C, F) for 10. Since the (B, E) pair is not in the same block of P_1, it follows that A and D are not equivalent. The successors of C and F are (A, A) for 00 and (C, F) for 10; each pair is in a single block. Thus the second partition is

$$P_2 = (A)(D)(B)(CF)(E)$$

The successors of C and F in P_2 are in the same block of P_2, which means that

$$P_3 = P_2$$

The conclusion is that rows C and F are equivalent. Combining them into a single row and changing all Fs into Cs gives the flow table in Figure 9.27.

Next we can try to merge some rows in the flow table by exploiting the existence of unspecified entries. The only row that can be merged with others is C. It can be merged with either A or E, but not both. Merging C with A would mean that the new state has to generate $z = 0$ when it is stable under the input valuation 00 and has to produce $z = 1$ when stable under 10. This can be achieved only by using the Mealy model. The alternative is to merge C and E, in which case the new state is stable under $DN = 01$ and 10, producing the output of 1. This can be achieved with the Moore model. Merging C and E into a single state C and changing all Es into Cs yields the reduced flow table in Figure 9.12. Observe that when C and E are merged, the new row C must include all specifications in both rows C and E. Both rows specify A as the next state if $DN = 00$. Row E specifies a stable state for $DN = 01$; hence the new row (called C) must also specify a stable state for the same valuation. Similarly, row C specifies a stable state for $DN = 10$, which must be reflected in the new row. Therefore, the next-state entries in the new row are A, (C), and (C) for the input valuations 00, 01, and 10, respectively.

Present state	Next state				Output z
	$DN = 00$	01	10	11	
A	(A)	B	C	–	0
B	D	(B)		–	0
C	A	–	(C)	–	1
D	(D)	E	C	–	0
E	A	(E)	–	–	1

Figure 9.27 First-step reduction of the FSM in Figure 9.26*b*.

Merging Procedure

In Example 9.7 it was easy to decide which rows should be merged because the only possibilities are to merge row C with either A or E. We chose to merge C and E because this can be done preserving the Moore model, which is likely to lead to a simpler expression that realizes the output z.

In general, there can be many possibilities for merging rows in larger flow tables. In such cases it is necessary to have a more structured procedure for making the choice. A useful procedure can be defined using the concept of *compatibility* of states.

Definition 9.1 – *Two states (rows in a flow table), S_i and S_j, are said to be compatible if there are no state conflicts for any input valuation. Thus for each input valuation, one of the following conditions must be true:*

- *both S_i and S_j have the same successor, or*
- *both S_i and S_j are stable, or*
- *the successor of S_i or S_j, or both, is unspecified.*

Moreover, both S_i and S_j must have the same output whenever specified.

Consider the primitive flow table in Figure 9.28. Let us examine the compatibility between different states, assuming that we would like to retain the Moore-type specification of outputs for this FSM. State A is compatible only with state H. State B is compatible with states F and G. State C is not compatible with any other state. State D is compatible with state E; so are state F with G and state G with H. In other words, the following compatible pairs exist: (A, H), (B, F), (B, G), (D, E), (F, G), and (G, H). The compatibility relationship among various states can be represented conveniently in the form of a *merger diagram*, as follows:

- Each row of the flow table is represented as a point, labeled by the name of the row.
- A line is drawn connecting any two points that correspond to compatible states (rows).

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	H	B	–	0
B	F	–	(B)	C	0
C	–	H	–	(C)	1
D	A	(D)	–	E	1
E	–	D	G	(E)	1
F	(F)	D	–	–	0
G	F	–	(G)	–	0
H	–	(H)	–	E	0

Figure 9.28 A primitive flow table.

From the merger diagram the best merging possibility can be chosen, and the reduced flow table can be derived.

Figure 9.29 gives the merger diagram for the primitive flow table in Figure 9.28. The diagram indicates that row A can be merged with H, but only if H is not merged with G, because there is no line joining A and G. Row B can be merged with rows F and G. Since it is also possible to merge F and G, it follows that B, F, and G are pairwise compatible. Any set of rows that are pairwise compatible for all pairs in the set can be merged into a single state. Thus states B, F, and G can be merged into a single state, but only if states G and H are not merged. State C cannot be merged with any other state. States D and E can be merged.

A prudent strategy is to merge the states so that the resulting flow table has as few states as possible. In our example the best choice is to merge the compatibles (A, H), (B, F, G), and (D, E), which leads to the reduced flow table in Figure 9.30. When a new row is created by merging two or more rows, all entries in the new row have to be specified to cover the individual requirements of the constituent rows. Replacing rows A and H with a new row A requires making A stable for both $w_2w_1 = 00$ and 01, because the old A has to be stable

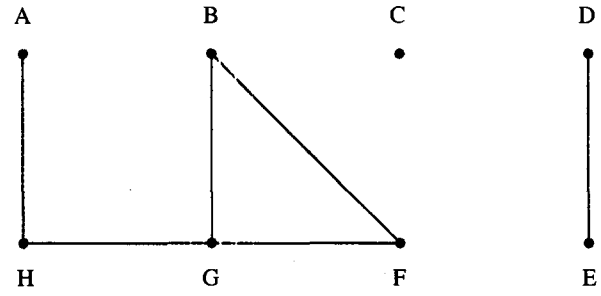


Figure 9.29 Merger diagram for the flow table in Figure 9.28, which preserves the Moore model.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	B	D	0
B	(B)	D	(B)	C	0
C	–	A	–	(C)	1
D	A	(D)	B	(D)	1

Figure 9.30 Reduced Moore-type flow table for the FSM in Figure 9.28.

for 00 and H has to be stable for 01. It also requires specifying B as the next-state for $w_2w_1 = 10$ and E as the next state for $w_2w_1 = 11$. Since the old state E becomes D, after merging D and E, the new row A must have the next-state entries (A), (A), B, and D for the input valuations 00, 01, 10, and 11, respectively. Replacing rows B, F, and G with a new row B requires making B stable for $w_2w_1 = 00$ and 10. The next-state entry for $w_2w_1 = 01$ has to be D to satisfy the requirement of the old state F. The next-state entry for $w_2w_1 = 11$ has to be C, as dictated by the old state B. Observe that the old state G imposes no requirements for transitions under $w_2w_1 = 01$ and 11, because its corresponding next-state entries are unspecified. Row C remains the same as before except that the name of the next-state entry for $w_2w_1 = 01$ has to be changed from H to A. Rows D and E are replaced by a new row D, using similar reasoning. Note that the flow table in Figure 9.30 is still of Moore type.

So far we considered merging only those rows that would allow us to retain the Moore-type specification of the FSM in Figure 9.28. If we are willing to change to the Mealy model, then other possibilities exist for merging. Figure 9.31 shows the complete merger diagram for the FSM of Figure 9.28. Black lines connect the compatible states that can be merged into a new state that has a Moore-type output; this corresponds to the merger diagram in Figure 9.29. Blue lines connect the states that can be merged only if Mealy-type outputs are used.

In this case going to the Mealy model is unlikely to result in a simpler circuit. Although several merger possibilities exist, they all require at least four states in the reduced flow table, which is not any better than the solution obtained in Figure 9.30. For example, one possibility is to perform the merge based on the partition (A, H), (B, C, G) (D, E) (F). Another possibility is to use (A, C) (B, F) (D, E) (G, H). We will not pursue these possibilities and will discuss the issues involved in specifying the Mealy-type outputs in Example 9.9.

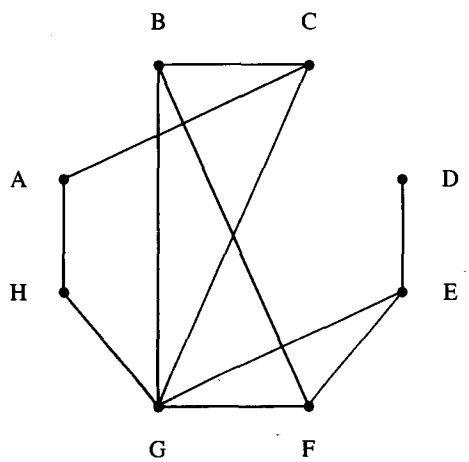


Figure 9.31 Complete merger diagram for Figure 9.28.

State Reduction Procedure

We can summarize the steps needed to generate the reduced flow table from a primitive flow table as follows:

1. Use the partitioning procedure from section 8.6.1 to eliminate the equivalent states in a primitive flow table.
2. Construct a merger diagram for the resulting flow table.
3. Choose subsets of compatible states that can be merged, trying to minimize the number of subsets needed to cover all states. Each state must be included in only one of the chosen subsets.
4. Derive the reduced flow table by merging the rows in chosen subsets.
5. Repeat steps 2 to 4 to see whether further reductions are possible.

Choosing an optimal subset of compatible states for merging can be a very complicated task because for large FSMs there may be many possibilities that should be investigated. A trial-and-error approach is a reasonable way to tackle this problem.

Example 9.8 Consider the initial flow table in Figure 9.32. To apply the partitioning procedure, we identify state pairs (A, G), (B, L), and (H, K) as being potentially equivalent rows, because both rows in each pair have the same outputs and their don't-care entries are in the same

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	K	0
G	(G)	B	J	–	0
H	–	L	E	(H)	1
J	G	–	(J)	–	0
K	–	B	E	(K)	1
L	A	(L)	–	K	1

Figure 9.32 Flow table for Example 9.8.

column. The remaining rows are distinct in this respect. Therefore, the first partition is

$$P_1 = (AG)(BL)(C)(D)(E)(F)(HK)(J)$$

Now the successors of (A, G) are (A, G) for $w_2w_1 = 00$, (F, B) for 01, and (C, J) for 10. Since F and B, as well as C and J, are not in the same block, it follows that A and G are not equivalent. The successors of (B, L) are (A, A), (B, L), and (H, K), respectively. All are in single blocks. The successors of (H, K) are (L, B), (E, E), and (H, K), which are all contained in single blocks. Therefore, the second partition is

$$P_2 = (A)(G)(BL)(C)(D)(E)(F)(HK)(J)$$

Repeating the successor test shows that the successors of (B, L) and (H, K) are still in single blocks; hence

$$P_3 = P_2$$

Combining rows B and L under the name B and rows H and K under the name H leads to the flow table in Figure 9.33.

A merger diagram for this flow table is given in Figure 9.34. It indicates that rows B and H should be merged into one row, which we will label as B. The merger diagram also suggests that rows D and E should be merged; we will call the new row D. The remaining rows present more than one choice for merging. Rows A and F can be merged, but in that case F and J cannot be merged. Rows C and J can be merged, or G and J can be merged. We will choose to merge the rows A and F into a new row called A and rows G and J into a new row G. The merger choice is indicated in blue in the diagram. The resultant flow table is shown in Figure 9.35. To see whether this table offers any further opportunities

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	F	C	–	0
B	A	(B)	–	H	1
C	G	–	(C)	D	0
D	–	F	–	(D)	1
E	G	–	(E)	D	1
F	–	(F)	–	H	0
G	(G)	B	J	–	0
H	–	B	E	(H)	1
J	G	–	(J)	–	0

Figure 9.33 Reduction obtained by using the partitioning procedure.

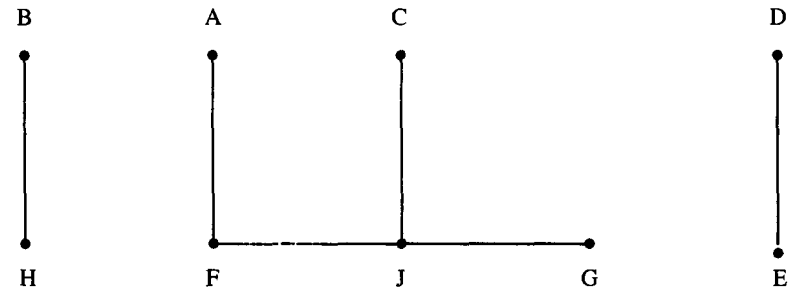


Figure 9.34 Merger diagram for Figure 9.33.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	G	–	(C)	D	0
D	G	A	(D)	(D)	1
G	(G)	B	(G)	–	0

Figure 9.35 Reduction obtained from the merger diagram in Figure 9.34.

for merging, we can construct the merger diagram in Figure 9.36. From this diagram it is apparent that rows C and G can be merged; let the new row be called C. This leads to the flow table in Figure 9.37, which cannot be reduced any more.

Example 9.9 Consider the flow table in Figure 9.38. Applying the partitioning procedure to this table gives

$$P_1 = (AFK)(BJ)(CG)(D)(E)(H)$$
$$P_2 = (A)(FK)(BJ)(C)(G)(D)(E)(H)$$
$$P_3 = P_2$$

Combining B and J into a new state B, and F and K into F, gives the flow table in Figure 9.39.

Figure 9.36 Merger diagram for Figure 9.35.

Present state	Next state				Output z
	$w_2w_1 = 00$	01	10	11	
A	(A)	(A)	C	B	0
B	A	(B)	D	(B)	1
C	(C)	B	(C)	D	0
D	C	A	(D)	(D)	1

Figure 9.37 Reduced flow table for Example 9.8.

Present state	Next state				Output z
	$w_2w_1 = 00$	01	10	11	
A	(A)	G	E	–	0
B	K	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	J	–	0
G	K	(G)	–	D	1
H	–	–	E	(H)	1
J	F	–	(J)	D	0
K	(K)	C	B	–	0

Figure 9.38 Flow table for Example 9.9.

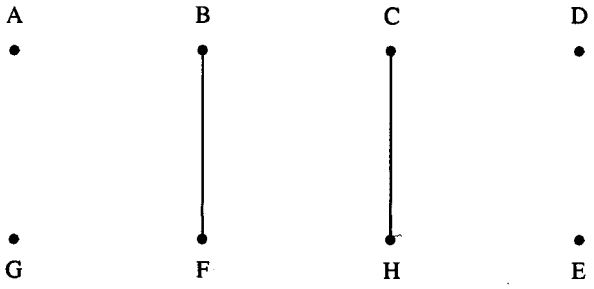
Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	G	E	–	0
B	F	–	(B)	D	0
C	F	(C)	–	H	1
D	–	C	E	(D)	0
E	A	–	(E)	D	1
F	(F)	C	B	–	0
G	F	(G)	–	D	1
H	–	–	E	(H)	1

Figure 9.39 Reduction resulting from the partitioning procedure.

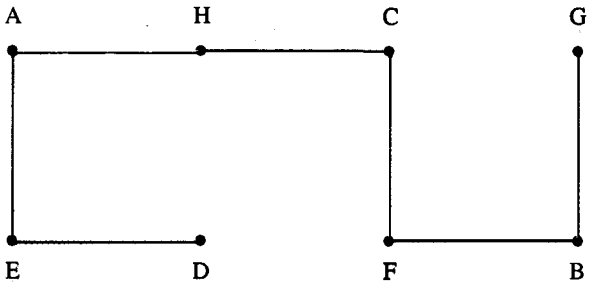
Figure 9.40*a* gives a merger diagram for this flow table, indicating the possibilities for merger if the Moore model of the FSM is to be preserved. In this case B and F can be merged, as well as C and H, resulting in a six-row flow table.

Next we should consider the merging possibilities if we are willing to change to the Mealy model. When going from the Moore model to the Mealy model, a stable state in the Mealy model must generate the same output as it had in the Moore model. It is also important to ensure that transitions in the Mealy model will not produce undesirable glitches in the output signal.

Figure 9.41 indicates how the FSM of Figure 9.39 can be represented in the Mealy form. The next-state entries are unchanged. In Figure 9.41, for each stable state the output value must be the same as for the corresponding row of the Moore-type table. For example, $z = 0$ when the state A is stable under $w_2w_1 = 00$. Also, $z = 0$ when the states B, D, and F are stable under $w_2w_1 = 10$, 11, and 00, respectively. Similarly, $z = 1$ when C, E, G, and H are stable under $w_2w_1 = 01$, 10, 01, and 11, respectively. If a transition from one stable state to another requires the output to change from 0 to 1, or from 1 to 0, then the exact time when the change takes place is not important, as we explained in section 9.1 when discussing Figure 9.3. For instance, suppose that the FSM is stable in A under $w_2w_1 = 00$, producing $z = 0$. If the inputs then change to $w_2w_1 = 01$, a transition to state G must be made, where $z = 1$. Since it is not essential that z becomes 1 before the circuit reaches the state G, the output entry in row A that corresponds to this transition can be treated as a don't care; therefore, it is left unspecified in the table. From the stable state A, it is also possible to change to E, which allows specifying another don't care because z changes from 0 to 1. A different situation arises in row B. Suppose that the circuit is stable in B under $w_2w_1 = 10$ and that the inputs change to 11. This has to cause a change to stable state D, and z must remain at 0 throughout the change in states. Hence the output in row B under $w_2w_1 = 11$ is specified as 0. If it were left unspecified, to be used as a don't care, then it is



(a) Preserving the Moore model



(b) Complete merger diagram

Figure 9.40 Merger diagrams for Figure 9.39.

possible that in the implementation of the circuit this don't care may be treated as a 1. This would cause a glitch in z, which would change $0 \rightarrow 1 \rightarrow 0$ as the circuit moves from B to D when the inputs change from 10 to 11. The same situation occurs for the transition from B to F when the inputs change from 10 to 00. We can use the same reasoning to determine other output entries in Figure 9.41.

From Figure 9.41 we can derive the merger diagram in Figure 9.40*b*. The blue lines connect the rows that can be merged only by specifying the output in the Mealy style. The black lines connect the rows that can be merged even if the outputs are of Moore type; they correspond to the diagram in Figure 9.40*a*. Choosing the subsets of compatible states (A, H), (B, G), (C, F), and (D, E), the FSM can be represented using only four states. Merging the states A and H into a new state A, states B and G into B, states C and F into C, and D and E into D, we obtain the reduced flow table in Figure 9.42. Each entry in this table meets the requirements specified in the corresponding rows that were merged.

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	G	E	–	0	–	–	–
B	F	–	(B)	D	0	–	0	0
C	F	(C)	–	H	–	1	–	1
D	–	C	E	(D)	–	–	–	0
E	A	–	(E)	D	–	–	1	–
F	(F)	C	B	–	0	–	0	–
G	F	(G)	–	D	–	1	–	–
H	–	–	E	(H)	–	–	1	1

Figure 9.41 The FSM of Figure 9.39 specified in the form of the Mealy model.

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	B	D	(A)	0	–	1	1
B	C	(B)	(B)	D	0	1	0	0
C	(C)	(C)	B	A	0	1	0	1
D	A	C	(D)	(D)	–	–	1	0

Figure 9.42 Reduced flow table for Example 9.9.

Example 9.10 As another example consider the flow table in Figure 9.43. The partitioning procedure gives

$$P_1 = (AF)(BEG)(C)(D)(H)$$
$$P_2 = (AF)(BE)(G)(C)(D)(H)$$
$$P_3 = P_2$$

Replacing state F with A, and state E with B, results in the flow table in Figure 9.44. The corresponding merger diagram is presented in Figure 9.45. It is apparent that states A, B,

and C can be merged and replaced with a new state A. Also D, G, and H can be merged into a new state D. The result is the reduced flow table in Figure 9.46, which has only two rows. Again we have used the Mealy model because the merged stable states D and H have $z = 1$ while G has $z = 0$.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	F	(B)	–	H	0
C	F	–	(C)	H	0
D	(D)	G	C	–	1
E	A	(E)	–	H	0
F	(F)	E	C	–	0
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figure 9.43 Flow table for Example 9.10.

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	(A)	B	C	–	0
B	A	(B)	–	H	0
C	A	–	(C)	H	0
D	(D)	G	C	–	1
G	D	(G)	–	H	0
H	–	G	C	(H)	1

Figure 9.44 Reduction after the partitioning procedure.

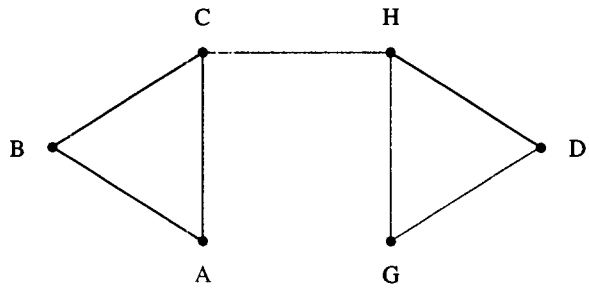


Figure 9.45 Merger diagram for Figure 9.44.

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	(A)	(A)	D	0	0	0	–
D	(D)	(D)	A	(D)	1	0	–	1

Figure 9.46 Reduced flow table for Example 9.10.

9.5 STATE ASSIGNMENT

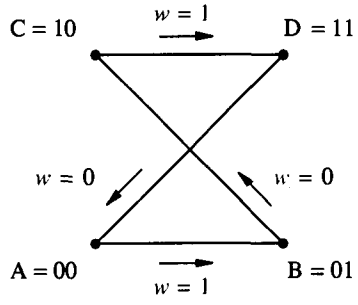
The examples in section 9.3 illustrate that the state assignment task for asynchronous FSMs is complex. The time needed to change the value of a state variable depends on the propagation delays in the circuit. Thus it is impossible to ensure that a change in the values of two or more variables will take place at exactly the same time. To achieve reliable operation of the circuit, the state variables should change their values one at a time in controlled fashion. This is accomplished by designing the circuit such that a change from one state to another entails a change in one state variable only.

States in FSMs are encoded as bit strings that represent different valuations of the state variables. The number of bit positions in which two given bit strings differ is called the *Hamming distance* between the strings. For example, for bit strings 0110 and 0100 the Hamming distance is 1, while for 0110 and 1101 it is 3. Using this terminology, an ideal state assignment has a Hamming distance of 1 for all transitions from one stable state to another. When the ideal state assignment is not possible, an alternative that makes use of unspecified states and/or transitions through unstable states must be sought. Sometimes it is necessary to increase the number of state variables to provide the needed flexibility.

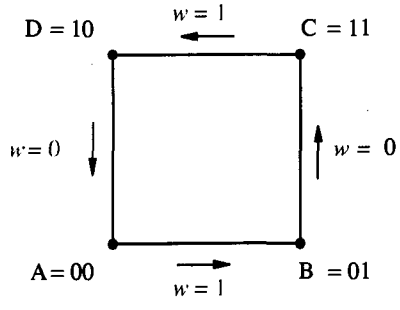
Example 9.11

Consider the parity-generating FSM in Figure 9.13. Two possible state assignments for this FSM are given in Figure 9.14. The transitions between states, as specified in Figure 9.13*b*, can be described in pictorial form as shown in Figure 9.47. Each row of the flow table is represented by a point. The four points needed to represent the rows are placed as vertices of a square. Each vertex has an associated code that represents a valuation of the state variables, y_2y_1. The codes shown in the figure, with $y_2y_1 = 00$ in the lower-left corner and so on, correspond to the coordinates of the two-dimensional cube presented in section 4.8. Figure 9.47*a* shows what happens if the state assignment in Figure 9.14*a* is used; namely, if $A = 00$, $B = 01$, $C = 10$, and $D = 11$. There is a transition from A to B if $w = 1$, which requires a change in y_1 only. A transition from C to D occurs if $w = 1$, which also requires a change in y_1 only. However, a transition from B to C caused by $w = 0$ involves a change in the values of both y_2 and y_1. Similarly, both state variables must change in going from D to A if $w = 0$. A change in both variables corresponds to a diagonal path in the diagram.

Figure 9.47*b* shows the effect of the state assignment in Figure 9.14*b*, which reverses the valuations assigned to C and D. In this case all four transitions are along the edges of the two-dimensional cube, and they involve a change in only one of the state variables. This is the desirable state assignment.



(a) Corresponding to Figure 9.14*a*

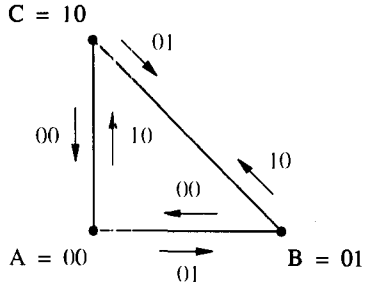


(b) Corresponding to Figure 9.14*b*

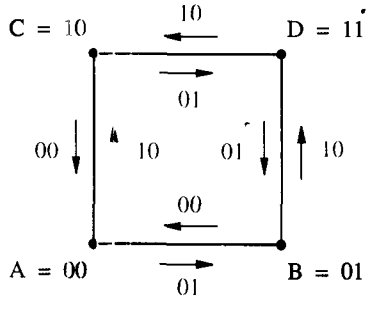
Figure 9.47 Transitions in Figure 9.13.

Example 9.12 The flow table for an arbiter FSM is given in Figure 9.21*a*. Transitions for this FSM are shown in Figure 9.48*a*, using the state assignment $A = 00$, $B = 01$, and $C = 10$. In this case multiple transitions are possible between the states. For example, there are two transitions between A and B: from B to A if $r_2r_1 = 00$ and from A to B if $r_2r_1 = 01$. Again there is a diagonal path, corresponding to transitions between B and C, which should be avoided. A possible solution is to introduce a fourth state, D, as indicated in Figure 9.48*b*. Now the transitions between B and C can take place via the unstable state D. Thus instead of going directly from B to C when $r_2r_1 = 10$, the circuit will go first from B to D and then from D to C.

Using the arrangement in Figure 9.48*b* requires modifying the flow table as shown in Figure 9.49. The state D is not stable for any input valuation. It cannot be reached if $r_2r_1 = 00$ or 11; hence these entries are left unspecified in the table. Also observe that we have specified the output $g_2g_1 = 10$ for state D, rather than leaving it unspecified. When a transition from one stable state to another takes place via an unstable state, the output of the unstable state must be the same as the output of one of the two stable states involved



(a) Transitions in Figure 9.21 *a*



(b) Using the extra state *D*

Figure 9.48 Transitions for the arbiter FSM in Figure 9.21.

Present state	Next state				Output
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(A)	B	C	–	00
B	A	(B)	D	(B)	01
C	A	D	(C)	(C)	10
D	–	B	C	–	10

Figure 9.49 Modified flow table based on the transitions in Figure 9.48*b*.

in the transition to ensure that a wrong output is not generated while passing through the unstable state.

It is interesting to compare this flow table with the excitation table in Figure 9.21*b*, which is also based on using the extra state *D*. In Figure 9.21*b* the state *D* specifies the necessary transitions should the circuit accidentally find itself in this state as a result of a race in changing the values of both state variables. In Figure 9.49 the state *D* is used in orderly transitions, which are not susceptible to any race conditions.

9.5.1 TRANSITION DIAGRAM

A diagram that illustrates the transitions specified in a flow table is called a *transition diagram*. In some books such diagrams are called *state-adjacency diagrams*. These diagrams provide a convenient aid in searching for a suitable state assignment.

A good state assignment results if the transition diagram does not have any diagonal paths. A general way of stating this requirement is to say that it must be possible to *embed* the transition diagram onto a k-dimensional cube, because in a cube all transitions between adjacent vertices involve the Hamming distance of 1. Ideally, a transition diagram for an FSM with n state variables can be embedded onto an n-dimensional cube, as is the case in the examples in Figures 9.47*b* and 9.48*b*. If this is not possible, then it becomes necessary to introduce additional state variables, as we will see in later examples.

The diagrams in Figures 9.47 and 9.48 present all information pertinent to transitions between the states in the given FSMs. For larger FSMs such diagrams take on a cluttered appearance. A simpler form can be used instead, as described below.

A transition diagram has to show the state transitions for each valuation of the input variables. The direction of a transition, for example from *A* to *B* or from *B* to *A*, is not important, because it is only necessary to ensure that all transitions involve the Hamming distance of 1. The transition diagram has to show the effect of individual transitions into each stable state, which may involve passing through unstable states. For a given row of a

flow table, it is possible to have two or more stable-state entries for different input valuations. It is useful to identify the transitions leading into these stable states with distinct labels in a transition diagram. To give each stable-state entry a distinct label, we will denote the stable-state entries with numbers 1, 2, 3, $\cdots$. Thus if state A is stable for two input valuations, we will replace the label A with 1 for one input valuation and with 2 for the other valuation.

Figure 9.50 shows a relabeled version of the flow table in Figure 9.21*a*. We have arbitrarily chosen to label (A) as 1, the two appearances of (B) as 2 and 3, and the two appearances of (C) as 4 and 5. All entries in each next-state column are labeled using this scheme. The transitions identified by these labels are presented in Figure 9.51*a*. The same information is given in Figure 9.48*a*. Actually, the diagram in Figure 9.48*a* contains more information because arrowheads show the direction of each transition. Note also that the edges in that diagram are labeled with input values r_2r_1, whereas the edges in Figure 9.51*a* are labeled with numerical stable-state labels as explained above.

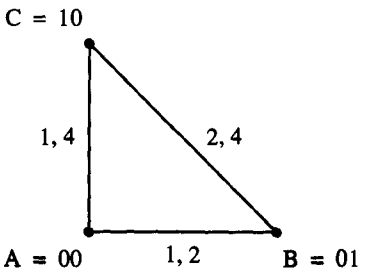
Figure 9.50 indicates that the stable state 2, which is one instance of the stable state B, can be reached either from state A or from state C. There is a corresponding label 2 on the paths connecting the vertices in the diagram in Figure 9.51*a*. The difficulty from the state-assignment point of view is that the path from C to B is diagonal. In Example 9.12 this problem was resolved by introducing a new state D. By examining the flow table in Figure 9.50 more closely, we can see that the functional behavior of the required arbiter FSM can be achieved if the transition from C to B takes place via state A. Namely, if the circuit is stable in C, then the input $r_2r_1 = 01$ can cause the change to A, from which the circuit immediately proceeds to state B. We can indicate the possibility of using this path by placing the label 2 on the edge that connects C and A in Figure 9.51*a*.

A similar situation exists for the transition from B to C, which is labeled 4. An alternative path can be realized by causing the circuit to go from state B to state A if $r_2r_1 = 10$ and then immediately proceed to C. This can be indicated by placing the label 4 on the edge that connects B and A in Figure 9.51*a*.

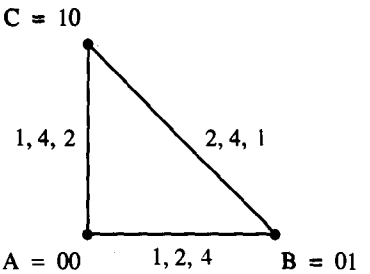
A possibility of having an alternative path for a transition exists whenever two states have the same uncircled label in the relabeled flow diagram. In Figure 9.50 there is a third such possibility if $r_2r_1 = 00$, using the label 1. This possibility is not useful because changing from either B or C to A involves a change in only one state variable using the state assignment in Figure 9.51*a*. Hence there would be no benefit in having a transition between B and C for this input valuation.

Present state	Next state				Output
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	(1)	2	4	–	00
B	1	(2)	4	(3)	01
C	1	2	(4)	(5)	10

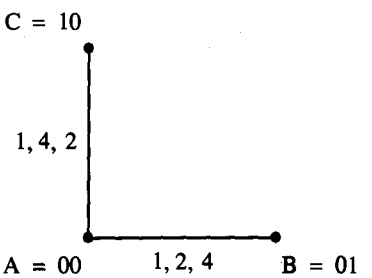
Figure 9.50 Relabeled flow table of Figure 9.21*a*.



(a) Transitions in Figure 9.50



(b) Complete transition diagram



(c) Selected transition diagram

Figure 9.51 Transition diagrams for Figure 9.50.

To depict the possibility of having alternative paths, we will indicate in blue the corresponding transitions on the diagram. Thus a complete transition diagram will show all direct transitions to stable states in black and possible indirect transitions through unstable states in blue. Figure 9.51*b* shows the complete transition diagram for the flow table in Figure 9.21*a*.

The transition diagram in Figure 9.51*b* cannot be embedded on the two-dimensional cube, because some transitions require a diagonal path. The blue label 1 on the path between B and C is of no concern, because it represents only an alternative path that does not have to be used. But the transitions between B and C labeled 2 and 4 are required. The diagram shows an alternative path, through A, having the labels 2 and 4. Therefore, the alternative path can be used, and the diagonal connection in the diagram can be eliminated. This leads to the transition diagram in Figure 9.51*c*, which can be embedded on the two-dimensional cube. The conclusion is that the state assignment $A = 00$, $B = 01$, and $C = 10$ is good, but the flow table must be modified to specify the transitions through alternative paths. The modified table is the same as the flow table designed earlier using an ad hoc approach, shown in Figure 9.23*a*.

As a final comment on this example, note the impact of alternative paths on the outputs produced by the FSM. If $r_2r_1 = 01$, then a change from a stable state C through unstable A to stable B generates the outputs $g_2g_1 = 10 \rightarrow 00 \rightarrow 01$, rather than $10 \rightarrow 01$ as specified in Figure 9.21*a*. For the arbiter FSM this presents no problem, as explained in Example 9.6.

Procedure for Deriving Transition Diagrams

The transition diagram is derived from a flow table as follows:

- Derive the relabeled flow table as explained above. For a given input valuation, all transitions that lead to the same stable state are labeled with the same number. Transitions through unstable states that eventually lead to a stable state are given the same number as the stable-state entry.
- Represent each row of the flow table by a vertex.
- Join two vertices, V_i and V_j, by an edge if they have the same number in any column of the relabeled flow table.
- For each column in which V_i and V_j have the same number, label the edge between V_i and V_j with that number. We will use black labels for direct transitions to circled (stable) states and blue labels when the next-state entries for both V_i and V_j in the flow table are uncircled.

Note that the first point says that in the relabeled flow table the transitions through unstable states are given the label of the stable state to which they lead for a given input valuation. For example, to derive a transition diagram starting from the flow table in Figure 9.23*a*, the table would be relabeled to give the table in Figure 9.50. The transition from stable A to stable B, when $r_2r_1 = 01$, has the label 2. The same label is given to the transition from stable C to unstable A because this transition ultimately leads to stable B.

9.5.2 EXPLOITING UNSPECIFIED NEXT-STATE ENTRIES

Unspecified entries in a flow table provide some flexibility in finding good state assignments. The following example presents a possible approach. The example also illustrates all steps in the derivation of a transition diagram.

Example 9.13

Consider the flow table in Figure 9.52*a*. This FSM has seven stable-state entries. Labeling these entries in order, from 1 to 7, results in the table in part (*b*) of the figure. In this case states 1 and 2 correspond to state *A*, 3 and 4 to state *B*, 5 and 6 to state *C*, and 7 to state *D*. In the column $w_2w_1 = 00$ there is a transition from *C* to *A*, which is labeled 1, and a transition from *D* to *B*, which is labeled 3, because 1 and 3 are the successor stable states in these transitions. Similarly, in column 11 there are transitions from *B* to *C* and from *D* to *A*, which are labeled 6 and 2, respectively. In column 01 there is a transition from *A* to *B*, which is labeled 4. State *C* is stable for this input valuation; it is labeled 5. There is no transition specified that leads to this stable state. The state can be reached only if *C* is stable under $w_2w_1 = 11$, which is labeled 6, and then the inputs change to $w_2w_1 = 01$. Note that the FSM remains stable in *C* if the inputs change from 11 to 01, or vice versa. Column 10 illustrates how unstable states are treated. From the stable state *A*, a transition to the unstable state *C* is specified. As soon as the FSM reaches state *C*, it proceeds to change to the stable state *D*, which is labeled 7. Thus 7 is used as the label for the entire transition sequence from *A* to *C* to *D*.

Taking rows *A*, *B*, *C*, and *D* as the four vertices, a first attempt at drawing the transition diagram is given in Figure 9.53*a*. The diagram shows transitions between all pairs of states,

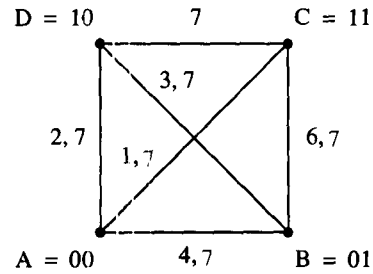
Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(A)	B	C	(A)	00
B	(B)	(B)	D	C	01
C	A	(C)	D	(C)	10
D	B	–	(D)	A	11

(a) Flow table

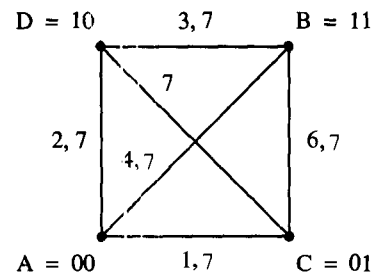
Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z_2z_1
A	(1)	4	7	(2)	00
B	(3)	(4)	7	6	01
C	1	(5)	7	(6)	10
D	3	–	(7)	2	11

(b) Relabeled flow table

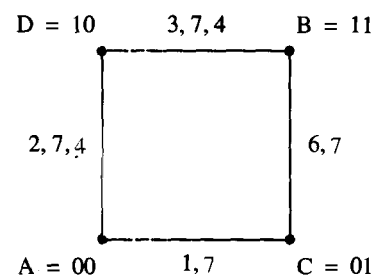
Figure 9.52 Flow tables for Example 9.13.



(a) First transition diagram



(b) Second transition diagram



(c) Augmented transition diagram

Figure 9.53 Transition diagrams for Figure 9.52.

which seems to suggest that it is impossible to have a state assignment where all transitions are characterized by a Hamming distance of 1. If the state assignment $A = 00$, $B = 01$, $C = 11$, and $D = 10$ is used, then the diagonal transition between A and C, or B and D, requires both state variables to change their values. The diagonal path from B to D with the label 7 is not needed, because an alternative path from B to D exists under label 7 that passes either through state A or through state C. Unfortunately, the diagonal paths labeled 1 and 3 cannot be removed, because there are no alternative paths for these transitions.

As the next attempt at finding a suitable state assignment, we will reverse the codes given to B and C, which yields the transition diagram in Figure 9.53*b*. Now the same argument about the alternative paths labeled 7 indicates that the diagonal from C to D can be omitted. Also, the label 7 on the diagonal between A and B can be omitted. However, this diagonal must remain because of the label 4 for which there is no alternative path between A and B. Looking at the flow table in Figure 9.52*b*, we see an unspecified entry in the column $w_2w_1 = 01$. This entry can be exploited by replacing it with the label 4, in which case the transition graph would show the label 4 on the edges connecting A and D, as well as B and D. Thus the diagonal between A and B could be removed, producing the transition diagram in Figure 9.53*c*. This diagram can be embedded on a two-dimensional cube, which means that the state assignment $A = 00$, $B = 11$, $C = 01$, and $D = 10$ can be used.

For the transition diagram in Figure 9.53*c* to be applicable, the flow table for the FSM must be modified as shown in Figure 9.54*a*. The unspecified entry in Figure 9.52*a* now specifies a transition to state B. According to Figure 9.53*c*, the change from state A to B under input valuation $w_2w_1 = 01$ must pass through state D; hence the corresponding entry in the first row is modified to ensure that this will take place. Also, when $w_2w_1 = 10$, the

Present state	Next state				Output z_2z_1			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	(A)	D	D	(A)	00	00	11	00
B	(B)	(B)	D	C	01	01	11	01
C	A	(C)	B	(C)	-0	10	1-	10
D	B	B	(D)	A	-1	0-	11	00

(a) Modified flow table

	Present state	Next state				Output			
		$w_2w_1 = 00$	01	10	11	00	01	10	11
	y_2y_1	Y_2Y_1				z_2z_1			
A	00	(00)	10	10	(00)	00	00	11	00
B	11	(11)	(11)	10	01	01	01	11	01
C	01	00	(01)	11	(01)	-0	10	1-	10
D	10	11	11	(10)	00	-1	0-	11	00

(b) Excitation table

Figure 9.54 Realization of the FSM in Figure 9.52*a*.

FSM must go to state D. If it happens to be in state C, then this change has to occur either via state A or state B. We have chosen the path via state B in Figure 9.54*a*.

The original flow table in Figure 9.52*a* is defined in the form of the Moore model. The modified flow table in Figure 9.54*a* requires the use of the Mealy model because the previously described transitions through unstable states must produce correct outputs. Consider first the change from A if $w_2w_1 = 01$. While stable in state A, the circuit must produce the output $z_2z_1 = 00$. Upon reaching the stable state B, the output must become 01. The problem is that this transition requires a short visit to state D, which in the Moore model would produce $z_2z_1 = 11$. Thus a glitch would be generated on the output signal z_2, which would undergo the change $0 \rightarrow 1 \rightarrow 0$. To avoid this undesirable glitch, the output in state D must be $z_2 = 0$ for this input valuation, which requires the use of the Mealy model as shown in the Figure 9.54*a*. Observe that while z_2 must be 0 in D for $w_2w_1 = 01$, z_1 can be either 0 or 1 because it is changing from 0 in state A to 1 in state B. Therefore, z_1 can be left unspecified so that this case can be treated as a don't-care condition. A similar situation arises when the circuit changes from C to D via B if $w_2w_1 = 10$. The output must change from 10 to 11, which means that z_2 must remain at 1 throughout this change, including the short time in state B where the Moore model output would be 01.

The modified flow table and the chosen state assignment lead to the excitation table in Figure 9.54*b*. From this table the next-state and output expressions are derived, as in the examples in section 9.3.

9.5.3 STATE ASSIGNMENT USING ADDITIONAL STATE VARIABLES

In Figure 9.52*a* there is an unspecified transition that can be exploited to find a suitable state assignment, as shown in section 9.5.2. In general, such flexibility may not exist. It may be impossible to find a race-free state assignment using $\log_2 n$ state variables for a flow table that has n rows. The problem can be solved by adding extra state variables. This can be done in three ways, as illustrated in the examples that follow.

Example 9.14 **USING EXTRA UNSTABLE STATES** Consider the FSM specified by the flow table in Figure 9.55*a*. The flow table is relabeled in part (*b*) of the figure. A corresponding transition diagram is depicted in Figure 9.56*a*. It indicates that there are transitions between all pairs of vertices (rows). No rearrangement of the existing vertices would allow mapping of the transition diagram onto a two-dimensional cube.

Let us now introduce one more state variable so that we can look for a way to map the transition diagram onto a three-dimensional cube. With three state variables the assignment for state A can be a Hamming distance of 1 different from the assignments for B, C, and D. For example, we could have $A = 000$, $B = 001$, $C = 100$, and $D = 010$. But it would then be impossible to have the pairs (B, C), (B, D), and (C, D) within the Hamming distance of 1. The solution here is to insert extra vertices in the transition paths, as shown in Figure 9.56*b*. Vertex E separates B from D, while vertices F and G break the paths (B, C) and (C, D). The labels associated with the transitions are attached to both segments of a

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	D	(B)	01
C	(C)	B	(C)	D	10
D	C	A	(D)	(D)	11

(a) Flow table

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
A	(1)	(2)	6	4	00
B	1	(3)	7	(4)	01
C	(5)	3	(6)	8	10
D	5	2	(7)	(8)	11

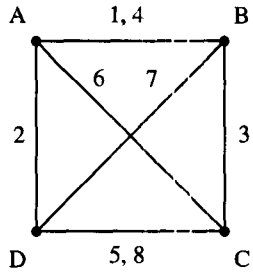
(b) Relabeled flow table

Figure 9.55 FSM for Example 9.14.

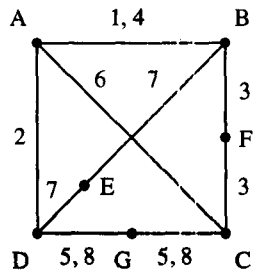
broken path. The resulting transition diagram can be embedded onto a three-dimensional cube as indicated in Figure 9.56*c*, where the black portion of the cube comprises the desired paths. Now the transition from B to D takes place via vertex E if $w_2w_1 = 10$ (label 7). The transition from C to B occurs via F if $w_2w_1 = 01$ (label 3). The transition from C to D goes through G if $w_2w_1 = 11$ (label 8), and the transition from D to C goes via G if $w_2w_1 = 00$ (label 5). Therefore, the flow table has to be modified as shown in Figure 9.57*a*. The three extra states are unstable because the circuit will not remain in these states for any valuation of the inputs. The circuit will merely pass through these states in the process of changing from one stable state to another. Observe that each of the states E, F, and G is needed to facilitate the transitions caused by only one or two valuations of inputs. Thus it is not necessary to specify the actions that might be caused by other input valuations, because such situations will never occur in a properly functioning circuit.

The outputs in Figure 9.57*a* can be specified using the Mealy model. It is essential that a proper output is generated when passing through unstable states, to avoid undesirable glitches in the output signals.

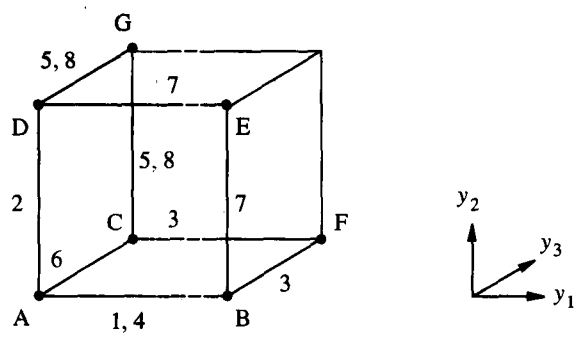
If we assign the state variables as shown on the right of Figure 9.56*c*, the modified flow table leads to the excitation table in Figure 9.57*b*. From this table, deriving the next-state and output expressions is a straightforward task.



(a) Transition diagram



(b) Augmented transition diagram



(c) Embedded transition diagram

Figure 9.56 Transition diagrams for Figure 9.55.

Example 9.15 **USING PAIRS OF EQUIVALENT STATES** Another approach is to increase the flexibility in state assignment by introducing an equivalent new state for each existing state. Thus state A can be replaced with two states $A1$ and $A2$ such that the final circuit produces the same outputs for $A1$ and $A2$ as it would for A. Similarly, other states can be replaced by equivalent pairs of states. Figure 9.58 shows how a three-dimensional cube can be used to find a good

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
A	(A)	(A)	C	B	00
B	A	(B)	E	(B)	01
C	(C)	F	(C)	G	10
D	G	A	(D)	(D)	11
E	–	–	D	–	–1
F	–	B	–	–	01
G	C	–	–	D	1–

(a) Modified flow table

	Present state $y_3y_2y_1$	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
		$Y_3Y_2Y_1$				
A	000	(000)	(000)	100	001	00
B	001	000	(001)	011	(001)	01
C	100	(100)	101	(100)	110	10
D	010	110	000	(010)	(010)	11
E	011	–	–	010	–	–1
F	101	–	001	–	–	01
G	110	100	–	–	010	1–

(b) Excitation table

Figure 9.57 Modified tables for Example 9.14.

state assignment for a four-row flow table. The four equivalent pairs are arranged so that the minimum Hamming distance of 1 exists between all pairs. For example, the pair ($B1$, $B2$) has the Hamming distance of 1 with respect to $A1$ (or $A2$), $C2$, and $D2$.

The transition diagram in Figure 9.56*a* can be embedded onto the three-dimensional cube as shown in Figure 9.58. Since there is a choice of two vertices on the cube for each

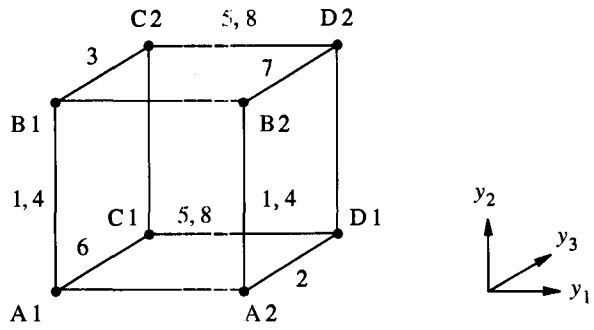


Figure 9.58 Embedded transition diagram if two nodes per row are used.

vertex in the transition diagram in Figure 9.56*a*, the embedded transition diagram does not involve any diagonal paths. Using this assignment of states, the flow table in Figure 9.55*a* has to be modified as presented in Figure 9.59*a*. The entries in the table are made to allow each transition in the original flow table to be realized using a transition between the corresponding pairs of equivalent states. Both states in an equivalent pair are stable for the input valuations for which the original state is stable. Thus $A1$ and $A2$ are stable if $w_2w_1 = 00$ or 01, $B1$ and $B2$ are stable if $w_2w_1 = 01$ or 11, and so on. At any given time the FSM may be in either of the two equivalent states that represent an original state. Then a change to another state must be possible from either of these states. For example, Figure 9.55*a* specifies that the FSM must change from the stable state A to state B if the input is $w_2w_1 = 11$. The equivalent transition in the modified flow table is the change from state $A1$ to $B1$ or from state $A2$ to $B2$. If the FSM is stable in A and the input changes from 00 to 10, then a change to C is required. The equivalent transition in the modified flow table is from state $A1$ to $C1$; if the FSM happens to be in state $A2$, it will first have to change to $A1$. The remaining entries in Figure 9.59*a* are derived using the same reasoning.

The outputs are specified using the Moore model, because the only unstable states are those involved in changing from one member of the equivalent pair to another, and both members generate the same outputs. For instance, in the previously described transition from A to C, if the starting point is $A2$, it is necessary to go first to $A1$ and then to $C1$. Even though $A1$ is unstable for $w_2w_1 = 10$, there is no problem because its output is the same as that of $A2$. Therefore, if the original flow table is defined using the Moore model, then the modified flow table can also be done using the Moore model.

Using the assignment of the state variables in Figure 9.58 gives the excitation table in Figure 9.59*b*.

9.5.4 ONE-HOT STATE ASSIGNMENT

The previously described schemes based on embedding the flow table in a cube may lead to an optimal state assignment, but they require a trial-and-error approach that becomes awkward for large machines. A straightforward, but more expensive, alternative is to use

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
A1	(A1)	(A1)	C1	B1	00
A2	(A2)	(A2)	A1	B2	00
B1	A1	(B1)	B2	(B1)	01
B2	A2	(B2)	D2	(B2)	01
C1	(C1)	C2	(C1)	D1	10
C2	(C2)	B1	(C2)	D2	11
D1	C1	A2	(D1)	(D1)	11
D2	C2	D1	(D2)	(D2)	11

(a) Modified flow table

	Present state $y_3y_2y_1$	Next state $w_2w_1 = 00$	01	10	11	Output z_2z_1
		$Y_3Y_2Y_1$				
A1	000	(000)	(000)	100	010	00
A2	001	(001)	(001)	000	011	00
B1	010	000	(010)	011	(010)	01
B2	011	001	(011)	111	(011)	01
C1	100	(100)	110	(100)	101	10
C2	110	(110)	010	(110)	111	10
D1	101	100	001	(101)	(101)	11
D2	111	110	101	(111)	(111)	11

(b) Excitation table

Figure 9.59 Modified flow and excitation tables for Example 9.15.

State assignment	Present State	Next state				Output
		$w_2w_1 = 00$	01	10	11	z_2z_1
0001	A	(A)	(A)	E	F	00
0010	B	F	(B)	G	(B)	01
0100	C	(C)	H	(C)	I	10
1000	D	I	J	(D)	(D)	11
0101	E	–	–	C	–	–0
0011	F	A	–	–	B	0–
1010	G	–	–	D	–	–1
0110	H	–	B	–	–	01
1100	I	C	–	–	D	1–
1001	J	–	A	–	–	00

Figure 9.60 State assignment with one-hot encoding.

one-hot codes. If each row in the flow table of an FSM is assigned a one-hot code, then race-free state transitions can be achieved by passing through unstable states that are at a Hamming distance of 1 from the two stable states involved in the transition. For example, suppose that state A is assigned the code 0001 and state B the code 0010. Then a race-free transition from A to B can pass through an unstable state 0011. Similarly, if C is assigned the code 0100, then a transition from A to C can be done via the unstable state 0101.

Using this approach, the flow table in Figure 9.55*a* can be modified as illustrated in Figure 9.60. The four states, A, B, C, and D, are assigned one-hot codes. As seen in the figure, it is necessary to introduce six unstable states, E through J, to handle the necessary transitions. These unstable states have to be specified only for the specific transitions, whereas for other input valuations they may be treated as don't cares.

The outputs can be specified using the Moore model. In some cases it does not matter when a particular output signal changes its value. For instance, state E is used to facilitate the transition from state A to C. Since $z_2z_1 = 00$ in A and 10 in C, it is not important if z_2 changes when passing through state E.

While straightforward to implement, the one-hot encoding is expensive because it requires n state variables to implement an n-row flow table. Simplicity of design and the cost of implementation often provide a challenging trade-off in designing logic circuits!

9.6 HAZARDS

In asynchronous sequential circuits it is important that undesirable glitches on signals should not occur. The designer must be aware of the possible sources of glitches and ensure that the transitions in a circuit will be glitch free. The glitches caused by the structure of a given

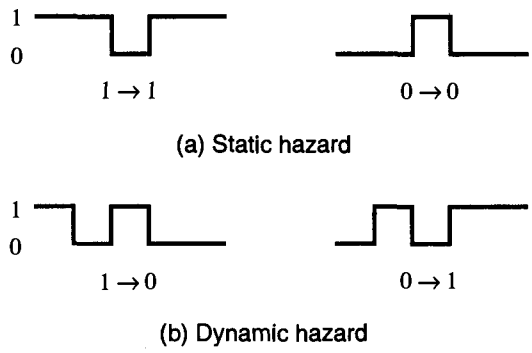


Figure 9.61 Definition of hazards.

circuit and propagation delays in the circuit are referred to as *hazards*. Two types of hazards are illustrated in Figure 9.61.

A *static hazard* exists if a signal is supposed to remain at a particular logic value when an input variable changes its value, but instead the signal undergoes a momentary change in its required value. As shown in Figure 9.61*a*, one type of static hazard is when the signal at level 1 is supposed to remain at 1 but dips to 0 for a short time. Another type is when the signal is supposed to remain at level 0 but rises momentarily to 1, thus producing a glitch.

A different type of hazard may occur when a signal is supposed to change from 1 to 0 or from 0 to 1. If such a change involves a short oscillation before the signal settles into its new level, as illustrated in Figure 9.61*b*, then a *dynamic hazard* is said to exist.

9.6.1 STATIC HAZARDS

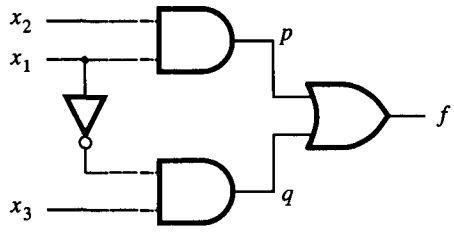
Figure 9.62*a* shows a circuit with a static hazard. Suppose that the circuit is in the state where $x_1 = x_2 = x_3 = 1$, in which case $f = 1$. Now let x_1 change from 1 to 0. Then the circuit is supposed to maintain $f = 1$. But consider what happens when the propagation delays through the gates are taken into account. The change in x_1 will probably be observed at point p before it will be seen at point q because the path from x_1 to q has an extra gate (NOT) in it. Thus the signal at p will become 0 before the signal at q becomes equal to 1. For a short time both p and q will be 0, causing f to drop to 0 before it recovers back to 1. This gives rise to the signal depicted on the left side of Figure 9.61*a*.

The glitch on f can be prevented as follows. The circuit implements the function

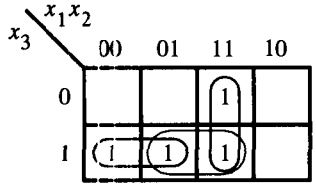
$$f = x_1x_2 + \bar{x}_1x_3$$

The corresponding Karnaugh map is given in Figure 9.62*b*. The two product terms realize the prime implicants encircled in black. The hazard explained above occurs when there is a transition from the prime implicant x_1x_2 to the prime implicant $\bar{x}_1x_3$. The hazard can be eliminated by including the third prime implicant, encircled in blue. Then the function would be implemented as

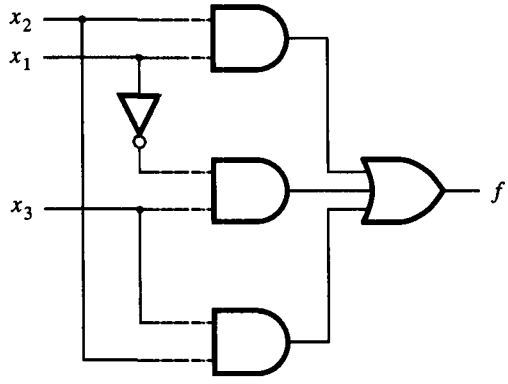
$$f = x_1x_2 + \bar{x}_1x_3 + x_2x_3$$



(a) Circuit with a hazard



(b) Karnaugh map



(c) Hazard-free circuit

Figure 9.62 An example of a static hazard.

Now the change in x_1 from 1 to 0 would have no effect on the output f because the product term x_2x_3 would be equal to 1 if $x_2 = x_3$, regardless of the value of x_1. The resulting hazard-free circuit is depicted in Figure 9.62*c*.

A potential hazard exists wherever two adjacent 1s in a Karnaugh map are not covered by a single product term. Therefore, a technique for removing hazards is to find a cover

in which some product term includes each pair of adjacent 1s. Then, since a change in an input variable causes a transition between two adjacent 1s, no glitch can occur because both 1s are included in a product term.

In asynchronous sequential circuits a hazard can cause the circuit to change to an incorrect stable state. Example 9.16 illustrates this situation.

Example 9.16

In Example 9.2 we analyzed the circuit that realizes a master-slave D flip-flop. From the excitation table in Figure 9.6*a*, one could attempt to synthesize a minimum-cost circuit that realizes the required functions, Y_m and Y_s. This would give

$$\begin{aligned} Y_m &= CD + \overline{C}y_m \\ &= (C \uparrow D) \uparrow (\overline{C} \uparrow y_m) \\ Y_s &= \overline{C}y_m + Cy_s \\ &= (\overline{C} \uparrow y_m) \uparrow (C \uparrow y_s) \end{aligned}$$

The corresponding circuit is presented in Figure 9.63*a*. At first glance this circuit may seem more attractive than the flip-flops discussed in Chapter 7 because it is less expensive. The problem is that the circuit contains a static hazard.

Figure 9.63*b* shows the Karnaugh maps for the functions Y_m and Y_s. The minimum-cost implementation is based on the prime implicants encircled in black. To see how this circuit is affected by static hazards, assume that presently $Y_s = 1$ and $C = D = 1$. The circuit generates $Y_m = 1$. Now let C change from 1 to 0. For the flip-flop to behave properly, Y_s must remain equal to 1. In Figure 9.63*a*, when C changes to 0, both p and r become 1. Due to the delay through the NOT gate, q may still be 1, causing the circuit to generate $Y_m = Y_s = 0$. The feedback from Y_m will maintain $q = 1$. Hence the circuit remains in an incorrect stable state with $Y_s = 0$.

To avoid the hazards, it is necessary to also include the terms encircled in blue, which gives rise to the expressions

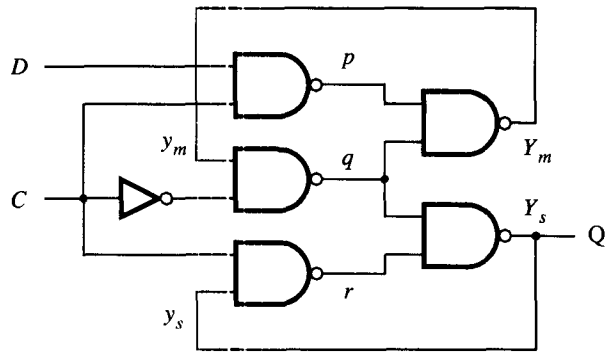
$$\begin{aligned} Y_m &= CD + \overline{C}y_m + Dy_m \\ Y_s &= \overline{C}y_m + Cy_s + y_m y_s \end{aligned}$$

The resulting circuit, implemented with NAND gates, is shown in Figure 9.63*c*.

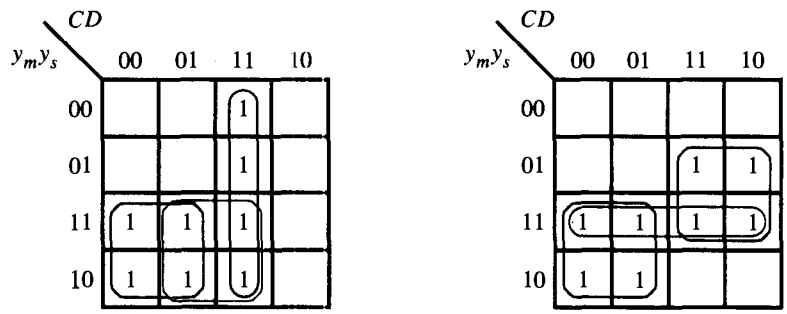
Note that we can obtain another NAND-gate implementation by rewriting the expressions for Y_m and Y_s as

$$\begin{aligned} Y_m &= CD + (\overline{C} + D)y_m \\ &= (C \uparrow D) \uparrow ((\overline{C} + D) \uparrow y_m) \\ &= (C \uparrow D) \uparrow ((C \uparrow \overline{D}) \uparrow y_m) \\ Y_s &= \overline{C}y_m + (C + y_m)y_s \\ &= (\overline{C} \uparrow y_m) \uparrow ((\overline{C} \uparrow \overline{y}_m) \uparrow y_s) \end{aligned}$$

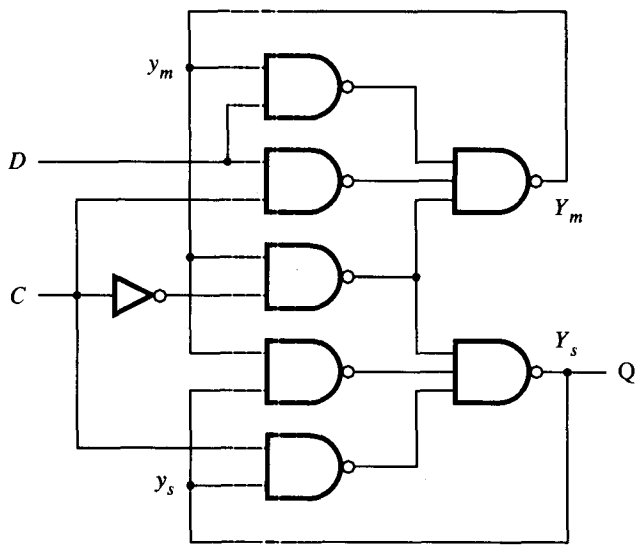
These expressions correspond exactly to the circuit in Figure 7.13.



(a) Minimum-cost circuit



(b) Karnaugh maps for Y_m and Y_s in Figure 9.6*a*



(c) Hazard-free circuit

Figure 9.63 Two-level implementation of master-slave D flip-flop.

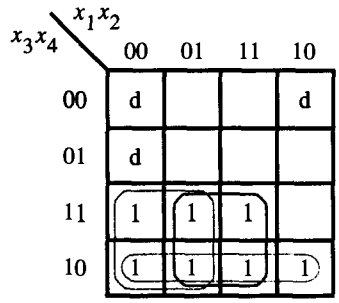


Figure 9.64 Function for Example 9.17.

Example 9.17

From the previous examples, it seems that static hazards can be avoided by including all prime implicants in a sum-of-products circuit that realizes a given function. This is indeed true. But it is not always necessary to include all prime implicants. It is only necessary to include product terms that cover the adjacent pairs of 1s. There is no need to cover the don't-care vertices.

Consider the function in Figure 9.64. A hazard-free circuit that implements this function should include the encircled terms, which gives

$$f = \bar{x}_1 x_3 + x_2 x_3 + x_3 \bar{x}_4$$

The prime implicant $\bar{x}_1 \bar{x}_2$ is not needed to prevent hazards, because it would account only for the two 1s in the left-most column. These 1s are already covered by $\bar{x}_1 x_3$.

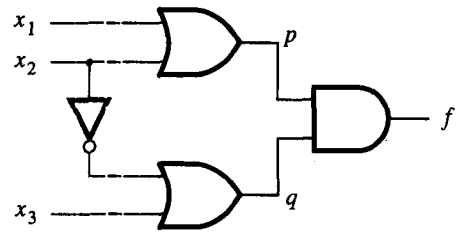
Example 9.18

Static hazards can also occur in other types of circuits. Figure 9.65*a* depicts a product-of-sums circuit that contains a hazard. If $x_1 = x_3 = 0$ and x_2 changes from 0 to 1, then f should remain at 0. However, if the signal at p changes earlier than the signal at q, then p and q will both be equal to 1 for a short time, causing a glitch $0 \rightarrow 1 \rightarrow 0$ on f.

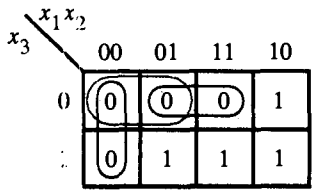
In a POS circuit, it is the transitions between adjacent 0s that may lead to hazards. Thus to design a hazard-free circuit, it is necessary to include sum terms that cover all pairs of adjacent 0s. In this example the term in blue in the Karnaugh map must be included, giving

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

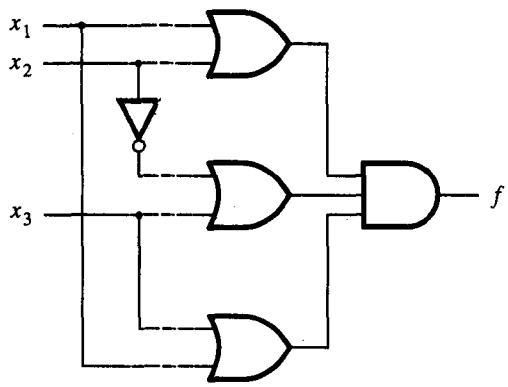
The circuit is shown in Figure 9.65*c*.



(a) Circuit with a hazard



(b) Karnaugh map

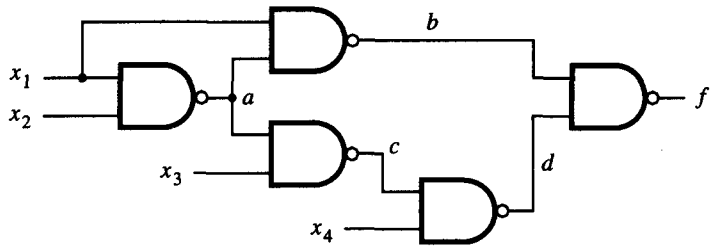


(c) Hazard-free circuit

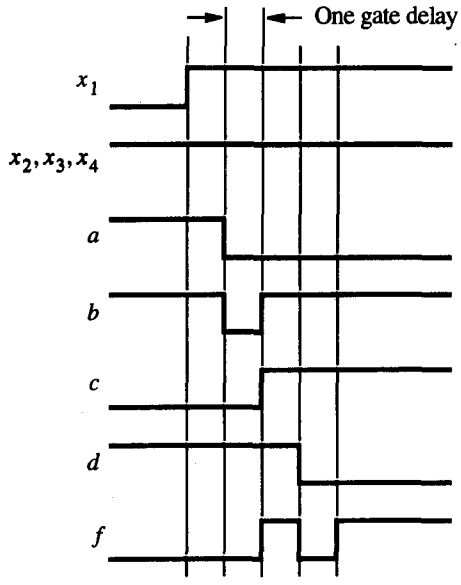
Figure 9.65 Static hazard in a POS circuit.

9.6.2 DYNAMIC HAZARDS

A dynamic hazard causes glitches on $0 \rightarrow 1$ or $1 \rightarrow 0$ transitions of an output signal. An example is given in Figure 9.66. Assuming that all NAND gates have equal delays, a timing diagram can be constructed as shown. The time elapsed between two vertical lines corresponds to a gate delay. The output f exhibits a glitch that should be avoided.



(a) Circuit



(b) Timing diagram

Figure 9.66 Circuit with a dynamic hazard.

It is interesting to consider the function implemented by this circuit, which is

$$f = x_1\bar{x}_2 + \bar{x}_3x_4 + x_1x_4$$

This is the minimum-cost sum-of-products expression for the function. If implemented in this form, the circuit would not have either a static or a dynamic hazard.

A dynamic hazard is caused by the structure of the circuit, where there exist multiple paths for a given signal change to propagate along. If the output signal changes its value three times, $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ in the example, then there must be at least three paths along which a change from a primary input can propagate. A circuit that has a dynamic hazard must also have a static hazard in some part of it. As seen in Figure 9.66*b*, there is a static hazard involving the signal on wire *b*.

Dynamic hazards are encountered in multilevel circuits obtained using factoring or decomposition techniques, which were discussed in Chapter 4. Such hazards are neither easy to detect nor easy to deal with. The designer can avoid dynamic hazards simply by using two-level circuits and ensuring that there are no static hazards.

9.6.3 SIGNIFICANCE OF HAZARDS

A glitch in an asynchronous sequential circuit can cause the circuit to enter an incorrect state and possibly become stable in that state. Therefore, the circuitry that generates the next-state variables must be hazard free. It is sufficient to eliminate hazards due to changes in the value of a single variable because the basic premise in an asynchronous sequential circuit is that the values of both the primary inputs and the state variables must change one at a time.

In combinational circuits, discussed in Chapters 4 through 6, we did not worry about hazards, because the output of a circuit depends solely on the values of the inputs. In synchronous sequential circuits the input signals must be stable within the setup and hold times of flip-flops. It does not matter whether glitches occur outside the setup and hold times with respect to the clock signal.

9.7 A COMPLETE DESIGN EXAMPLE

In the previous sections we examined the various design aspects of asynchronous sequential circuits. In this section we give a complete design example, which covers all necessary steps.

9.7.1 THE VENDING-MACHINE CONTROLLER

The control mechanism of a vending machine is a good vehicle for illustrating a possible application of a digital circuit. We used it in the synchronous environment in Chapter 8. A small example of a vending machine served as an object of analysis in section 9.2. Now we will consider a vending-machine controller similar to the one in Example 8.7 to see how it can be implemented using an asynchronous sequential circuit. The specification for the controller is:

- It accepts nickels and dimes.
- A total of 15 cents is needed to release the candy from the machine.
- No change is given if 20 cents is deposited.

Coins are deposited one at a time. The coin-sensing mechanism generates signals $N = 1$ and $D = 1$ when it sees a nickel or a dime, respectively. It is impossible to have $N = D = 1$ at the same time. Following the insertion of a coin for which the sum equals or exceeds 15 cents, the machine releases the candy and resets to the initial state.

Figure 9.67 shows a state diagram for the required FSM. It is derived using a straightforward approach in which all possible sequences of depositing nickels and dimes are

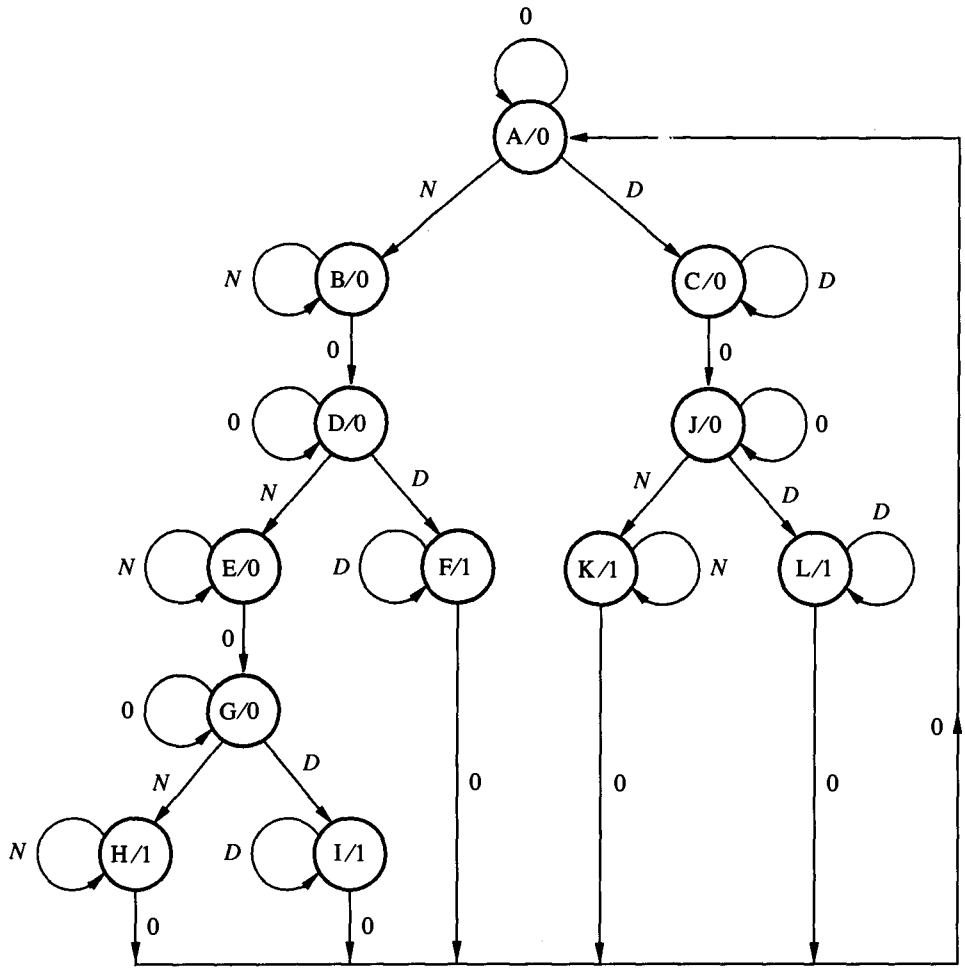


Figure 9.67 Initial state diagram for the vending-machine controller.

enumerated in a treelike structure. To keep the diagram uncluttered, the labels D and N denote the input conditions $DN = 10$ and $DN = 01$, respectively. The condition $DN = 00$ is labeled simply as 0. The candy is released in states F, H, and K, which are reached after 15 cents has been deposited, and in states I and L, upon a deposit of 20 cents.

The corresponding flow table is given in Figure 9.68. It can be reduced using the partitioning procedure as follows

$$P_1 = (ADGJ)(BE)(C)(FIL)(HK)$$
$$P_2 = (A)(D)(GJ)(B)(E)(C)(FIL)(HK)$$
$$P_3 = P_2$$

Using G to represent the equivalent states G and J, F to represent F, I, and L, and H to represent H and K yields a partially reduced flow table in Figure 9.69. The merger diagram

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	J	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	(I)	–	0
H	A	(H)	–	–	1
I	A	–	I	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	A	(L)	–	1

Figure 9.68 Initial flow table for the vending-machine controller.

Present state	Next state				Output
	DN = 00	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	A	–	(F)	–	1
G	(G)	H	F	–	0
H	A	(H)	–	–	1

Figure 9.69 First step in state minimization.

for this table is presented in Figure 9.70. It indicates that states C and E can be merged, as well as F and H. Thus the reduced flow table is obtained as shown in Figure 9.71*a*. The same information is depicted in the form of a state diagram in Figure 9.72.

Next a suitable state assignment must be found. The flow table is relabeled in Figure 9.71*b* to associate a unique number with each stable state. Then the transition diagram

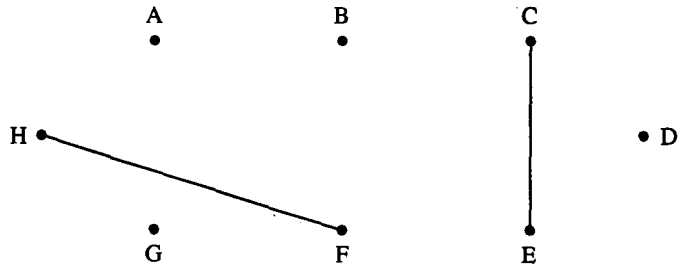


Figure 9.70 Merger diagram for Figure 9.69.

Present state	Next state				Output
	$DN = 00$	01	10	11	z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	G	(C)	(C)	–	0
D	(D)	C	F	–	0
F	A	(F)	(F)	–	1
G	(G)	F	F	–	0

(a) Minimized flow table

Present state	Next state				Output
	$DN = 00$	01	10	11	z
A	(1)	2	4	–	0
B	5	(2)	–	–	0
C	8	(3)	(4)	–	0
D	(5)	3	7	–	0
F	1	(6)	(7)	–	1
G	(8)	6	7	–	0

(b) Relabeled flow table

Figure 9.71 Reduced flow tables.

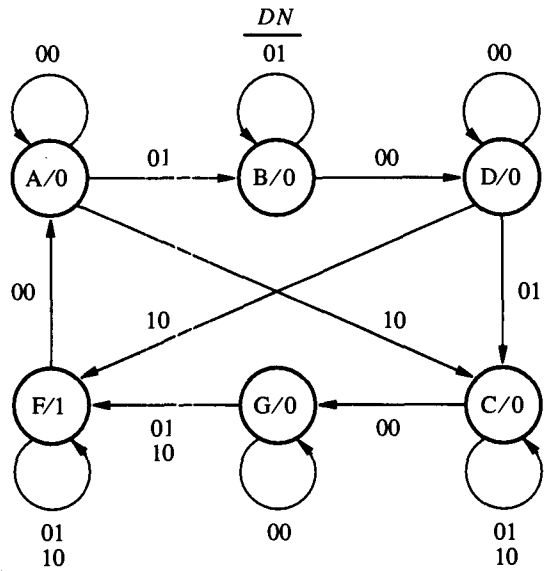


Figure 9.72 State diagram for the vending-machine controller.

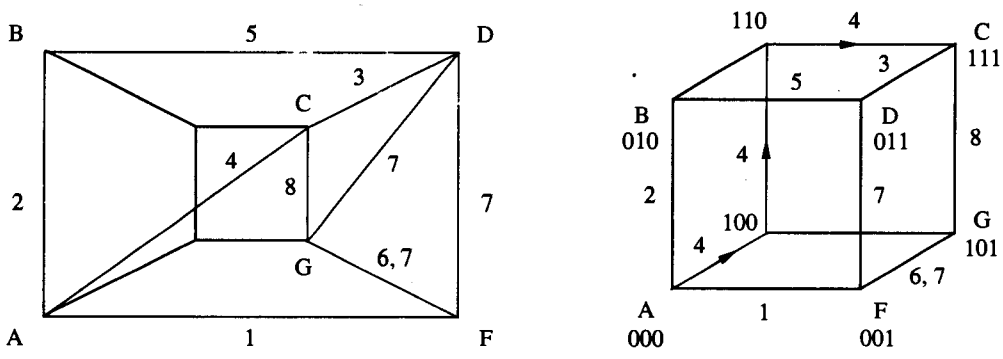
in Figure 9.73*a* is obtained. Since we wish to try to embed the diagram onto a three-dimensional cube, eight vertices are shown in the figure. The diagram shows two diagonal transitions. The transition between *D* and *G* (label 7) does not matter, because it is only an alternative path. The transition from *A* to *C* (label 4) is required, and it can be realized via unused states as indicated in blue in Figure 9.73*b*. Therefore, the transition diagram can be embedded onto a three-dimensional cube as shown. Using the state assignment from this figure, the excitation table in Figure 9.74 is derived.

The Karnaugh maps for the next-state functions are given in Figure 9.75. From these maps the following hazard-free expressions are obtained

$$Y_1 = \overline{N}y_2 + Ny_1 + Dy_1 + y_1y_3 + y_1y_2$$
$$Y_2 = N\overline{y}_1 + Ny_2 + \overline{y}_1y_3 + \overline{D}y_2\overline{y}_3 + Dy_2y_3$$
$$Y_3 = D\overline{y}_1 + y_2y_3 + Ny_1y_2 + \overline{D}y_3\overline{N}$$

All product terms in these expressions are needed for a minimum-cost POS implementation except for y_1y_2, which is included to prevent hazards in the expression for Y_1. The output expression is

$$z = y_1\overline{y}_2\overline{y}_3$$



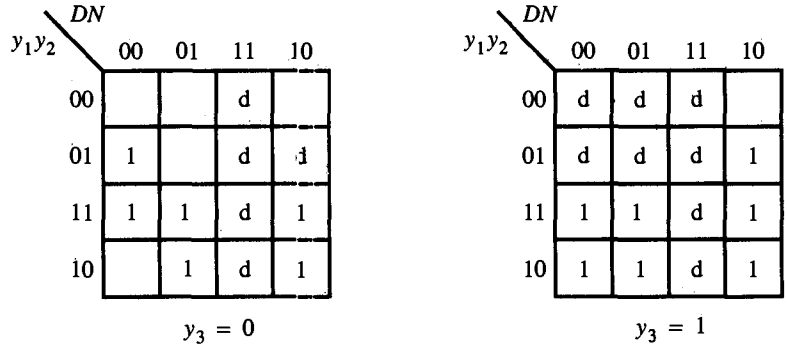
(a) Transition diagram

(b) Embedded on the cube

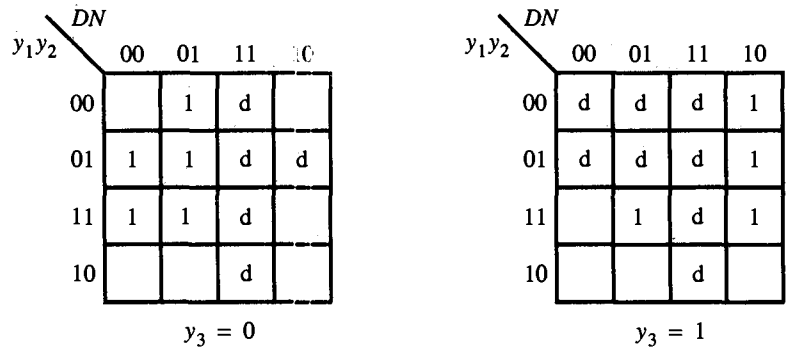
Figure 9.73 Determination of the state assignment.

	Present state $y_3y_2y_1$	Next state $DN = 00$	01	10	11	Output z
		$Y_3Y_2Y_1$				
A	000	(000)	010	100	–	0
B	010	011	(010)	–	–	0
C	111	101	(111)	(111)	–	0
D	011	(011)	111	001	–	0
F	001	000	(001)	(001)	–	1
G	101	(101)	001	001	–	0
	100	–	–	110	–	0
	110	–	–	111	–	0

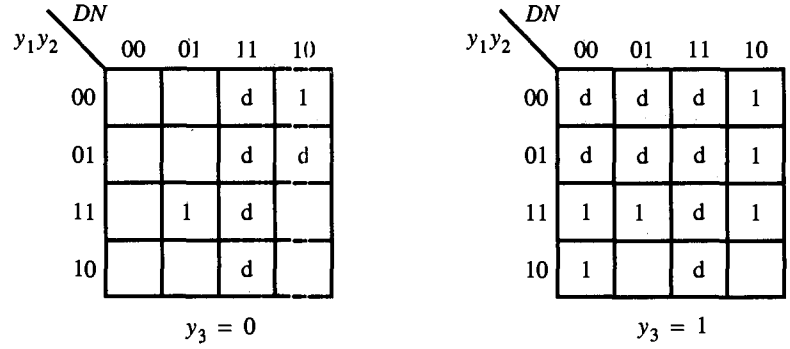
Figure 9.74 Excitation table based on the state assignment in Figure 9.73*b*.



(a) Map for Y_1



(b) Map for Y_2



(c) Map for Y_3

Figure 9.75 Karnaugh maps for the functions in Figure 9.74.

9.8 CONCLUDING REMARKS

Asynchronous sequential circuits are more difficult to design than the synchronous sequential circuits. The difficulties with race conditions present a problem that must be handled carefully. At the present time there is little CAD support for designing asynchronous circuits. For these reasons, most designers resort to synchronous sequential circuits in practical applications.

An important advantage of asynchronous circuits is their speed of operation. Since there is no clock involved, the speed of operation depends only on the propagation delays in the circuit. In an asynchronous system that comprises several circuits, some circuits may operate faster than others, thus potentially improving the overall performance of the system. In contrast, in synchronous systems the clock period has to be long enough to accommodate the slowest circuit, and it has a large effect on the performance.

Asynchronous circuit techniques are also useful in designing systems that consist of two or more synchronous circuits that operate under the control of different clocks. The signals exchanged between such circuits often appear to be asynchronous in nature.

From the reader's point of view, it is useful to view asynchronous circuits as an excellent vehicle for gaining a deeper understanding of the operation of digital circuits in general. These circuits illustrate the consequences of propagation delays and race conditions that may be inherent in the structure of a circuit. They also illustrate the concept of stability, demonstrated through the existence of stable and unstable states. For further discussion of asynchronous sequential circuits, the reader may consult references [1–6].

PROBLEMS

9.1 Derive a flow table that describes the behavior of the circuit in Figure P9.1. Compare your solution with the tables in Figure 9.21. Is there any similarity?

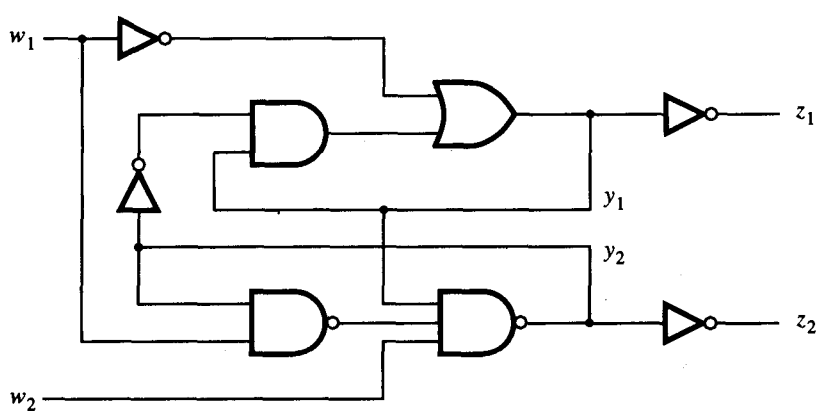


Figure P9.1 Circuit for problem 9.1.

9.2 Consider the circuit in Figure P9.2. Draw the waveforms for the signals C, z_1, and z_2. Assume that C is a square-wave clock signal and that each gate has a propagation delay Δ. Express the behavior of the circuit in the form of a flow table that would produce the desired signals. (Hint: use the Mealy model.)

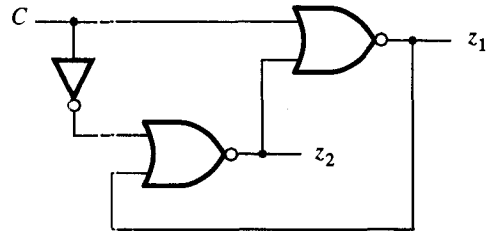


Figure P9.2 Circuit for problem 9.2.

9.3 Derive the minimal flow table that specifies the same functional behavior as the flow table in Figure P9.3.

9.4 Derive the minimal Moore-type flow table that specifies the same functional behavior as the flow table in Figure P9.4.

9.5 Find a suitable state assignment using as few states as possible and derive the next-state and output expressions for the flow table in Figure 9.42.

9.6 Find a suitable state assignment for the flow table in Figure 9.42, using pairs of equivalent states, as explained in Example 9.15. Derive the next-state and output expressions.

9.7 Find a state assignment for the flow table in Figure 9.42, using one-hot encoding. Derive the next-state and output expressions.

9.8 Implement the FSM specified in Figure 9.39, using the merger diagram in Figure 9.40*a*.

9.9 Find a suitable state assignment for the FSM defined by the flow table in Figure P9.5. Derive the next-state and output expressions for the FSM using this state assignment.

9.10 Find a hazard-free minimum-cost implementation of the function

$$f(x_1, \cdots, x_4) = \sum m(0, 4, 11, 13, 15) + D(2, 3, 5, 10)$$

9.11 Repeat problem 9.10 for the function

$$f(x_1, \cdots, x_5) = \sum m(0, 4, 5, 24, 25, 29) + D(8, 13, 16, 21)$$

9.12 Find a hazard-free minimum-cost POS implementation of the function

$$f(x_1, \cdots, x_4) = \Pi M(0, 2, 3, 7, 10) + D(5, 13, 15)$$

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z
A	(A)	B	C	–	0
B	D	(B)	–	–	0
C	P	–	(C)	–	0
D	(D)	E	F	–	0
E	G	(E)	–	–	0
F	M	–	(F)	–	0
G	(G)	H	I	–	0
H	J	(H)	–	–	0
I	A	–	(I)	–	1
J	(J)	K	L	–	0
K	A	(K)	–	–	1
L	A	–	(L)	–	1
M	(M)	N	O	–	0
N	A	(N)	–	–	1
O	A	–	(O)	–	1
P	(P)	R	S	–	0
R	T	(R)	–	–	0
S	A	–	(S)	–	1
T	(T)	U	V	–	0
U	A	(U)	–	–	1
V	A	–	(V)	–	1

Figure P9.3 Flow table for problem 9.3.

9.13 Repeat problem 9.12 for the function

$$f(x_1, \cdots, x_5) = \Pi M\,(2, 6, 7, 25, 28, 29) + D(0, 8, 9, 10, 11, 21, 24, 26, 27, 30)$$

9.14 Consider the circuit in Figure P9.6. Does this circuit exhibit any hazards?

9.15 Design an original circuit that exhibits a dynamic hazard.

9.16 A control mechanism for a vending machine accepts nickels and dimes. It dispenses merchandise when 20 cents is deposited; it does not give change if 25 cents is deposited.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	–	0
B	K	(B)	–	H	0
C	F	–	(C)	M	0
D	(D)	E	J	–	1
E	A	(E)	–	M	0
F	(F)	L	J	–	0
G	D	(G)	–	H	0
H	–	G	J	(H)	1
J	F	–	(J)	H	0
K	(K)	L	C	–	1
L	A	(L)	–	H	0
M	–	G	C	(M)	1

Figure P9.4 Flow table for problem 9.4.

Present state	Next state w_2w_1 = 00	01	10	11	Output z
A	(A)	B	C	–	0
B	D	(B)	–	G	0
C	F	–	(C)	G	0
D	(D)	E	C	–	1
E	A	(E)	–	G	0
F	(F)	E	C	–	0
G	–	B	C	(G)	1

Figure P9.5 Flow table for problem 9.9.

Design the FSM that implements the required control, using as few states as possible. Find a suitable state assignment and derive the next-state and output expressions.

9.17 Design an asychronous circuit that meets the following specifications. The circuit has two inputs: a clock input c and a control input w. The output, z, replicates the clock pulses when

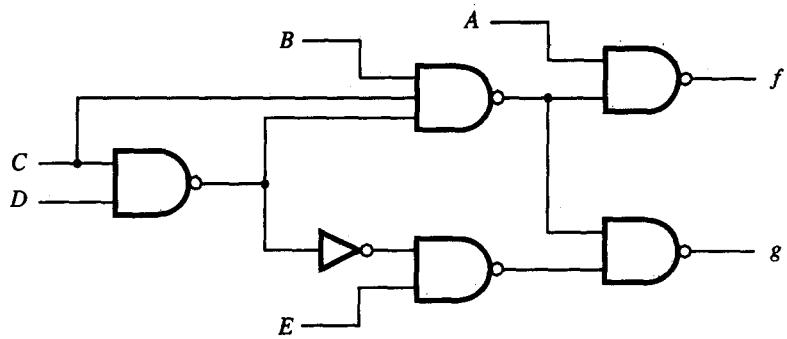


Figure P9.6 Circuit for problem 9.14.

$w = 1$; otherwise, $z = 0$. The pulses appearing on z must be full pulses. Consequently, if $c = 1$ when w changes from 0 to 1, then the circuit will not produce a partial pulse on z, but will wait until the next clock pulse to generate $z = 1$. If $c = 1$ when w changes from 1 to 0, then a full pulse must be generated; that is, $z = 1$ as long as $c = 1$. Figure P9.7 illustrates the desired operation.

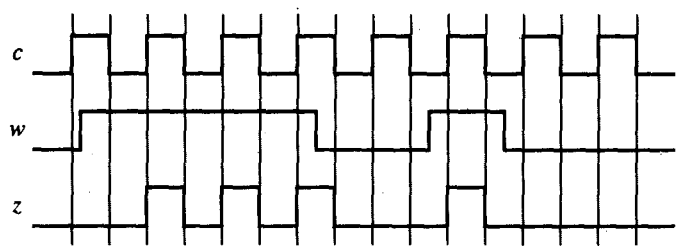


Figure P9.7 Waveforms for problem 9.17.

9.18 Repeat problem 9.17 but with the following change in the specification. While $w = 1$, the output z should have only one pulse; if several pulses occur on c, only the first one should be reproduced on z.

9.19 Example 9.6 describes a simple arbiter for two devices contending for a shared resource. Design a similar arbiter for three devices that use a shared resource. In case of simultaneous requests, namely, if one device has been granted access to the shared resource and before it releases its request the other two devices make requests of their own, let the priority of the devices be Device 1 > Device 2 > Device 3.

9.20 In the discussion of Example 9.6, we mentioned a possible use of the mutual exclusion element (ME) to prevent both request inputs to the FSM being equal to 1 at the same time. Design an arbiter circuit for this case.

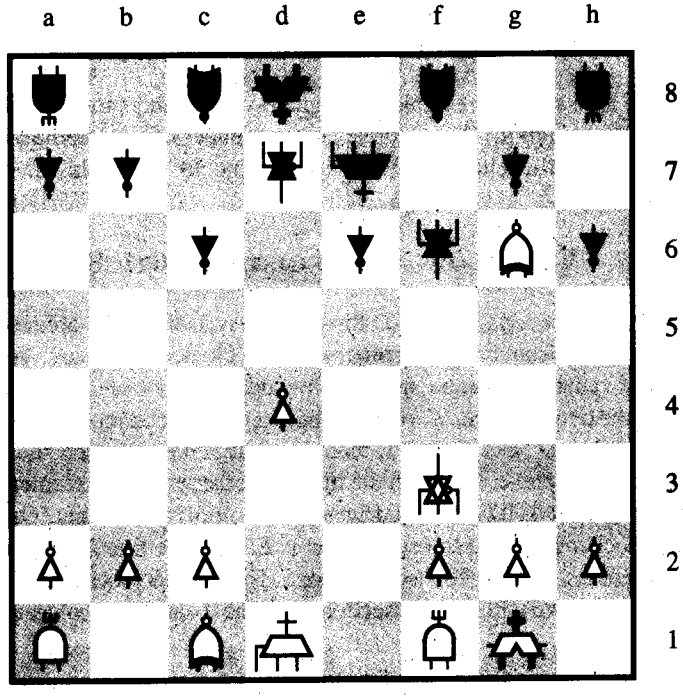
REFERENCES

1. K. J. Breeding, *Digital Design Fundamentals*, (Prentice-Hall: Englewood Cliffs, NJ, 1989).
2. F. J. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., (Wiley: New York, 1993).
3. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*, (Prentice-Hall: Englewood Cliffs, NJ, 1995).
4. N. L. Pappas, *Digital Design*, (West: St. Paul, MN, 1994).
5. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed., (West: St. Paul, MN, 1993).
6. C. J. Myers, *Asynchronous Circuit Design*, (Wiley: New York, 2001).

chapter

10

DIGITAL SYSTEM DESIGN



10. Bd3–g6+, Ke8–d8

In the previous chapters we showed how to design many types of simple circuits, such as multiplexers, decoders, flip-flops, registers, and counters, which can be used as building blocks. In this chapter we provide examples of more complex circuits that can be constructed using the building blocks as subcircuits. Such larger circuits form a *digital system*. For practical reasons our examples of digital systems will not be large, but the design techniques presented are applicable to systems of any size. After presenting several examples, we will discuss some practical issues, such as how to ensure reliable clocking of flip-flops in individual and multiple chips, how to deal with input signals that are not synchronized to the clock signal, and the like.

A digital system consists of two main parts, called the datapath circuit and the control circuit. The *datapath circuit* is used to store and manipulate data and to transfer data from one part of the system to another. Datapath circuits comprise building blocks such as registers, shift registers, counters, multiplexers, decoders, adders, and so on. The *control circuit* controls the operation of the datapath circuit. In Chapter 8 we referred to the control circuits as finite state machines.

10.1 BUILDING BLOCK CIRCUITS

We will give several examples of digital systems and show how to design their datapath and control circuits. The examples use a number of the building block circuits that were presented in earlier chapters. Some building blocks used in this chapter are described below.

10.1.1 FLIP-FLOPS AND REGISTERS WITH ENABLE INPUTS

In many applications that use D flip-flops, it is useful to be able to prevent the data stored in the flip-flop from changing when an active clock edge occurs. We showed in Figure 7.61 how this capability can be provided by adding a multiplexer to the flip-flop. Figure 10.1 depicts the circuit. When $E = 0$, the flip-flop output cannot change, because the multiplexer connects Q to D. But if $E = 1$, then the multiplexer connects the R input to D. Instead of using the multiplexer shown in the figure, another way to implement the enable feature is to use a two-input AND gate that drives the flip-flop's clock input. One input to the AND gate is the clock signal, and the other input is E. Then setting $E = 0$ prevents the clock signal from reaching the flip-flop's clock input. This method seems simpler than the multiplexer approach, but we will show in section 10.3 that it can cause problems in practical operation.

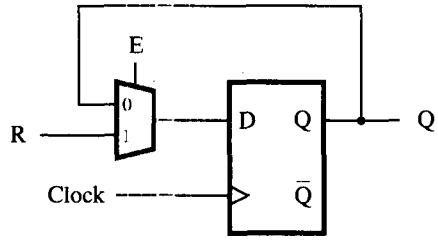


Figure 10.1 A flip-flop with an enable input.

We will prefer the multiplexer-based approach over gating the clock with an AND gate in this chapter.

Verilog code for a D flip-flop with an asynchronous reset input and an enable input is given in Figure 10.2. We can extend the enable capability to registers with n bits by using n 2-to-1 multiplexers controlled by E. The multiplexer for each flip-flop, i, selects either the external data bit, R_i, or the flip-flop's output, Q_i. Verilog code for an n-bit register with an asynchronous reset input and an enable input is given in Figure 10.3.

10.1.2 SHIFT REGISTERS WITH ENABLE INPUTS

It is useful to be able to inhibit the shifting operation in a shift register by using an enable input, E. We showed in Figure 7.19 that shift registers can be constructed with a parallel-load capability, which is implemented using a multiplexer. Figure 10.4 shows how

```
module rege (R, Clock, Resetn, E, Q);
   input R, Clock, Resetn, E;
   output Q;
   reg Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else if (E)
         Q <= R;

endmodule
```

Figure 10.2 Code for a D flip-flop with enable.

```
module regne (R, Clock, Resetn, E, Q);
   parameter n = 8;
   input [n-1:0] R;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else if (E)
         Q <= R;

endmodule
```

Figure 10.3 An n-bit register with an enable input.

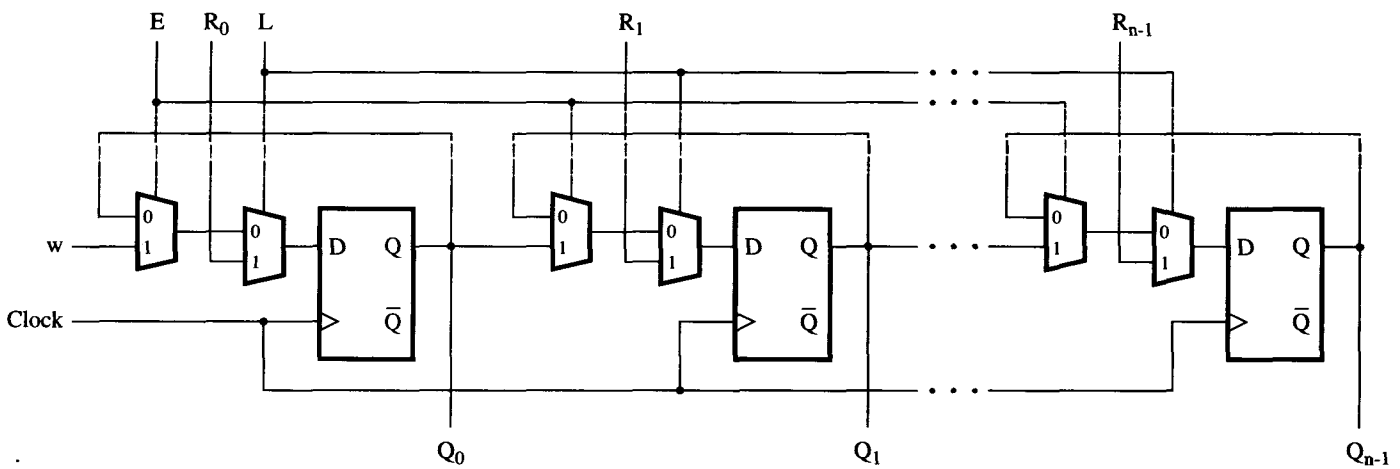


Figure 10.4 A shift register with parallel-load and enable control inputs.

the enable feature can be added by using an additional multiplexer. If the parallel-load control input, L, is 1, the flip-flops are loaded in parallel. But if $L = 0$, the additional multiplexer selects new data to be loaded into the flip-flops only if the enable E is 1.

Verilog code that represents the circuit in Figure 10.4 is given in Figure 10.5. When $L = 1$, the register is loaded in parallel from the R input. When $L = 0$ and $E = 1$, the data in the shift register is shifted in a right-to-left direction.

10.1.3 STATIC RANDOM ACCESS MEMORY (SRAM)

We have introduced several types of circuits that can be used to store data. Assume that we need to store a large number, m, of data items, each of which consists of n bits. One possibility is to use an n-bit register for each data item. We would need to design circuitry to control access to each register, both for loading (writing) data into it and for reading data out.

When m is large, it is awkward to use individual registers to store the data. A better approach is to make use of a *static random access memory* (SRAM) block. An SRAM block is a two-dimensional array of SRAM cells, where each cell can store one bit of information.

```
module shiftlne (R, L, E, w, Clock, Q);
   parameter n = 4;
   input [n−1:0] R;
   input L, E, w, Clock;
   output [n−1:0] Q;
   reg [n−1:0] Q;
   integer k;

   always @(posedge Clock)
   begin
      if (L)
         Q <= R;
      else if (E)
      begin
         Q[0] <= w;
         for (k = 1; k < n; k = k+1)
            Q[k] <= Q[k−1];
      end
   end

endmodule
```

Figure 10.5 A right-to-left shift register with an enable input.

If we need to store m items with n bits each, we can use an array of $m \times n$ SRAM cells. The dimensions of the SRAM array are called its *aspect ratio*.

An SRAM cell is similar to the storage cell that was shown in Figure 7.3. Since an SRAM block may contain a large number of SRAM cells, each cell must take as little space on an integrated circuit chip as possible. For this reason, the storage cell should use as few transistors as possible. One popular storage cell used in practice is depicted in Figure 10.6. It operates as follows. To store data into the cell, the *Sel* input is set to 1, and the data value to be stored is placed on the *Data* input. The SRAM cell may include a separate input for the complement of the data, indicated by the transistor shown in blue in the figure. For simplicity we assume that this transistor is not included in the cell. After waiting long enough for the data to propagate through the feedback path formed by the two NOT gates, *Sel* is changed to 0. The stored data then remains in the feedback loop indefinitely. A possible problem is that when $Sel = 1$, the value of *Data* may not be the same as the value being driven by the small NOT gate in the feedback path. Hence the transistor controlled by *Sel* may attempt to drive the stored data to one logic value while the output of the small NOT gate has the opposite logic value. To resolve this problem, the NOT gate in the feedback path is built using small (weak) transistors, so that its output can be overridden with new data.

To read data stored in the cell, we simply set *Sel* to 1. In this case the *Data* node would not be driven to any value by external circuitry, so that the SRAM cell can place the stored data on this node. The *Data* signal is passed through a buffer, not shown in the figure, and provided as an output of the SRAM block.

An SRAM block contains an array of SRAM cells. Figure 10.7 shows an array with two rows of two cells each. In each column of the array, the *Data* nodes of the cells are connected together. Each row, i, has a separate select input, Sel_i, that is used to read or write the contents of the cells in that row. Larger arrays are formed by connecting more cells to Sel_i in each row and by adding more rows. The SRAM block must also contain circuitry that controls access to each row in the array. Figure 10.8 depicts a $2^m \times n$ array of the type in Figure 10.7, which has a decoder that drives the *Sel* inputs in each row of the array. The inputs to the decoder are called *Address* inputs. This term derives from the notion that the

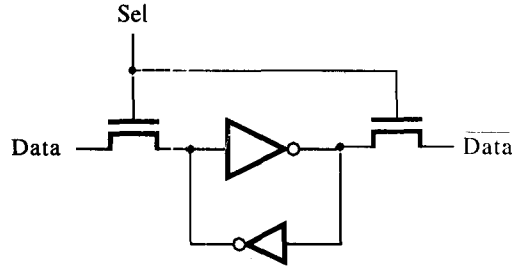


Figure 10.6 An SRAM cell.

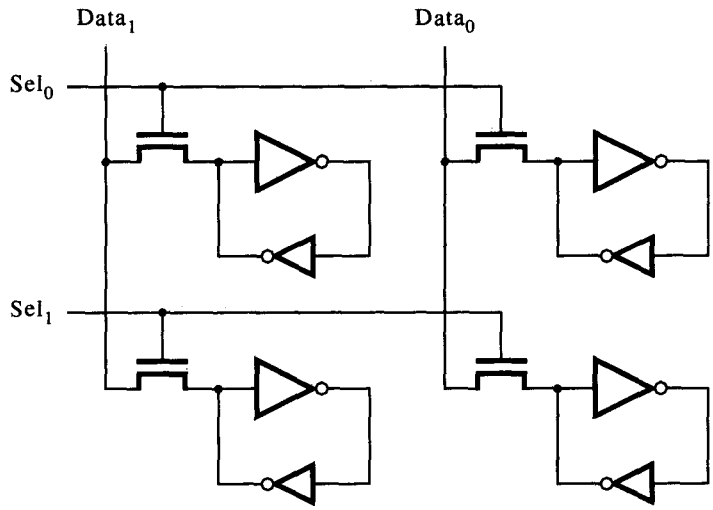


Figure 10.7 A 2 × 2 array of SRAM cells.

location of a row in the array can be thought of as the "address" of the row. The decoder has *m Address* inputs and produces 2^m select outputs. If the *Write* control input is 1, then the data bits on the inputs $d_{n-1}, \ldots, d_0$ are stored in the cells of the row selected by the *Address* inputs. If the *Read* control input is 1, then the data stored in the row selected by the *Address* inputs appears on the outputs $q_{n-1}, \ldots, q_0$. In many practical applications the data inputs and data outputs are connected together. Thus the *Write* and *Read* inputs must never have the value 1 at the same time.

The design of memory blocks has been the subject of intensive research and development. We have described only the basic operation of one type of memory block. The reader can refer to books on computer organization for more information [1, 2].

10.1.4 SRAM BLOCKS IN PLDS

Some PLDs contain SRAM blocks that can be used as part of circuits implemented in the chips. One popular chip has a number of SRAM blocks, each of which contains 2048 SRAM cells. The SRAM blocks can be configured to provide different aspect ratios, depending on the needs of the design being implemented. Aspect ratios from 256 × 8 to 2048 × 1 can be realized using a single SRAM block, and multiple blocks can be combined to form larger memory arrays. To include SRAM blocks in a circuit, designers use prebuilt modules that are provided in a library as part of the CAD tools.

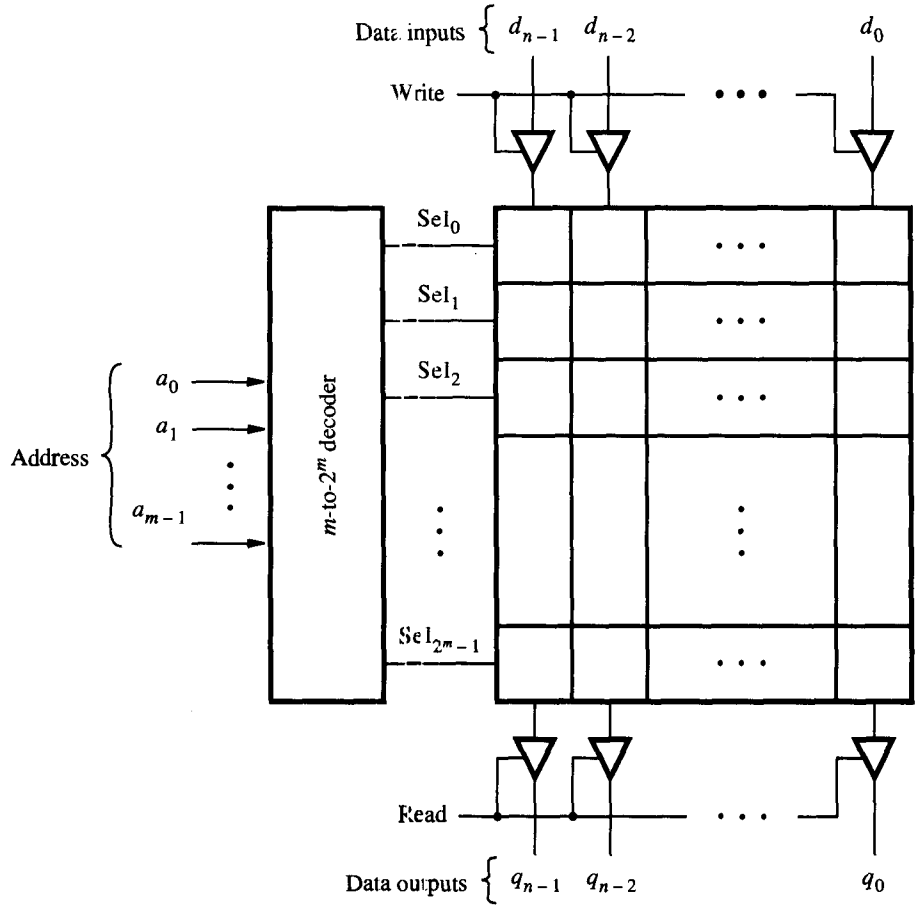


Figure 10.8 A $2^m \times n$ SRAM block.

10.2 DESIGN EXAMPLES

We introduced algorithmic state machine (ASM) charts in section 8.10 and showed how they can be used to describe finite state machines. ASM charts can also be used to describe digital systems that include both datapath and control circuits. We will illustrate how the ASM charts can be used as an aid in designing digital systems by giving several examples.

10.2.1 A BIT-COUNTING CIRCUIT

Suppose that we wish to count the number of bits in a register, A, that have the value 1. Figure 10.9 shows pseudo-code for a step-by-step procedure, or *algorithm*, that can be

```
B = 0 ;
while A ≠ 0 do
    if a0 = 1 then
        B = B + 1 ;
    end if ;
    Right-shift A ;
end while ;
```

Figure 10.9 Pseudo-code for the bit counter.

used to perform the required task. It assumes that A is stored in a register that can shift its contents in the left-to-right direction. The answer produced by the algorithm is stored in the variable named B. The algorithm terminates when A does not contain any more 1s, that is when $A = 0$. In each iteration of the while loop, if the least-significant bit (LSB) of A is 1, then B is incremented by 1; otherwise, B is not changed. A is shifted one bit to the right at the end of each loop iteration.

Figure 10.10 gives an ASM chart that represents the algorithm in Figure 10.9. The state box for the starting state, $S1$, specifies that B is initialized to 0. We assume that an input signal, s, exists, which is used to indicate when the data to be processed has been loaded into A, so that the machine can start. The decision box labeled s stipulates that the machine remains in state $S1$ as long as $s = 0$. The conditional output box with *Load A* written inside it indicates that A is loaded from external data inputs if $s = 0$ in state $S1$.

When s becomes 1, the machine changes to state $S2$. The decision box below the state box for $S2$ checks whether $A = 0$. If so, the bit-counting operation is complete; hence the machine should change to state $S3$. If not, the FSM remains in state $S2$. The decision box at the bottom of the chart checks the value of a_0. If $a_0 = 1$, B is incremented, which is indicated in the chart as $B \leftarrow B + 1$. If $a_0 = 0$, then B is not changed. In state $S3$, B contains the result, which is the number of bits in A that were 1. An output signal, *Done*, is set to 1 to indicate that the algorithm is finished; the FSM stays in $S3$ until s goes back to 0.

10.2.2 ASM CHART IMPLIED TIMING INFORMATION

In section 8.10 we said that ASM charts are similar to traditional flowcharts, except that the ASM chart implies timing information. We can use the bit-counting example to illustrate this concept. Consider the ASM block for state $S2$, which is shaded in blue in Figure 10.10. In a traditional flowchart, when state $S2$ is entered, the value of A would first be shifted to the right. Then we would examine the value of A and if A's LSB is 1, we would immediately add 1 to B. But, since the ASM chart represents a sequential circuit, changes in A and B, which represent the outputs of flip-flops, take place after the active clock edge. The same clock signal that controls changes in the state of the machine also controls changes in A and B. Hence in state $S2$, the decision box that tests whether $A = 0$, as well as the box that checks the value of a_0, check the bits in A before they are shifted. If $A = 0$, then the FSM will change to state $S3$ on the next clock edge (this clock edge also shifts A, which has no effect because A is already 0 in this case.) On the other hand, if $A \neq 0$, then the

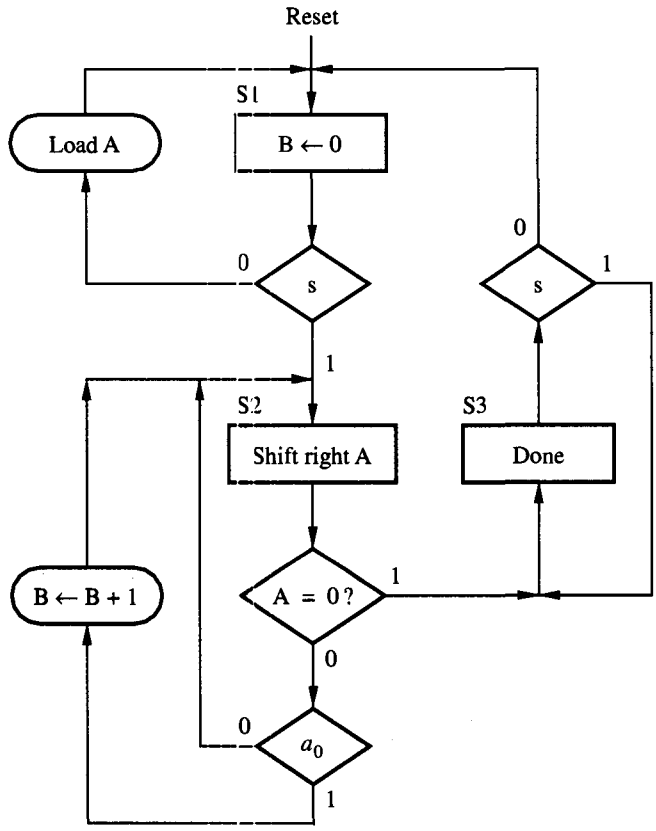


Figure 10.10 ASM chart for the pseudo-code in Figure 10.9.

FSM does not change to $S3$, but remains in $S2$. At the same time, A is still shifted, and B is incremented if a_0 has the value 1. These timing issues are illustrated in Figure 10.14, which represents a simulation result for a circuit that implements the ASM chart. We show how the circuit is designed in the following discussion.

Datapath Circuit

By examining the ASM chart for the bit-counting circuit, we can infer the type of circuit elements needed to implement its datapath. We need a shift register that shifts left-to-right to implement A. It must have the parallel-load capability because of the conditional output box in state $S1$ that loads data into the register. An enable input is also required because shifting should occur only in state $S2$. A counter is needed for B, and it needs a parallel-load capability to initialize the count to 0 in state $S1$. It is not wise to rely on the counter's reset input to clear B to 0 in state $S1$. In practice, the reset signal is used in a digital system for only two purposes: to initialize the circuit when power is first applied, or to recover from

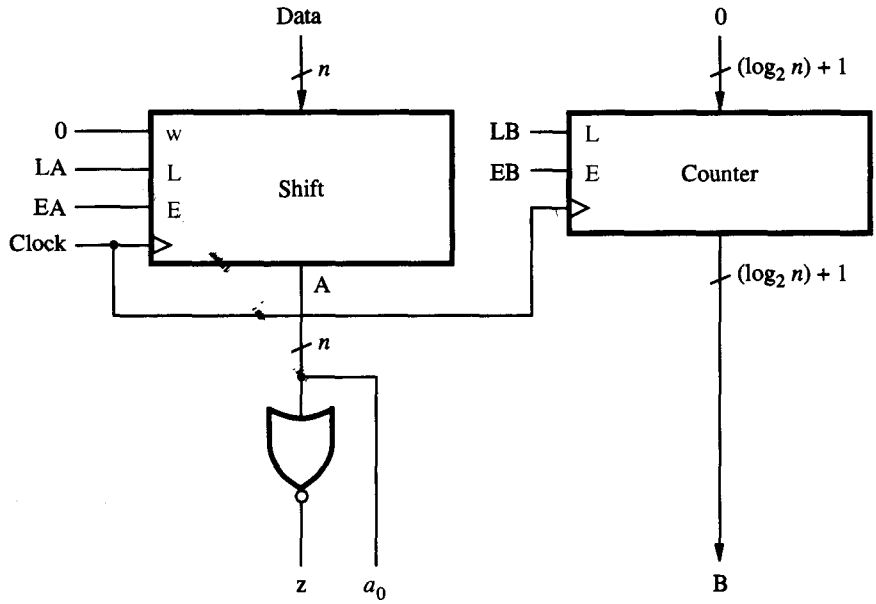


Figure 10.11 Datapath for the ASM chart in Figure 10.10.

an error. The machine changes from state $S3$ to $S1$ as a result of $s = 0$; hence we should not assume that the reset signal is used to clear the counter.

The datapath circuit is depicted in Figure 10.11. The serial input to the shift register, w, is connected to 0, because it is not needed. The load and enable inputs on the shift register are driven by the signals *LA* and *EA*. The parallel input to the shift register is named *Data*, and its parallel output is A. An n-input NOR gate is used to test whether $A = 0$. The output of this gate, z, is 1 when $A = 0$. Note that the figure indicates the n-input NOR gate by showing a single input connection to the gate, with the label n attached to it. The counter has $log_2(n)$ bits, with parallel inputs connected to 0 and parallel outputs named B. It also has a parallel load input *LB* and enable input *EB* control signals.

Control Circuit

For convenience we can draw a second ASM chart that represents only the FSM needed for the control circuit, as shown in Figure 10.12. The FSM has the inputs s, a_0, and z and generates the outputs *EA*, *LB*, *EB*, and *Done*. In state $S1$, *LB* is asserted, so that 0 is loaded in parallel into the counter. Note that for the control signals, like *LB*, instead of writing *LB* = 1, we simply write *LB* to indicate that the signal is asserted. We assume that external circuitry drives *LA* to 1 when valid data is present at the parallel inputs of the shift register, so that the shift register contents are initialized before s changes to 1. In state $S2$, *EA* is asserted to cause a shift operation, and the count enable for B is asserted only if $a_0 = 1$.

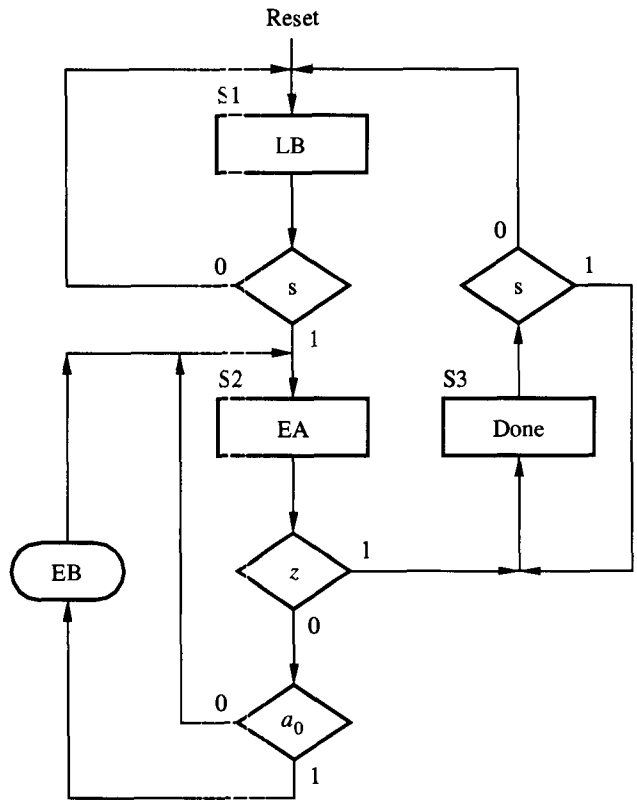


Figure 10.12 ASM chart for the bit counter datapath circuit.

Verilog Code

The bit-counting circuit can be described in Verilog code as shown in Figure 10.13. We have chosen to define A as an eight-bit vector and B as a 4-bit vector signal. The ASM chart in Figure 10.12 can be directly translated into code that describes the required control circuit. The signal y is used to represent the present state of the FSM, and Y represents the next state. The FSM is described with three **always** blocks: the block labeled *State_table* specifies the state transitions, the block labeled *State_flipflops* represents the state flip-flops, and the block labeled *FSM_outputs* specifies the generated outputs in each state. A default value is specified at the beginning of the *FSM_outputs* block for each output signal, and then individual output values are specified in the **case** statement.

The fourth **always** block defines the up-counter that implements B. The shift register for A is instantiated at the end of the code, and the z signal is defined using the reduction NOR operator. We implemented the code in Figure 10.13 in a chip and performed a timing simulation. Figure 10.14 gives the results of the simulation for $A = 00111011$. After the circuit is reset, the input signal LA is set to 1, and the desired data, $(3B)_{16}$, is placed on the

```
module bitcount (Clock, Resetn, LA, s, Data, B, Done);
   input Clock, Resetn, LA, s;
   input [7:0] Data;
   output [3:0] B;
   output Done;
   wire [7:0] A;
   wire z;
   reg [1:0] Y, y;
   reg [3:0] B;
   reg Done, EA, EB, LB;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

   always @(s or y or z)
   begin: State_table
      case (y)
         S1: if (!s) Y = S1;
             else Y = S2;
         S2: if (z == 0) Y = S2;
             else Y = S3;
         S3: if (s) Y = S3;
             else Y = S1;
         default: Y = 2'bxx;
      endcase
   end

   always @(posedge Clock or negedge Resetn)
   begin: State_flipflops
      if (Resetn == 0)
         y <= S1;
      else
         y <= Y;
   end
```

... continued in Part *b*.

Figure 10.13 Verilog code for the bit-counting circuit (Part *a*).

Data input. When *s* changes to 1, the next active clock edge causes the FSM to change to state *S*2. In this state, each active clock edge increments *B* if a_0 is 1, and shifts *A*. When $A = 0$, the next clock edge causes the FSM to change to state *S*3, where *Done* is set to 1 and *B* has the correct result, $B = 5$. To check more thoroughly that the circuit is designed correctly, we should try different values of input data.

```
    always @(y or A[0])
    begin: FSM_outputs
        // defaults
        EA = 0; LB = 0; EB = 0; Done = 0;
        case (y)
            S1: LB = 1;
            S2: begin
                    EA = 1;
                    if (A[0]) EB = 1;
                    else EB = 0;
                end
            S3: Done = 1;
        endcase
    end

// datapath circuit

    // counter B
    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            B <= 0;
        else if (LB)
            B <= 0;
        else if (EB)
            B <= B + 1;

    shiftrne ShiftA (Data, LA, EA, 0, Clock, A);
    assign z = ~| A;

endmodule
```

Figure 10.13 Verilog code for the bit-counting circuit (Part *b*).

10.2.3 SHIFT-AND-ADD MULTIPLIER

We presented a circuit that multiplies two unsigned n-bit binary numbers in Figure 5.36. The circuit uses a two-dimensional array of identical subcircuits, each of which contains a full-adder and an AND gate. For large values of n, this approach may not be appropriate because of the large number of gates needed. Another approach is to use a shift register in combination with an adder to implement the traditional method of multiplication that is done by "hand." Figure 10.15*a* illustrates the manual process of multiplying two binary numbers. The product is formed by a series of addition operations. For each bit i in the multiplier that is 1, we add to the product the value of the multiplicand shifted to the left i times. This algorithm can be described in pseudo-code as shown in Figure 10.15*b*, where A is the multiplicand, B is the multiplier, and P is the product.

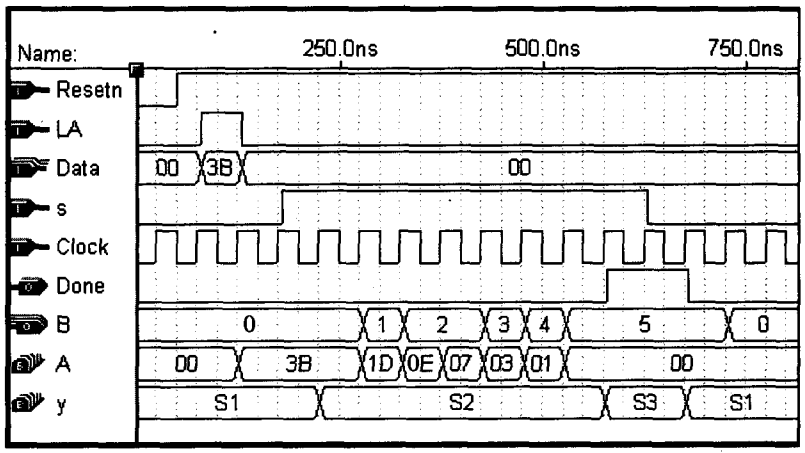


Figure 10.14 Simulation results for the bit-counting circuit.

Decimal		Binary	
13		1 1 0 1	Multiplicand
× 11		× 1 0 1 1	Multiplier
13		1 1 0 1	
13		1 1 0 1	
143		0 0 0 0	
		1 1 0 1	
		1 0 0 0 1 1 1 1	Product

(a) Manual method

$P = 0$;
for $i = 0$ to $n - 1$ do
    if $b_i = 1$ then
        $P = P + A$;
    end if;
    Left-shift A;
end for;

(b) Pseudo-code

Figure 10.15 An algorithm for multiplication.

An ASM chart that represents the algorithm in Figure 10.15*b* is given in Figure 10.16. We assume that an input s is used to control when the machine begins the multiplication process. As long as s is 0, the machine stays in state $S1$ and the data for A and B can be loaded from external inputs. In state $S2$ we test the value of the LSB of B, and if it is 1, we add A to P. Otherwise, P is not changed. The machine moves to state $S3$ when B contains 0, because P has the final product in this case. For each clock cycle in which the machine is in state $S2$, we shift the value of A to the left, as specified in the pseudo-code in Figure 10.15*b*. We shift the contents of B to the right so that in each clock cycle b_0 can be used to decide whether or not A should be added to P.

Datapath Circuit

We can now define the datapath circuit. To implement A we need a right-to-left shift register that has $2n$ bits. A $2n$-bit register is needed for P, and it must have an enable input because the assignment $P \leftarrow P + A$ in state $S2$ is inside a conditional output box. A $2n$-bit adder is needed to produce $P + A$. Note that P is loaded with 0 in state $S1$, and P is loaded

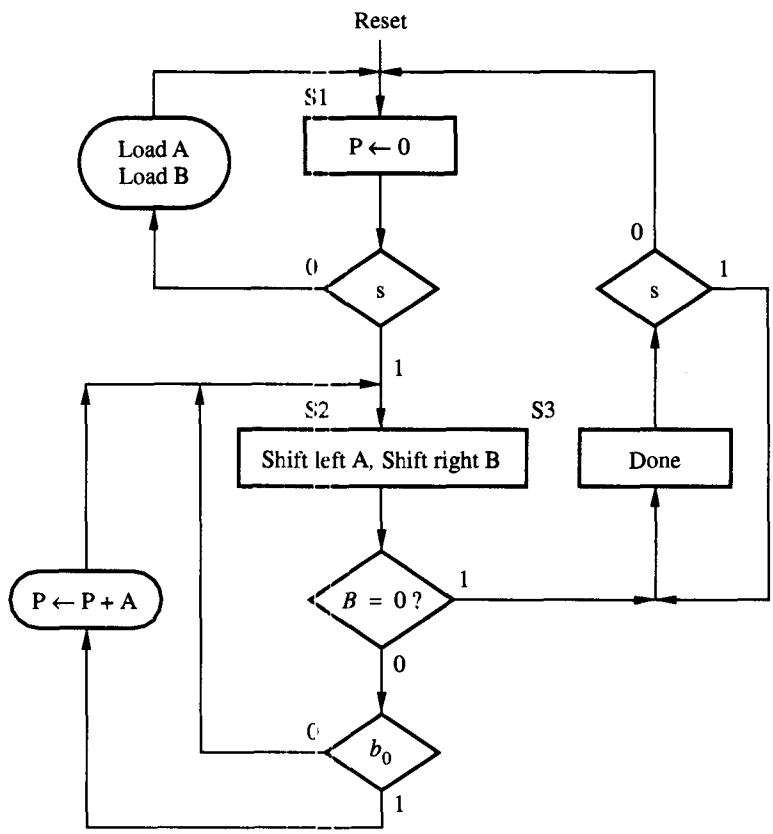


Figure 10.16 ASM chart for the multiplier.

from the output of the adder in state $S2$. We cannot assume that the reset input is used to clear P, because the machine changes from state $S3$ back to $S1$ based on the s input, not the reset input. Hence a 2-to-1 multiplexer is needed for each input to P, to select either 0 or the appropriate sum bit from the adder. An n-bit left-to-right shift register is needed for B, and an n-input NOR gate can be used to test whether $B = 0$.

Figure 10.17 shows the datapath circuit and labels the control signals for the shift registers. The input data for the shift register that holds A is named *DataA*. Since the shift register has $2n$ bits, the most-significant n data inputs are connected to 0. A single

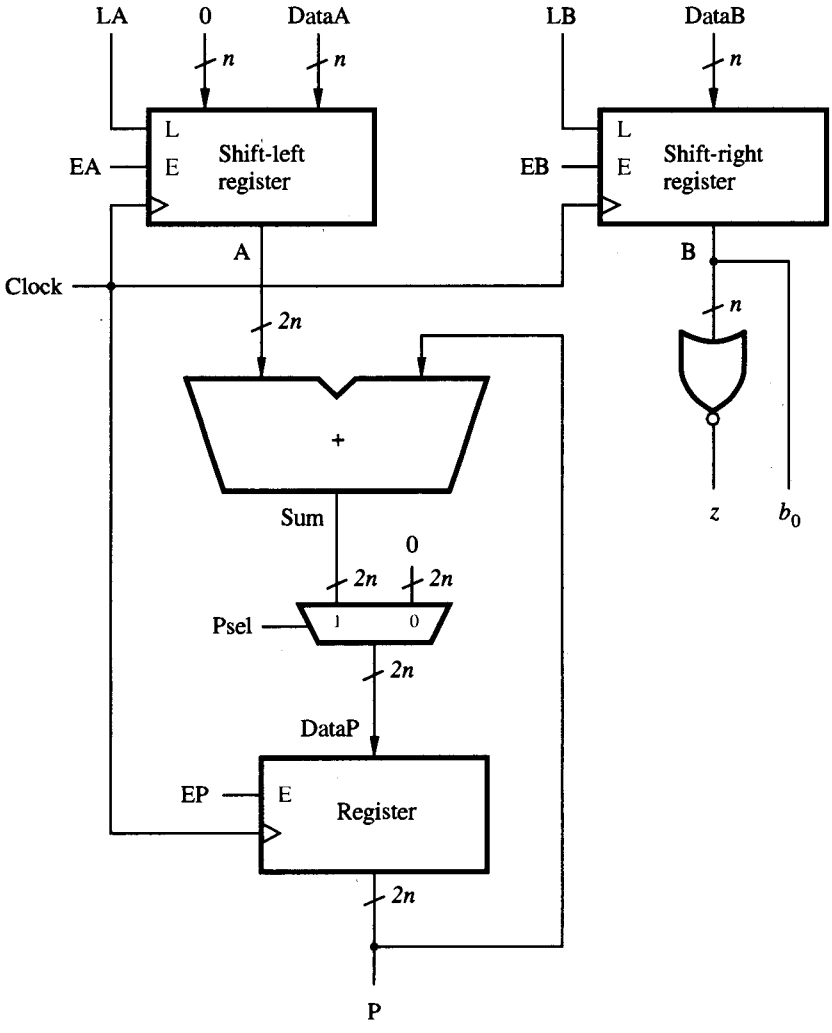


Figure 10.17 Datapath circuit for the multiplier.

multiplexer symbol is shown connected to the register that holds P. This symbol represents $2n$ 2-to-1 multiplexers that are each controlled by the *Psel* signal.

Control Circuit

An ASM chart that represents only the control signals needed for the multiplier is given in Figure 10.18. In state $S1$, *Psel* is set to 0 and *EP* is asserted, so that register P is cleared. When $s = 0$, parallel data can be loaded into shift registers A and B by an external circuit that controls their parallel load inputs *LA* and *LB*. When $s = 1$, the machine changes to state $S2$, where *Psel* is set to 1 and shifting of A and B is enabled. If $b_0 = 1$, the enable for P is asserted. The machine changes to state $S3$ when $z = 1$, and then remains in $S3$ and sets *Done* to the value 1 as long as $s = 1$.

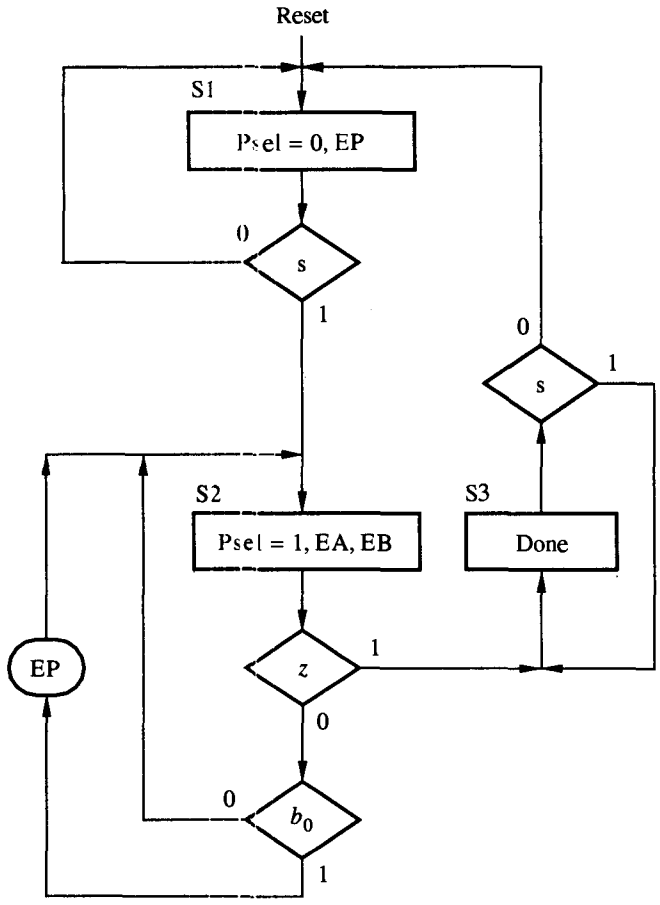


Figure 10.18 ASM chart for the multiplier control circuit.

Verilog Code

Verilog code for the multiplier is given in Figure 10.19. The number of bits in A and B is set by the parameter n. For registers that are $2n$ bits wide, the number of bits is set to $n + n$. By changing the value of the parameters, the code can be used for numbers of any size. The **always** blocks labeled *State_table* and *State_ flipflops* define the state transitions and state flip-flops, respectively. The control circuit outputs are specified in the **always** block labeled *FSM_outputs*. The parallel data input on the shift register A is $2n$ bits wide, but *DataA* is only n bits wide. Hence the concatenate operation {{n{1'b0}}, DataA} is used to prepend n zeros onto *DataA* for loading into the shift register. The multiplexer needed for register P is defined using a **for** loop that defines $2n$ 2-to-1 multiplexers. Figure 10.20 gives a simulation result for the circuit generated from the code. After the circuit is reset, *LA* and *LB* are set to 1, and the numbers to be multiplied are placed on the *DataA* and *DataB* inputs. After s is set to 1, the FSM (y) changes to state $S2$, where it remains until $B = 0$. For each clock cycle in state $S2$, A is shifted to the left, and B is shifted to the right. In three of the clock cycles in state $S2$, the contents of A are added to P, corresponding to the three bits in B that have the value 1. When $B = 0$, the FSM changes to state $S3$ and P contains the correct product, which is $(64)_{16} \times (19)_{16} = (9C4)_{16}$. The decimal equivalent of this result is $100 \times 25 = 2500$.

The number of clock cycles that the circuit requires to generate the final product is determined by the left-most digit in B that is 1. It is possible to reduce the number of clock cycles needed by using more complex shift registers for A and B. If the two right-most bits in B are both 0, then both A and B could be shifted by two bit positions in one clock cycle. Similarly, if the three lowest digits in B are 0, then a three bit-position shift can be done, and so on. A shift register that can shift by multiple bit positions at once can be built using a *barrel shifter*. We leave it as an exercise for the reader to modify the multiplier to make use of a barrel shifter.

10.2.4 DIVIDER

The preceding example implements the traditional method of performing multiplication by hand. In this example we will design a circuit that implements the traditional long-hand division. Figure 10.21*a* gives an example of long-hand division. The first step is to try to divide the divisor 9 into the first digit of the dividend 1, which does not work. Next, we try to divide 9 into 14, and determine that 1 is the first digit in the quotient. We perform the subtraction $14 - 9 = 5$, bring down the last digit from the dividend to form 50, and then determine that the next digit in the quotient is 5. The remainder is $50 - 45 = 5$, and the quotient is 15. Using binary numbers, as illustrated in Figure 10.21*b*, involves the same process, with the simplification that each digit of the quotient can be only 0 or 1.

Given two unsigned n-bit numbers A and B, we wish to design a circuit that produces two n-bit outputs Q and R, where Q is the quotient A/B and R is the remainder. The procedure illustrated in Figure 10.21*b* can be implemented by shifting the digits in A to the left, one digit at a time, into a shift register R. After each shift operation, we compare R with B. If $R \geq B$, a 1 is placed in the appropriate bit position in the quotient and B is subtracted from R. Otherwise, a 0 bit is placed in the quotient. This algorithm is described

```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
   parameter n = 8;
   input Clock, Resetn, LA, LB, s;
   input [n-1:0] DataA, DataB;
   output [n+n-1:0] P;
   output Done;
   wire z;
   reg [n+n-1:0] A, DataP;
   wire [n+n-1:0] Sum;
   reg [1:0] y, Y;
   reg [n-1:0] B;
   reg Done, EA, EB, EP, Psel;
   integer k;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

   always @(s or y or z)
   begin: State_table
      case (y)
         S1: if (s == 0) Y = S1;
             else Y = S2;
         S2: if (z == 0) Y = S2;
             else Y = S3;
         S3: if (s == 1) Y = S3;
             else Y = S1;
         default: Y = 2'bxx;
      endcase
   end

   always @(posedge Clock or negedge Resetn)
   begin: State_flipflops
      if (Resetn == 0)
         y <= S1;
      else
         y <= Y;
      end
```

... continued in Part *b*.

Figure 10.19 Verilog code for the multiplier circuit (Part a).

```
    always @(s or y or B[0])
    begin: FSM_outputs
      // defaults
      EA = 0; EB = 0; EP = 0; Done = 0; Psel = 0;
      case (y)
        S1: EP = 1;
        S2: begin
              EA = 1; EB = 1; Psel = 1;
              if (B[0]) EP = 1;
              else EP = 0;
            end
        S3: Done = 1;
      endcase
    end

  //datapath circuit

    shiftrne ShiftB (DataB, LB, EB, 0, Clock, B);
      defparam ShiftB.n = 8;
    shiftlne ShiftA ({{n{1'b0}}, DataA}, LA, EA, 0, Clock, A);
      defparam ShiftA.n = 16;

    assign z = (B == 0);
    assign Sum = A + P;

    // define the 2n 2-to-1 multiplexers
    always @(Psel or Sum)
      for (k = 0; k < n+n; k = k+1)
        DataP[k] = Psel ? Sum[k] : 0;

    regne RegP (DataP, Clock, Resetn, EP, P);
      defparam RegP.n = 16;

  endmodule
```

Figure 10.19 Verilog code for the multiplier circuit (Part *b*).

using pseudo-code in Figure 10.21*c*. The notation $R||A$ is used to represent a $2n$-bit shift register formed using R as the left-most n bits and A as the right-most n bits.

The pseudo-code for the multiplier in Figure 10.15*b* examines one digit, b_i, in each loop iteration. In the ASM chart in Figure 10.16, we shift B to the right so that b_0 always contains the digit needed. Similarly, in the long-division pseudo-code, each loop iteration results in setting a digit q_i to either 1 or 0. A straightforward way to accomplish this is to shift 1 or 0 into the least-significant bit of Q in each loop iteration. An ASM chart that

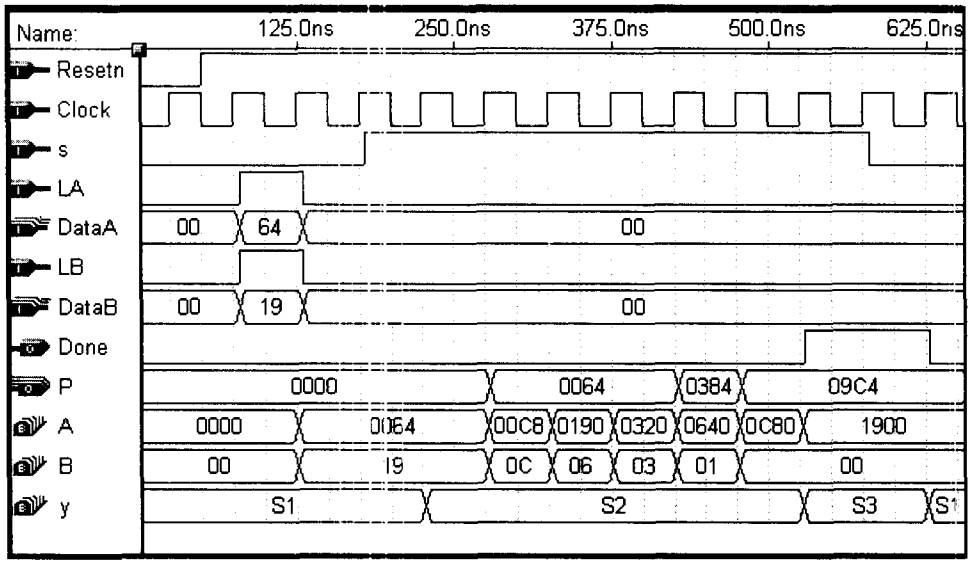


Figure 10.20 Simulation results for the multiplier circuit.

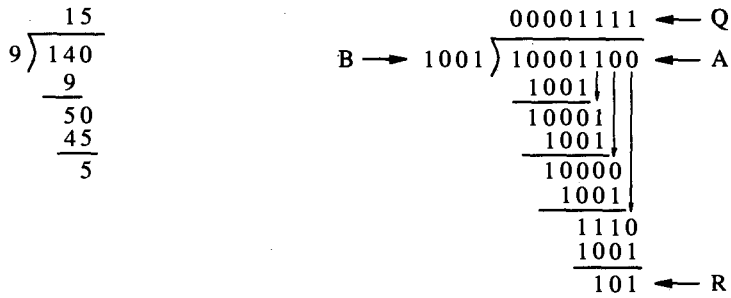
represents the divider circuit is shown in Figure 10.22. The signal C represents a counter that is initialized to $n - 1$ in the starting state $S1$. In state $S2$, both R and A are shifted to the left, and then in state $S3$, B is subtracted from R if $R \geq B$. The machine changes to state $S4$ when $C = 0$.

Datapath Circuit

We need n-bit shift registers that shift right to left for A, R, and Q. An n-bit register is needed for B, and a subtractor is needed to produce $R - B$. We can use an adder module in which the carry-in is set to 1 and B is complemented. The carry-out, c_{out}, of this module has the value 1 if the condition $R \geq B$ is true. Hence the carry-out can be connected to the serial input of the shift register that holds Q, so that it is shifted into Q in state $S3$. Since R is loaded with 0 in state $S1$ and from the outputs of the adder in state $S3$, a multiplexer is needed for the parallel data inputs on R. The datapath circuit is depicted in Figure 10.23. Note that the down-counter needed to implement C and the NOR gate that outputs a 1 when $C = 0$ are not shown in the figure.

Control Circuit

An ASM chart that shows only the control signals needed for the divider is given in Figure 10.24. In state $S3$ the value of c_{out} determines whether or not the sum output of the adder is loaded into R. The shift enable on Q is asserted in state $S3$. We do not have to specify whether 1 or 0 is loaded into Q, because c_{out} is connected to Q's serial input in the datapath circuit. We leave it as an exercise for the reader to write Verilog code that represents the ASM chart in Figure 10.24 and the datapath circuit in Figure 10.23.



(a) An example using decimal numbers

(b) Using binary numbers

```
R = 0;
for i = 0 to n - 1 do
    Left-shift R||A;
    if R ≥ B then
        q_i = 1;
        R = R - B;
    else
        q_i = 0;
    end if;
end for;
```

(c) Pseudo-code

Figure 10.21 An algorithm for division.

Enhancements to the Divider Circuit

Using the ASM chart in Figure 10.24 causes the circuit to loop through states $S2$ and $S3$ for $2n$ clock cycles. If these states can be merged into a single state, then the number of clock cycles needed can be reduced to n. In state $S3$, if $c_{out} = 1$, we load the sum output (result of the subtraction) from the adder into R, and (assuming $z = 0$) change to state $S2$. In state $S2$ we then shift R (and A) to the left. To combine $S2$ and $S3$ into a new state, called $S2$, we need to be able to place the sum into the left-most bits of R while at the same time shifting the MSB of A into the LSB of R. This step can be accomplished by using a separate flip-flop for the LSB of R. Let the output of this flip-flop be called rr_0. It is initialized to 0 when $s = 0$ in state $S1$. Otherwise, the flip-flop is loaded from the MSB of A. In state $S2$, if $c_{out} = 0$, R is shifted left and rr_0 is shifted into R. But if $c_{out} = 1$, R is loaded in parallel from the sum outputs of the adder.

Figure 10.25 illustrates how the division example from Figure 10.21*b* can be performed using n clock cycles. The table in the figure shows the values of R, rr_0, A, and Q in each step

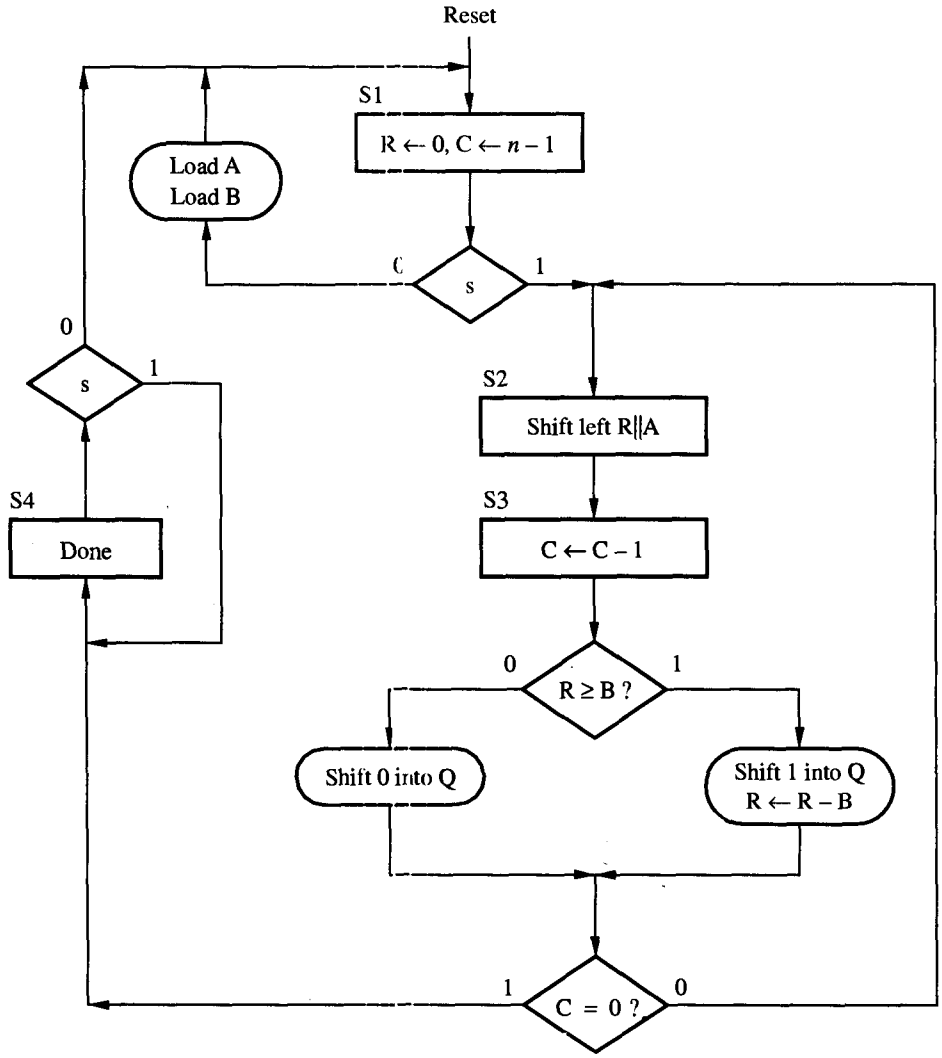


Figure 10.22 ASM chart for the divider.

of the division. In the datapath circuit in Figure 10.23, we use a separate shift register for Q. This register is not actually needed, because the digits in the quotient can be shifted into the least-significant bit of the register used for A. In Figure 10.25 the digits of Q that are shifted into A are shown in blue. The first row in the table represents loading of initial data into registers A (and B) and clearing R and rr_0 to 0. In the second row of the table, labeled clock cycle 0, the diagonal blue arrow shows that the left-most bit of A (1) is shifted into rr_0. The number in $R||rr_0$ is now 000000001, which is smaller than B (1001). In clock cycle 1, rr_0 is

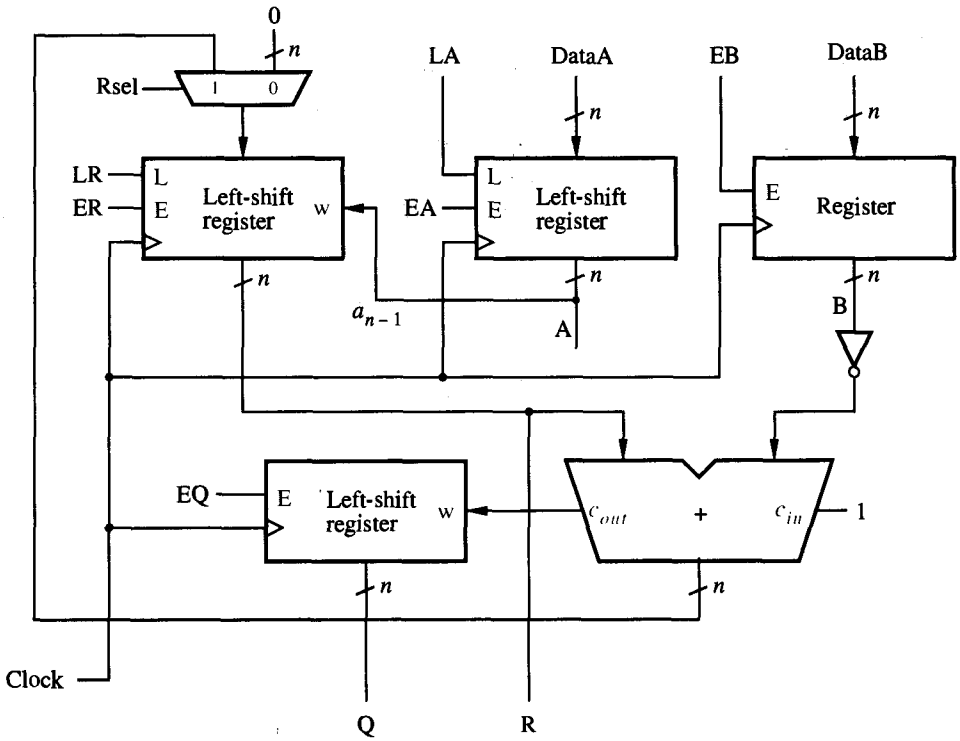


Figure 10.23 Datapath circuit for the divider.

shifted into R, and the MSB of A is shifted into rr_0. Also, as shown in blue, a 0 is shifted into the LSB of Q (A). The number in $R||rr_0$ is now 000000010, which is still smaller than B. Hence, in clock cycle 2 the same actions are performed as for clock cycle 1. These actions are also performed in clock cycles 3 and 4, at which point $R||rr_0 = 000010001$. Since this is larger than B, in clock cycle 5 the result of the subtraction $000010001 - 1001 = 00001000$ is loaded into R. The MSB of A (1) is still shifted into rr_0, and a 1 is shifted into Q. In clock cycles 6, 7, and 8, the number in $R||rr_0$ is larger than B; hence in each of these cycles the result of the subtraction $R||rr_0 - B$ is loaded into R, and a 1 is loaded into Q. After clock cycle 8 the correct result, Q = 00001111 and $R = 00000101$, is obtained. The bit rr_0 is not a part of the final result.

An ASM chart that shows the values of the required control signals for the enhanced divider is depicted in Figure 10.26. The signal ER0 is used in conjunction with the flip-flop that has the output rr_0. When ER0 = 0, the value 0 is loaded into the flip-flop. When ER0 is set to 1, the MSB of shift register A is loaded into the flip-flop. In state $S1$, if $s = 0$, then LR is asserted to initialize R to 0. Registers A and B can be loaded with data from external inputs. When s changes to 1, the machine makes a transition to state $S2$ and at the same time shifts $R||R0||A$ to the left. In state $S2$, if $c_{out} = 1$, then R is loaded in parallel from

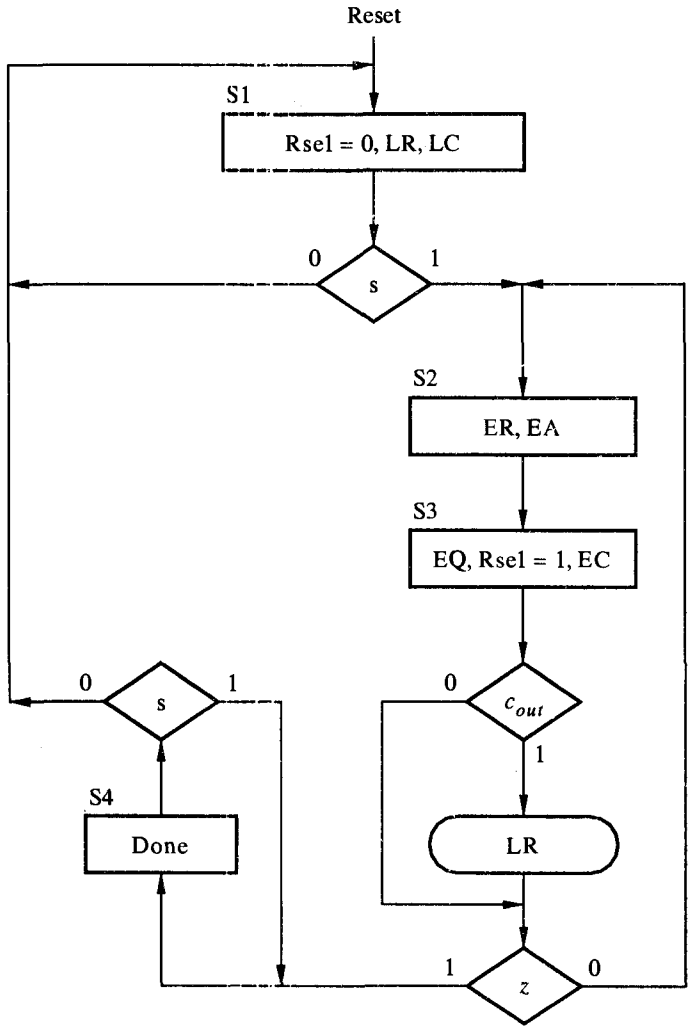


Figure 10.24 ASM chart for the divider control circuit.

the sum outputs of the adder. At the same time, $R0||A$ is shifted left (rr_0 is not shifted into R in this case). If $c_{out} = 0$, then $R|R0||A$ is shifted left. The ASM chart shows how the parallel-load and enable inputs on the registers have to be controlled to achieve the desired operation.

The datapath circuit for the enhanced divider is illustrated in Figure 10.27. As discussed for Figure 10.25, the digits of the quotient Q are shifted into register A. Note that one of the n-bit data inputs on the adder module is composed of the $n - 1$ least-significant bits in register R concatenated with bit rr_0 on the right.

Verilog Code

Figure 10.28 shows Verilog code that represents the enhanced divider. The parameter n sets the number of bits in the operands. The *State_table*, *State_flipflops*, and *FSM_outputs*

B ⟶ 1001 ⟌ 10001100 ⟵ A

	Clock cycle	R								rr_0	A/Q							
	Load A, B	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0
0	Shift left	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
1	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0
2	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0
3	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0
4	Shift left, $Q_0 \leftarrow 0$	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
5	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1
6	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
7	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	1	1
8	Subtract, $Q_0 \leftarrow 1$	0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1

Figure 10.25 An example of division using $n = 8$ clock cycles.

always blocks describe the control circuit, as in the previous examples. The shift registers and counters in the datapath circuit are instantiated at the bottom of the code. The signal rr_0 in Figure 10.25 is represented in the code by the signal R0. This signal is implemented as the output of the *muxdff* component; the code for this subcircuit is shown in Figure 7.52. Note that the adder that produces the *Sum* signal has one input defined as the concatenation of R with R0. The multiplexer needed for the input to R is represented by the *DataR* signal. This multiplexer is defined in the last statement of the code.

A simulation result for the circuit produced from the code is given in Figure 10.29. The data A = A6 and $B = 8$ is loaded, and then s is set to 1. The circuit changes to state $S2$ and concurrently shifts R, $R0$, and A to the left. The output of the shift register that holds A is labeled Q in the simulation results because this shift register contains the quotient when the division operation is complete. On the first three active clock edges in state $S2$, the number represented by $R||R0$ is less than the number in B (8); hence $R||R0||A$ is shifted left on each clock edge, and 0 is shifted into Q. In the fourth consecutive clock cycle for which the FSM has been in state $S2$, the contents of R are $00000101 = (5)_{10}$, and $R0$ is 0; hence $R||R0 = 000001010 = (10)_{10}$. On the next active clock edge, the output of the adder, which is $10 - 8 = 2$, is loaded into R, and 1 is shifted into Q. After n clock cycles in state $S2$, the circuit changes to state $S3$, and the correct result, $Q = 14 = (20)_{10}$ and $R = 6$, is obtained.

10.2.5 ARITHMETIC MEAN

Assume that k n-bit numbers are stored in a set of registers $R_0, \ldots, R_{k-1}$. We wish to design a circuit that computes the mean M of the numbers in the registers. The pseudo-code for a suitable algorithm is shown in Figure 10.30*a*. Each iteration of the loop adds the contents of one of the registers, denoted R_i, to a *Sum* variable. After the sum is computed, M is obtained as *Sum*/k. We assume that integer division is used, so a remainder R, not shown in the code, is produced as well.

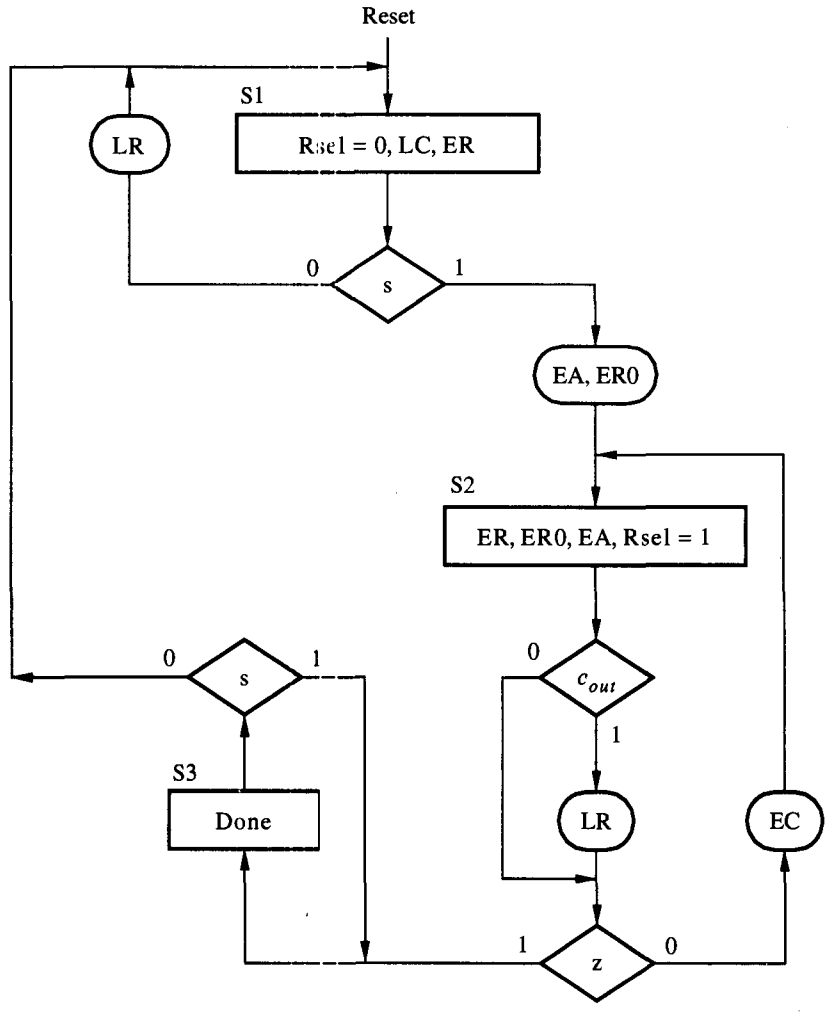


Figure 10.26 ASM chart for the enhanced divider control circuit.

An ASM chart is given in Figure 10.30*b*. While the start input, *s*, is 0, the registers can be loaded from external inputs. When *s* becomes 1, the machine changes to state *S*2, where it remains while $C \neq 0$, and computes the summation (C is a counter that represents i in Figure 10.30*a*). When $C = 0$, the machine changes to state *S*3 and computes $M = Sum/k$. From the previous example, we know that the division operation requires multiple clock cycles, but we have chosen not to indicate this in the ASM chart. After computing the division operation, state *S*4 is entered and *Done* is set to 1.

Datapath Circuit

The datapath circuit for this task is more complex than in our previous examples. It is depicted in Figure 10.31. We need a register with an enable input to hold *Sum*. For simpli-

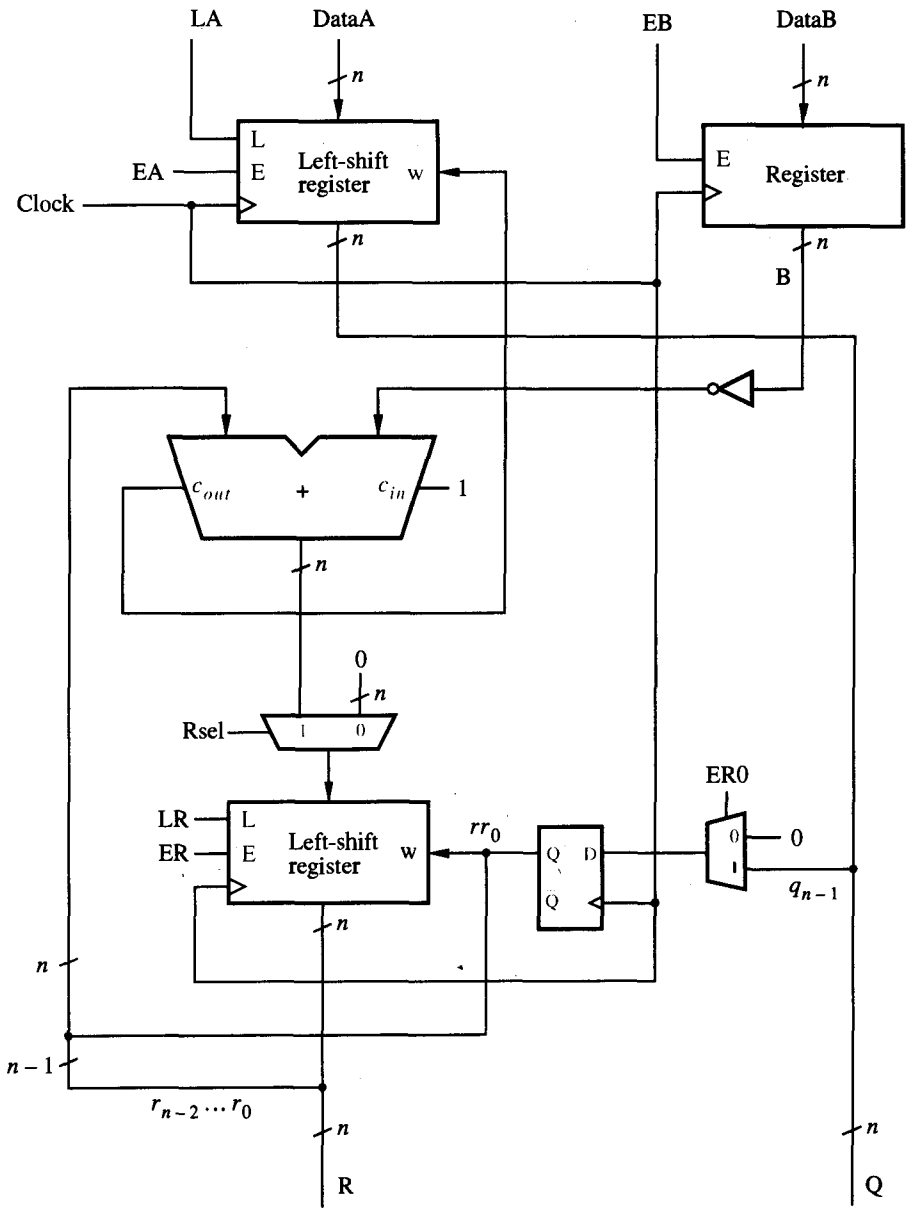


Figure 10.27 Datapath circuit for the enhanced divider.

city, assume that the sum can be represented in n bits without overflowing. A multiplexer is required on the data inputs on the *Sum* register, to select 0 in state $S1$ and the sum outputs of an adder in state $S2$. The *Sum* register provides one of the data inputs to the adder. The other input has to be selected from the data outputs of one of the k registers. One way to select among the registers is to connect them to the data inputs of a k-to-1 multiplexer that is connected to the adder. The select lines on the multiplexer can be controlled by the

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done);
   parameter n = 8, logn = 3;
   input Clock, Resetn, s, LA, EB;
   input [n−1:0] DataA, DataB;
   output [n−1:0] R, Q;
   output Done;
   wire Cout, z;
   wire [n−1:0] DataR;
   wire [n:0] Sum;
   reg [1:0] y, Y;
   reg [n−1:0] A, B;
   reg [logn−1:0] Count;
   reg Done, EA, Rsel, LR, ER, ER0, LC, EC, R0;
   integer k;

// control circuit

   parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

   always @(s or y or z)
   begin: State_table
      case (y)
         S1: if (s == 0) Y = S1;
             else Y = S2;
         S2: if (z == 0) Y = S2;
             else Y = S3;
         S3: if (s == 1) Y = S3;
             else Y = S1;
         default: Y = 2'bxx;
      endcase
   end

   always @(posedge Clock or negedge Resetn)
   begin: State_flipflops
      if (Resetn == 0)
         y <= S1;
      else
         y <= Y;
   end
```

... continued in Part *b*.

Figure 10.28 Verilog code for the divider circuit (Part a).

```
always @(y or s or Cout or z)
begin: FSM_outputs
   // defaults
   LR = 0; ER = 0; ER0 = 0; LC = 0; EC = 0; EA = 0;
   Rsel = 0; Done = 0;
   case (y)
      S1: begin
            LC = 1; ER = 1;
            if (s == 0)
            begin
               LR = 1; ER0 = 0;
            end
            else
            begin
               LR = 0; EA = 1; ER0 = 1;
            end
          end
      S2: begin
            Rsel = 1; ER = 1; ER0 = 1; EA = 1;
            if (Cout) LR = 1;
            else LR = 0;
            if (z == 0) EC = 1;
            else EC = 0;
          end
      S3: Done = 1;
   endcase
end
```

... continued in Part *c*.

Figure 10.28 Verilog code for the divider circuit (Part *b*).

counter C. To compute the division operation, we can use the divider circuit designed in section 10.2.4.

The circuit in Figure 10.31 is based on $k = 4$, but the same circuit structure can be used for larger values of k. Note that the enable inputs on the registers R_0 through R_3 are connected to the outputs of a 2-to-4 decoder that has the two-bit input *RAdd*, which stands for "register address." The decoder enable input is driven by the *ER* signal. All registers are loaded from the same input lines, *Data*. Since $k = 4$, we could perform the division operation simply by shifting *Sum* two bits to the right, which can be done in one clock cycle with a shift register that shifts by two digits. To obtain a more general circuit that works for any value of k, we use the divider circuit designed in section 10.2.4.

```
// datapath circuit

    regne  RegB (DataB, Clock, Resetn, EB, B);
       defparam RegB.n = n;
    shiftlne  ShiftR (DataR, LR, ER, R0, Clock, R);
       defparam ShiftR.n = n;
    muxdff  FF_R0 (0, A[n-1], ER0, Clock, R0);
    shiftlne  ShiftA (DataA, LA, EA, Cout, Clock, A);
       defparam ShiftA.n = n;
    assign  Q = A;
    downcount  Counter (Clock, EC, LC, Count);
       defparam Counter.n = logn;

    assign  z = (Count == 0);
    assign  Sum = {R, R0} + (~B + 1);
    assign  Cout = Sum[n];

    // define the n 2-to-1 multiplexers
    assign  DataR = Rsel ? Sum : 0;

endmodule
```

Figure 10.28 Verilog code for the divider circuit (Part c).

Control Circuit

Figure 10.32 gives an ASM chart for the FSM needed to control the circuit in Figure 10.31. While in state $S1$, data can be loaded into registers $R_0, \ldots, R_{k-1}$. But no control signals have to be asserted for this purpose, because the registers are loaded under control of the *ER* and *RAdd* inputs, as discussed above. When $s = 1$, the FSM changes to state $S2$, where it asserts the enable *ES* on the *Sum* register and allows *C* to decrement. When the counter reaches 0 ($z = 1$), the machine enters state $S3$, where it asserts the *LA* and *EB* signals to load the *Sum* and *k* into the *A* and *B* inputs of the divider circuit, respectively. The FSM then enters state $S4$ and asserts the *Div* signal to start the division operation. When it is finished, the divider circuit sets $zz = 1$, and the FSM moves to state $S5$. The mean *M* appears on the Q and *R* outputs of the divider circuit. The *Div* signal must still be asserted in state $S5$ to prevent the divider circuit from reinitializing its registers. Note that in the ASM chart in Figure 10.30*b*, only one state is shown for computing $M = Sum/k$, but in Figure 10.32, states $S3$ and $S4$ are used for this purpose. It is possible to combine states $S3$ and $S4$, which we will leave as an exercise for the reader (problem 10.6).

Alternative Datapath Circuits

In Figure 10.31 registers $R_0, \ldots, R_{k-1}$ are connected to the adder using a multiplexer. Another way to achieve the desired connection is to add tri-state buffers to the outputs of the *k* registers and to connect all tri-state buffers for a given bit position to the corresponding

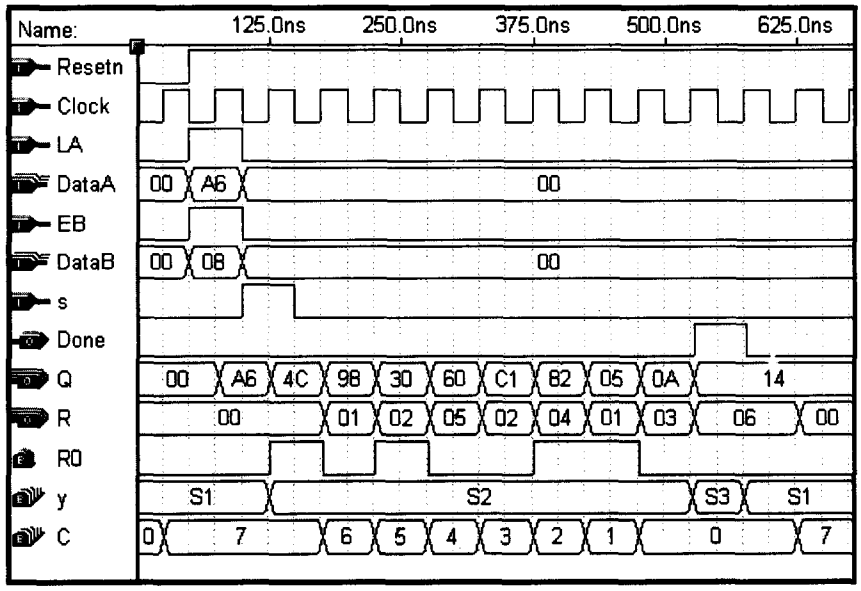


Figure 10.29 Simulation results for the divider circuit.

input of the adder. The down-counter C can be used to enable each tri-state buffer at the proper time (when the FSM is in state $S2$), by connecting a 2-to-4 decoder to the outputs of the counter and using one output of the decoder to enable each tri-state buffer. We will show an example of using tri-states buffers in this manner in Figure 10.42.

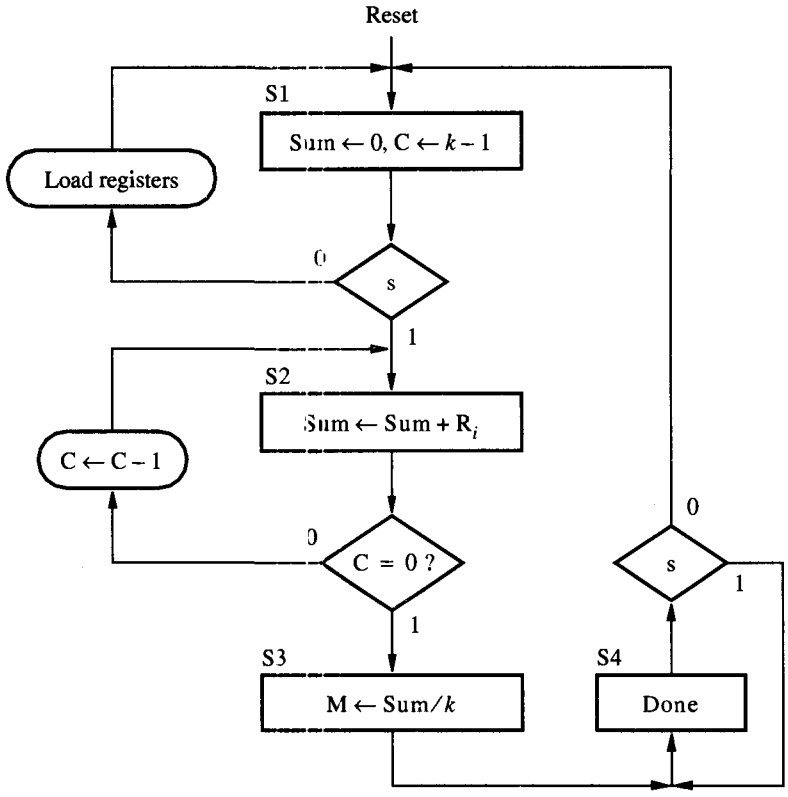
For large values of k, it is preferable to use an SRAM block with k rows and n columns, instead of using k registers. Predefined modules that represent SRAM blocks are usually provided by CAD tools. If the circuit being designed is to be implemented in a custom chip, then the CAD tools ensure that the desired SRAM block is included on the chip. Some PLDs include SRAM blocks that can be configured to implement various numbers of rows and columns. The CAD system that accompanies the book provides the *lpm_ram_dq* module, which is a part of the LPM standard library.

Figure 10.33 gives a schematic diagram for the arithmetic mean circuit, using the parameters $k = 16$ and $n = 8$. This schematic was created using the CAD tools that accompany the book. Four of the graphical symbols in the schematic represent subcircuits described using Verilog code, namely *downcnt*, *regne*, *divider*, and *meancntl*. The code for the *divider* subcircuit is shown in Figure 10.28. The *meancntl* subcircuit represents the FSM in Figure 10.32. The Verilog code for this FSM is not shown. The schematic also includes a multiplexer connected to the *Sum* register, an adder, and a NOR gate that detects when the counter C reaches 0. The outputs of the counter provide the address inputs to the SRAM block, called *MReg*.

The SRAM block has 16 rows and eight columns. In Figure 10.31 a decoder controls the loading of data into each of the k registers. To read the data from the registers, the counter C is used. To keep the schematic in Figure 10.33 simple, we have included the

```
Sum = 0 ;
for i = k − 1 down to 0 do
    Sum = Sum + R_i
end for ;
M = Sum ÷ k ;
```

(a) Pseudo-code



(b) ASM chart

Figure 10.30 An algorithm for finding the mean of k numbers.

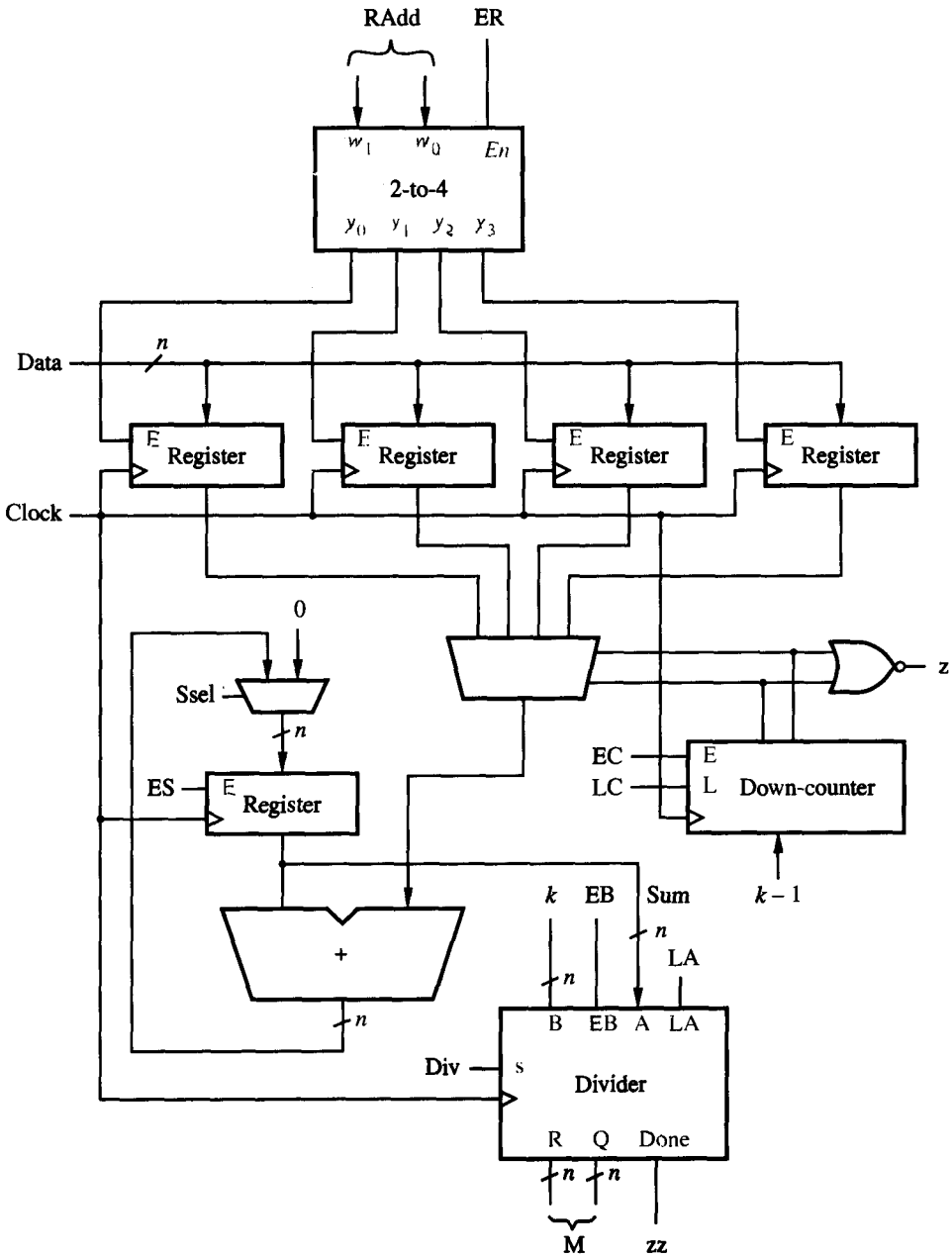


Figure 10.31 Datapath circuit for the mean operation.

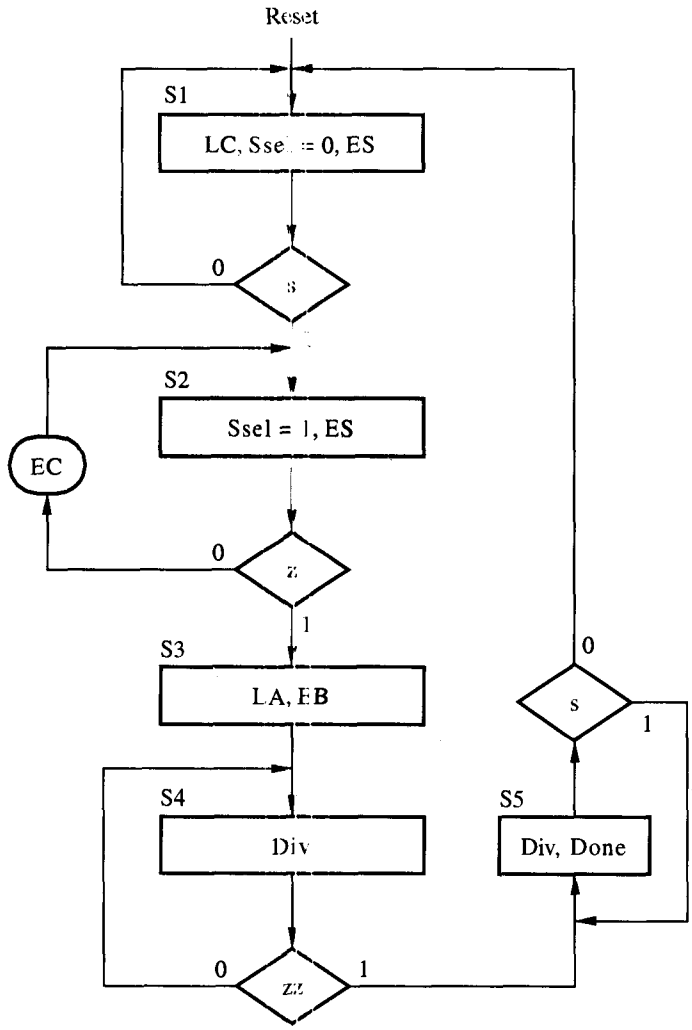


Figure 10.32 ASM chart for the mean operation control circuit.

counter to read data from the SRAM block, but we have ignored the issue of writing data into the SRAM block. It is possible to modify the *meancntl* code to allow the counter C to address the SRAM block for loading the initial data, but we will not pursue this issue here.

For simulation purposes we can use a feature of the CAD system that allows initial data to be stored in the SRAM block. We chose to store 0 in R_0 (row 0 of the SRAM block); 1 in $R_1, \ldots$; and 15 in R_{15}. The results of a timing simulation for the circuit implemented in an FPGA chip are shown in Figure 10.34. Only a part of the simulation, from the point

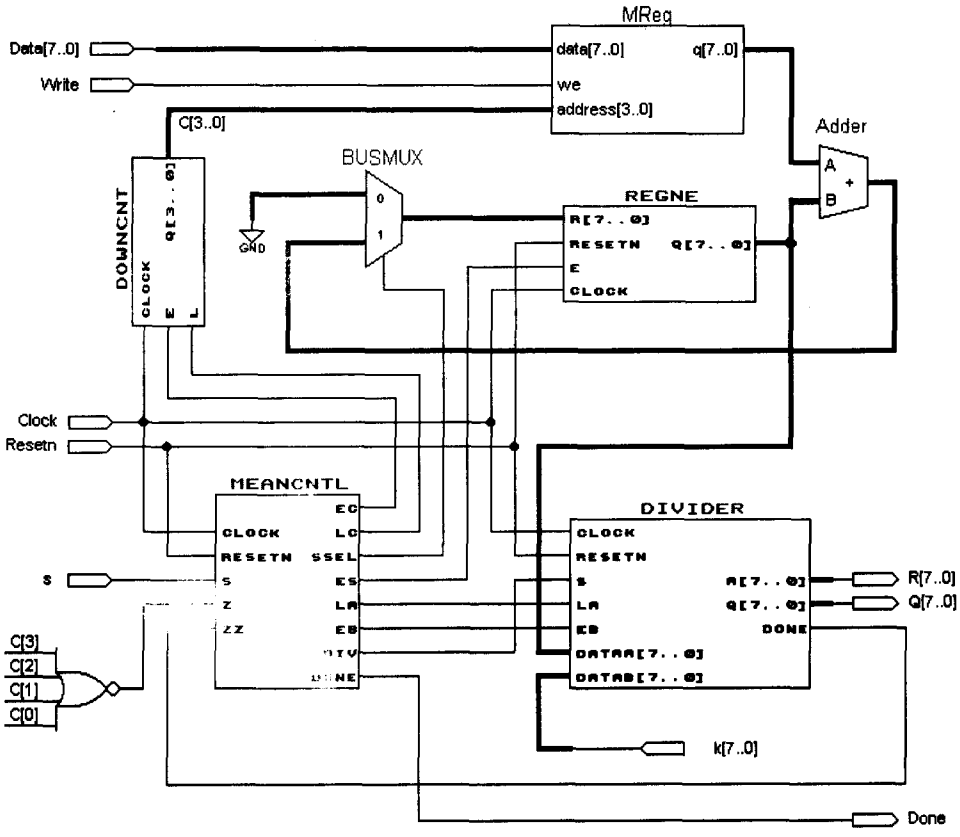


Figure 10.33 Schematic of the mean circuit with an SRAM block.

where $C = 5$, is shown in the figure. At this point the *meancntl* FSM is in state $S2$, and the *Sum* is being accumulated. When C reaches 0, *Sum* has the correct value, which is $0 + 1 + 2 + \ldots + 15 = 120 = (78)_{16}$. The FSM changes to state $S3$ for one clock cycle and then remains in state $S4$ until the division operation is complete. The correct result, Q $= 7$ and $R = 8$, is obtained when the FSM changes to state $S5$.

10.2.6 SORT OPERATION

Given a list of k unsigned n-bit numbers stored in a set of registers $R_0, \ldots, R_{k-1}$, we wish to design a circuit that can sort the list (contents of the registers) in ascending order. Pseudo-code for a simple sorting algorithm is shown in Figure 10.35. It is based on finding the smallest number in the sublist $R_i, \ldots, R_{k-1}$ and moving that number into R_i, for $i = 1, 2, \ldots, k-2$. Each iteration of the outer loop places the number in R_i into A. Each iteration of the inner loop compares this number to the contents of another register R_j. If

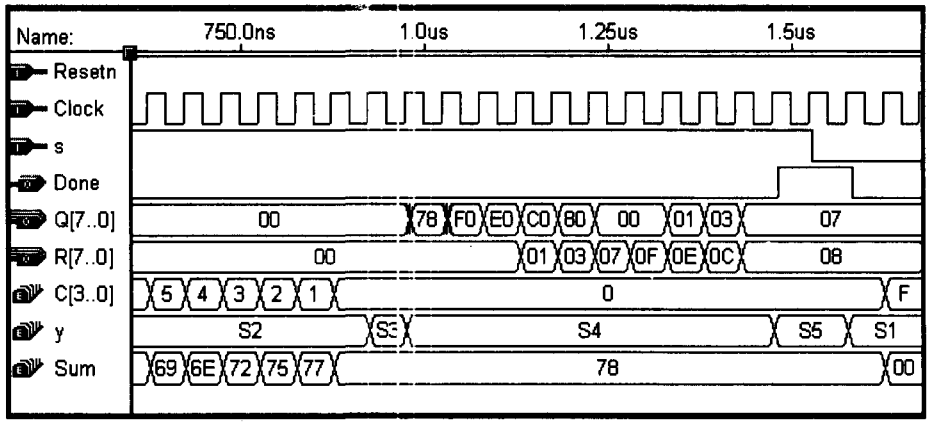


Figure 10.34 Simulation results for the mean circuit using SRAM.

```
for i = 0 to k − 2 do
    A = R_i ;
    for j = i + 1 to k − 1 do
        B = R_j ;
        if B < A then
            R_i = B ;
            R_j = A ;
            A = R_i ;
        end if ;
    end for ;
end for ;
```

Figure 10.35 Pseudo-code for the sort operation.

the number in R_j is smaller than A, the contents of R_i and R_j are swapped and A is changed to hold the new contents of R_i.

An ASM chart that represents the sorting algorithm is shown in Figure 10.36. In the initial state $S1$, while $s = 0$ the registers are loaded from external data inputs and a counter C_i that represents i in the outer loop is cleared. When the machine changes to state $S2$, A is loaded with the contents of R_i. Also, C_j, which represents j in the inner loop, is initialized to the value of i. State $S3$ is used to initialize j to the value $i + 1$, and state $S4$ loads the value of R_j into B. In state $S5$, A and B are compared, and if $B < A$, the machine moves to state $S6$. States $S6$ and $S7$ swap the values of R_i and R_j. State $S8$ loads A from R_i. Although this step is necessary only for the case where $B < A$, the flow of control is simpler if this operation is performed in both cases. If C_j is not equal to $k - 1$, the machine changes from $S8$ to $S4$, thus remaining in the inner loop. If $C_j = k - 1$ and C_i is not equal to $k - 2$, then the machine stays in the outer loop by changing to state $S2$.

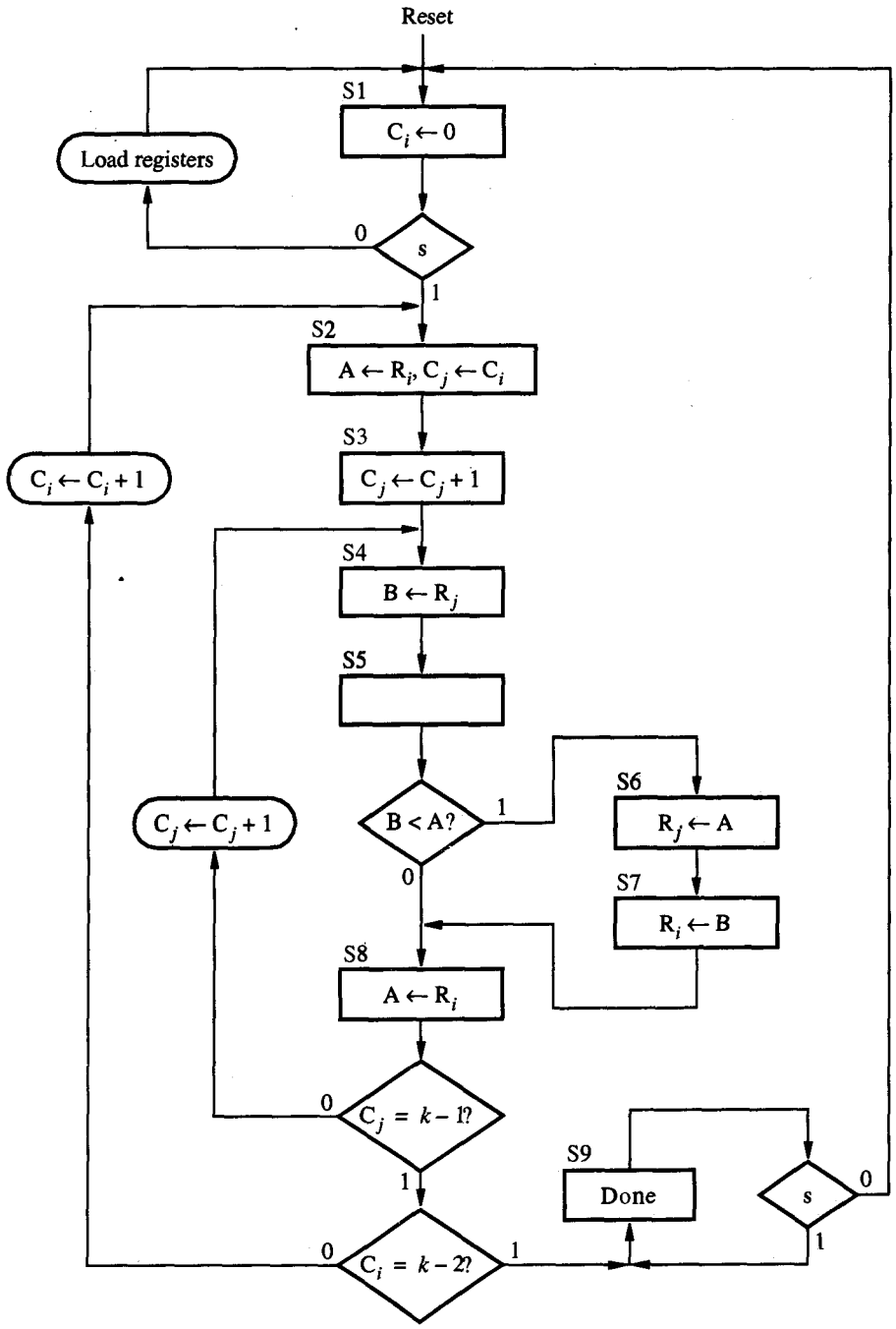


Figure 10.36 ASM chart for the sort operation.

Datapath Circuit

There are many ways to implement a datapath circuit that meets the requirements of the ASM chart in Figure 10.36. One possibility is illustrated in Figures 10.37 and 10.38. Figure 10.37 shows how the registers $R_0, \ldots, R_{k-1}$ can be connected to registers A and B using 4-to-1 multiplexers. We assume the value $k = 4$ for simplicity. Registers A and B are connected to a comparator subcircuit and, through multiplexers, back to the inputs of the registers $R_0, \ldots, R_{k-1}$. The registers can be loaded with initial (unsorted) data using the *DataIn* lines. The data is written (loaded) into each register by asserting the *WrInit* control signal and placing the address of the register on the *RAdd* input. The tri-state buffer driven by the *Rd* control signal is used to output the contents of the registers on the *DataOut* output.

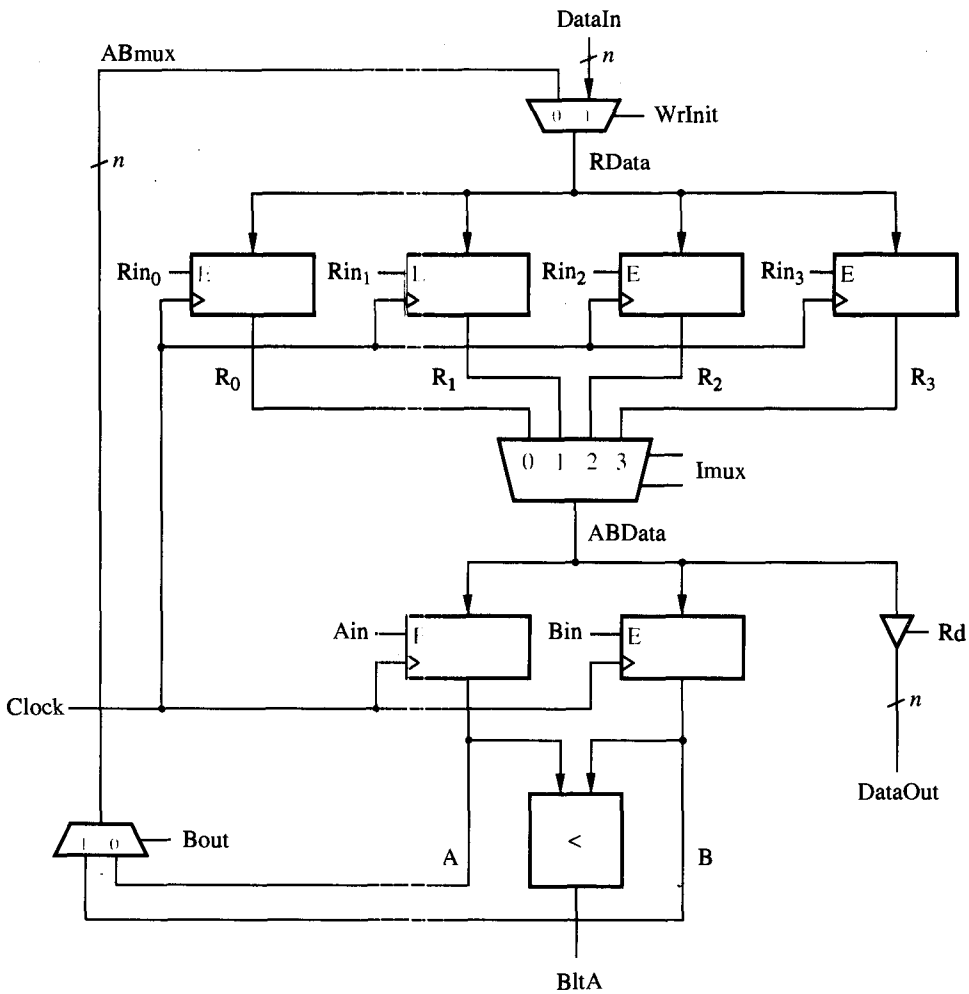


Figure 10.37 A part of the datapath circuit for the sort operation.

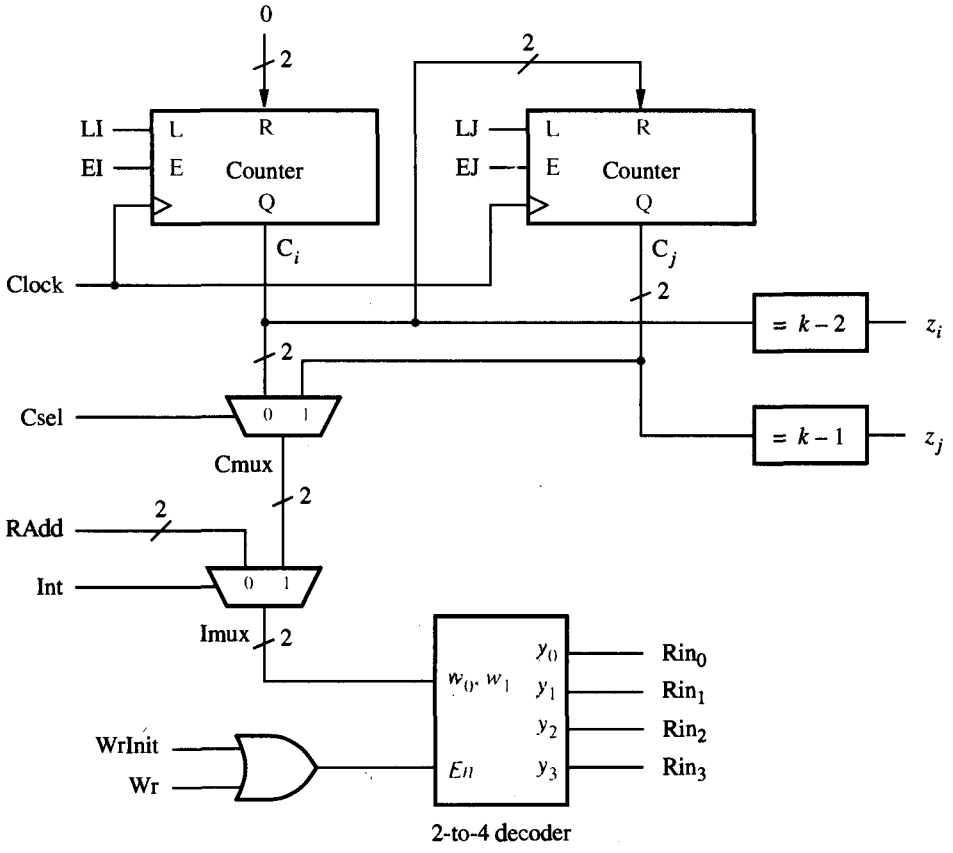


Figure 10.38 A part of the datapath circuit for the sort operation.

The signals $Rin_0, \ldots, Rin_{k-1}$ are controlled by the 2-to-4 decoder shown in Figure 10.38. If $Int = 1$, the decoder is driven by one of the counters C_i or C_j. If $Int = 0$, then the decoder is driven by the external input *RAdd*. The signals z_i and z_j are set to 1 if $C_i = k - 2$ and $C_j = k - 1$, respectively. An ASM chart that shows the control signals used in the datapath circuit is given in Figure 10.39.

Verilog Code

Verilog code for the sorting operation is presented in Figure 10.40. The FSM that controls the sort operation is described in the same way as in previous examples, using the **always** blocks *State_table*, *State_flipflops*, and *FSM_outputs*. Following these blocks, the code instantiates the registers R_0 to R_3, as well as A and B. The counters C_i and C_j have the instance names *OuterLoop* and *InnerLoop*, respectively. The multiplexers with the outputs *CMux* and *IMux* are specified using the conditional operator. The 4-to-1 multiplexer in Figure 10.37 is defined by the **case** statement that specifies the value of the *ABData* signal for each value of *IMux*. The 2-to-4 decoder in Figure 10.38 with the outputs $Rin_0, \ldots, Rin_3$

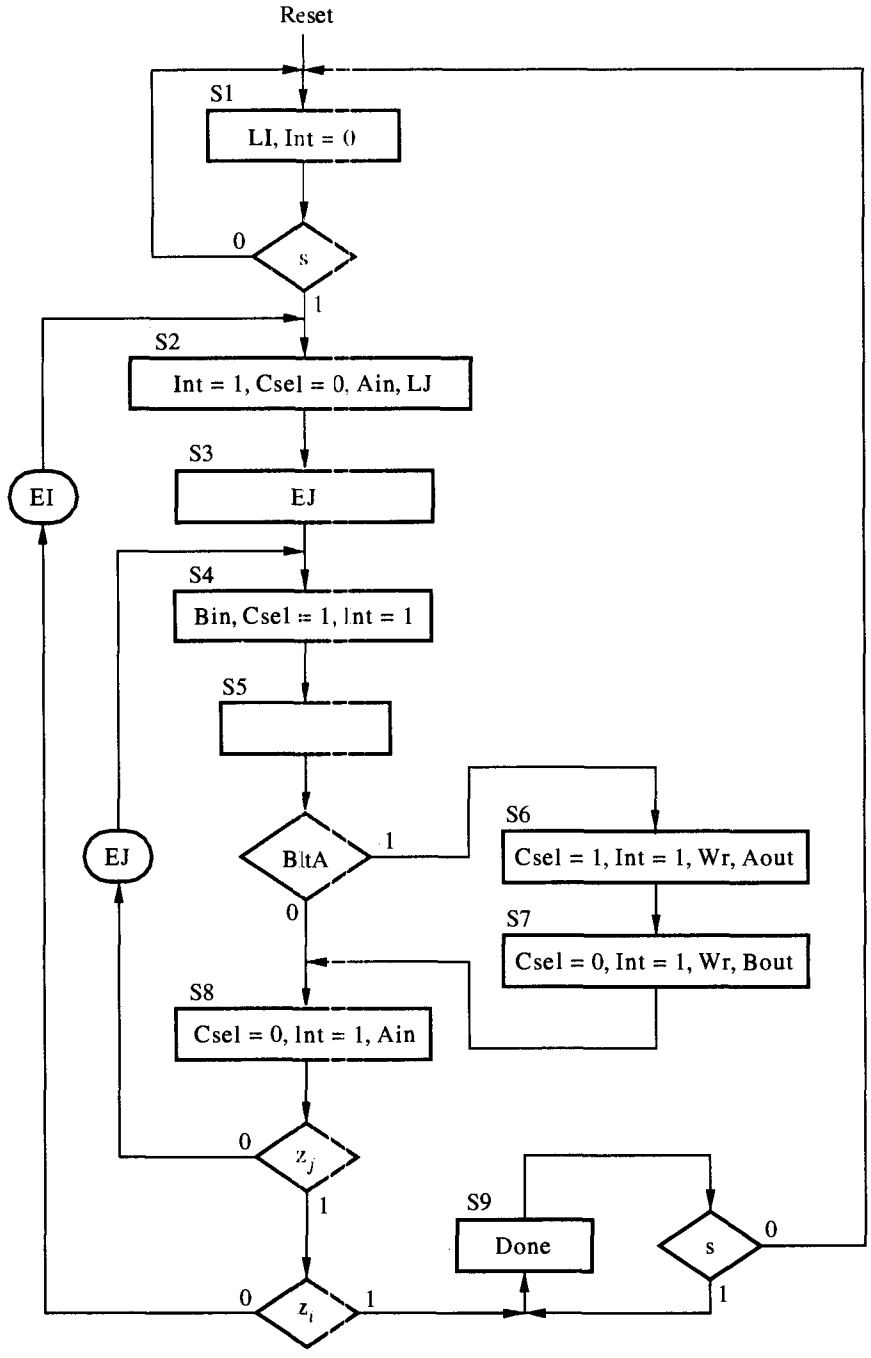


Figure 10.39 ASM chart for the control circuit.

```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
   parameter n = 4;
   input Clock, Resetn, s, WrInit, Rd;
   input [n-1:0] DataIn;
   input [1:0] RAdd;
   output [n-1:0] DataOut;
   output Done;
   wire [1:0] Ci, Cj, CMux, IMux;
   wire [n-1:0] R0, R1, R2, R3, A, B, RData, ABMux;
   wire BltA, zi, zj;
   reg Int, Csel, Wr, Ain, Bin, Aout, Bout;
   reg LI, LJ, EI, EJ, Done, Rin0, Rin1, Rin2, Rin3;
   reg [3:0] y, Y;
   reg [n-1:0] ABData;

// control circuit
   parameter S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
   parameter S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

   always @(s or BltA or zj or zi)
   begin: State_table
      case (y)
         S1: if (s == 0) Y = S1;
             else Y = S2;
         S2: Y = S3;
         S3: Y = S4;
         S4: Y = S5;
         S5: if (BltA) Y = S6;
             else Y = S8;
         S6: Y = S7;
         S7: Y = S8;
         S8: if (!zj) Y = S4;
             else if (!zi) Y = S2;
             else Y = S9;
         S9: if (s) Y = S9;
             else Y = S1;
         default: Y = 4'bx;
      endcase
   end
```

. . . continued in Part *b*.

Figure 10.40 Verilog code for the sorting circuit (Part a).

```
always @(posedge Clock or negedge Resetn)
begin: State_flipflops
   if (Resetn == 0)
      y <= S1;
   else
      y <= Y;
end

always @(y or zj or zi)
begin: FSM_outputs
   // defaults
   Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0;
   Wr = 0; Ain = 0; Bin = 0; Aout = 0; Bout = 0;
   case (y)
      S1:  begin LI = 1; Int = 0; end
      S2:  begin Ain = 1; LJ = 1; end
      S3:  EJ = 1;
      S4:  begin Bin = 1; Csel = 1; end
      S5:; // no ouputs asserted in this state
      S6:  begin Csel = 1; Wr = 1; Aout = 1; end
      S7:  begin Wr = 1; Bout = 1; end
      S8:  begin
              Ain = 1;
              if (!zj) EJ = 1;
              else
              begin
                 EJ = 0;
                 if (!zi) EI = 1;
                 else EI = 0;
              end
           end
      S9:  Done = 1;
   endcase
end
```

... continued in Part *c*.

Figure 10.40 Verilog code for the sorting circuit (Part *b*).

is defined by the **case** statement that sets the value of the concatenated signals {Rin3, Rin2, Rin1, Rin0}. Finally, the code specifies the values of the z_i and z_j signals, and defines the tri-state buffers for the *DataOut* output.

We implemented the code in Figure 10.40 in an FPGA chip. Figure 10.41 gives an example of a simulation result. Part (*a*) of the figure shows the first half of the simulation,

```
//datapath circuit

regne  Reg0 (RData, Clock, Resetn, Rin0, R0);
   defparam Reg0.n = n;
regne  Reg1 (RData, Clock, Resetn, Rin1, R1);
   defparam Reg1.n = n;
regne  Reg2 (RData, Clock, Resetn, Rin2, R2);
   defparam Reg2.n = n;
regne  Reg3 (RData, Clock, Resetn, Rin3, R3);
   defparam Reg3.n = n;

regne  RegA (ABData, Clock, Resetn, Ain, A);
   defparam RegA.n = n;
regne  RegB (ABData, Clock, Resetn, Bin, B);
   defparam RegB.n = n;

assign  BltA = (B < A) ? 1 : 0;
assign  ABMux = (Bout == 0) ? A : B;
assign  RData = (WrInit == 0) ? ABMux : DataIn;

upcount  OuterLoop (0, Resetn, Clock, EI, LI, Ci);
upcount  InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

assign  CMux = (Csel == 0) ? Ci : Cj;
assign  IMux = (Int == 1) ? CMux : RAdd;
```

... continued in Part *d*.

Figure 10.40 Verilog code for the sorting circuit (Part *c*).

from 0 to 1.25 μs, and part (*b*) shows the second half, from 1.25 μs to 2.5 μs. After resetting the circuit, *WrInit* is set to 1 for four clock cycles, and unsorted data is written into the four registers using the *DataIn* and *RAdd* inputs. After *s* is changed to 1, the FSM changes to state $S2$. States $S2$ to $S4$ load A with the contents of R_0 (3) and B with the contents of R_1 (2). State $S5$ compares B with A, and since $B < A$, the FSM uses states $S6$ and $S7$ to swap the contents of registers R_0 and R_1. In state $S8$, A is reloaded from R_0, which now contains 2. Since z_j is not asserted, the FSM increments the counter C_j and changes back to state $S4$. Register B is now loaded with the contents of R_2 (4), and the FSM changes to state $S5$. Since $B = 4$ is not less than $A = 2$, the machine changes to $S8$ and then back to $S4$. Register B is now loaded with the contents of R_3 (1), which is then compared against $A = 2$ in state $S5$. The contents of R_0 and R_3 are swapped, and the machine changes to $S8$. At this point, the register contents are $R_0 = 1$, $R_1 = 3$, $R_2 = 4$, and $R_3 = 2$. Since $z_j = 1$ and $z_i = 0$, the FSM performs the next iteration of the outer loop by changing to state $S2$. Jumping forward in the simulation time, in Figure 10.41*b* the circuit reaches the state in which $C_i = 2$, $C_j = 3$, and the FSM is in state $S8$. The FSM then changes to state $S9$ and

```
always @(WrInit or Wr or IMux)
begin
   case (IMux)
      0:  ABData = R0;
      1:  ABData = R1;
      2:  ABData = R2;
      3:  ABData = R3;
   endcase

   if (WrInit || Wr)
      case (IMux)
         0:  {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
         1:  {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
         2:  {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
         3:  {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
      endcase
   else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;
end

assign  zi = (Ci == 2);
assign  zj = (Cj == 3);
assign  DataOut = (Rd == 0) ? 'bz : ABData;

endmodule
```

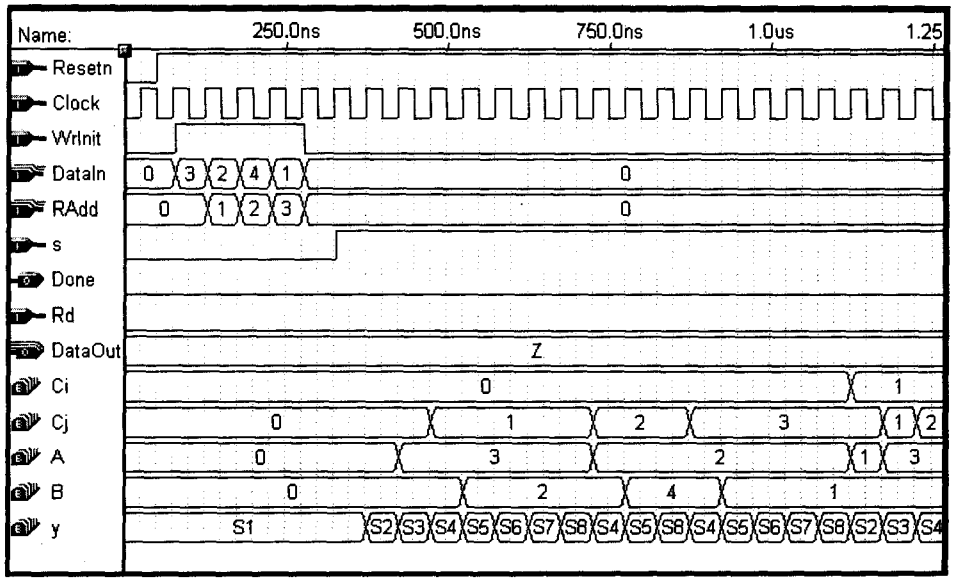
Figure 10.40 Verilog code for the sorting circuit (Part *d*).

sets *Done* to the value 1. The correctly sorted data is read out of the registers by setting the signal $Rd = 1$ and using the *RAdd* inputs to select each of the registers.

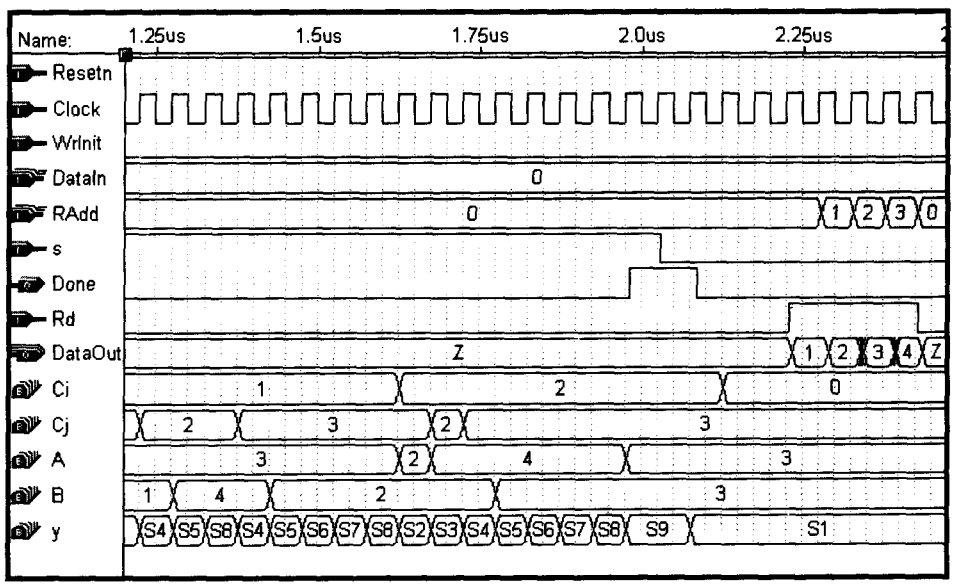
Alternative Datapath Circuits

In Figure 10.37 we use multiplexers to connect the various registers in the datapath circuit. Another approach is to use tri-state buffers to interconnect the registers, as illustrated in Figure 10.42. As we said in section 7.14, the set of n common wires that connect the registers is called a *bus*. The circuit in Figure 10.42 has two buses, one that connects the outputs of registers $R_0, \ldots, R_3$ to the inputs of registers A and B and another that connects the outputs of A and B back to the inputs of $R_0, \ldots, R_{k-1}$. When multiplexers provide the connection between registers, as shown in Figure 10.37, the term *bus* can still be used to refer to the connection between registers.

The circuit in Figure 10.42 uses the circuit in Figure 10.38 with one modification. In Figure 10.38 the *IMux* signal is connected to a 2-to-4 decoder that generates $Rin_0, \ldots, Rin_3$. If the circuit in Figure 10.42 is used, then a second decoder connected to *IMux* is required



(a) Loading the registers and starting the sort operation



(b) Completing the sort operation and reading the registers

Figure 10.41 Simulation results for the sort operation.

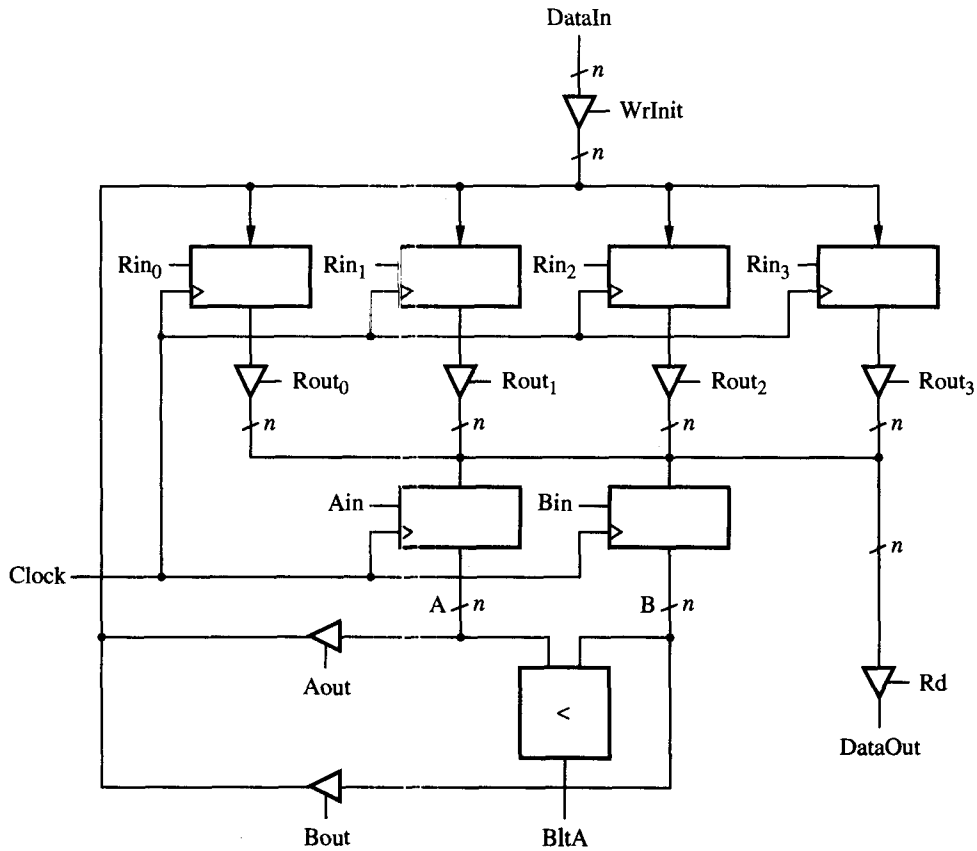


Figure 10.42 Using tri-state buffers in the datapath circuit.

to generate the control signals $Rout_0, \ldots, Rout_3$. The control circuit described in the ASM chart in Figure 10.39 can be used for the datapath circuit in Figure 10.42.

We said in section 10.2.5 that for large values of k, it is better to use an SRAM block to store the data, instead of individual registers. The sorting circuit can be changed to make use of an SRAM block with k rows and n columns. In this case the datapath circuit is similar to the one in Figure 10.37, but does not require the 4-to-1 multiplexers, because the data outputs from the SRAM block are connected directly to registers A and B. We still need to use the circuit in Figure 10.38, except that the 2-to-4 decoder is not required, because the *IMux* signal is connected to the address inputs on the SRAM block. The *write* input on the SRAM block is driven by the OR gate with the inputs *WrInit* and *Wr*. Verilog code can be written for the sorting circuit, in which a component that represents the SRAM block is instantiated from a library of predefined modules. The code for the control circuit shown in Figure 10.40 does not have to be changed (see problem 10.11).

10.3 CLOCK SYNCHRONIZATION

In the previous section we provided several examples of circuits that contain many flip-flops. In Chapter 9 we showed that to ensure proper operation of sequential circuits it is essential to give careful consideration to the timing aspects associated with the storage elements. This section discusses some of the timing aspects of synchronous sequential circuits.

10.3.1 CLOCK SKEW

Figure 10.1 shows how an enable input can be used to prevent a flip-flop from changing its stored value when an active clock edge occurs. Another way to implement the clock enable feature is shown in Figure 10.43. The circuit uses an AND gate to force the clock input to have the value 0 when $E = 0$. This circuit is simpler than the one in Figure 10.1 but can cause problems in practice. Consider a sequential circuit that has many flip-flops, some of which have an enable input and others that do not. If the circuit in Figure 10.43 is used, then the flip-flops without the enable input will observe changes in the clock signal slightly earlier than the flip-flops that have the enable input. This situation, in which the clock signal arrives at different times at different flip-flops, is known as *clock skew*. Figure 10.43 shows only one possible source of clock skew. Similar problems arise in a chip in which the clock signal is distributed to different flip-flops by wires whose lengths vary appreciably.

To understand the possible problems caused by clock skew, consider the datapath circuit for the bit-counting example in Figure 10.11. The shift register's LSB, a_0, is used as a control signal that determines whether or not a counter is incremented. Assume that clock skew exists that causes the clock signal to arrive earlier at the shift-register flip-flops than at the counter. The clock skew may cause the shift register to be shifted *before* the value of a_0 is used to cause the counter to increment. Therefore, the signal *EB* in Figure 10.11 may fail to cause the counter to be incremented even if the value of a_0 was 1.

For proper operation of synchronous sequential circuits, it is essential to minimize the clock skew as much as possible. Chips that contain many flip-flops, such as PLDs, use carefully designed networks of wires to distribute the clock signal to the flip-flops. Figure 10.44 gives an example of a clock-distribution network. Each node labeled *ff* represents the clock input of a flip-flop; for clarity, the flip-flops are not shown. The buffer on the left of the figure produces the clock signal. This signal is distributed to the flip-flops such that the length of the wire between each flip-flop and the clock source is the same. Due to

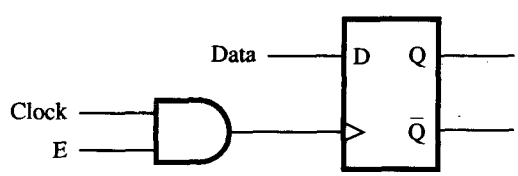


Figure 10.43 Clock enable circuit.

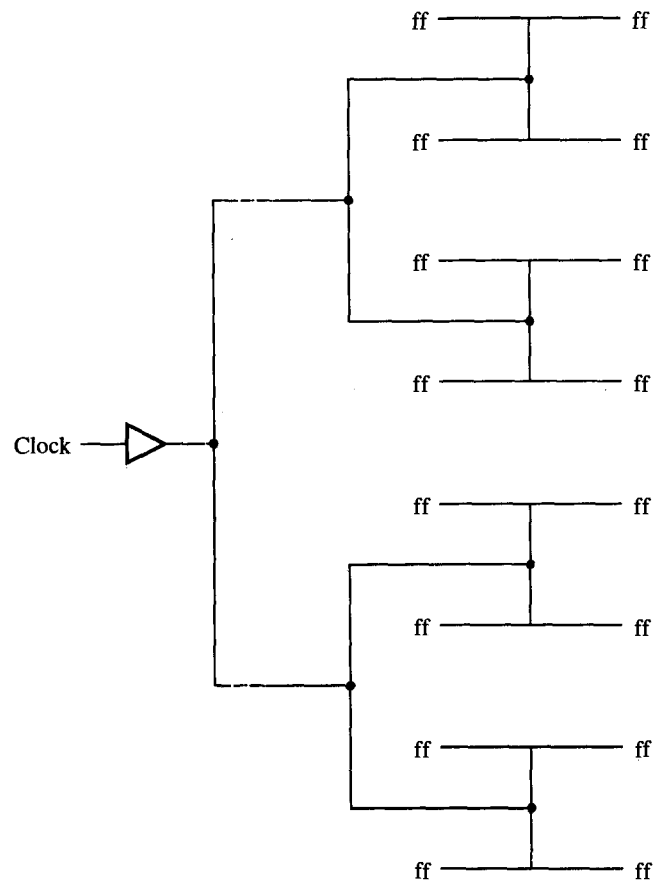


Figure 10.44 An H tree clock distribution network.

the appearance of sections of the wires, which resemble the letter H, the clock distribution network is known as an *H tree*. In PLDs the term *global clock* refers to the clock network. A PLD chip usually provides one or more global clocks that can be connected to all flip-flops. When designing a circuit to be implemented in such a chip, a good design practice is to connect all the flip-flops in the circuit to a single global clock. Connecting logic gates to the clock inputs of flip-flops, as discussed for the enable circuit in Figure 10.43, should be avoided.

It is useful to be able to ensure that a sequential circuit is reset into a known state when power is first applied to the circuit. A good design practice is to connect the asynchronous reset (clear) inputs of all flip-flops to a wiring network that provides a low-skew reset signal. PLDs usually provide a *global reset* wiring network for this purpose.

10.3.2 FLIP-FLOP TIMING PARAMETERS

We discussed the timing parameters for storage elements in section 7.3.1. Data to be clocked into a flip-flop must be stable t_{su} before the active clock edge and must remain stable t_h after the clock edge. A change in the value of the output Q appears after the *register delay*, t_{rd}. An *output delay time*, t_{od}, is required for the change in Q to propagate to an output pin on the chip. These timing parameters account for the behavior of an individual flip-flop without considering how the flip-flop is connected to other circuitry in an integrated circuit chip.

Figure 10.45 depicts a flip-flop as part of an integrated circuit. Connections are shown from the flip-flop's clock, *D*, and Q terminals to pins on the chip package. There is an input buffer associated with each pin on the chip. Other circuitry may also be connected to the flip-flop; the shaded box represents a combinational circuit connected to *D*. The propagation delays between the pins on the chip package and the flip-flop are labeled in the figure as t_{Data}, t_{Clock}, and t_{od}.

In digital systems the output signals from one chip are used as the input signals to another chip. In most cases the flip-flops in all chips are driven by a common clock that has low skew. The signals must propagate from the Q outputs of flip-flops in one chip to the *D* inputs of flip-flops in another chip. To ensure that all timing specifications are met, it is necessary to consider the output delays in one chip and the input delays in another.

The t_{co} delay determines how long it takes from when an active clock edge occurs at the clock pin on the chip package until a change in the output of a flip-flop appears at an output pin on the chip. This delay consists of three main parts. The clock signal must first propagate from its input pin on the chip to the flip-flop's *Clock* input. This delay is labeled t_{Clock} in Figure 10.45. After the register delay t_{rd}, the flip-flop produces a new output, which takes t_{od} to propagate to the output pin. An example of timing parameters taken from a commercial CPLD chip is $t_{Clock} = 1.5$ ns, $t_{rd} = 1$ ns, and $t_{od} = 2$ ns. These parameters give the delay from the active clock edge to the change on the output pin as $t_{co} = 4.5$ ns.

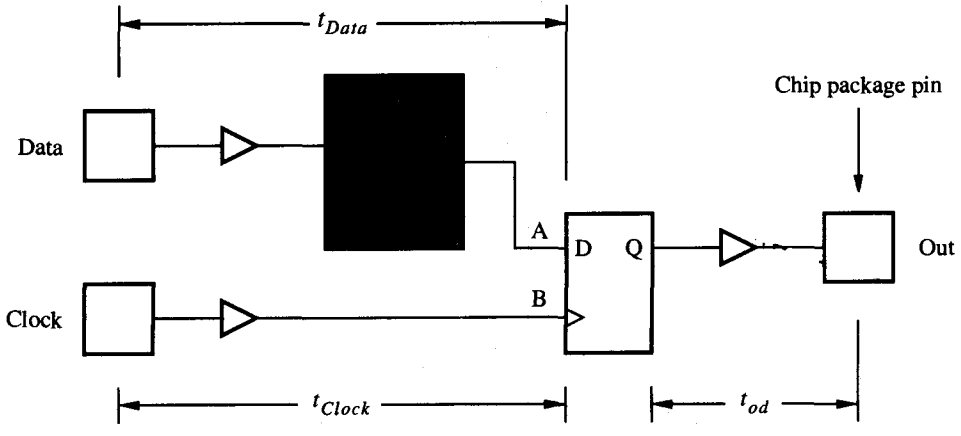


Figure 10.45 A flip-flop in an integrated circuit chip.

If chips are separated by a large distance, the propagation delays between them must be taken into consideration. But in most cases the distance between chips is small, and the propagation time of signals between the chips is negligible. Once a signal reaches the input pin on a chip, the relative values of t_{Data} and t_{Clock} (see Figure 10.45) must be considered. For example, in Figure 10.46 we assume that $t_{Data} = 4.5$ ns and $t_{Clock} = 1.5$ ns. The setup time for the flip-flops in the chip is specified as $t_{su} = 3$ ns. In the figure the *Data* signal changes from low to high 3 ns before the positive clock edge, which should meet the setup requirements. The *Data* signal takes 4.5 ns to reach the flip-flop, whereas the *Clock* signal takes only 1.5 ns. The signal labeled *A* and the clock signal labeled *B* reach the flip-flop at the same time. The setup time requirement is violated, and the flip-flop may become unstable. To avoid this condition, it is necessary to increase the setup time as seen from outside the chip.

The hold time for flip-flops is also affected by chip-level delays. The result is usually a reduction in the hold time, rather than an increase. For example, with the timing parameters in Figure 10.46 assume that the hold time is $t_h = 2$ ns. Assume that the signal at the *Data* pin on the chip changes value at exactly the same time that an active edge occurs at the *Clock* pin. The change in the *Clock* signal will reach node *B* 4.5 − 1.5 = 3 ns before the change in *Data* reaches node *A*. Hence even though the external change in *Data* is coincident with the clock edge, the required hold time of 2 ns is not violated.

For large circuits, ensuring that flip-flop timing parameters are properly adhered to is a challenge. Both the timing parameters of the flip-flops themselves and the relative delays incurred by the clock and data signals must be considered. CAD systems provide tools that can check the setup and hold times at all flip-flops automatically. This task is done using timing simulation, as well as special-purpose timing-analysis tools.

10.3.3 ASYNCHRONOUS INPUTS TO FLIP-FLOPS

In our examples of synchronous sequential circuits, we have assumed that changes in all input signals occur shortly after an active clock edge. The rationale for this assumption is that the inputs to one circuit are produced as the outputs of another circuit, and the same clock signal is used for both circuits. In practice, some of the inputs to a circuit may be

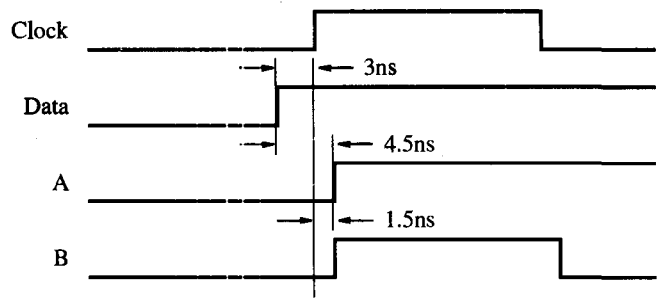


Figure 10.46 Flip-flop timing in a chip.

generated asynchronously with respect to the clock signal. If these signals are connected to the *D* input of a flip-flop, then the setup or hold times may be violated.

When a flip-flop's setup or hold times are violated, the flip-flop's output may assume a voltage level that does not correspond to either logic value 0 or 1. We say that the flip-flop is in a *metastable* state. The flip-flop eventually settles in one of the stable states, 0 or 1, but the time required to recover from the metastable state is not predictable. A common approach for dealing with asynchronous inputs is illustrated in Figure 10.47. The asynchronous data input is connected to a two-bit shift register. The output of the first flip-flop, labeled *A* in the figure, will sometimes become metastable. But if the clock period is sufficiently long, then *A* will recover to a stable logic value before the next clock pulse occurs. Hence the output of the second flip-flop will not become metastable and can safely be connected to other parts of the circuit. The synchronization circuit introduces a delay of one clock cycle before the signal can be used by the rest of the circuit.

Commercial chips, such as PLDs, specify the minimum allowable clock period that has to be used for the circuit in Figure 10.47 to solve the metastability problem. In practice, it is not possible to *guarantee* that node *A* will always be stable before a clock edge occurs. The data sheets specify a probability of node *A* being stable, as a function of the clock period. We will not pursue this issue further; the interested reader can refer to references [10, 11] for a more detailed discussion.

10.3.4 SWITCH DEBOUNCING

Inputs to a logic circuit are sometimes generated by mechanical switches. A problem with such switches is that they bounce away from their contact points when changed from one position to the other. Figure 10.48*a* shows a single-pole single-throw switch that provides an input to a logic circuit. If the switch is open, then the *Data* signal has the value 1. When the switch is thrown to the closed position, *Data* becomes 0, but the switch bounces for some time, causing *Data* to oscillate between 1 and 0. The bouncing typically persists for about 10 ms.

There is no simple way of dealing with the bouncing problem using the single-pole single-throw switch. If this type of switch must be used, then a possible solution is to use a circuit, such as a counter, to measure an appropriately long delay to wait for the bouncing to stop (see problem 10.23).

A better approach for dealing with switch bouncing is depicted in Figure 10.48*b*. It uses a single-pole double-throw switch and a basic SR latch to generate an input to a logic

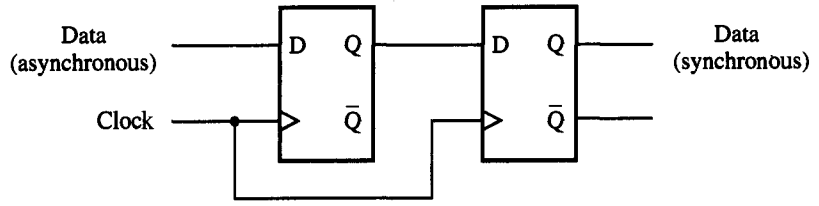
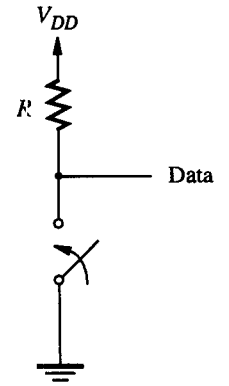
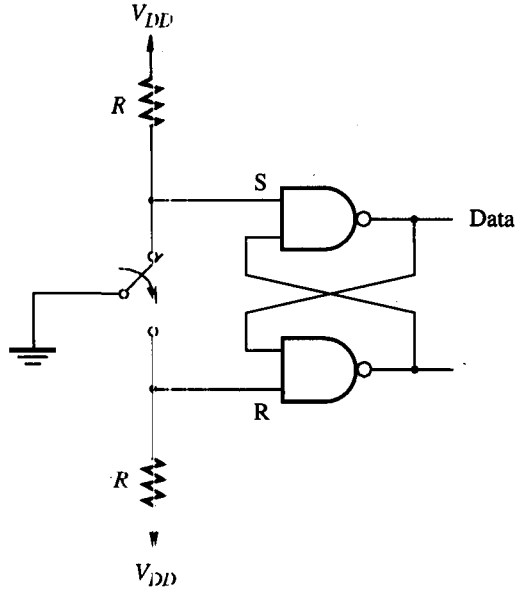


Figure 10.47 Asynchronous inputs.



(a) Single-pole single-throw switch



(b) Single-pole double-throw switch with a basic SR latch

Figure 10.48 Switch debouncing circuit.

circuit. When the switch is in the bottom position, the R input on the latch is 0 and *Data* = 0. When the switch is thrown to the top position, the S input on the latch becomes 0, which sets *Data* to 1. If the switch bounces away from the top position, the inputs to the

latch become $R = S = 1$ and the value $Data = 1$ is stored by the latch. When the switch is thrown to the bottom position, *Data* changes to 0 and this value is stored in the latch if the switch bounces. Note that when a switch bounces, it cannot bounce fully between the S and R terminals; it only bounces slightly away from one of the terminals and then back to it.

10.4 CONCLUDING REMARKS

This chapter has provided several examples of digital systems that include one or more FSMs as well as building blocks like adders, registers, shift registers, and counters. We have shown how ASM charts can be used as an aid for designing a digital system, and we have shown how the circuits can be described using Verilog code. A number of practical issues have been discussed, such as clock skew, synchronization of asynchronous inputs, and switch debouncing. Some notable books that also cover the material presented in this chapter include [3–10].

PROBLEMS

10.1 The circuit in Figure 10.4 gives a shift register in which the parallel-load control input is independent of the enable input. Show a different shift register circuit in which the parallel-load operation can be performed only when the enable input is also asserted.

10.2 The ASM chart in Figure 10.10, which describes the bit-counting circuit, includes Moore-type outputs in states $S1$, $S2$, and $S3$, and it has a Mealy-type output in state $S2$.
(a) Show how the ASM chart can be modified such that it has only Moore-type outputs in state $S2$.
(b) Give the ASM chart for the control circuit corresponding to part (*a*).
(c) Give Verilog code that represents the modified control circuit.

10.3 Figure 10.17 shows the datapath circuit for the shift-and-add multiplier. It uses a shift register for B so that b_0 can be used to decide whether or not A should be added to P. A different approach is to use a normal register to hold operand B and to use a counter and multiplexer to select bit b_i in each stage of the multiplication operation.
(a) Show the ASM chart that uses a normal register for B, instead of a shift register.
(b) Show the datapath circuit corresponding to part (*a*).
(c) Give the ASM chart for the control circuit corresponding to part (*b*).
(d) Give Verilog code that represents the multiplier circuit.

10.4 Write Verilog code for the divider circuit that has the datapath in Figure 10.23 and the control circuit represented by the ASM chart in Figure 10.24.

10.5 Section 10.2.4 shows how to implement the traditional long division that is done by "hand." A different approach for implementing integer division is to perform repeated subtraction as indicated in the pseudo-code in Figure P10.1.

```
Q = 0 ;
R = A ;
while ((R − B) > 0) do
    R = R − B ;
    Q = Q + 1 ;
end while ;
```

Figure P10.1 Pseudo-code for integer division.

(a) Give an ASM chart that represents the pseudo-code in Figure P10.1.
(b) Show the datapath circuit corresponding to part (*a*).
(c) Give the ASM chart for the control circuit corresponding to part (*b*).
(d) Give Verilog code that represents the divider circuit.
(e) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit designed in section 10.2.4.

10.6 In the ASM chart in Figure 10.32, the two states $S3$ and $S4$ are used to compute the mean $M = Sum/k$. Show a modified ASM chart that combines states $S3$ and $S4$ into a single state, called $S3$.

10.7 Write Verilog code for the FSM represented by your ASM chart defined in problem 10.6.

10.8 In the ASM chart in Figure 10.36, we specify the assignment $C_j \leftarrow C_i$ in state $S2$, and then in state $S3$ we increment C_j by 1. Is it possible to eliminate state $S3$ if the assignment $C_j \leftarrow C_i + 1$ is performed in $S2$? Explain any implications that this change has on the control and datapath circuits.

10.9 Figure 10.35 gives pseudo-code for the sorting operation in which the registers being sorted are indexed using variables i and j. In the ASM chart in Figure 10.36, variables i and j are implemented using the counters C_i and C_j. A different approach is to implement i and j using two shift registers.
(a) Redesign the circuit for the sorting operation using the shift registers instead of the counters to index registers $R_0, \ldots, R_3$.
(b) Give Verilog code for the circuit designed in part (*a*).
(c) Discuss the relative merits and drawbacks of your circuit in comparison with the circuit that uses the counters C_i and C_j.

10.10 Figure 10.42 shows a datapath circuit for the sorting operation that uses tri-state buffers to access the registers. Using the MAX+plusII Graphic Editor, draw the schematic in Figure 10.42. For the tri-state buffers, use the module named *tri* from the *Primitives* library. Use the *lpm_compare* module for the comparator subcircuit. Create the other necessary subcircuits using Verilog code and create graphical symbols that represent them. Describe the control circuit using Verilog code, create a graphical symbol for it, and connect this symbol to the

datapath modules in the schematic. Give a simulation result for your circuit implemented in a chip of your choosing. See Appendices B, C, and D for instructions on using the CAD tools.

10.11 Figure 10.40 gives Verilog code for the sorting circuit. Show how to modify this code to make use of a subcircuit that represents a $k \times n$ SRAM block. Use the *lpm_ram_dq* module for the SRAM block. Choose the synchronous SRAM option so that all changes to the SRAM contents are synchronized to the clock signal. (Hint: use the complement of the clock signal to synchronize the SRAM operations because this approach allows the Verilog code for the FSM shown in Figure 10.40 to be used without changes.)

10.12 Design a circuit that finds the log_2 of an operand that is stored in an n-bit register. Show all steps in the design process and state any assumptions made. Give Verilog code that describes your circuit.

10.13 Figure 10.33 shows a schematic for the circuit that computes the mean operation. Write Verilog code that represents this circuit. Use an array of registers instead of an SRAM block. For the *divider* subcircuit, use a shift operation that divides by four, instead of using the divider circuit designed in section 10.2.4.

10.14 The circuit designed in section 10.2.5 uses an adder to compute the sum of the contents of the registers. The divider subcircuit used to compute $M = Sum/k$ also includes an adder. Show how the circuit can be redesigned so that it contains only a single adder subcircuit that is used both for the summation operation and the division operation. Show only the extra circuitry needed to connect to the adder; and explain its operation.

10.15 Give Verilog code for the circuit designed in problem 10.14, including both the datapath and control circuits.

10.16 The pseudo-code for the sorting operation given in Figure 10.35 uses registers A and B to hold the contents of the registers being sorted. Show pseudo-code for the sorting operation that uses only register A to hold temporary data during the sorting operation. Give a corresponding ASM chart that represents the datapath and control circuits needed. Use multiplexers to interconnect the registers, in the style shown in Figure.10.37. Give a separate ASM chart that represents the control circuit.

10.17 Give Verilog code for the sorting circuit designed in problem 10.16.

10.18 In section 7.14.1 we showed a digital system with three registers, $R1$ to $R3$, and we designed a control circuit that can be used to swap the contents of registers $R1$ and $R2$. Give an ASM chart that represents this digital system and the swap operation.

10.19 (a) For the ASM chart derived in problem 10.18, show another ASM chart that specifies the required control signals to control the datapath circuit. Assume that multiplexers are used to implement the bus that connects the registers, as shown in Figure 7.65.
(b) Write complete Verilog code for the system in problem 10.18, including the control circuit described in part (a).
(c) Synthesize a circuit from the Verilog code written in part (b) and show a timing simulation that illustrates correct functionality of the circuit.

10.20 In section 7.14.2 we gave the design for a circuit that works as a processor. Give an ASM chart that describes the functionality of this processor.

10.21 (a) For the ASM chart derived in problem 10.20, show another ASM chart that specifies the required control signals to control the datapath circuit in the processor. Assume that multiplexers are used to implement the bus that connects the registers, $R0$ to $R3$, in the processor.
(b) Write complete Verilog code for the system in problem 10.20, including the control circuit described in part (*a*).
(c) Synthesize a circuit from the Verilog code written in part (*b*) and show a timing simulation that illustrates correct functionality of the circuit.

10.22 Consider the design of a circuit that controls the traffic lights at the intersection of two roads. The circuit generates the outputs $G1$, $Y1$, $R1$ and $G2$, $Y2$, $R2$. These outputs represent the states of the green, yellow, and red lights, respectively, on each road. A light is turned on if the corresponding output signal has the value 1. The lights have to be controlled in the following manner: when $G1$ is turned on it must remain on for a time period called t_1 and then be turned off. Turning off $G1$ must result in $Y1$ being immediately turned on; it should remain on for a time period called t_2 and then be turned off. When either $G1$ or $Y1$ is on, $R2$ must be on and $G2$ and $Y2$ must be off. Turning off $Y1$ must result in $G2$ being immediately turned on for the t_1 time period. When $G2$ is turned off, $Y2$ is turned on for the t_2 time period. Of course, when either $G2$ or $Y2$ is turned on, $R1$ must be turned on and $G1$ and $Y1$ must be off.
(a) Give an ASM chart that describes the traffic-light controller. Assume that two down-counters exist, one that is used to measure the t_1 delay and another that is used to measure t_2. Each counter has parallel load and enable inputs. These inputs are used to load an appropriate value representing either the t_1 or t_2 delay and then allow the counter to count down to 0.
(b) Give an ASM chart for the control circuit for the traffic-light controller.
(c) Write complete Verilog code for the traffic-light controller, including the control circuit from part (*a*) and counters to represent t_1 and t_2. Use any convenient clock frequency to clock the circuit and assume convenient count values to represent t_1 and t_2. Give simulation results that illustrate the operation of your circuit.

10.23 Assume that you need to use a single-pole single-throw switch as shown in Figure 10.48*a*. Show how a counter can be used as a means of debouncing the *Data* signal produced by the switch. (Hint: design an FSM that has *Data* as an input and produces the output z, which is the debounced version of *Data*. Assume that you have access to a *Clock* input signal with the frequency 102.4 kHz, which can be used as needed.)

10.24 Clock signals are usually generated using special purpose chips. One example of such a chip is the 555 programmable timer, which is depicted in Figure P10.2. By choosing particular values for the resistors R_a and R_b and the capacitor C_1, the 555 timer can be used to produce a desired clock signal. It is possible to choose both the period of the clock signal and its duty cycle. The term *duty cycle* refers to the percentage of the clock period for which the signal is high. The following equations define the clock signal produced by the chip

$$\text{Clock period} = 0.7(R_a + 2R_b)C_1$$

$$\text{Duty cycle} = \frac{R_a + R_b}{R_a + 2R_b}$$

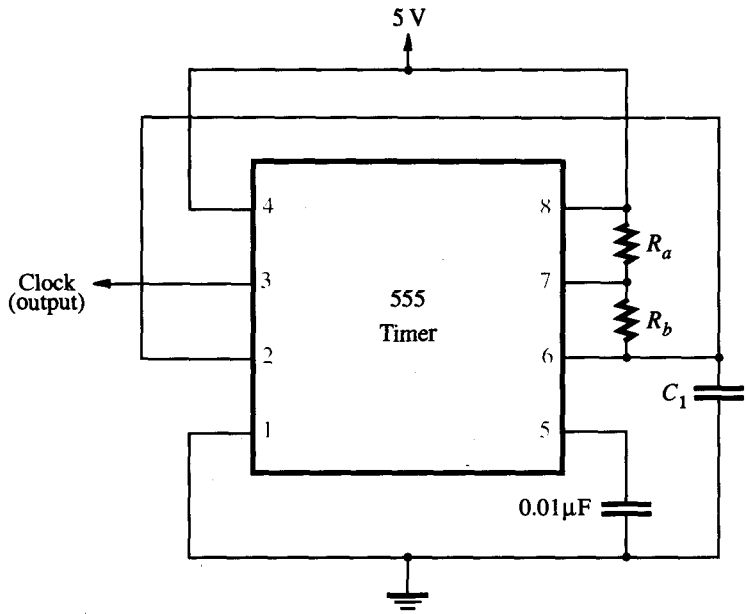


Figure P10.2 The 555 programmable timer chip.

(a) Determine the values of R_a, R_b, and C_1 needed to produce a clock signal with a 50 percent duty cycle and a frequency of about 500 kHz.
(b) Repeat part (*a*) for a duty cycle of 75 percent.

REFERENCES

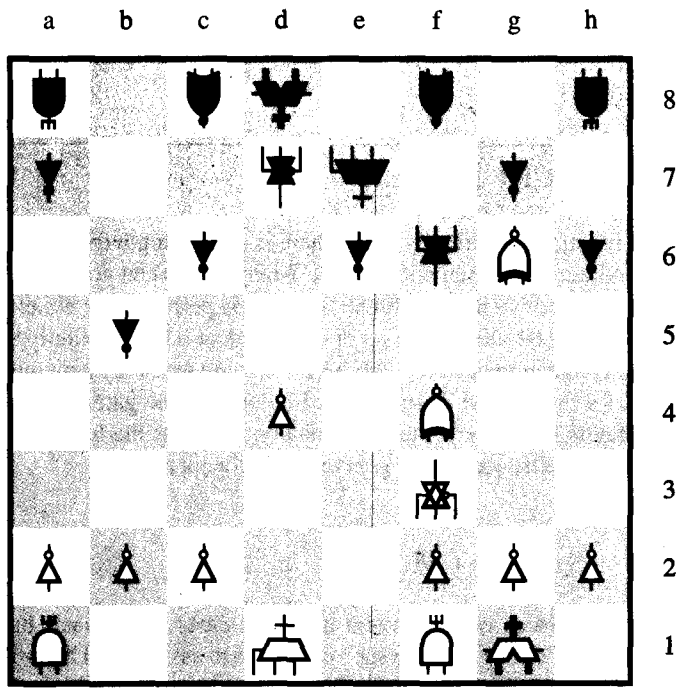
1. V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 5th ed. (McGraw-Hill: New York, 2001).
2. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design—The Hardware/Software Interface*, 2nd ed. (Morgan Kaufmann: San Francisco, CA, 1998).
3. D. D. Gajski, *Principles of Digital Design* (Prentice-Hall: Upper Saddle River, NJ, 1997).
4. M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals* (Prentice-Hall: Upper Saddle River, NJ, 1997).
5. J. P. Daniels, *Digital Design from Zero to One* (Wiley: New York, 1996).
6. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).

7. R. H. Katz, *Contemporary Logic Design* (Benjamin/Cummings: Redwood City, CA, 1994).
8. J. P. Hayes, *Introduction to Logic Design* (Addison-Wesley: Reading, MA, 1993).
9. C. H. Roth Jr., *Fundamentals of Logic Design*, 4th ed. (West: St. Paul, MN, 1993).
10. J. F. Wakerly, *Digital Design Principles and Practices* (Prentice-Hall: Englewood Cliffs, NJ, 1990).
11. C. J. Myers, *Asynchronous Circuit Design*, (Wiley: New York, 2001).

chapter

11

TESTING OF LOGIC CIRCUITS



11. Bc1–f4, b7–b5

In the previous chapters we have discussed the design of logic circuits. Following a sound design procedure, we expect that the designed circuit will perform as required. But how does one verify that the final circuit indeed achieves the design objectives? It is essential to ascertain that the circuit exhibits the required functional behavior and that it meets any timing constraints that are imposed on the design. We have discussed the timing issues in several places in the book. In this chapter we will discuss some testing techniques that can be used to verify the functionality of a given circuit.

There are several reasons for testing a logic circuit. When the circuit is first developed, it is necessary to verify that the designed circuit meets the required functional and timing specifications. When multiple copies of a correctly designed circuit are being manufactured, it is essential to test each copy to ensure that the manufacturing process has not introduced any flaws. It is also necessary to test circuits used in equipment that is installed in the field when it is suspected that there may be something wrong.

The basis of all testing techniques is to apply predefined sets of inputs, called *tests*, to a circuit and compare the outputs observed with the patterns that a correctly functioning circuit is supposed to produce. The challenge is to derive a relatively small number of tests that provide an adequate indication that the circuit is correct. The exhaustive approach of applying all possible tests is impractical for large circuits because there are too many possible tests.

11.1 FAULT MODEL

A circuit functions incorrectly when there is something wrong with it, such as a transistor fault or an interconnection wiring fault. Many things can go wrong, leading to a variety of faults. A transistor switch can break so that it is permanently either closed or open. A wire in the circuit can be shorted to V_{DD} or to ground, or it can be simply broken. There can be an unwanted connection between two wires. A logic gate may generate a wrong output signal because of a fault in the circuitry that implements the gate. Dealing with many different types of faults is cumbersome. Fortunately, it is possible to restrict the testing process to some simple faults, and obtain generally satisfactory results.

11.1.1 STUCK-AT MODEL

Most circuits discussed in this text use logic gates as the basic building blocks. A good model for representing faults in such circuits is to assume that all faults manifest themselves as some wires (inputs or outputs of gates) being permanently stuck at logic value 0 or 1. We indicate that a wire, w, has an undesirable signal that always corresponds to the logic value 0 by saying that w is *stuck-at-0*, which is denoted as $w/0$. If w has an undesirable signal that is always equal to logic 1, then w is *stuck-at-1*, which is denoted as $w/1$.

An obvious example of a stuck-at fault is when an input to a gate is incorrectly connected to a power supply, either V_{DD} or ground. But the stuck-at model is also useful for dealing with faults of other types, which often cause the same problems as if a wire were stuck at a particular logic value. The exact impact of a fault in the circuitry that implements a logic gate depends on the particular technology used. We will restrict our attention to the stuck-at faults and will examine the testing process assuming that these are the only faults that can occur.

11.1.2 SINGLE AND MULTIPLE FAULTS

A circuit can be faulty because it has either a single fault or possibly many faults. Dealing with multiple faults is difficult because each fault can occur in many different ways. A pragmatic approach is to consider single faults only. Practice has shown that a set of tests that can detect all single faults in a given circuit can also detect the vast majority of multiple faults.

A fault is detected if the output value produced by the faulty circuit is different from the value produced by the good circuit when an appropriate test is applied as input. Each test is supposed to be able to detect the occurrence of one or more faults. A complete set of tests used for a given circuit is referred to as the *test set.*

11.1.3 CMOS CIRCUITS

CMOS logic circuits present some special problems in terms of faulty behavior. The transistors may fail in permanently open or shorted (closed) state. Many such failures manifest themselves as stuck-at faults. But some produce entirely different behavior. For example, transistors that fail in the shorted state may cause a continuous flow of current from V_{DD} to ground, which can create an intermediate output voltage that may not be determined as either logic 0 or 1. Transistors failing in the open state may lead to conditions where the output capacitor retains its charge level because the switch that is supposed to discharge it is broken. The effect is that a combinational CMOS circuit starts behaving as a sequential circuit.

Specific techniques for testing of CMOS circuits are beyond the scope of this book. An introductory discussion of this topic can be found in references [1–3]. Testing of CMOS circuits has been the subject of considerable research [4–6]. We will assume that a test set developed using the stuck-at model will provide an adequate coverage of faults in all circuits.

11.2 COMPLEXITY OF A TEST SET

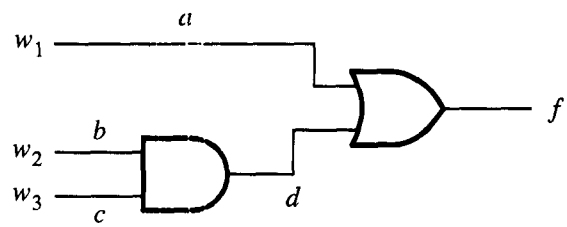
There is large difference in testing combinational and sequential circuits. Combinational circuits can be tested adequately regardless of their design. Sequential circuits present a much greater challenge because the behavior of a circuit under test is influenced not only by the tests that are applied to the external inputs but also by the states that the circuit is in when the tests are applied. It is very difficult to test a sequential circuit designed by a designer who does not take its testability into account. However, it is possible to design such circuits to make them more easily testable, as we will discuss in section 11.6. We will start by considering the testing of combinational circuits.

An obvious way to test a combinational circuit is to apply a test set that comprises all possible input valuations. Then it is only necessary to check if the output values produced by the circuit are the same as specified in a truth table that defines the circuit. This approach works well for small circuits, where the test set is not large, but it becomes totally impractical

for large circuits with many input variables. Fortunately, it is not necessary to apply all 2^n valuations as tests for an n-input circuit. A complete test set, capable of detecting all single faults, usually comprises a much smaller number of tests.

Figure 11.1*a* gives a simple three-input circuit for which we want to determine the smallest test set. An exhaustive test set would include all eight input valuations. This circuit involves five wires, labeled a, b, c, d, and f in the figure. Using our fault model, each wire can be stuck either at 0 or 1.

Figure 11.1*b* enumerates the utility of the eight input valuations as possible tests for the circuit. The valuation $w_1w_2w_3 = 000$ can detect the occurrence of a stuck-at-1 fault on wires a, d, and f. In a good circuit this test results in the output $f = 0$. However, if any of the faults $a/1$, $d/1$, or $f/1$ occurs, then the circuit will produce $f = 1$ when the input valuation 000 is applied. The test 001 causes $f = 0$ in the good circuit, and it results in



(a) Circuit

Test $w_1w_2w_3$	Fault detected									
	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$f/0$	$f/1$
000		√						√		√
001		√		√				√		√
010		√				√		√		√
011			√		√		√		√	
100	√								√	
101	√								√	
110	√								√	
111									√	

(b) Faults detected by the various input valuations

Figure 11.1 Fault detection in a simple circuit.

$f = 1$ if any of the faults $a/1$, $b/1$, $d/1$, or $f/1$ occurs. This test can detect the occurrence of four different faults. We say that it *covers* these faults. The last test, 111, can detect only one fault, $f/0$.

A minimal test set that covers all faults in the circuit can be derived from the table by inspection. Some faults are covered by only one test, which means that these tests must be included in the test set. The fault $b/1$ is covered only by 001. The fault $c/1$ is covered only by 010. The faults $b/0$, $c/0$, and $d/0$ are covered only by 011. Therefore, these three tests are essential. For the remaining faults there is a choice of tests that can be used. Selecting the tests 001, 010, and 011 covers all faults except $a/0$. This fault can be covered by three different tests. Choosing 100 arbitrarily, a complete test set for the circuit is

$$\text{Test set} = \{001, 010, 011, 100\}$$

The conclusion is that all possible stuck-at faults in this circuit can be detected using four tests, rather than the eight tests that would be used if we simply tried to test the circuit using its complete truth table.

The size of the complete test set for a given n-input circuit is generally much smaller than 2^n. But this size may still be unacceptably large in practical terms. Moreover, deriving the minimal test set is likely to be a daunting task for even moderately sized circuits. Certainly, the simple approach of Figure 11.1 is not practical. In the next section we will explore a more interesting approach.

Deriving a test set by considering the individual faults on all wires in a circuit, as done in section 11.2, is not attractive from the practical point of view. There are too many wires and too many faults to consider. A better alternative is to deal with several wires that form a path as an entity that can be tested for several faults using a single test. It is possible to activate a path so that the changes in the signal that propagates along the path have a direct impact on the output signal.

Figure 11.2 illustrates a path from input w_1 to output f, through three gates, which consists of wires a, b, c, and f. The path is activated by ensuring that other paths in the circuit do not determine the value of the output f. Thus the input w_2 must be set to 1 so that the signal at b depends only on the value at a. The input w_3 must be 0 so that it does

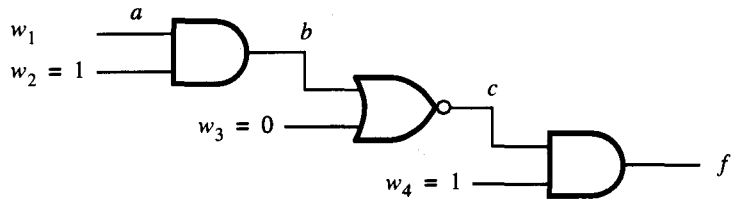


Figure 11.2 A sensitized path.

not affect the NOR gate, and w_4 must be 1 to not affect the AND gate. Then if $w_1 = 0$ the output will be $f = 1$, whereas $w_1 = 1$ will cause $f = 0$. Instead of saying that the path from w_1 to f is activated, a more specific term is used in technical literature, which says that the path is *sensitized*.

To sensitize a path through an input of an AND or NAND gate, all other inputs must be set to 1. To sensitize a path through an input of an OR or NOR gate, all other inputs must be 0.

Consider now the effect of faults along a sensitized path. The fault $a/0$ in Figure 11.2 will cause $f = 1$ even if $w_1 = 1$. The same effect occurs if the faults $b/0$ or $c/1$ are present. Thus the test $w_1w_2w_3w_4 = 1101$ detects the occurrence of faults $a/0$, $b/0$, and $c/1$. Similarly, if $w_1 = 0$, the output should be $f = 1$. But if any of the faults $a/1$, $b/1$, or $c/0$ is present, the output will be $f = 0$. Hence these three faults are detectable using the test 0101. The presence of any stuck-at fault along the sensitized path is detectable by applying only two tests.

The number of paths in a given circuit is likely to be much smaller than the number of individual wires. This suggests that it may be attractive to derive a test set based on the sensitized paths. This possibility is illustrated in the next example.

Example 11.1 **PATH-SENSITIZED TESTS** Consider the circuit in Figure 11.3. This circuit has five paths. The path $w_1 - c - f$ is sensitized by setting $w_2 = 1$ and $w_4 = 0$. It doesn't matter whether w_3 is 0 or 1, because $w_2 = 1$ causes the signal on wire b to be equal to 0, which forces $d = 0$ regardless of the value of w_3. Thus the path is sensitized by setting $w_2w_3w_4 = 1x0$, where the symbol x means that the value of w_3 does not matter. Now the tests $w_1w_2w_3w_4 =$ 01x0 and 11x0 detect all faults along this path. The second path, $w_2 - c - f$, is tested using 1000 and 1100. The path $w_2 - b - d - f$ is tested using 0010 and 0110. The tests for the path $w_3 - d - f$ are x000 and x010. The fifth path, $w_4 - f$, is tested with 0x00 and 0x01. Instead of using all ten of these tests, we can observe that the test 0110 serves also as the test 01x0, the test 1100 serves also as 11x0, the test 1000 serves also as x000, and the test 0010 serves also as x010. Therefore, the complete test set is

$$\text{Test set} = \{0110, 1100, 1000, 0010, 0x00, 0x01\}$$

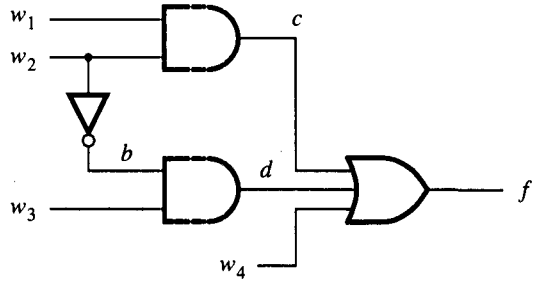


Figure 11.3 Circuit for Example 11.1.

While this approach is simpler, it is still impractical for large circuits. But the concept of path sensitizing is very useful, as we will see in the discussion that follows.

11.3.1 DETECTION OF A SPECIFIC FAULT

Suppose that we suspect that the circuit in Figure 11.3 has a fault where the wire b is stuck-at-1. A test that determines the presence of this fault can be obtained by sensitizing a path that propagates the effect of the fault to the output, f, where it can be observed. The path goes from b to d to f. It is necessary to set $w_3 = 1$, $w_4 = 0$, and $c = 0$. The latter can be accomplished by setting $w_1 = 0$. If b is stuck-at-1, then it is necessary to apply an input that would normally produce the value of 0 on the wire b, so that the output values in good and faulty circuits would be different. Hence w_2 must be set to 1. Therefore, the test that detects the $b/1$ fault is $w_1w_2w_3w_4 = 0110$.

In general, the fault on a given wire can be detected by propagating the effect of the fault to the output, sensitizing an appropriate path. This involves assigning values to other inputs of the gates along the path. These values must be obtainable by assigning specific values to the primary inputs, which may not always be possible. Example 11.2 illustrates the process.

Example 11.2

FAULT PROPAGATION As the effect of a fault propagates through the gates along a sensitized path, the polarity of the signal will change when passing through an inverting gate. Let the symbol D denote a stuck-at-0 fault in general. The effect of the stuck-at-0 fault will be unaltered when passed through an AND or OR gate. If D is on an input of an AND (OR) gate and the other inputs are set to 1 (0), then the output of the gate will behave as having D on it. But if D is on an input of a NOT, NAND, or NOR gate, then the output will appear to be stuck-at-1, which is denoted as $\overline{D}$.

Figure 11.4 shows how the effect of a fault can be propagated using the D and $\overline{D}$ symbols. Suppose first that there is a stuck-at-0 fault on wire b; that is, $b/0$. We want to propagate the effect of this fault along the path $b - h - f$. This can be done as indicated in Figure 11.4*b*. Setting $g = 1$ propagates the fault to the wire h. Then h appears to be stuck-at-1, denoted by $\overline{D}$. Next the effect is propagated to f by setting $k = 1$. Since the last NAND also inverts the signal, the output becomes equal to D, which is equivalent to $f/0$. Thus in a good circuit the output should be 1, but in a faulty circuit it will be 0. Next we must ascertain that it is possible to have $g = 1$ and $k = 1$ by assigning the appropriate values to the primary input variables. This is called the *consistency check*. By setting $c = 0$, both g and k will be forced to 1, which can be achieved with $w_3 = w_4 = 1$. Finally, to cause the propagation of the fault D on wire b, it is necessary to apply a signal that causes b to have the value 1, which means that either w_1 or w_2 has to be 0. Then the test $w_1w_2w_3w_4 = 0011$ detects the fault $b/0$.

Suppose next that the wire g is stuck-at-1, denoted by $\overline{D}$. We can try to propagate the effect of this fault through the path $g - h - f$ by setting $b = 1$ and $k = 1$. To make $b = 1$,

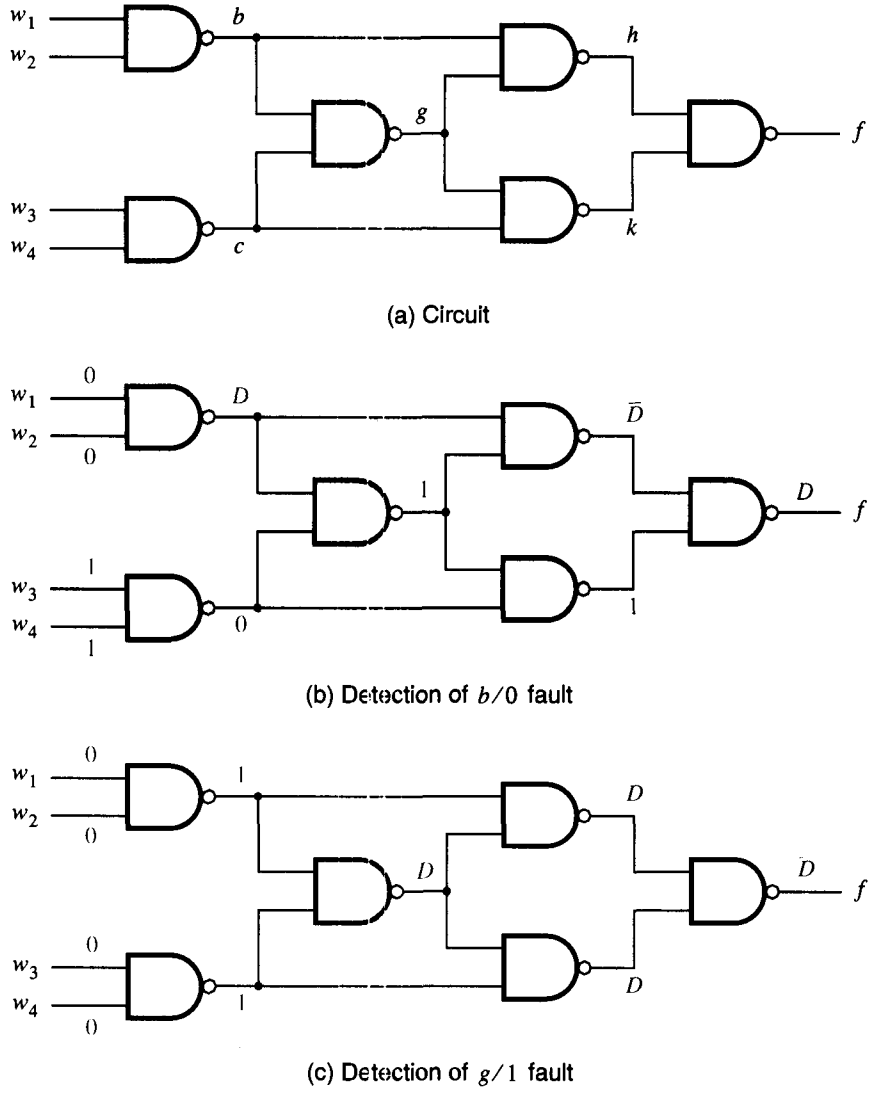


Figure 11.4 Detection of faults.

we set $w_1 = w_2 = 0$. To make $k = 1$, we have to make $c = 0$. But it is also necessary to cause the propagation of the $\bar{D}$ fault on g by means of a signal that makes $g = 0$ in the good circuit. This can be done only if $b = c = 1$. The problem is that at the same time we need $c = 0$, to make $k = 1$. Therefore, the consistency check fails, and the fault $g/1$ cannot be propagated in this way.

Another possibility is to propagate the effect of the fault along two paths simultaneously, as shown in Figure 11.4*c*. In this case the fault is propagated along the paths $g - h - f$ and

$g - k - f$. This requires setting $b = 1$ and $c = 1$, which also happens to be the condition needed to cause the propagation as explained above. The test 0000 achieves the desired objective of detecting $g/1$. Observe that if D (or $\overline{D}$) appears on both inputs of a NAND gate, the output value will be $\overline{D}$ (or D).

The idea of propagating the effect of faults using path sensitizing has been exploited in a number of methods for derivation of test sets for fault detection. The scheme illustrated in Figure 11.4 indicates the essence of the D-algorithm, which was one of the first practical schemes developed for fault detection purposes [7]. Other techniques have grown from this basic approach [8].

11.4 CIRCUITS WITH TREE STRUCTURE

Circuits with a treelike structure, where each gate has a fan-out of 1, are particularly easy to test. The most common forms of such circuits are the sum-of-products or product-of-sums. Since there is a unique path from each primary input to the output of the circuit, it is sufficient to derive the tests for faults on the primary inputs. We will illustrate this concept by means of the sum-of-products circuit in Figure 11.5.

If any input of an AND gate is stuck-at-0, this condition can be detected by setting all inputs of the gate to 1 and ensuring that the other AND gates produce 0. This makes $f = 1$ in the good circuit, and $f = 0$ in the faulty circuit. Three such tests are needed because there are three AND gates.

Testing for stuck-at-1 faults is slightly more involved. An input of an AND gate is tested for the stuck-at-1 fault by driving it with the logic value 0, while the other inputs of the gate have the logic value 1. Thus a good gate produces the output of 0, and a faulty

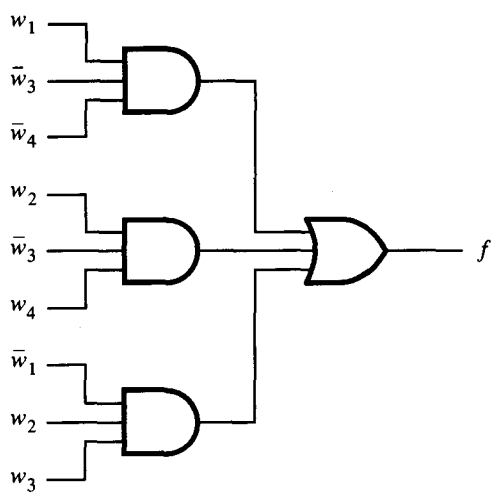


Figure 11.5 Circuit with a tree structure.

	No.	Product term			Test
		$w_1\overline{w}_3\overline{w}_4$	$w_2\overline{w}_3w_4$	$\overline{w}_1w_2w_3$	$w_1w_2w_3w_4$
Stuck-at-0 tests	1	1 1 1	0 1 0	0 0 0	1 0 0 0
	2	0 1 0	1 1 1	1 1 0	0 1 0 1
	3	0 0 0	1 0 1	1 1 1	0 1 1 1
Stuck-at-1 tests	4	0 1 1	1 1 0	1 1 0	0 1 0 0
	5	1 0 1	1 0 0	0 1 1	1 1 1 0
	6	1 1 0	0 1 1	0 0 0	1 0 0 1
	7	1 0 0	1 0 1	0 1 1	1 1 1 1
	8	0 0 0	0 0 1	1 0 1	0 0 1 1

Figure 11.6 Derivation of tests for the circuit in Figure 11.5.

gate generates 1. At the same time, the other AND gates must have the output of 0, which is accomplished by making at least one input of these gates equal to 0.

Figure 11.6 shows the derivation of the necessary tests. The first three tests are for the stuck-at-0 faults. Test 4 detects a stuck-at-1 fault on either the first input of the top AND gate or the third inputs of the other two gates. Observe that in each case the tested input is driven by logic 0, while the other inputs are equal to 1. This yields the test vector $w_1w_2w_3w_4 = 0100$. Clearly, it is useful to test inputs on as many gates as possible using a single test vector. Test 5 detects a fault on either the second input of the top gate or the first input of the bottom gate; it does not test any inputs of the middle gate. The required test pattern is 1110. Three more tests are needed to detect stuck-at-1 faults on the remaining inputs of the AND gates. Therefore, the complete test set is

$$\text{Test set} = \{1000, 0101, 0111, 0100, 1110, 1001, 1111, 0011\}$$

11.5 RANDOM TESTS

So far we have considered the task of deriving a *deterministic* test set for a given circuit, primarily relying on the path-sensitizing concept. In general, it is difficult to generate such test sets when circuits become larger. A useful alternative is to choose the tests at random, which we will explore in this section.

Figure 11.7 gives all functions of two variables. For an n-variable function, there are 2^{2^n} possible functions; hence there are $2^{2^2} = 16$ two-variable functions. Consider the XOR function, implemented as shown in Figure 11.8. Let us consider the possible stuck-at-0 and

w_1w_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 11.7 All two-variable functions.

stuck-at-1 faults on wires b, c, d, h, and k in this circuit. Each fault transforms the circuit into a faulty circuit that implements a function other than XOR, as indicated in Figure 11.9. To test the circuit, we can apply one or more input valuations to distinguish the good circuit from the possible faulty circuits listed in Figure 11.9. Choose arbitrarily $w_1w_2 = 01$ as the first test. This test will distinguish the good circuit, which must generate $f = 1$, from the faulty circuits that realize f_0, f_2, f_3, and f_{10}, because each of these would generate $f = 0$. Next, arbitrarily choose the test $w_1w_2 = 11$. This test distinguishes the good circuit from the faulty circuits that realize f_5, f_7, and f_{15}, in addition to f_3, which we have already tested for using $w_1w_2 = 01$. Let the third test be $w_1w_2 = 10$; it will distinguish the good circuit from f_4 and f_{12}. These three tests, chosen in a seemingly random way, detect all faulty circuits that involve the faults in Figure 11.9. Moreover, note that the first two tests distinguish seven of the nine possible faulty circuits.

This example suggests that it may be possible to derive a suitable test set by selecting the tests randomly. How effective can random testing be? Looking at Figure 11.7, we see that any of the four possible tests distinguishes the correct function from eight faulty functions, because they produce different output values for this input valuation. These eight faulty functions detectable by a single test are one-half of the total number of possible functions (2^{2^2-1} for the two-variable case). The test cannot distinguish between the correct function and the seven faulty functions that produce the same output value. The application of the second test distinguishes four of the remaining seven functions because they produce

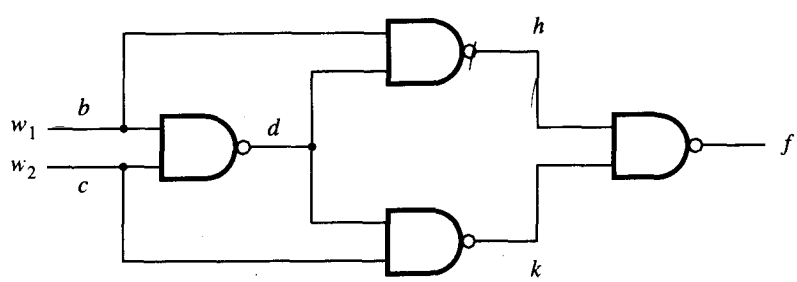


Figure 11.8 The XOR circuit.

Fault	Circuit implements
$b/0$	$f_5 = w_2$
$b/1$	$f_{10} = \overline{w}_2$
$c/0$	$f_3 = w_1$
$c/1$	$f_{12} = \overline{w}_1$
$d/0$	$f_0 = 0$
$d/1$	$f_7 = w_1 + w_2$
$h/0$	$f_{15} = 1$
$h/1$	$f_4 = \overline{w}_1 w_2$
$k/0$	$f_{15} = 1$
$k/1$	$f_2 = w_1 \overline{w}_2$

Figure 11.9 The effect of various faults.

an output value that is different from the correct function. Thus each application of a new test essentially cuts in half the number of faulty functions that have not been detected. Consequently, the probability that the first few tests will detect a large portion of all possible faults is high. More specifically, the probability that each faulty circuit can be detected by the first test is

$$P_1 = \frac{1}{2^{2^2} - 1} \cdot 2^{2^2-1} = \frac{8}{15} = 0.53$$

This is the ratio of the number of faulty circuits that produce an output value different from the good circuit, to the total number of faulty circuits.

This reasoning is readily extended to n-variable functions. In this case the first test detects 2^{2^n-1} out of a total of $2^{2^n} - 1$ possible faulty functions. Therefore, if m tests are applied, the probability that a faulty circuit will be detected is

$$P_m = \frac{1}{2^{2^n} - 1} \cdot \sum_{i=1}^{m} 2^{2^n-i}$$

This expression is depicted in graphical form in Figure 11.10. The conclusion is that random testing is very effective and that after a few tens of tests the existence of a fault is likely to be detected even in very large circuits.

Random testing works particularly well for circuits that do not have high fan-in. If fan-in is high, then it may be necessary to resort to other testing schemes. For example, suppose that an AND gate has a large number of inputs. Then there is a problem with detecting stuck-at-1 faults on its inputs, which may not be covered by random tests. But it is possible to test for these faults using the approach described in section 11.4.

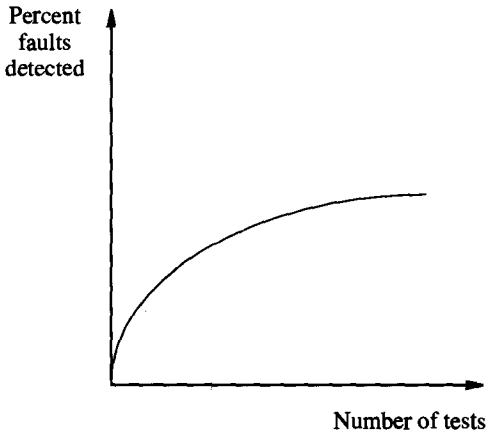


Figure 11.10 Effectiveness of random testing.

The simplicity of random testing is a very attractive feature. For this reason, coupled with good effectiveness of tests, this technique is often used in practice.

11.6 TESTING OF SEQUENTIAL CIRCUITS

As seen in the previous sections, combinational circuits can be tested effectively, using either deterministic or random test sets. It is much more difficult to test sequential circuits. The presence of memory elements allows a sequential circuit to be in various states, and the response of the circuit to externally applied test inputs depends on the state of the circuit.

A combinational circuit can be tested by comparing its behavior with the functionality specified in the truth table. An equivalent attempt would be to test a sequential circuit by comparing its behavior with the functionality specified in the state table. This entails checking that the circuit performs correctly all transitions between states and that it produces a correct output. This approach may seem easy, but in reality it is extremely difficult. A big problem is that it is difficult to ascertain that the circuit is in a specific state if the state variables are not observable on the external pins of the circuit, which is usually the case. Yet for each transition to be tested, it is necessary to verify with complete certainty that the correct destination state was reached. Such an approach may work for very small sequential circuits, but it is not feasible for practical-size circuits. A much better approach is to design the sequential circuits so that they are easily testable.

11.6.1 DESIGN FOR TESTABILITY

A synchronous sequential circuit comprises the combinational circuit that implements the output and next-state functions, as well as the flip-flops that hold the state information during a clock cycle. A general model for the sequential circuits is shown in Figure 8.88.

The inputs to the combinational network are the primary inputs, w_1 through w_n, and the present state variables, y_1 through y_k. The outputs of the network are the primary outputs, z_1 through z_m, and the next-state variables, Y_1 through Y_k. The combinational network could be tested using the techniques presented in the previous sections if it were possible to apply tests on all of its inputs and observe the results on all of its outputs. Applying the test vectors to the primary inputs poses no difficulty. Also, it is easy to observe the values on the primary outputs. The question is how to apply the test vectors on the present-state inputs and how to observe the values on the next-state outputs.

A possible approach is to include a two-way multiplexer in the path of each present-state variable so that the input to the combinational network can be either the value of the state variable (obtained from the output of the corresponding flip-flop) or the value that is a part of the test vector. A significant drawback of this approach is that the second input of each multiplexer must be directly accessible through external pins, which requires many pins if there are many state variables. An attractive alternative is to provide a connection that allows shifting the test vector into the circuit one bit at a time, thus trading off pin requirements for the time it takes to perform a test. Several such schemes have been proposed, one of which is described below.

Scan-Path Technique

A popular technique, called the *scan path*, uses multiplexers on flip-flop inputs to allow the flip-flops to be used either independently during normal operation of the sequential circuit, or as a part of a shift register for testing purposes. Figure 11.11 presents the general scan-path structure for a circuit with three flip-flops. A 2-to-1 multiplexer connects the D input of each flip-flop either to the corresponding next-state variable or to the serial path that connects all flip-flops into a shift register. The control signal $\overline{Normal}/Scan$ selects the active input of the multiplexer. During the normal operation the flip-flop inputs are driven by the next-state variables, Y_1, Y_2, and Y_3.

For testing purposes the shift-register connection is used to scan in the portion of each test vector that involves the present-state variables, y_1, y_2, and y_3. This connection has Q_i connected to D_{i+1}. The input to the first flip-flop is the externally accessible pin *Scan-in*. The output comes from the last flip-flop, which is provided on the *Scan-out* pin.

The scan-path technique involves the following steps:

1. The operation of the flip-flops is tested by scanning into them a pattern of 0s and 1s, for example, 01011001, in consecutive clock cycles, and observing whether the same pattern is scanned out.
2. The combinational circuit is tested by applying test vectors on $w_1w_2 \cdots w_ny_1y_2y_3$ and observing the values generated on $z_1z_2 \cdots z_mY_1Y_2Y_3$. This is done as follows:

    - The $y_1y_2y_3$ portion of the test vector is scanned into the flip-flops during three clock cycles, using $\overline{Normal}/Scan = 1$.
    - The $w_1w_2 \cdots w_n$ portion of the test vector is applied as usual and the normal operation of the sequential circuit is performed for one clock cycle, by setting $\overline{Normal}/Scan = 0$. The outputs $z_1z_2 \cdots z_m$ are observed. The generated values of $Y_1Y_2Y_3$ are loaded into the flip-flops at this time.
    - The select input is changed to $\overline{Normal}/Scan = 1$, and the contents of the flip-flops are scanned out during the next three clock cycles, which makes the

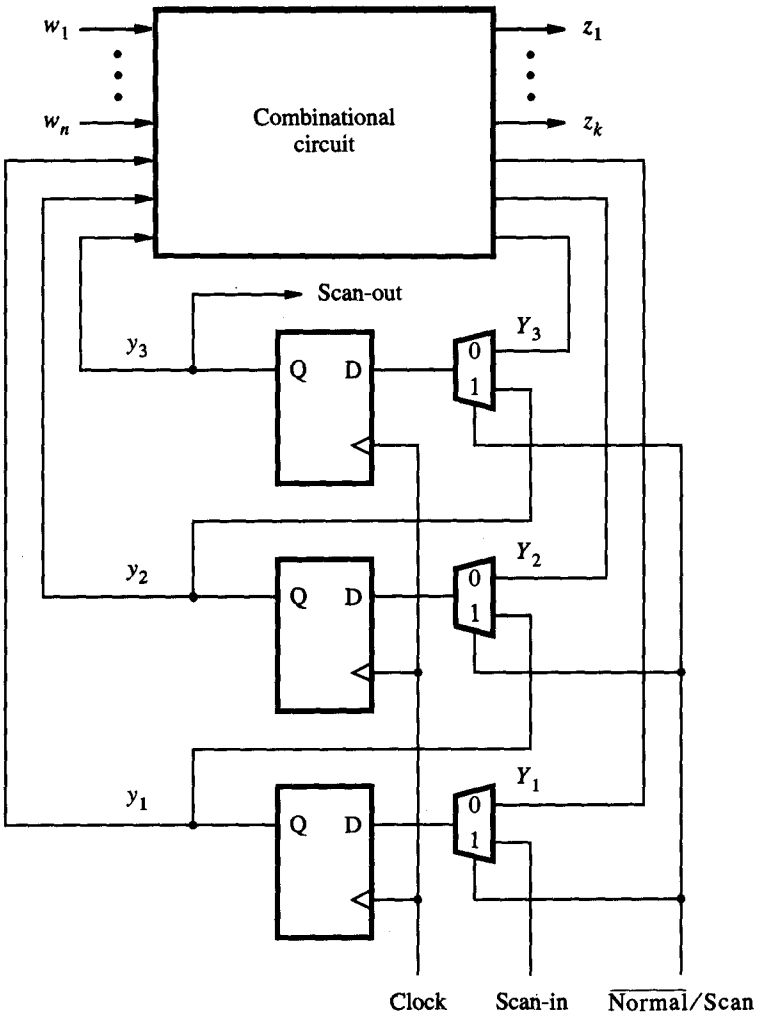


Figure 11.11 Scan-path arrangement.

$Y_1Y_2Y_3$ portion of the test result observable externally. At the same time, the next test vector can be scanned in to reduce the total time needed to test the circuit.

The next example shows a specific circuit that is designed for scan-path testing.

Example 11.3

Figure 8.78 shows a circuit that recognizes a specific input sequence, which was discussed in section 8.9. The circuit can be made easily testable by modifying it for scan path as shown in Figure 11.12. The combinational part, consisting of four AND and two OR gates, is the same in both figures.

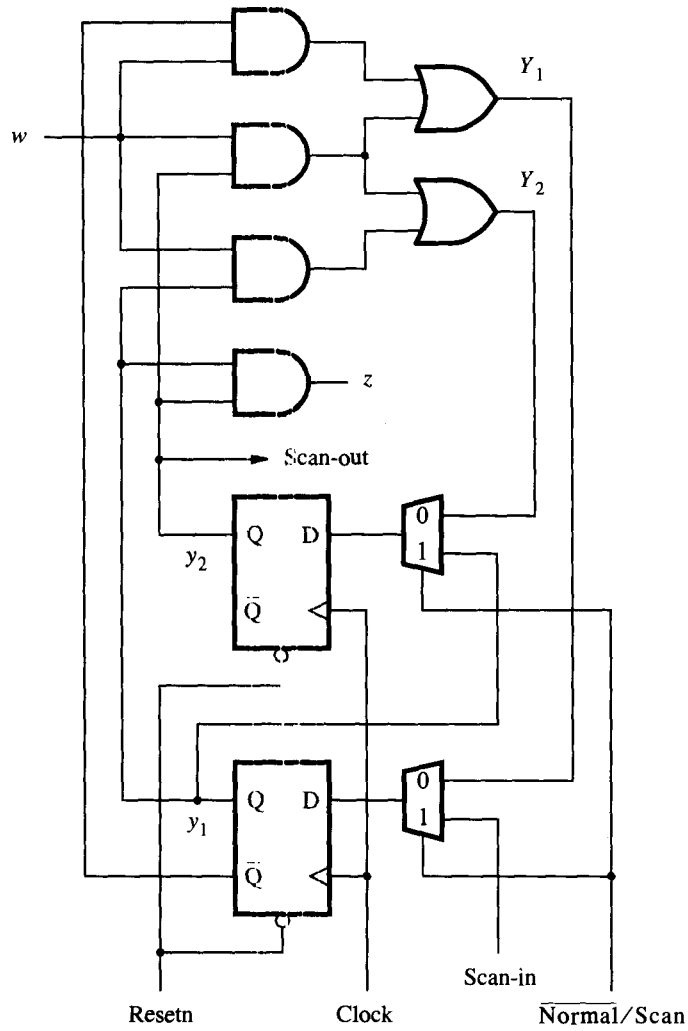


Figure 11.12 Circuit for Example 11.3.

The flip-flops can be tested by scanning through them a sequence of 0s and 1s as explained above. The combinational circuit can be tested by applying test vectors on w, y_1, and y_2. Let us use the random-testing approach, choosing arbitrarily four test vectors $wy_1y_2 = 001$, 110, 100, and 111. To apply the first test vector, the pattern $y_1y_2 = 01$ is scanned into the flip-flops during two clock cycles. Then for one clock cycle, the circuit is made to operate in the normal mode with $w = 0$. This essentially applies the vector $wy_1y_2 = 001$ to the AND-OR circuit. The result of this test should be $z = 0$, $Y_1 = 0$, and $Y_2 = 0$. The value of z can be observed directly. The values of Y_1 and Y_2 are loaded into the respective flip-flops, and they are scanned out during the next two clock cycles. As these

values are being scanned out, the next test pattern $y_1y_2 = 10$ can be scanned in. Thus it takes five cycles to perform one test, but the last two cycles are overlapped with the second test. The third and fourth tests are performed in the same way. The total time needed to perform all four tests is 14 clock cycles.

The preceding approach is based on testing a sequential circuit by testing its combinational part using the techniques developed in the previous sections. The scan-path facility makes it also possible to test the sequential circuit by making it go through all transitions specified in the state table. The circuit can be placed into a given state simply by scanning into the flip-flops the valuation of the state variables that denotes this state. The result of the transition can be checked by observing the primary outputs and by scanning out the valuation that presents the destination state. We leave it to the reader to develop the details of this approach (see problem 11.16).

One limitation of the scan-path technique is that it does not work well if the asynchronous preset and reset features of the flip-flops are used during normal operation. We have already suggested that it is better to use synchronous preset and reset. If the designer wishes to use the asynchronous preset and reset capability, then a testable circuit can be designed using techniques such as the *level-sensitive scan design* [1, 9]. The reader can consult the references for a description of this technique.

11.7 BUILT-IN SELF-TEST

Until now we have assumed that testing of logic circuits is done by externally applying the test inputs and comparing the results with the expected behavior of the circuit. This requires connecting external equipment to the circuit under test. An interesting question is whether it is possible to incorporate the testing capability within the circuit itself so that no external equipment is needed. Such built-in capability would allow the circuit to be self-testable. This section presents a scheme that provides the *built-in self-test* (BIST) capability.

Figure 11.13 shows a possible BIST arrangement in which a test vector generator produces the test vectors that must be applied to the circuit under test. In section 11.5 we explained that randomly chosen test vectors give good results, with the fault coverage depending on the number of tests performed. For each test vector applied to the circuit, it is

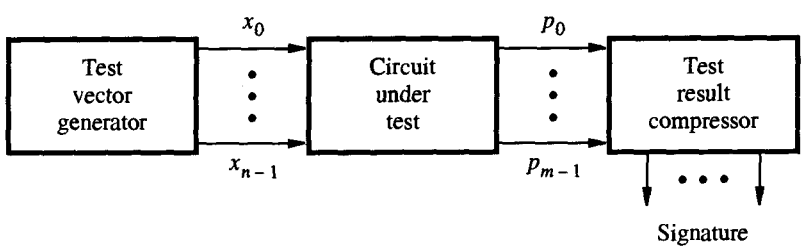
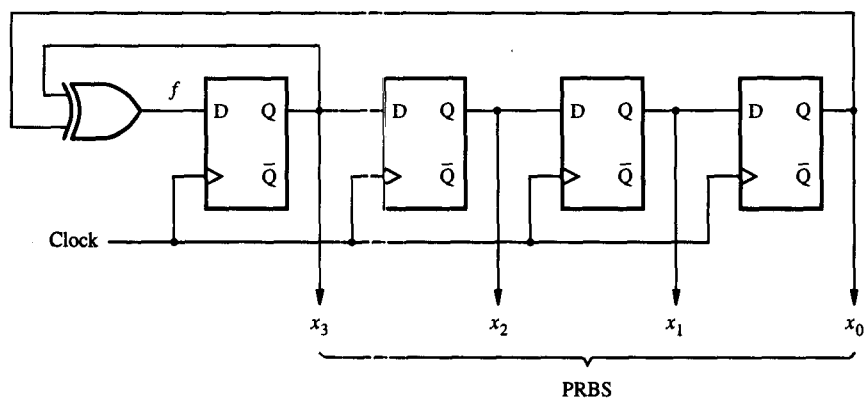


Figure 11.13 The testing arrangement.

necessary to determine the required response of the circuit. The response of a good circuit may be determined using the simulator tool of a CAD system. The expected responses to the applied tests must be stored on the chip so that a comparison can be made when the circuit is being tested.

A practical approach for generating the test vectors on-chip is to use *pseudorandom* tests, which have the same characteristics as random tests but are produced deterministically and can be repeated at will. The generator for pseudorandom tests is easily constructed using a feedback shift-register circuit. A small example of a possible generator is given in Figure 11.14. A four-bit shift register, with the signals from the first and fourth stages fed back through an XOR gate, generates 15 different patterns during successive clock cycles. If the shift register is set at the beginning to $x_3x_2x_1x_0 = 1000$, then the generated patterns are as shown in part (*b*) of the figure. Observe that the pattern 0000 cannot be used, because the circuit would be locked in this pattern indefinitely.

The circuit in Figure 11.14 is representative of a class of circuits known as *linear feedback shift registers* (LFSRs). Using feedback from the various stages of an n-bit shift



(a) Circuit

x_3	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	···
x_2	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	0	···
x_1	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	0	···
x_0	0	0	0	1	1	1	1	0	1	0	1	1	0	0	1	0	···
f	1	1	1	0	1	0	1	1	0	0	1	0	0	0	1	1	···

(b) Generated sequence

Figure 11.14 Pseudorandom binary sequence generator (PRBSG).

register, connected to the first stage by means of XOR gates, it is possible to generate a sequence of $2^n - 1$ patterns that have the characteristics of randomly generated numbers. Such circuits are used extensively in error-correcting codes. The theory of operation of these circuits is presented in a number of books [1–3, 10]. A table of the feedback connections for various values of n, which generate a maximum-length pseudorandom sequence, is given in Peterson and Weldon [11].

The pseudorandom binary sequence generator (PRBSG) gives a simple method of generating tests. The required response of the circuit under test can be determined by using the simulator tool of the CAD system. The remaining question is how to check whether the circuit indeed produces the required response. It is not attractive to have to store a large number of responses to the tests on a chip that also includes the main circuit. A practical solution is to compress the results of the tests into a single pattern. This can be done using an LFSR circuit. Instead of just providing the feedback signals as the input, a compressor circuit includes the output signals produced by the circuit under test. Figure 11.15 shows a single-input compressor circuit (SIC), which uses the same feedback connections as the PRBSG of Figure 11.14. The input p is the output of a circuit under test. After applying a number of test vectors, the resulting values of p drive the SIC and, coupled with the LFSR functionality, produce a four-bit pattern. The pattern generated by the SIC is called a *signature* of the tested circuit for the given sequence of tests. The signature represents a single pattern that may be interpreted as a result of all the applied tests. It can be compared against a predetermined pattern to see if the tested circuit is working properly. Storing a single n-bit pattern for comparison purposes presents only a small overhead. The randomizing nature of the compressor circuits based on LFSRs provides a good coverage of patterns that may result from a faulty circuit [12].

If the circuit under test has more than one output, then an LSFR with multiple inputs can be used. Figure 11.16 illustrates how four inputs, p_0 through p_3, can be added to the basic circuit of Figure 11.14. Again the four-bit signature provides a good mechanism for distinguishing among different sequences of four-bit patterns that may appear on the inputs of this multiple-input compressor circuit (MIC).

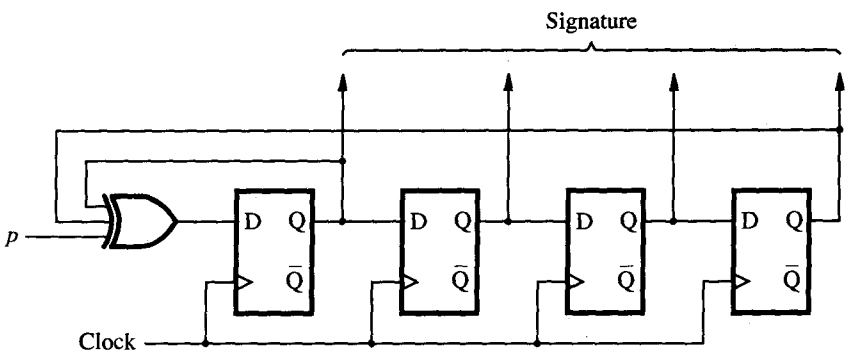


Figure 11.15 Single-input compressor circuit (SIC).

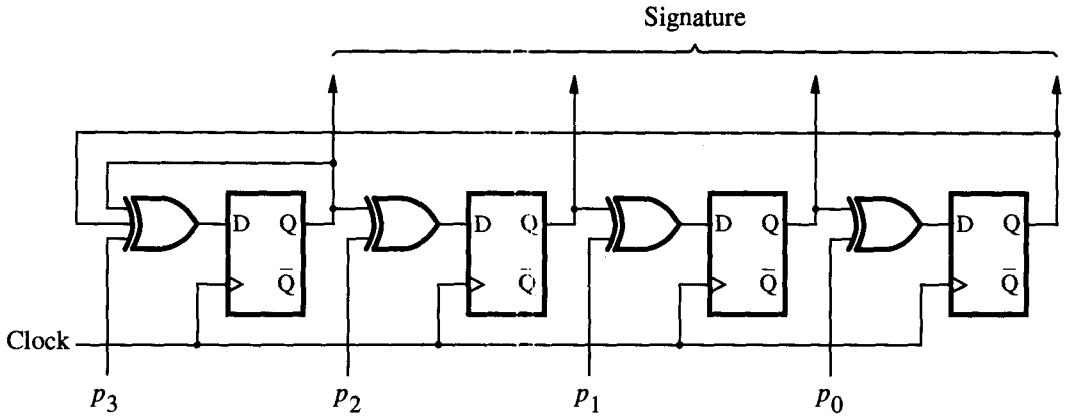


Figure 11.16 Multiple-input compressor circuit (MIC).

A complete BIST scheme for a sequential circuit may be implemented as indicated in Figure 11.17. The scan-path approach is used to provide a testable circuit. The test patterns that would normally be applied on the primary inputs $W = w_1 w_2 \cdots w_n$ are generated internally as the patterns on $X = x_1 x_2 \cdots x_n$. Multiplexers are needed to allow switching from W to X, as inputs to the combinational circuit. A pseudorandom binary sequence

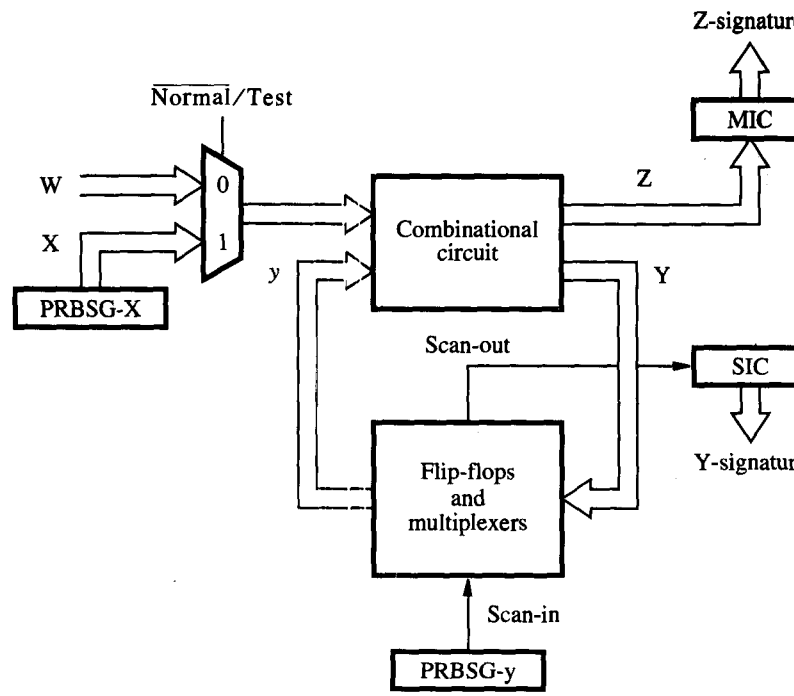


Figure 11.17 BIST in a sequential circuit.

generator, PRBSG-X, generates the test patterns for X. The portion of the tests applied via the next-state signals, y, is generated by the second PRBS generator, PRBSG-y. These patterns are scanned into the flip-flops as explained in section 11.6.

The test outputs are compressed using the two compressor circuits. The patterns on the primary outputs, $Z = z_1 z_2 \cdots z_m$, are compressed using the MIC circuit, and those on the next-state wires $Y = Y_1 Y_2 \cdots Y_k$, by the SIC circuit. These circuits produce the Z-signature and Y-signature, respectively. The testing procedure is the same as given in Example 11.3 except that the comparison with the test result that a good circuit is supposed to give is done only once; at the end of the testing process the two signatures are compared with the stored patterns. Figure 11.17 does not show the circuitry needed to store these patterns and perform the comparison. Instead of storing the signature patterns of the required results as a part of the designed circuit, it is possible to shift out the contents of MIC and SIC shift registers onto two output pins and to perform the necessary comparison with the expected signatures externally. Note that using signature testing in this way reduces the testing time significantly, compared to the time it would take to test the circuit by scanning out the results of individual tests and comparing them with predetermined patterns.

The effectiveness of the BIST approach depends on the length of the LFSR generator and compressor circuits. Longer shift registers give better results [13]. One reason for failing to detect that the circuit under test may be faulty is that the pseudorandomly generated tests do not have perfect coverage of all possible faults. Another reason is that a signature generated by compressing the outputs of a faulty circuit may coincidentally end up being the same as the signature of the good circuit. This can occur because the compression process results in a loss of some information, such that two distinct output patterns may be compressed into the same signature. This is known as the *aliasing* problem.

11.7.1 BUILT-IN LOGIC BLOCK OBSERVER

The essence of BIST is to have internal capability for generation of tests and for compression of the results. Instead of using separate circuits for these two functions, it is possible to design a single circuit that serves both purposes. Figure 11.18 shows the structure of a possible circuit, known as the *built-in logic block observer* (BILBO) [14]. This four-bit circuit has the same feedback connections as the circuit of Figure 11.14.

The BILBO circuit has four modes of operation, which are controlled by the mode bits, M_1 and M_2. The modes are as follows:

- $M_1M_2 = 11$ — Normal system mode in which all flip-flops are independently controlled by the signals on inputs p_0 through p_3. In this mode each flip-flop may be used to implement a state variable of a finite state machine by using p_0 to p_3 as y_0 to y_3.
- $M_1M_2 = 00$ — Shift-register mode in which the flip-flops are connected into a shift register. This mode allows test vectors to be scanned in, and the results of applied tests to be scanned out, if the control input $\overline{G}/S$ is equal to 1. If $\overline{G}/S = 0$, then the circuit acts as the PRBS generator.
- $M_1M_2 = 10$ — Signature mode in which a series of patterns applied on inputs p_0 through p_3 are compressed into a signature available as a pattern on q_0 through q_3.
- $M_1M_2 = 01$ — Reset mode in which all flip-flops are reset to 0.

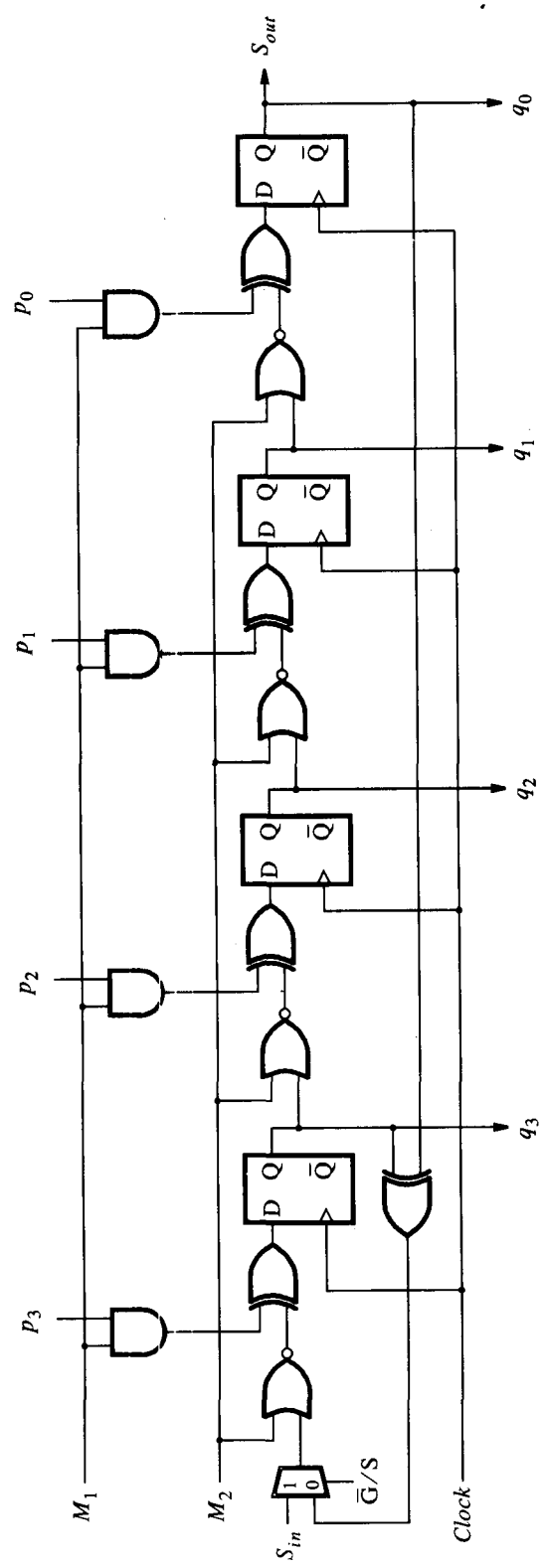


Figure 11.18 A four-bit built-in logic block observer (BILBO).

An efficient way of using BILBO circuits is presented in Figure 11.19. A combinational circuit can be tested by partitioning it into two (or more) parts. A BILBO circuit is used to provide inputs to one part and to accept outputs from the other part. The testing process involves a two-phase approach. First, BILBO1 is used as a PRBS generator that provides test patterns for combinational network 1 (CN1). During this time BILBO2 acts as a compressor and produces a signature for the test. The signature is shifted out by placing BILBO2 into the shift-register mode. Next, the roles of BILBO1 and BILBO2 are reversed, and the process is repeated to test CN2.

The detailed steps in the testing process are

1. Scan the initial test pattern into BILBO1 and reset all flip-flops in BILBO2.
2. Use BILBO1 as the PRBS generator for a given number of clock cycles and use BILBO2 to produce a signature.
3. Scan out the contents of BILBO2 and externally compare the signature; then scan into it the initial test pattern for testing CN2. Reset the flip-flops in BILBO1.
4. Use BILBO2 as the PRBS generator for a given number of clock cycles and use BILBO1 to produce a signature.
5. Scan out the signature in BILBO1 and externally compare it with the required pattern.

The BILBO circuits are used in this way for testing purposes. At other times the normal system mode is used.

11.7.2 SIGNATURE ANALYSIS

We have explained the use of signatures in the context of implementing an efficient built-in testing mechanism. The main idea of compressing a long sequence of test results into a single signature was originally developed as the basis for an instrument manufactured by Hewlett-Packard in the 1970s, known as the Signature Analyzer [15]. Thus the name *signature analysis* was coined to refer to the testing schemes that use signatures to represent the results of applied tests.

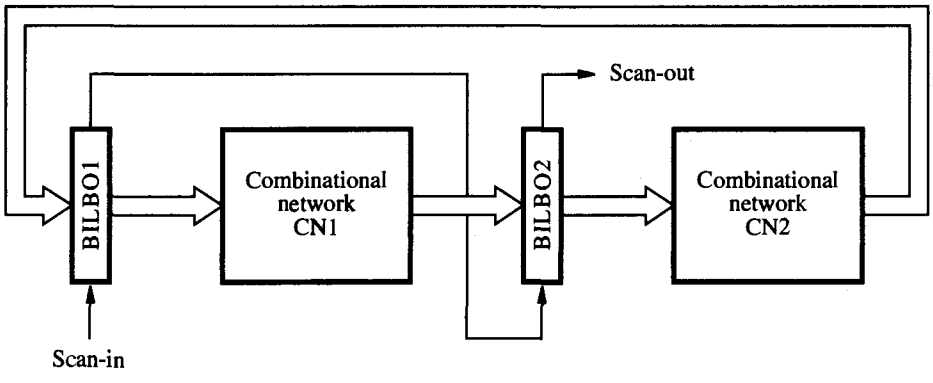


Figure 11.19 Using BILBO circuits for testing.

Signature analysis is particularly suitable for digital systems that naturally include an ability to generate the desired test patterns. Such is the case with computer systems in which various parts of the system can be stimulated by test patterns produced under software control.

11.7.3 BOUNDARY SCAN

The testing techniques discussed in the previous sections are equally applicable to circuits that are implemented on single chips or on printed circuit boards that contain a number of chips. A circuit can be tested only if it is possible to apply the tests to it and observe the outputs produced. This involves having access to the primary inputs and outputs.

When chips are soldered onto a printed circuit board, it often becomes impossible to attach test probes to pins. This hinders the testing process unless some indirect access to the pins is provided. The scan-path concept can be extended to the board level to deal with the problem. Suppose that each primary input or output pin on a chip is connected through a D flip-flop and that a provision is made for a test mode in which all flip-flops can be connected into a shift register. Then the test information can be scanned in and scanned out using the shift-register path, via two pins that serve as serial input and output. Connecting the serial output pin of one chip to the serial input pin of another chip results in the pins of all chips being connected into a board-wide shift register for testing purposes. This approach has become popular in practice and has been embodied into the IEEE Standard 1149.1 [16].

11.8 PRINTED CIRCUIT BOARDS

Design and testing techniques presented in this book can be applied to any logic circuit, whether the circuit is realized on a single chip or its implementation involves a number of chips placed on a printed circuit board (PCB). In this section we discuss some practical issues that arise when one or more circuits that form a larger digital system are implemented on a PCB.

A typical PCB contains multiple layers of wiring. When the board is manufactured, the wiring pattern on each layer is generated. The layers are separated by insulating material and pressed together in sandwichlike fashion to form the board. Connections between different wiring levels are made through holes that are provided for this purpose. Chips and other components are then soldered to the top and possibly to the bottom layers.

In preceding chapters we have discussed in considerable detail the CAD tools used for designing circuits that can be implemented on a single chip, such as a PLD. For a multiple-chip implementation, we need a different set of CAD tools to design a PCB that incorporates the chips and connections needed to realize the complete digital system. Such tools are available from a number of companies, for example, Cadence Design Systems and Mentor Graphics. These tools can automatically determine where each chip should be placed on the PCB, but the designer can also specify the location of particular chips. This is called the *placement* process. Given a specific placement of chips and other components (such as connectors and capacitors), the tools generate a layout for each layer of wiring traces that provide the required connections on the board. This process is referred to as *routing*.

Again the designer can intervene and manually route some connections. However, since the number of connections can be in the tens of thousands, it is crucial to obtain a good automated solution.

In addition to the design issues discussed in the previous chapters, a large circuit implemented on a PCB is subject to some other constraints. Signals on the wiring traces may be affected by noise problems caused by crosstalk, spikes in the power supply voltage, and reflections from the end points of long traces.

Crosstalk

Two closely spaced wires that run parallel to each other are capacitively coupled, and a pulse on one wire can induce a similar (but usually much smaller) pulse on the adjoining wire. This is referred to as *crosstalk*. Its existence is undesirable because it contributes to noise problems.

When drawing timing diagrams, we usually draw ideal waveforms with sharp edges, which have well-defined voltage levels for the logic values 0 and 1. In an actual circuit the corresponding signals may depart significantly from the desired behavior. As explained in section 3.8.4, noise in a circuit can affect voltage levels, which can be troublesome. For example, if at some point in time the noise diminishes the value of a signal that should be at logic 1 to a level where this signal is interpreted by the next gate as being logic 0, then a malfunction in the circuit is likely to occur. Since the noise effects tend to be random, they are often difficult to detect.

To minimize crosstalk, it is prudent to avoid having long wires running parallel in close proximity to each other. This may be difficult to achieve because of limited space on a PCB and the need to provide a large number of wires. Using additional layers (planes) of wiring helps in coping with crosstalk problems.

Power Supply Noise

When a CMOS circuit changes its state, there is a momentary flow of current in the circuit, which is manifested as a current pulse on the power supply (V_{DD} and *Ground*) wires. Since a wiring trace on a PCB has a small "line inductance," such a current pulse causes a voltage spike (short pulse) on these lines. The cumulative effect of such voltage spikes can cause a malfunction of the circuit.

The induced voltage spikes can be reduced significantly by connecting a small capacitor between the V_{DD} and *Ground* wires, in close proximity to the chip that causes the spikes to occur. Since these spikes have the characteristic of a very high frequency signal, the path through the capacitor is essentially a short circuit for them. Thus the voltage spikes "bypass" the power supply lines and do not affect other chips connected to the same lines. Such capacitors are called *bypass capacitors*. They do not affect the DC voltage on the power supply lines.

Large chips, such as PLDs, often require more than one V_{DD} and *Ground* connection. In this case it is advisable to use one bypass capacitor for each pair of V_{DD} and *Ground* pins on the chip. For example, with PLDs the manufacturers recommend using a 0.2 μF capacitor for each such pair of pins, placed as close as possible to the PLD chip.

Reflections and Terminations

Wiring traces on a PCB act as simple wires in circuits when the clock frequency is low. However, at higher clock frequencies it becomes necessary to worry about so-called

transmission-line effects. When a signal propagates along a long wire, it is attenuated due to the small resistance of the wire, it picks up crosstalk that manifests itself as noise, and it may be reflected when it reaches the end of the wire. The reflection causes a problem if its effect does not die down before the next active clock edge. The discussion of transmission-line effects is beyond the scope of this book. We will only mention that the reflection of signals can be prevented by placing a suitable "termination" component on the line. This termination can be as simple as a resistor whose resistance matches the apparent resistance of the line, known as the *characteristic impedance* of the line. Other forms of termination are also possible. For details of such schemes, the reader may consult other references [17–18].

11.8.1 TESTING OF PCBS

The manufactured PCB has to be tested thoroughly. Flaws in the manufacturing process may cause some connections to be broken and others to be shorted by a solder blob that touches two adjacent wires. There may be problems caused by design errors that were not discovered during the design process. Finally, some chips and other components on the PCB may be defective.

Power Up

The first step is to turn on the power supply. In the worst case this may cause some chip to be destroyed because of a fatal short-circuit condition (in an extreme case a chip package may actually blow apart). Assuming that this is not the case, it is essential to check if any of the chips is becoming inordinately hot. Overheating is a symptom of a serious problem that must be corrected.

It is also necessary to check that the power and ground connections are properly made on each chip and that the voltage level is as specified.

Reset

The next step is to reset all circuitry on the PCB to reach a predetermined starting point. This typically implies resetting the flip-flops, which is usually achieved by asserting a common reset line. It is important to verify that the starting state is correctly established.

Low-Level Functional Testing

Since practical circuits can be extremely complex, it is prudent to test the basic functionality first. A key test is to verify that the control signals are working correctly.

Using the divide-and-conquer approach, simple functions are tested first, followed by the more complex ones.

Full Functional Testing

Having verified the operation of smaller subcircuits, it is necessary to test the functionality of the entire system on the PCB. The number of errors often depends on the thoroughness of the simulation done during the design process. In general, it is difficult to simulate large digital systems fully, so some errors are likely to be found on the PCB. Typical errors are due to

- Manufacturing errors, such as wrong wiring traces, blown components, or incorrect power supply voltage.
- Incorrect specifications.
- Designer's misinterpretation of information on the data sheets that describe some chips.
- Incorrect information on the data sheets provided by the chip manufacturer.

As mentioned earlier, PCBs contain multiple layers of wiring. Each layer may have several thousands of wires in it. Finding and fixing errors can be a difficult and time-consuming task, especially if errors involve wires in internal (as opposed to the top or bottom) wiring layers.

Timing

It is next necessary to verify the timing of the circuit. A good strategy is to start with a slow clock. If the circuit works properly, then the clock frequency is gradually increased until the required operating frequency is reached.

Timing problems arise because of propagation delays through various paths in a circuit. These delays are caused by the logic gates and the wiring that interconnects them. It is essential to ensure that all data inputs to flip-flops in the circuit are stable before the active edge of the clock signal arrives, as required by the setup time.

Reliability

A digital system is expected to operate reliably for a long time. Its reliability may be affected by several factors, such as timing, noise, and crosstalk problems.

The timing of signals has to provide some safety margin to allow for small changes in propagation delays. If the timing is too tight, then it is likely that the circuit will operate correctly for some period of time, but will eventually fail because of a timing error. The timing of chips may change with temperature, so failures can occur if thermal constraints are not adhered to. Cooling is usually provided by means of fans.

11.8.2 INSTRUMENTATION

Testing of circuits implemented in PCBs requires some specialized instruments.

Oscilloscope

The details of individual signals can be examined using an oscilloscope. This instrument displays the voltage waveform of a signal, showing the potential problems with respect to propagation delay and noise. The waveform displayed on an oscilloscope shows the actual voltage levels of the signal; it does not depict the simplified view of ideal waveforms that have perfectly square edges. If the user wants to see only the logic values of a signal (0 or 1), then a different type of instrument called a logic analyzer can be used.

Logic Analyzer

While an oscilloscope allows simultaneous examination of a few signals, a logic analyzer allows examination of tens or even hundreds of signals at the same time. It takes inputs from a set of points in the circuit, by means of probes attached to these points, and

digitizes and displays the detected signals in the form of waveforms on a screen. A powerful feature of the logic analyzer is that it has internal storage capable of recording a sequence of changes in the signals over a substantial period of time. Then any segment of this information can be displayed as desired by the operator. Typically, it is possible to record a few milliseconds' worth of events, which involves many cycles of a normal digital clock.

Looking at the waveforms taken when the circuit under test is working properly is not helpful in the debugging process. It is essential to see the waveforms generated when a malfunction takes place. The logic analyzer can be "triggered" to record a window of events that occurred before and after the trigger event. The user must specify the trigger event. For example, suppose that a malfunction is suspected to be caused by two control signals, A and B, being asserted at the same time, even though the design specification requires that these signals be mutually exclusive. A useful trigger point can then be established as the time when the AND of A and B has the value 1. Finding suitable trigger events can be difficult, and the user must rely on intuition and experience.

To use a logic analyzer effectively, it must be possible to connect the probes to some useful (for testing purposes) points in the circuit. Thus it is important to provide such "test" points when a PCB is being designed.

11.9 CONCLUDING REMARKS

Manufactured products must be tested to ensure that they perform as expected. All of the techniques discussed in this chapter are relevant for this type of testing. The development of tests and the required responses is based on the assumption that the circuits are designed correctly. Thus it is the validity of the physical implementation that is being tested.

Another aspect of testing occurs during the design process. The designer has to ascertain that the designed circuit meets the specifications. From the testing point of view, this poses a significant problem because there exists no provably good circuit that can be used to generate the desired tests. CAD tools are helpful in deriving tests for a designed circuit, but they cannot determine whether the circuit is indeed what the designer intended to achieve in terms of its functionality. A design error usually results in a circuit that has somewhat different functionality than required by the specification.

Small circuits can be tested fully to verify their functionality. A combinational circuit can be tested to see if it performs according to its truth table. A sequential circuit can be tested by checking the transitions specified in the state table. This is much easier to do if the circuit is designed for testability, as explained in section 11.6.1. Large circuits cannot be tested exhaustively, because a vast number of tests would have to be applied. In this case the designer's ingenuity is needed to determine a manageable set of tests that will hopefully demonstrate the correctness of the circuit.

PROBLEMS

11.1 Derive a table similar to Figure 11.1*b* for the circuit in Figure P11.1 to show the coverage of the various stuck-at-0 and stuck-at-1 faults by the eight possible tests. Find a minimal test set for this circuit.

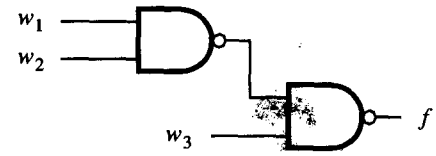


Figure P11.1 Circuit for problem 11.1.

11.2 Repeat problem 11.1 for the circuit in Figure P11.2.

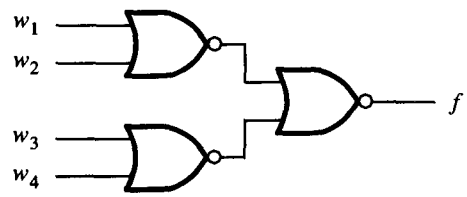


Figure P11.2 Circuit for problem 11.2.

11.3 Devise a test to distinguish between two circuits that implement the following expressions

$$f = x_1x_2x_3 + x_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_4 + \bar{x}_1x_3\bar{x}_4$$
$$g = (\bar{x}_1 + x_2)(x_3 + x_4)$$

11.4 Consider the circuit in Figure P11.3. Sensitize each path in this circuit to obtain a complete test set that comprises a minimum number of tests.

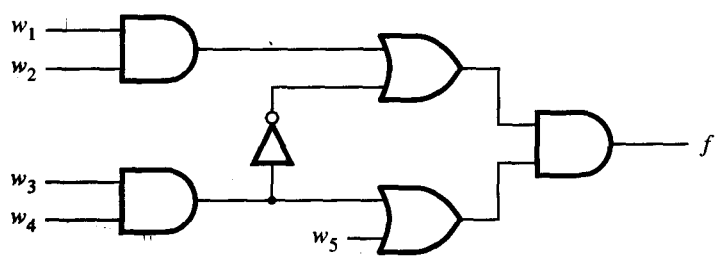


Figure P11.3 Circuit for problem 11.4.

11.5 For the circuit of Figure 11.4*a*, show the tests that can detect each of the faults: $w_1/0$, $w_4/1$, $g/0$, and $c/1$.

11.6 Suppose that the tests $w_1w_2w_3w_4 = 0100$, 1010, 0011, 1111, and 0110 are chosen randomly to test the circuit in Figure 11.3. What percentage of single faults are detected using these tests?

11.7 Repeat problem 11.6 for the circuit in Figure 11.4*a*.

11.8 Repeat problem 11.6 for the circuit in Figure 11.5.

11.9 Consider the circuit in Figure P11.4. Are all single stuck-at-0 and stuck-at-1 faults in this circuit detectable? If not, explain why.

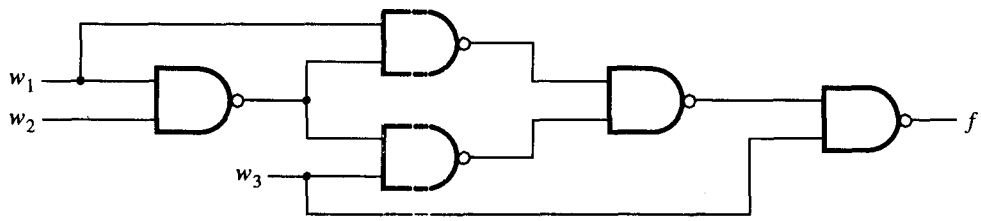


Figure P11.4 Circuit for problem 11.9.

11.10 Prove that in a circuit in which all gates have a fan-out of 1, any set of tests that detects all single faults on the input wires detects all single faults in the entire circuit.

11.11 The circuit in Figure P11.5 determines the parity of a four-bit data unit. Derive a minimal test set that can detect all single stuck-at-0 and stuck-at-1 faults in this circuit. Would your test set work if the XOR gates are implemented using the circuit in Figure 4.26*c*? Can your result be extended to a general case that involves n-bit data units?

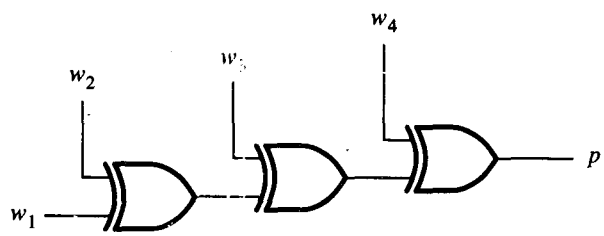


Figure P11.5 Circuit for problem 11.11.

11.12 Derive a test set that can detect all single faults in the decoder circuit in Figure 6.16*c*.

11.13 List all single faults in the circuit in Figure 11.4*a* that can be detected using each of the tests $w_1w_2w_3w_4 = 1100$, 0010, and 0110.

11.14 Sensitize each path in the combinational part of the circuit in Figure 11.12 to obtain a complete test set that comprises as few tests as possible. Show how your test set can be applied to test this circuit. How many clock cycles are needed to perform the necessary tests?

11.15 Derive an ASM chart that represents the flow of control needed to test the circuit in Figure 11.12.

11.16 The circuit in Figure 11.12 provides an easily testable implementation of the FSM in Figure 8.79. In Example 11.3 we showed how this circuit may be tested by testing the combinational part using randomly chosen tests. A different approach to testing may be to attempt to determine whether the circuit actually realizes the functionality specified in the state table in Figure 8.79*b*. This can be done by making the circuit go through all transitions given in the state table. For example, after applying the *Resetn* = 0 signal, the circuit begins in state *A*. It must be verified that the circuit is indeed forced into state *A* by scanning out the expected valuation $y_2y_1 = 00$. Next each transition must be checked. To verify the transition $A \rightarrow A$ if $w = 0$, it is necessary to make the input w equal to 0 and allow the normal operation to take place for one clock cycle by making $\overline{Normal}/Scan = 0$. The value of the output z must be observed. This is followed by scanning out the values of y_2 and y_1 to see if $y_2y_1 = 00$. At the same time, the valuation for the next test should be scanned in. If this test involves verifying that $B \rightarrow A$ if $w = 0$, then the valuation $y_2y_1 = 01$ is scanned in. This process continues until all transitions have been verified.

Indicate in the form of a table the values of the signals $\overline{Normal}/Scan$, *Scan-in, Scan-out,* w, and z, as well as the transition tested, for each clock cycle necessary to perform the complete test for this circuit.

11.17 Write Verilog code that represents the circuit in Figure 11.12.

11.18 Derive an ASM chart that describes the control needed to test a digital system that uses the BILBO structure in Figures 11.18 and 11.19.

REFERENCES

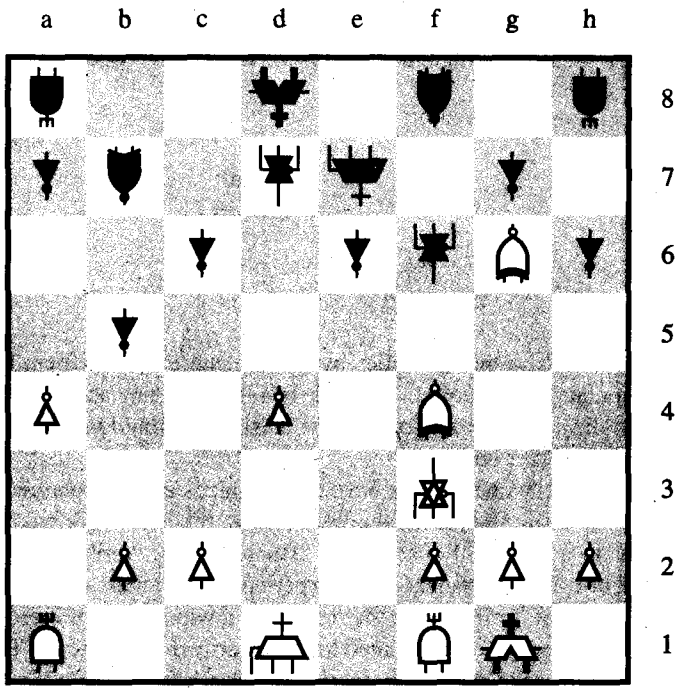
1. A. Miczo, *Digital Logic Testing and Simulation* (Wiley: New York, 1986).
2. P. K. Lala, *Practical Digital Logic Design and Testing* (Prentice-Hall: Englewood Cliffs, NJ, 1996).
3. F. H. Hill and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed. (Wiley: New York, 1993).
4. Y. M. El Ziq, "Automatic Test Generation for Stuck-Open Faults in CMOS VLSI," Proc. 18th Design Automation Conf., 1981, pp. 347–54.
5. D. Baschiera and B. Courtois, "Testing CMOS: A Challenge," *VLSI Design*, October 1984, pp. 58–62.
6. P. S. Moritz and L. M. Thorsen, "CMOS Circuit Testability," *IEEE Journal of Solid State Circuits* SC-21 (April 1986), pp. 306–9.

7. J. P. Roth et al., "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Computers* EC-16, no. 5, (October 1967), pp. 567–80.
8. J. Abraham and V. K. Agarwal, "Test Generation for Digital Systems," in D. K. Pradhan, *Fault-Tolerant Computing*, vol. 1, (Prentice-Hall: Englewood Cliffs, NJ, 1986).
9. T. W. Williams and K. P. Parker, "Design for Testability—a Survey," *IEEE Transactions on Computers* C-31 (January 1982), pp. 2–15.
10. V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design* (Prentice-Hall: Englewood Cliffs, NJ, 1995).
11. W. W. Peterson and E. J. Weldon Jr., *Error-Correcting Codes*, 2nd ed. (MIT Press: Boston, MA, 1972).
12. J. E. Smith, "Measures of Effectiveness of Fault Signature Analysis," *IEEE Transactions on Computers* C-29, no. 7 (June 1980), pp. 510–4.
13. R. David, "Testing by Feedback Shift Register," *IEEE Transactions on Computers* C-29, no. 7 (July 1980), pp. 668–73.
14. B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Techniques," Proceedings 1977 Test Conference, IEEE Pub. 79CH1609-9C, October 1979, pp. 37–41.
15. A. Y. Chan, "Easy-to-Use Signature Analyzer Accurately Troubleshoots Complex Logic Circuits," *Hewlett-Packard Journal*, May 1997, pp. 9–14.
16. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.
17. *High-Speed Board Designs*, Application Note 75, Altera Corporation, January 1998.
18. L. Y. Levesque, "High-Speed Interconnection Techniques," Technical Report, Texas Instruments Inc., 1994.

appendix

A

VERILOG REFERENCE



12. a2–a4, Bc8–b7

This appendix describes the features of Verilog used in this book. It is meant to serve as a convenient reference for the reader, hence only brief descriptions are provided, along with examples. The vast majority of the examples presented are consistent with the original Verilog 1995 standard, but we also introduce some of the most important features of Verilog 2001 [8]. The reader is encouraged to first study the introduction to Verilog in section 2.10.

This appendix is not meant to be a comprehensive Verilog manual. While we discuss almost all the features of Verilog that are useful in the synthesis of logic circuits, we do not discuss many of the features that are useful only for simulation of circuits. Although the omitted features are not needed for any of the examples used in this book, a reader who wishes to learn more about Verilog can refer to specialized texts [1–8].

How to Write Verilog Code

The tendency for the novice is to write Verilog code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. The task of a synthesis tool is to analyze a piece of Verilog code and determine, according to the semantics of the language, what circuit can be used to implement the code. Consider a code fragment such as

```
f = w0;
if (s == 1)
   f = w1;
```

We can understand the semantics by considering each statement in sequence, in the way that a simulation tool would. The code results in f being assigned the value of either w_0 or w_1, depending on the value of s. A synthesis tool would usually implement this behavior using a multiplexer circuit.

In general, synthesis tools have to recognize certain structures in code, like the above multiplexer. From the practical point of view, this will work only if users write code that conforms to a commonly used style. The beginning Verilog user should therefore adopt the style of code recommended by experienced designers. This book contains more than 140 examples of Verilog code that represent a wide range of logic circuits. In all of these examples the code is easily related to the described logic circuit. The reader is encouraged to adopt the same style of code. A good approach is to "write Verilog code that obviously represents the intended circuit."

Although Verilog is a fairly straightforward language to learn and use, the novice designer will tend to make some common errors in syntax and semantics. A list of typical errors is given in section A.15, as well as a set of guidelines that expert Verilog coders recommend as good style for writing clear and effective code.

Once complete Verilog code is written for a particular design, it is useful to analyze the resulting circuit synthesized by the CAD tools. Much can be learned about Verilog, logic circuits, and logic synthesis through this process.

A.1 DOCUMENTATION IN VERILOG CODE

Documentation can be included in Verilog code by writing a comment. A *short* comment begins with the double slash, //, and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters /* and */. Examples of comments are

```
// This is a short comment
/*This is a long Verilog comment
  that spans two lines */
```

A.2 WHITE SPACE

White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler. Multiple statements can be written on a single line, such as

```
f = w0; if (s == 1) f = w1;
```

Although legal, this code is hard to read and uses poor style. Placing each statement on a separate line, and using indentation within blocks of code, such as an **if-else** statement, are good ways to increase the readability of code.

A.3 SIGNALS IN VERILOG CODE

In Verilog, a signal in a circuit is represented as a *net* or a *variable* with a specific type. The term *net* is derived from the electrical jargon, where it refers to the interconnection of two or more points in a circuit. A net or variable declaration has the form

type [range] signal_name{, signal_name};

The square brackets indicate an optional field, and the curly brackets indicate that additional entries are permitted. We will use this syntax throughout the appendix. The *signal_name* is an identifier, as defined in the next section. Without the *range* field the declared net or variable is scalar and represents a single-bit signal. The range is used to specify vectors that correspond to multibit signals, as explained in section A.6.

A.4 IDENTIFIER NAMES

Identifiers are the names of variables and other elements in Verilog code. The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ underscore and $ characters. There are two caveats: an identifier must not begin with a digit and it should not be a Verilog keyword. Examples of legal identifiers are *f*, *x*1, *x_y*, and *Byte*.

Some examples of illegal names are *1x*, *+y*, *x*y*, and *258*. Verilog is case sensitive, hence k is not the same as K, and BYTE is not the same as Byte.

For special purposes Verilog allows a second form of identifier, called an *escaped* identifier. Such identifiers begin with the (\) backslash character, which can then be followed by any printable ASCII characters except white spaces. Examples of escaped identifiers are \123, *sig-name*, and *a+b*. Escaped identifiers should not be used in normal Verilog code; they are intended for use in code produced automatically when other languages are translated into Verilog.

A.5 SIGNAL VALUES, NUMBERS, AND PARAMETERS

Verilog supports scalar nets and variables that represent individual signals, and vectors that correspond to multiple signals. Each individual signal can have four possible values:

0 = logic value 0
1 = logic value 1
z = tri-state (high impedance)
x = unknown value

The z and x values can also be denoted by the capital letters Z and X. The value x can be used to denote a don't-care condition in Verilog code; the symbol ? can also be used for this purpose. The value of a vector variable is specified by giving a constant of the form

[size]['radix]constant

where *size* is the number of bits in the constant, and *radix* is the number base. Supported radices are

d = decimal
b = binary
h = hexadecimal
o = octal

When no radix is specified, the default is decimal. If size specifies more bits than are needed to represent the given constant, then in most cases the constant is padded with zeros. The exceptions to this rule are when the first character of the constant is either x or z, in which case the padding is done using that value. Some examples of constants include

0	the number 0
10	the decimal number 10
'b10	the binary number 10 = $(2)_{10}$
'h10	the hex number 10 = $(16)_{10}$
4'b100	the binary number 0100 = $(4)_{10}$
4'bx	an unknown 4-bit value xxxx
8'b1000_0011	_ can be inserted for readability
8'hfx	equivalent to 8'b1111_xxxx

A.5.1 PARAMETERS

A *parameter* associates an identifier name with a constant. Let the Verilog code include the following declarations.

```
parameter n = 4;
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
```

Then the identifier *n* can be used in place of the number 4, the name *S0* can be substituted for the value 2'b00, and so on. An important use of parameters is in the specification of parameterized subcircuits, which is described in section A.12.

A.6 NET AND VARIABLE TYPES

Verilog defines a number of types of nets and variables. These types are defined by the language itself, and user-defined types are not permitted.

A.6.1 NETS

A *net* represents a node in a circuit. To distinguish between different types of circuit nodes there exist several types of nets, called *wire*, *tri*, and a number of others that are not needed for synthesis, and are not used in this book.

The **wire** type is employed to connect an output of one logic element in a circuit to an input of another logic element. The following are examples of scalar **wire** declarations.

```
wire x;
wire Cin, AddSub;
```

A vector **wire** represents multiple nodes, such as

```
wire [3:0] S;
wire [1:2] Array;
```

The square brackets are the syntax for specifying a vector's range. The range $[R_a{:}R_b]$ can be either increasing or decreasing, as shown. In either case, R_a is the index of the most-significant (leftmost) bit in the vector, and R_b is the index of the least-significant (rightmost) bit. The indices R_a and R_b can be either positive or negative integers.

The net *S* can be used as a four-bit quantity, or each bit can be referred to individually as *S*[3], *S*[2], *S*[1], and *S*[0]. If a value is assigned to *S* such as S = 4'b0011, the result is $S[3] = 0, S[2] = 0, S[1] = 1$, and $S[0] = 1$. The assignment of a single bit in a vector to another net, such as $f = S[0]$, is called a *bit-select* operation. A range of values from one vector can be assigned to another vector, which is called a *part-select* operation. If we assign Array = *S*[2:1], this produces $Array[1] = S[2]$ and $Array[2] = S[1]$. The index used in a bit-select operation can involve a variable, such as $S[i]$, while the indices used with a part-select operation have to be constant expressions, such as *S*[2:1].

The *tri* type denotes circuit nodes that are connected in a tri-state fashion. Examples of **tri** nets are

```
tri z;
tri [7:0] DataOut;
```

These nets are treated in the same manner as the **wire** type, and they are used only to enhance the readability of code that includes tri-state gates.

A.6.2 VARIABLES

Nets provide a means for interconnecting logic elements, but they do not allow a circuit to be described in terms of its *behavior*. For this purpose, Verilog provides *variables*. A variable can be assigned a value in one Verilog statement, and it retains this value until it is overwritten in a subsequent assignment statement. There are two types of variables, *reg* and *integer*. Consider the code fragment

```
Count = 0;
for (k = 0; k < 4; k = k + 1)
   if (S[k])
      Count = Count + 1;
```

The **for** and **if** statements are described in section A.11. This code stores in *Count* the number of bits in *S* that have the value 1. Since it models the behavior of a circuit, *Count* has to be declared as a variable, rather than a simple **wire**. If *Count* has three bits, then the declaration is

```
reg [2:0] Count;
```

The keyword **reg** does *not* denote a storage element, or register. In Verilog code, **reg** variables can be used to model either combinational or sequential parts of a circuit. In our example, the variable *k* serves as a loop index. Such variables are declared as type **integer** in the statement

```
integer k;
```

Integer variables are useful for describing the behavior of a module, but they do not directly correspond to nodes in a circuit. In this book we use **integers** as loop control variables.

Verilog Terminology: Register versus Variable

The original Verilog standard used the term *register* to refer to the **reg** and **integer** types. Since these types are used to model both combinational and sequential circuit elements, this terminology is confusing and not intuitive. The Verilog 2001 standard omits the term register and uses the word *variable* instead, as we have done in this book. We mention this point here because the reader is likely to encounter the term *register* used in the style from the original Verilog standard in many other books.

A.6.3 MEMORIES

A memory is a two-dimensional array of bits. Verilog allows such a structure to be declared as a variable (**reg** or **integer**) that is an array of vectors, such as

reg [7:0] R [3:0];

This statement defines *R* as four eight-bit variables named *R*[3], *R*[2], *R*[1], and *R*[0]. In Verilog 1995, the only way to access individual bits of *R* is to first transfer one of the eight-bit vectors to another eight-bit variable. For example, to access the leftmost bit of *R*[3] we can use the assignment Byte = R[3], and then refer to *Byte*[7]. Memories cannot be net types, and they cannot be used as ports on a module.

Support for two-level indexing, such as R[3][7], is added in Verilog 2001. It also adds support for higher dimension arrays. An example of a three-dimensional array declaration in Verilog 2001 is **reg** [7:0] R [3:0] [1:0]. This statement declares a three-dimensional array of bits.

A.7 OPERATORS

Verilog has a large number of operators, as shown in Table A.1. The first column gives the category, the second column indicates how each operator is used, and the third column specifies the number of bits produced in the result. To aid in describing the table, we use operands named *A*, *B*, and *C*, which may be either vectors or scalars. The syntax ~*A* means

Table A.1 Verilog operators and bit lengths.

Category	Examples	Bit Length
Bitwise	$\sim A$, $+A$, $-A$ $A \& B$, $A \mid B$, $A \sim\hat{}\ B$, $A \hat{}\sim B$	L(*A*) MAX (L(*A*), L(*B*))
Logical	!*A*, *A*&&*B*, *A* || *B*	1 bit
Reduction	&*A*, ~&*A*, |*A*, ~|*A*, ^~*A*, ~^*A*	1 bit
Relational	$A == B$, $A! = B$, $A > B$, $A < B$ $A >= B$, $A <= B$ $A === B$, $A! == B$	1 bit
Arithmetic	$A + B$, $A - B$, $A * B$, A/B $A \% B$	MAX (L(*A*), L(*B*))
Shift	$A << B$, $A >> B$	L(*A*)
Concatenate	{*A*, ..., *B*}	L(*A*) + ··· + L(*B*)
Replication	{*B*{*A*}}	*B* * L(*A*)
Condition	*A* ? *B* : *C*	MAX (L(*B*), L(*C*))

that the ~ operator is applied to the variable A, and the syntax $L(A)$ means that the result has the same number of bits (length) as in A.

Most of the operators in Table A.1 are also listed in Table 6.2 and described in detail in section 6.6.5. The bitwise operators are 1's complement (~), unary plus (+), 2's complement (−), AND (&), OR (|), XOR (^), and XNOR (~^ or ^~). The bitwise operators produce multibit results, usually with the same number of bits as the operands. For example, if $A = a_1a_0$, $B = b_1b_0$, and $C = c_1c_0$, then the operation $C = A \,\&\, B$ results in $c_1 = a_1 \,\&\, b_1$ and $c_0 = a_0 \,\&\, b_0$. Note that unary plus just denotes a positive number; it has no effect.

The logical operators generate a one-bit result. They are NOT (!), AND (&&), and OR (||). If the operand of ! is a vector, then $!A$ produces 1 (True) only if all bits in A are 0; otherwise $!A$ gives 0 (False). The result of $A \,\&\&\, B$ is 1 if both A and B are nonzero, while $A \,||\, B$ produces 1 unless both A and B are zero. If an operand is ambiguous (contains an x), the result is x. The logical operators are normally used in conditional statements such as **if** $((A < B) \,\&\&\, (B < C))$.

The reduction operators use the same symbols as some of the bitwise operators, but have only one operand. The reduction $\&A$ produces the AND of all of the bits in A, while $\sim\&A$ produces the NAND. Similarly, the other reduction operators produce single-bit Boolean results.

The relational operators give a 1 (True) or 0 (False) result based on the specified comparison of A and B. For synthesis of logic circuits A and B are usually **wire** or **reg** types, and Verilog treats them as unsigned numbers. If **integer** variables are supported, they may be treated as signed numbers. The result of a relational operation is ambiguous (x) if either operand has any unspecified digits. An exception is for the === and !== operators, which check for equality and inequality, respectively. These operators compare equality of x and z digits in addition to 0 and 1 digits.

Verilog includes the normal arithmetic operators +, −, *, and /. The modulus operator (%) is also included, but it is usually not supported for synthesis, except for use in calculating a compile-time constant. The operation $A \,\%\, B$ returns the remainder of the integer division $A \div B$. Arithmetic operands of type **wire** and **reg** are treated as unsigned numbers. If the two operands are of unequal size, zero digits are padded on the left, and bits are truncated if the result has fewer digits than the largest operand. **Integer** variables are considered as 2's complement numbers.

The << and >> operators perform logical shifts to the left and right, respectively. For a left shift, zeros are shifted into the LSB, while for a right shift, zeros are shifted into the MSB. For synthesis, the operand B should be a constant.

The {,} concatenate operator allows vectors to be combined to produce a larger resulting vector. Any operand that is a constant must have a specified size, as in 4'b0011. Operands can be repeated multiple times by using the replication operator. The operation {{3{A}}, {2{B}}} is equivalent to $\{A, A, A, B, B\}$. The replication operator can be used to form an n-bit vector of digits: the operation n{1'b1} represents n ones.

The last item in Table A.1 is the ? : conditional operator. The result of A ? B : C is equal to B if operand A evaluates to 1 (True); otherwise, the result is C. In the case that A evaluates to x, the conditional operator generates a bitwise output; each bit in the result is 1 if the corresponding bits in both B and C are 1, 0 if these bits are 0, and x otherwise.

The precedence of Verilog operators follows similar rules as in arithmetic and Boolean algebra. For example, * has precedence over +, and & has precedence over |. A complete listing of the precedence rules is given in Table 6.3.

A.8 VERILOG MODULE

A circuit or subcircuit described with Verilog code is called a *module*. Figure A.1 gives the general structure of a **module** declaration. The module has a name, *module_name*, which can be any valid identifier, followed by a list of ports. The term *port* is adopted from the electrical jargon, in which it refers to an input or output connection in an electrical circuit. The ports can be of type **input**, **output**, or **inout** (bidirectional), and can be either scalar or vector. Examples of ports are

```
input Cin, x, y;
input [3:0] X, Y;
output Cout, s;
inout [7:0] Bus;
output [3:0] S;

wire Cout, s;
wire [7:0] Bus;
reg [3:0] S;
```

As shown, output and inout ports have an associated type. We assume that *Cout*, *s*, and *Bus* are nets in this example, while *S* is a variable. The **wire** declarations can actually be omitted, because Verilog assumes that signals are nets by default. However, any port used as a variable must be explicitly declared as such.

As Figure A.1 indicates, a module may contain any number of net (**wire** or **tri**) or variable (**reg** or **integer**) declarations, and a variety of other types of statements that are described later in this appendix.

Figure A.2 gives the Verilog code for a module *fulladd*, which represents a full-adder circuit. (The full-adder is discussed in section 5.2.) The input port *Cin* is the carry-in, and the bits to be added are the input ports *x* and *y*. The output ports are the sum, *s*, and

```
module module_name [(port_name{, port_name})];
   [parameter declarations]
   [input declarations]
   [output declarations]
   [inout declarations]
   [wire or tri declarations]
   [reg or integer declarations]
   [function or task declarations]
   [assign continuous assignments]
   [initial block]
   [always blocks]
   [gate instantiations]
   [module instantiations]
endmodule
```

Figure A.1 The general form of a module.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  assign s = x ^ y ^ Cin;
  assign Cout = (x & y) | (Cin & x) | (Cin & y);

endmodule
```

Figure A.2 A full-adder module.

```
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;

  assign {Cout, s} = x + y + Cin;

endmodule
```

Figure A.3 A full-adder module defined using the + operator.

the carry-out, *Cout*. The functionality of the full-adder is described with logic equations preceded by the keyword **assign**, which is discussed in section A.10.

There is usually more than one way to describe a given circuit using Verilog. Figure A.3 gives another version of the *fulladd* module, in which the functionality is specified by using the concatenate and addition operators. The statement

```
assign {Cout, s} = x + y + Cin;
```

assigns the least-significant bit in the result $x + y + Cin$ to the output s and the most-significant bit to *Cout*. The circuits generated from the modules in Figures A.2 and A.3 are the same.

A.9 GATE INSTANTIATIONS

Verilog includes predefined modules that implement basic logic gates. These gates allow a circuit's structure to be described using *gate instantiation* statements of the form

```
gate_name [instance_name] (output_port, input_port{, input_port});
```

The *gate_name* specifies the desired type of gate, and the *instance_name* is any unique identifier. Each gate may have a different number of ports, with the output port listed first, followed by a variable number of input ports. An example of using gates to realize a full-adder is given in Figure A.4. The code defines four **wire** nets, z1 to z4, that connect the gates together, and each gate has a specified instance name. Figure A.5 shows a simpler version,

```
// Structural specification of a full-adder
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;
   wire z1, z2, z3, z4;

   and And1 (z1, x, y);
   and And2 (z2, x, Cin);
   and And3 (z3, y, Cin);
   or Or1 (Cout, z1, z2, z3);
   xor Xor1 (z4, x, y);
   xor Xor2 (s, z4, Cin);

endmodule
```

Figure A.4 A full-adder described using gate instantiation.

```
// Structural specification of a full-adder
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   and (z1, x, y);
   and (z2, x, Cin);
   and (z3, y, Cin);
   or (Cout, z1, z2, z3);
   xor (z4, x, y);
   xor (s, z4, Cin);

endmodule
```

Figure A.5 A simplified version of Figure A.4.

in which instance names are not included and the declarations of z1 to z4 are omitted. Since the nets are not explicitly declared, they are implicitly assumed to be of type **wire**.

The logic gates supported in Verilog are summarized in Table A.2. The second column describes the function of each gate, and the rightmost column gives an example of instantiating the gate. Verilog allows gates with any number of inputs to be specified, but some CAD/systems set practical limits. The *notif* and *bufif* gates represent tri-state buffers. The gate *notif0* is an inverting tri-state buffer with active-low enable, and *notif1* provides the same functionality with an active-high enable. The *bufif0* and *bufif1* gates are tri-state buffers that do not invert the output.

Table A.2 Verilog gates.

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \bar{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e\ ?\ \bar{a} : 'bz)$	**notif0** (f, a, e)
notif1	$f = (e\ ?\ \bar{a} : 'bz)$	**notif1** (f, a, e)
bufif0	$f = (!e\ ?\ a : 'bz)$	**bufif0** (f, a, e)
bufif1	$f = (e\ ?\ a : 'bz)$	**bufif1** (f, a, e)

For simulation purposes, it is possible to set a parameter of the gate that represents its propagation delay. As an example, the following statement instantiates a three-input AND gate with a delay of five time units (the units of time are determined by the simulator being used).

```
and #(5) And3 (z, x1, x2, x3);
```

This type of delay parameter has no meaning when using Verilog for synthesis of logic circuits.

A.10 CONCURRENT STATEMENTS

In any hardware description language, including Verilog, the concept of a *concurrent statement* means that the code may include a number of such statements, and each represents a part of the circuit. We use the word *concurrent* because the statements are considered in parallel and the ordering of statements in the code does not matter. Gate instantiations are one type of concurrent statements. This section introduces another type of concurrent statement, called the *continuous assignment*.

A.10.1 CONTINUOUS ASSIGNMENTS

While gate instantiations allow the description of a circuit's structure, continuous assignments permit the description of a circuit's function. The general form of this statement is

assign net_assignment{, net_assignment};

The *net_assignment* can be any expression involving the operators listed in Table A.1. Examples of continuous assignments are

```
assign Cout = (x & y) | (x & Cin) | (y & Cin);
assign s = x ^ y ^ z;
```

Although they are not needed in terms of operator precedence, the parentheses in the expression for *Cout* are included for clarity. Multiple assignments can be specified in one **assign** statement, using commas to separate the assignments, as in

```
assign Cout = (x & y) | (x & Cin) | (y & Cin),
       s = x ^ y ^ z;
```

An example of a multibit assignment is

```
wire [1:3] A, B, C;
      ⋮
assign C = A & B;
```

This results in $c_1 = a_1b_1$, $c_2 = a_2b_2$, and $c_3 = a_3b_3$.

The arithmetic assignment

```
wire [3:0] X, Y, S;
      ⋮
assign S = X + Y;
```

represents a four-bit adder without carry-in and carry-out. If we declare a carry-in and carry-out,

```
wire carryin, carryout;
```

then the statement

```
assign {carryout, S} = X + Y + carryin;
```

represents the four-bit adder with carry-in and carry-out. We mentioned in section A.7 that Verilog treats the **wire** type as an unsigned number. Since a five-bit result is needed in {carryout, S}, each operand is padded with a zero. When using Verilog for synthesis, it is up to the compiler to determine, or *infer*, that a four-bit adder with carry-out is needed and to recognize the carry-in.

A complete example of arithmetic assignments is given in Figure A.6. There are two four-bit inputs, X and Y, and two eight-bit outputs, S and $S2s$. To produce the eight-bit

```
module adder_sign (X, Y, S, S2s);
  input [3:0] X, Y;
  output [7:0] S, S2s;

  assign S = X + Y,
         S2s = {{4{X[3]}}, X} + {{4{Y[3]}}, Y};

endmodule
```

Figure A.6 An example of arithmetic assignments and sign extension.

sum $S = X + Y$ the Verilog compiler automatically pads X and Y with four zeros. The assignment to $S2s$ shows how a signed (2's complement) result can be generated. Recall from section 5.3 that the leftmost bit in a 2's complement number is the sign bit. The assignment to $S2s$ uses the concatenate and replication operators to pad X and Y with four copies of their most-significant bit, thereby performing sign extensions.

As an example, assume that $X = 0011$ and $Y = 1101$. The unsigned result is $S = 0011 + 1101 = 00010000$, or $S = 3 + 13 = 16$. The signed result is $S2s = 0011 + 1101 = 00000000$, or $S2s = 3 + (-3) = 0$.

A.10.2 USING PARAMETERS

Figure A.6 specifies an adder for four-bit numbers. We can make this code more general by introducing a parameter that sets the number of bits in the adder. Figure A.7 gives the code for an n-bit adder module, *addern*. The number of bits to be added is defined with the **parameter** keyword, introduced in section A.5. The value of n defines the bit widths of X, Y, S, and $S2s$.

It is possible to combine a continuous assignment with a **wire** declaration. For example, the sum, s, and carry-out, c, of a half-adder could be defined as

```
wire s = x ^ y,
     c = x & y;
```

Verilog allows parameters, such as delays, to be associated with continuous assignments. These parameters have no meaning for synthesis, but we mention them for completeness. Consider the statement

```
wire #8 s = x ^ y,
     #5 c = x & y;
```

These assignments specify that the operation $x \wedge y$ has an associated propagation delay of eight time units, and $x \,\&\, y$ has a delay of five units. Such delays, which are useful only for simulation purposes, can also be associated with **wires**, as in

```
module addern (X, Y, S, S2s);
  parameter n = 4;
  input  [n-1:0] X, Y;
  output [2*n-1:0] S, S2s;

  assign  S = X + Y,
          S2s = {{n{X[n-1]}}, X} + {{n{Y[n-1]}}, Y};

endmodule
```

Figure A.7 Using a parameter in an *n*-bit adder.

```
wire   #2 c;
assign #5 c = x & y;
```

This code specifies that two time units of delay are incurred on the **wire** *c* in addition to the five time units for the AND gate that produces *x* & *y*.

A.11 PROCEDURAL STATEMENTS

In addition to the concurrent statements described in the previous section, Verilog also provides *procedural statements* (also called *sequential statements*). Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code. Verilog syntax requires that procedural statements be contained inside an **always** block.

A.11.1 ALWAYS AND INITIAL BLOCKS

An **always** block is a construct that contains one or more procedural statements. It has the form

```
always @(sensitivity_list)
[begin]
   [procedural assignment statements]
   [if-else statements]
   [case statements]
   [while, repeat, and for loops]
   [task and function calls]
[end]
```

Verilog includes several types of procedural statements. These statements permit the description of a circuit in terms of its *behavior* in a much more powerful way than is possible with continuous assignments or gate instantiations.

When multiple statements are included in an **always** block, the **begin** and **end** keywords are needed; otherwise, these keywords can be omitted. The **begin** and **end** keywords are also used with other Verilog constructs. We refer to the statements delimited by **begin** and **end** as a *begin-end block.*

The *sensitivity_list* is a list of signals that directly affect the output results generated by the **always** block. In some books the terms *event_expression* or *event_control* are used in addition to *sensitivity_list.* A simple example of an **always** block is

```
always @(x or y)
begin
   s = x ^ y;
   c = x & y;
end
```

Since the output variables s and c depend on x and y, these signals are included in the sensitivity list, separated by the keyword **or**. (In Verilog 2001, a comma can be substituted for **or**. Verilog 2001 also allows the special sensitivity list denoted @*. This syntax automatically adds all input signals used in the **always** block to the sensitivity list.) The semantics of the **always** block are as follows: if the value of a signal in the sensitivity list changes, then the statements inside the **always** block are evaluated in the order presented.

For simulation purposes, Verilog also provides the *initial* construct. The **initial** and **always** constructs have the same form, but the statements inside the **initial** construct are executed only once, at the start of a simulation. **Initial** constructs are not meaningful for synthesis, hence we do not discuss them further.

A Verilog module may include several **always** blocks, each representing a part of the circuit being modeled. While the statements inside each **always** block are evaluated in order, there is no meaningful order among the different **always** blocks. In this sense, each entire **always** block can be considered as a concurrent statement, because a Verilog compiler evaluates all **always** blocks concurrently.

Procedural Assignment Statements

Any signal assigned a value inside an **always** block has to be a variable of type **reg** or **integer**. A value is assigned to a variable with a *procedural assignment statement.* There are two kinds of assignments: *blocking* assignments, denoted by the = symbol, and *non-blocking* assignments, denoted by the <= symbol. The term *blocking* means that the assignment statement completes and updates its left-hand side before the subsequent statement is evaluated. This concept is best explained in the context of simulation. Consider the blocking assignments

```
S = X + Y;
p = S[0];
```

At simulation time t_i the statements are evaluated, in order. The first statement sets S using the current values of X and Y, and then the second statement sets p according to this

new value of S. Verilog also provides *non-blocking* assignments, specified as

```
S <= X + Y;
p <= S[0];
```

In this case, at simulation time t_i the statements are still evaluated in order, but they both use the values of variables that exist at the start of the simulation time t_i. The first statement determines a new value for S based on the current values of X and Y, but S is not actually changed to this value until all statements in the associated **always** block have been evaluated. Therefore, the value of p at time t_i is based on the value of S at time t_{i-1}. We can summarize the difference between blocking and non-blocking assignments as follows. For blocking assignments, the values of variables seen at time t_i by each statement are the new values set in t_i by any preceding statements in the **always** block. For non-blocking assignments, the values of variables seen at time t_i are the values set in time t_{i-1}.

Although we introduced the concepts of blocking and non-blocking assignments in the context of simulation, the semantics are the same for synthesis. For combinational circuits, only blocking assignments should be used, as we will explain in section A.11.7. We give a number of examples of combinational circuits in the following sections and also introduce the **if-else**, **case**, and loop statements. Section A.14 focuses on sequential circuits and explains that they should be designed with non-blocking assignments.

A.11.2 THE IF-ELSE STATEMENT

The general form of the **if-else** statement is given in Figure A.8. If *expression*1 is True, then the first statement is evaluated. When multiple statements are involved, they have to be included inside a **begin-end** block.

The **else if** and **else** clauses are optional. Verilog syntax specifies that when **else if** or **else** are included, they are paired with the most recent unfinished **if** or **else if**.

An example of an **if-else** statement used for combinational logic is

```
if (expression1)
begin
   statement;
end
else if (expression2)
begin
   statement;
end
else
begin
   statement;
end
```

Figure A.8 The form of the **if-else** statement.

```
always @(w0 or w1 or s)
   if (s == 0)
      f = w0;
   else
      f = w1;
```

This code defines a 2-to-1 multiplexer with data inputs w_0 and w_1, select input s, and output f.

A.11.3 STATEMENT ORDERING

Another way of describing the 2-to-1 multiplexer with an **if-else** statement is presented in Figure A.9. Instead of using an **else** clause, this code first makes the default assignment $f = w_0$ and then changes this assignment to $f = w_1$ if s has the value 1. The Verilog semantics specify that a signal assigned multiple values in an **always** construct retains the last assignment. This example highlights the importance of the ordering of statements in an **always** block. If the statements are reversed, as in

```
always @(w0 or w1 or s)
begin
   if ( s == 1 )
      f = w1;
   f = w0;
end
```

then the **if** statement would be evaluated first and the assignment $f = w_0$ would be performed last. Hence, the code would always result in f being set to the value of w_0.

```
module mux (w0, w1, s, f);
   input w0, w1, s;
   output f;
   reg f;

   always @(w0 or w1 or s)
   begin
      f = w0;
      if (s == 1)
         f = w1;
   end

endmodule
```

Figure A.9 Code for a 2-to-1 multiplexer.

Implied Memory

Consider the **always** block

```
always @( w0 or w1 or s )
begin
   if (s == 1)
      f = w1;
end
```

This is the same as the code in Figure A.9, except that the default assignment f = w0; has been removed. Since the code does not specify a value for the variable f when s is 0, the Verilog semantics specify that f must retain its current value. A synthesized circuit has to implement the functionality

$$f = s \cdot w_1 + \bar{s} \cdot f$$

Hence, when $s = 0$, the value of w_1 is "remembered" by a latch at the output f. This effect is called *implied memory*. We will show shortly that implied memory is the key concept used to describe sequential circuits.

A.11.4 THE CASE STATEMENT

The form of a **case** statement is illustrated in Figure A.10. The bits in *expression*, called the *controlling expression*, are checked for a match with each *alternative*. The first successful match causes the associated statements to be evaluated. Each digit in each alternative is compared for an exact match of the four values 0, 1, x, and z. A special case is the **default** clause, which takes effect if no other alternative matches. When using Verilog for simulation, an alternative can be a general expression, but for synthesis these items are restricted to a single constant, such as 1'b0:, or a list of constants separated by commas, such as 1, 2, 3:.

An example of a **case** statement is

```
case (expression)
   alternative1: begin
                    statement;
                 end
   alternative2: begin
                    statement;
                 end
   [default:     begin
                    statement;
                 end]
endcase
```

Figure A.10 The general form of the **case** statement.

```
always @(w0 or w1 or s)
  case (s)
    1'b0: f = w0;
    1'b1: f = w1;
  endcase
```

This code represents the same 2-to-1 multiplexer described in section A.11.2 using the **if-else** statement. When using Verilog for simulation, it is necessary to give alternatives for all possible valuations of the controlling expression. A default has to be included for any valuations not explicitly covered by the listed alternatives. In this example, s can have the four values 0, 1, x, or z; hence, we could include a default to handle the cases $s = x$ and $s = z$. We have not included the default clause here because we are concerned only with synthesis, and the synthesis tools require only the bit values 0 and 1 to be considered.

Figure A.11 demonstrates the use of a **case** statement to specify truth tables. This code represents the same full-adder that is described in Figures A.2 and A.3. The controlling expression in the **case** statement is the concatenated bits $\{Cin, x, y\}$, and the alternatives correspond to the rows of the truth table in Figure 5.4*a*.

The **case** statement is also important for representing some types of sequential circuits, such as finite state machines, which are discussed in section A.14.

```
// Full adder
module fulladd (Cin, x, y, s, Cout);
  input Cin, x, y;
  output s, Cout;
  reg s, Cout;

  always @(Cin or x or y)
  begin
    case ( {Cin, x, y} )
      3'b000: {Cout, s} = 'b00;
      3'b001: {Cout, s} = 'b01;
      3'b010: {Cout, s} = 'b01;
      3'b011: {Cout, s} = 'b10;
      3'b100: {Cout, s} = 'b01;
      3'b101: {Cout, s} = 'b10;
      3'b110: {Cout, s} = 'b10;
      3'b111: {Cout, s} = 'b11;
    endcase
  end

endmodule
```

Figure A.11 Using a **case** statement to specify a truth table.

```
module priority (W, Y, f);
   input [3:0] W;
   output [1:0] Y;
   output f;
   reg [1:0] Y;

   assign f = (W != 0);
   always @(W)
   begin
      casex (W)
         'b1xxx: Y = 3;
         'b01xx: Y = 2;
         'b001x: Y = 1;
         default: Y = 0;
      endcase
   end

endmodule
```

Figure A.12 A priority encoder described using a casex statement.

A.11.5 CASEZ AND CASEX STATEMENTS

In the **case** statement, the values x or z in an alternative are checked for an exact match with the same values in the controlling expression. The **casez** statement adds more flexibility, by treating a z digit in an alternative as a don't-care condition. The **casex** statement treats both x and z as don't cares. The alternatives do not have to be mutually exclusive. If they are not, then the first matching item has priority. Figure A.12 shows how **casex** can be used to describe a priority encoder with the 4-bit input W and the outputs Y and f. The priority encoder is defined in Figure 6.24 (the output z in Figure 6.24 is named f in the Verilog code in Figure A.12). The first alternative, 1xxx, specifies that if w_3 has the value 1, then the other inputs are treated as don't cares, hence the output is set to $Y = 3$. Similarly, the other alternatives describe the desired priority scheme.

A.11.6 LOOP STATEMENTS

Verilog includes four types of loop statements: **for**, **while**, **repeat**, and **forever**. Synthesis tools typically support the **for** loop, which has the general form

```
for (initial_index; terminal_index; increment)
begin
   statement;
end
```

```
module ripple (carryin, X, Y, S, carryout);
   parameter n = 4;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   reg [n-1:0] S;
   reg [n:0] C;
   reg carryout;
   integer k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k <= n-1; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure A.13 An n-bit ripple-carry adder using a **for** loop.

This syntax is very similar to the for loop in the C programming language. The *initial_index* is evaluated once, before the first loop iteration, and typically performs the initialization of the **integer** loop control variable, such as $k = 0$. In each loop iteration, the **begin-end** block is performed, and then the *increment* statement is evaluated. A typical increment statement is $k = k + 1$. Finally, the *terminal_index* condition is checked, and if it is True (1), then another loop iteration is done. For synthesis, the terminal_index condition has to compare the loop index to a constant value, such as $k < 8$.

An example of using a **for** loop to describe an n-bit ripple-carry adder is presented in Figure A.13. The effect of the loop is to repeat its **begin-end** block for the specified values of k. In this example, each loop iteration, k, defines a full-adder with the inputs x_k, y_k, and c_k, and the outputs s_k and c_{k+1}. It is possible to define the **integer** k (parameters can also be defined in this way) inside the **always** block if the **begin-end** block has a label. For example,

```
always @(X or Y or carryin)
begin: fulladders
   integer k;
   C[0] = carryin;
   for (k = 0; k <= n−1; k = k+1)
   begin
      S[k] = X[k] ^ Y[k] ^ C[k];
      C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
   end
   carryout = C[n];
end
```

A second **for**-loop example is given in Figure A.14. This code produces a count of the number of bits in the n-bit input X that have the value 1. Unrolling the loop, the first two iterations give

```
Count = Count + X[0];
Count = Count + X[1];
```

The first statement produces the value $Count = 0 + X[0] = X[0]$. The second assignment then gives Count = X[0] + X[1], and so on for the other loop iterations. At the end of the loop, we have

```
Count = X[0] + X[1] + ... + X[n−1]
```

```
module bit_count (X, Count);
   parameter n = 4;
   parameter logn = 2;
   input [n−1:0] X;
   output [logn:0] Count;
   reg [logn:0] Count;
   integer k;

   always @(X)
   begin
      Count = 0;
      for (k = 0; k < n; k = k+1)
         Count = Count + X[k];
   end

endmodule
```

Figure A.14 A bit-counting example.

A synthesis tool generates a circuit that has a cascade of adders to implement this expression. For example, for $n = 3$ a possible circuit that employs two-bit adders is shown in Figure A.15.

Figure A.16 shows the general forms of the **while** and **repeat** loops. The **while** loop has the same structure as the corresponding statement in the C language, and the **repeat** loop simply specifies a number of times to repeat its **begin-end** block. The **forever** loop, not shown in the figure, loops endlessly.

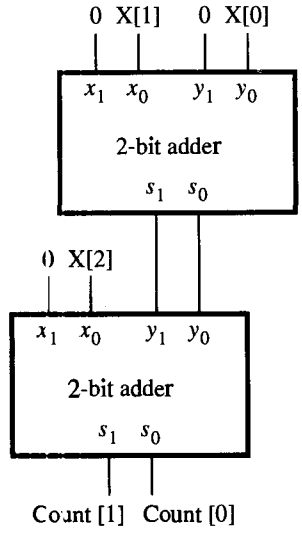


Figure A.15 A circuit that implements the code in Figure A.14.

```
while (condition)
begin
   statement;
end

⋮

repeat (constant_value)
begin
   statement;
end
```

Figure A.16 The general forms of the **while** and **repeat** statements.

```
always @(X)
begin
   Count = 0;
   for (k = 0; k < n; k = k+1)
      Count <= Count + X[k];
end
```

Figure A.17 Using non-blocking assignments for a combinational circuit.

A.11.7 BLOCKING VERSUS NON-BLOCKING ASSIGNMENTS FOR COMBINATIONAL CIRCUITS

All our previous examples of combinational circuits used blocking assignments, which is a good way to design such circuits. A natural question is whether combinational circuits can be described using non-blocking assignments. The answer is that this would work in many cases, but if subsequent assignments depend on the results of preceding assignments, non-blocking assignments can produce nonsensical combinational circuits. As an example, consider changing the **for** loop in Figure A.14 to use non-blocking assignments, as indicated in Figure A.17. For simplicity assume that $n = 3$, so that the unrolled loop is

```
Count <= Count + X[0];
Count <= Count + X[1];
Count <= Count + X[2];
```

Since non-blocking assignments are involved, each subsequent assignment statement sees the starting value of *Count* = 0 rather than a new *Count* value produced by the previous statements. The **for** loop thus degenerates to

```
Count <= 0 + X[0];
Count <= 0 + X[1];
Count <= 0 + X[2];
```

When there are multiple assignments to the same variable in an **always** block, Verilog semantics specify that the variable retains its last assignment. Therefore, the code produces the wrong result *Count* = *X*[2].

A.12 USING SUBCIRCUITS

A Verilog module can be included as a subcircuit in another module. For this to work, both modules must be defined in the same file or else the Verilog compiler must be told where each module is located (the mechanism for doing this varies from one compiler to the next). The general form of a *module instantiation* statement is similar to a gate instantiation statement

```
module_name [#(parameter overrides)] instance_name (
            .port_name ( [expression] ) {, .port_name ( [expression] )} );
```

The *instance_name* can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit. The same module can be instantiated multiple times in a given design provided that each instance name is unique. The #(parameter overrides) can be used to set the values of parameters defined inside the *module_name* module. We discuss this feature in the next section. Each *port_name* is the name of a port in the subcircuit, and each *expression* specifies a connection to that port. The syntax *.port_name* is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the **module** statement of the subcircuit. In Verilog jargon, this is called *named* port connections. If the port connections are given in the same order as in the subcircuit, then *.port_name* is not needed. This format is called *ordered* port connections.

An example is presented in Figure A.18. It gives the code for a four-bit ripple-carry adder built using four instances of the *fulladd* subcircuit in Figure A.2. The inputs to the adder are *carryin* and two four-bit numbers *X* and *Y*. The output is the four-bit sum, *S*, and *carryout*. A three-bit signal, *C*, represents the carries from stages 0, 1, and 2. This signal is declared as a **wire** vector with four bits.

The *adder4* module instantiates four copies of the full-adder subcircuit. In the first three instantiation statements, we use ordered port connections because the signals are listed in the same order as given in the declaration of the *fulladd* module in Figure A.2. The last instantiation statement gives an example of named port connections. The port connections used in the instantiation statements specify how the *fulladd* instances are interconnected by nets to create the adder module.

Figure A.19 gives an example of a hierarchical Verilog file containing two modules. The bottom module, *seg7*, presents the seven-segment converter from Figure 6.38. It has the four-bit *bcd* input, which represents a binary-coded-decimal digit, and the seven-bit *leds* output, which is intended to drive the seven segments *a* to *g* on a digit-oriented display.

```
module adder4 (carryin, X, Y, S, carryout);
  input carryin;
  input [3:0] X, Y;
  output [3:0] S;
  output carryout;
  wire [3:1] C;

  fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
  fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
  fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
  fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));

endmodule
```

Figure A.18 An example of module instantiation.

```
module group (Digits, Lights);
   input [11:0] Digits;
   output [1:21] Lights;

   seg7 digit0 (Digits[3:0], Lights[1:7]);
   seg7 digit1 (Digits[7:4], Lights[8:14]);
   seg7 digit2 (Digits[11:8], Lights[15:21]);

endmodule

module seg7(bcd, leds);
   input [3:0] bcd;
   output [1:7] leds;
   reg [1:7] leds;

   always @(bcd)
      case (bcd)      //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase

endmodule
```

Figure A.19 An example of a hierarchical design file.

Three copies of the seven-segment decoder are instantiated in the top module, *group*. It has a 12-bit input, *Digits*, and a 21-bit output, *Lights*, that are connected to the three instantiated subcircuits.

A.12.1 SUBCIRCUIT PARAMETERS

When a subcircuit includes parameters, their default values can be changed in an instantiation statement. Figure A.20 gives a module with two eight-bit inputs, X and Y, and a four-bit output, C. The module determines the number of bits in X and Y that are identical. The code first uses the bitwise XNOR operation to generate a signal T that has 1s in the bit

```
module common (X, Y, C);
   input [7:0] X, Y;
   output [3:0] C;
   wire [7:0] T;

   // Make T[i] = 1 if X[i] == Y[i]
   assign T = X ~^ Y;

   bit_count #(8,3) cbits (T, C);

endmodule
```

Figure A.20 Overriding module parameters using #.

```
module common (X, Y, C);
   input [7:0] X, Y;
   output [3:0] C;
   wire [7:0] T;

   // Make T[i] = 1 if X[i] == Y[i]
   assign T = X ~^ Y;

   bit_count cbits (T, C);
      defparam cbits.n = 8, cbits.logn = 3;

endmodule
```

Figure A.21 Overriding module parameters using the **defparam** statement.

positions where *X* and *Y* have identical values. Then it instantiates the subcircuit specified in Figure A.14 to count the number of 1s in *T*. The syntax #(8,3) overrides the default values of the subcircuit parameters in the order given in the code. Since Figure A.14 first defines the parameter *n*, followed by *logn*, then #(8,3) sets $n = 8$ and $logn = 3$. If only one parameter is specified, such as #(8), it overrides the first parameter given in the subcircuit.

An alternative syntax for overriding parameter values is shown in Figure A.21. Here, the subcircuit parameters are specified by the separate statement

```
defparam cbits.n = 8, cbits.logn = 3;
```

This statement is not a part of the instantiation statement; hence, it can appear anywhere in the code. The intended subcircuit is identified uniquely by its instance name, *cbits*. If the **defparam** statement is in a separate module from the corresponding instantiation statement, the subcircuit module name should be specified in addition to the instance name. An example is

```
defparam bit_count.cbits.n = 8, bit_count.cbits.logn = 3;
```

Another example using subcircuit parameters is shown in Figure A.22. It instantiates a module from the *Library of Parameterized Modules* (LPM), which is a standardized library of circuit building blocks that are generally useful for implementing logic circuits. We provide information on these modules, which are supported in a number of CAD systems, in Appendix D. The code in Figure A.22 instantiates the LPM module *lpm_add_sub*, which is an adder/subtractor circuit introduced in section 5.5. The **defparam** keyword sets the number of bits in the adder/subtractor to 8, and two additional **defparam** statements declare the number being used as unsigned and specify that we wish to perform addition only.

A.12.2 VERILOG 2001 GENERATE CAPABILITY

Figure A.18 instantiates four copies of the *fulladd* subcircuit to form a four-bit ripple-carry adder. A natural extension of this code is to add a parameter that sets the number of bits needed and then use a loop to instantiate the required subcircuits. Although Verilog 1995 does not allow subcircuits to be instantiated in a loop, this capability is added with the Verilog 2001 **generate** construct.

The **generate** construct has the simplified form

```
generate
   [for loops]
   [if-else statements]
   [case statements]
   [instantiation statements]
endgenerate
```

This construct enhances the flexibility of Verilog modules, because it allows instantiation statements to be included inside **for** loops and **if-else** statements. If a **for** loop is included

```
module adder_inst (carryin, X, Y, S, carryout);
   input [7:0] X, Y;
   input carryin;
   output [7:0] S;
   output carryout;

   lpm_add_sub adder8(.cin (carryin), .dataa (X), .datab (Y), .result (S), .cout (carryout));
      defparam adder8.lpm_width = 8;
      defparam adder8.lpm_representation = "UNSIGNED";
      defparam adder8.lpm_direction = "ADD";

endmodule
```

Figure A.22 Instantiating a module from a standard library (LPM).

```
module ripple_g (carryin, X, Y, S, carryout);
   parameter n = 4;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   wire [n:0] C;

   genvar i;
   assign C[0] = carryin;
   assign carryout = C[n];

   generate
      for (i = 0; i <= n-1; i = i+1)
      begin:addbit
         fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
      end
   endgenerate

endmodule
```

Figure A.23 An *n*-bit ripple-carry adder using Verilog 2001's **generate** statement.

in the **generate** block, the loop index variable has to be declared of type **genvar**. A **genvar** variable is similar to an **integer** variable, but it can have only positive values and it can be used only inside **generate** blocks.

Figure A.23 shows the *ripple_g* module, which instantiates *n fulladd* modules. Each instance generated in the **for** loop will have a unique instance name produced by the compiler based on the **for** loop label. The generated names are *addbit*[0].*stage*, ..., *addbit*[*n* − 1].*stage*. This code produces the same result as the code in Figure A.13.

The **generate** construct can include concurrent statements and procedural statements, but its main advantage is in the instantiation of gates and modules inside **for** loops and **if-else** statements.

A.13 FUNCTIONS AND TASKS

Figure 6.4 shows how a 16-to-1 multiplexer module can be created by instantiating five 4-to-1 multiplexers. Another way of modeling this circuit is to use a Verilog *function* which has the general form

```
function [range | integer] function_name;
   [input declarations]
   [parameter, reg, integer declarations]
   begin
      statement;
   end
endfunction
```

The purpose of a **function** is to allow the code to be written in a modular fashion without defining separate modules. A **function** is defined within a module, and it is called either in a continuous assignment statement or in a procedural assignment statement inside that module. A **function** can have more than one input, but it does not have an output, because the **function** name itself serves as the output variable. Figure A.24 shows how the 16-to-1 multiplexer module can be written using a function. To see how this code works, consider the assignment

```
module mux_f (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output f;
   reg f;
   reg [0:3] M;

   function mux4to1;
      input [0:3] W;
      input [1:0] S;

      if (S == 0) mux4to1 = W[0];
      else if (S == 1) mux4to1 = W[1];
      else if (S == 2) mux4to1 = W[2];
      else if (S == 3) mux4to1 = W[3];
   endfunction

   always @(W or S16)
   begin
      M[0] = mux4to1(W[0:3], S16[1:0]);
      M[1] = mux4to1(W[4:7], S16[1:0]);
      M[2] = mux4to1(W[8:11], S16[1:0]);
      M[3] = mux4to1(W[12:15], S16[1:0]);
      f = mux4to1(M[0:3], S16[3:2]);
   end

endmodule
```

Figure A.24 An example of a function.

```
f = mux4to1(M[0:3], S16[3:2]);
```

The effect of the **function** call is to have the Verilog compiler replace the assignment statement with the **function** body. The in-line equivalent of the above **function** call is

```
if (S16[3:2] == 0)  f = M[0];
else if (S16[3:2] == 1)  f = M[1];
else if (S16[3:2] == 2)  f = M[2];
else if (S16[3:2] == 3)  f = M[3];
```

```
module group_f (Digits, Lights);
  input [11:0] Digits;
  output [1:21] Lights;
  reg [1:21] Lights;

  function [1:7] leds;
    input [3:0] bcd;
    begin
      case (bcd)  // abcdefg
        0: leds = 7'b1111110;
        1: leds = 7'b0110000;
        2: leds = 7'b1101101;
        3: leds = 7'b1111001;
        4: leds = 7'b0110011;
        5: leds = 7'b1011011;
        6: leds = 7'b1011111;
        7: leds = 7'b1110000;
        8: leds = 7'b1111111;
        9: leds = 7'b1111011;
        default: leds = 7'bx;
      endcase
    end
  endfunction

  always @(Digits)
  begin
    Lights[1:7] = leds(Digits[3:0]);
    Lights[8:14] = leds(Digits[7:4]);
    Lights[15:21] = leds(Digits[11:8]);
  end

endmodule
```

Figure A.25 Using a function to implement the *group* module.

Similarly, each **function** call is inserted in-line, with the substitution of appropriate bits from *S16*, *W*, and *M*.

A second example of a **function** is shown in Figure A.25. The *group_ f* module is equivalent to the *group* module in Figure A.19. Where the *group* module instantiates three copies of the *seg7* subcircuit, *group_f* achieves the same effect by using a **function**. Since it returns a seven-bit value, the **function** is defined with the syntax

function [1:7] leds;

Consider again the 16-to-1 multiplexer example in Figure A.24. Another method of writing this code appears in Figure A.26. This code uses a Verilog *task*, which is similar to a **function**. While a **function** returns a value, a **task** does not; it has input and output variables, like a module. A **task** can be called only from inside an **always** (or **initial**) block. In a similar way as described above for the Verilog **function**, the compiler essentially inserts the code for the **task** body at the point in the code where it is called.

```
module mux_t (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output f;
   reg f;
   reg [0:3] M;

   task mux4to1;
      input [0:3] W;
      input [1:0] S;
      output Result;
      begin
         if (S == 0) Result = W[0];
         else if (S == 1) Result = W[1];
         else if (S == 2) Result = W[2];
         else if (S == 3) Result = W[3];
      end
   endtask

   always @(W or S16)
   begin
      mux4to1(W[0:3], S16[1:0], M[0]);
      mux4to1(W[4:7], S16[1:0], M[1]);
      mux4to1(W[8:11], S16[1:0], M[2]);
      mux4to1(W[12:15], S16[1:0], M[3]);
      mux4to1(M[0:3], S16[3:2], f);
   end

endmodule
```

Figure A.26 An example of a **task**.

Functions and **tasks** are not crucial for designing Verilog code, but they facilitate the writing of modular code without using separate modules. One advantage of **functions** and **tasks** is that they can be called from an **always** block, whereas these blocks are not allowed to contain instantiation statements. These features of Verilog become increasingly important as the size of the code being developed increases.

A.14 SEQUENTIAL CIRCUITS

While combinational circuits can be modeled with either continuous assignment or procedural assignment statements, sequential circuits can be described only with procedural statements. We now give some examples of sequential circuits.

A.14.1 A GATED D LATCH

Figure A.27 gives the code for a gated D latch. The sensitivity list for the **always** block includes both the data input, *D*, and clock, *clk*. The **if** statement specifies that Q should be set to the value of *D* whenever *clk* is 1. There is no **else** clause in the **if** statement. As we explained in section A.11.3, this situation implies that Q should retain its present value when the **if** condition is not met.

A.14.2 D FLIP-FLOP

Figure A.28 shows how flip-flops are described in Verilog. The **always** construct uses the special sensitivity list @(**posedge** Clock). This event expression tells the Verilog compiler that any **reg** variable assigned a value in the **always** construct is the output of a D flip-flop. The code in the figure generates a flip-flop with the input *D* and the output Q that is sensitive to the positive clock edge. A negative-edge sensitive flip-flop is specified by @(**negedge** Clock).

```
module latch (D, clk, Q);
   input D, clk;
   output Q;
   reg Q;

   always @(D or clk)
      if (clk)
         Q <= D;

endmodule
```

Figure A.27 A gated D latch.

```
module flipflop(D, Clock, Q);
   input D, Clock;
   output Q;
   reg Q;

   always @(posedge Clock)
      Q <= D;

endmodule
```

Figure A.28 A D flip-flop.

We said in section A.11.1 that sequential circuits should be described with non-blocking assignments, and we used this type of assignment in Figure A.28. The behavior of blocking and non-blocking assignments for sequential circuits is discussed in section A.14.6.

A.14.3 FLIP-FLOPS WITH RESET

Figure A.29 gives an **always** block that is similar to the one in Figure A.28. It describes a D flip-flop with an asynchronous reset (clear) input, *Resetn*. When *Resetn* = 0, the flip-flop output Q is set to 0. Appending the letter *n* to a signal name is a popular convention to denote an active-low signal.

Verilog syntax requires that a sensitivity list contains either all edge-sensitive events or all level-sensitive events but not a mixture; hence, the reset condition is checked using **negedge** Resetn. An active-high reset would require the event **posedge** Reset and the **if-else** statement would then check for the condition (Reset == 1).

```
module flipflop_ar (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;
   reg Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure A.29 A D flip-flop with asynchronous reset.

In general, Verilog offers a variety of ways to describe a given circuit. But, for specifying flip-flops, the format of the code is quite strict. Only minor variations of the code in Figures A.28 and A.29 can be made and still infer the desired flip-flops. For instance, the **if-else** statement could alternatively specify **if** (!Resetn), but this has to be the first statement in the **always** block. Note that nothing is special about the variable name *Clock*; the keyword **posedge** and the format of the rest of the **always** block are what allow the Verilog compiler to recognize the flip-flop clock signal.

Figure A.30 shows how a flip-flop with a synchronous reset input can be described. Since only the **posedge** Clock event appears in the sensitivity list of the **always** block, the reset operation has to be synchronized to the clock edge.

A.14.4 INSTANTIATING A FLIP-FLOP FROM A LIBRARY

Since flip-flops are widely used in logic circuits, most CAD systems provide an assortment of flip-flop components that can be instantiated in Verilog code. An example of this is provided in Figure A.31, which instantiates the *lpm_ff* component from the Library of Parameterized Modules.

```
module flipflop_sr (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;
   reg Q;

   always @(posedge Clock)
      if (Resetn == 0)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure A.30 A D flip-flop with synchronous reset.

```
module dff_inst (D, Clock, Resetn, Q);
   input D, Clock, Resetn;
   output Q;

   lpm_ff flipflop(.data(D), .aclr(Resetn), .clock(Clock), .q(Q));
      defparam flipflop.lpm_width = 1;

endmodule
```

Figure A.31 Instantiating a D flip-flop from a library.

A.14.5 REGISTERS

One possible approach for describing a multibit register is to create an entity that instantiates multiple flip-flops. A more convenient method is illustrated in Figure A.32. It gives the same code shown in Figure A.29 but using the four-bit input *D* and the four-bit output Q. The code describes a four-bit register with asynchronous clear.

Figure A.33 shows how the code in Figure A.32 can be extended to represent an *n*-bit register with an enable input, *E*. The number of flip-flops is set by the parameter *n*. When the active clock edge occurs, the flip-flops in the register cannot change their stored values if the enable *E* is 0. If $E = 1$, the register responds to the active clock edge in the normal way.

```
module reg4 (D, Clock, Resetn, Q);
   input [3:0] D;
   input Clock, Resetn;
   output [3:0] Q;
   reg [3:0] Q;

   always @(posedge Clock or negedge Resetn)
   if (Resetn == 0)
      Q <= 4'b0000;
   else
      Q <= D;

endmodule
```

Figure A.32 A four-bit register with asynchronous clear.

```
module regne (D, Clock, Resetn, E, Q);
   parameter n = 4;
   input [n-1:0] D;
   input Clock, Resetn, E;
   output [n-1:0] Q;
   reg [n-1:0] Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else if (E)
         Q <= D;

endmodule
```

Figure A.33 An *n*-bit register with asynchronous clear and enable.

A.14.6 SHIFT REGISTERS

An example of code that defines a three-bit shift register is provided in Figure A.34. The lines of code are numbered for ease of reference. The shift register has a serial input, *w*, and parallel outputs, Q. The right-most bit in the register is Q[3], and the left-most bit is Q[1]. Shifting is performed in the right-to-left direction. All assignments to Q are synchronized to the clock edge by the (**posedge** Clock) event, hence Q represents the outputs of flip-flops. The statement in line 7 specifies that Q[3] is assigned the value of *w*. The semantics of the non-blocking assignments mean that the subsequent statements do not see the new value of Q[3] until the next time the **always** block is evaluated (in the following clock cycle). In line 8, the current value of Q[3], before it is shifted as a result of line 7, is assigned to Q[2]. Line 9 completes the shift operation by assigning the current value of Q[2] to Q[1], before it is changed as a result of line 8.

The key point that has to be appreciated in the code in Figure A.34 is that the assignments in lines 7 to 9 do not take effect until the end of the **always** block. Hence, all flip-flops change their values at the same time, as required in the shift register. We could write the statements in lines 7 to 9 in any order without changing the meaning of the code.

Blocking Assignments for Sequential Circuits

We said previously that blocking assignments should not be used for sequential circuits. As an example of the semantics involved, Figure A.35 gives the **always** block from Figure A.34 with blocking assignments. The first assignment sets Q[3] = *w*. Since blocking assignments are involved, the next statement sees this new value of Q[3]; hence, it produces Q[2] = Q[3] = *w*. Similarly, the final assignment gives Q[1] = Q[2] = *w*. The code does not describe the desired shift register, but rather loads all flip-flops with the value on the input *w*.

For the code in Figure A.35 to correctly describe a shift register, the ordering of the three assignments has to be reversed. Then the first assignment sets Q[1] to the value of Q[2], the second sets Q[2] to the value of Q[3], and so on. Each successive assignment is

```
1   module shift3 (w, Clock, Q);
2     input w, Clock;
3     output [1:3] Q;
4     reg [1:3] Q;

5     always @(posedge Clock)
6     begin
7       Q[3] <= w;
8       Q[2] <= Q[3];
9       Q[1] <= Q[2];
10    end

11  endmodule
```

Figure A.34 A three-bit shift register.

```
module shift3 (w, Clock, Q);
   input w, Clock;
   output [1:3] Q;
   reg [1:3] Q;

   always @(posedge Clock)
   begin
      Q[3] = w;
      Q[2] = Q[3];
      Q[1] = Q[2];
   end

endmodule
```

Figure A.35 Wrong code for a three-bit shift register.

not affected by the one that precedes it; hence, the semantics of blocking assignments cause no problem.

To avoid the confusing dependence on the ordering of statements, blocking assignments should be avoided when modeling sequential circuits. Also, because they imply differing semantics, blocking and non-blocking assignments should never be mixed in a single **always** construct.

A.14.7 COUNTERS

Figure A.36 presents the code for a four-bit counter with an asynchronous reset input. The counter also has an enable input, E. On the positive clock edge, if E is 1, the count is incremented. If $E = 0$, the counter holds its current value. Since counters are commonly

```
module count4 (Clock, Resetn, E, Q);
   input Clock, Resetn, E;
   output [3:0] Q;
   reg [3:0] Q;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         Q <= 0;
      else if (E)
         Q <= Q + 1;

endmodule
```

Figure A.36 Code for a four-bit counter.

needed in logic circuits, most CAD systems provide (in a library) a selection of counters that can be instantiated in a design.

A.14.8 AN EXAMPLE OF A SEQUENTIAL CIRCUIT

An example of a sequential circuit is given in Figure A.37. It adds together the values of the k-bit input X over successive clock cycles, and stores the sum of these values into a k-bit register. Such a circuit is often called an *accumulator*. To store the result of each addition operation, the circuit includes a k-bit register with an asynchronous reset input, *Resetn*. It also has an enable input, E, which is controlled by a down-counter. The down-counter has an asynchronous load input and a count enable input. The circuit is operated by first setting *Resetn* to 0, which resets the contents of the k-bit register to 0 and loads the down-counter with the k-bit number on the Y input.

Then, in each clock cycle, the counter is decremented, and the sum outputs from the adder are loaded into the register. When the counter reaches 0, the enable inputs on both

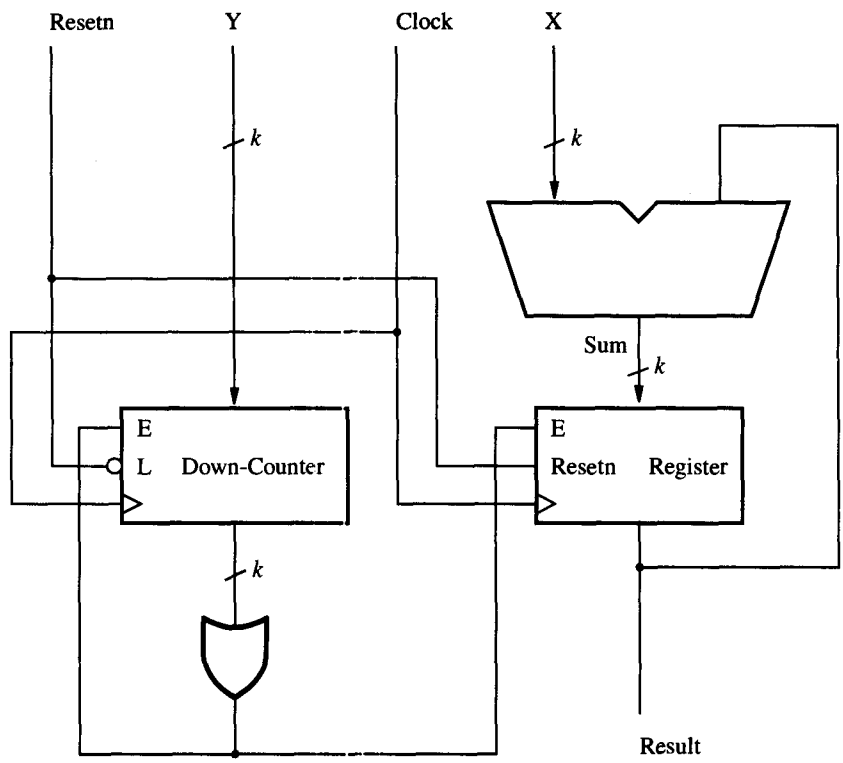


Figure A.37 The accumulator circuit.

the register and counter are set to 0 by the OR gate. The circuit remains in this state until it is reset again. The final value stored in the register is the sum of the values of X in each of the Y clock cycle.

We designed the accumulator circuit by using two subcircuits described in this appendix: *ripple* (Figure A.13) and *regne* (Figure A.33). The complete code is given in Figure A.38. It uses the parameter k to set the number of bits in the input X. Using this parameter in the code makes it easy to change the bit width at a later time if desired. The code defines the signal *Sum* to represent the outputs of the adder; for simplicity, we ignore the possibility of arithmetic overflow and assume that the sum will fit into k bits. The k-bit signal C represents the outputs from the down-counter. The *Go* signal is connected to the enable inputs on the register and counter.

A.14.9 MOORE-TYPE FINITE STATE MACHINES

Figure A.39 gives the state diagram of a simple Moore machine. Verilog code for this machine is shown in Figure A.40. The two-bit vector *y* represents the present state of the

```
module accum (X, Y, Clock, Resetn, Result);
   parameter k = 8;
   input [k−1:0] X, Y;
   input Clock, Resetn;
   output [k−1:0] Result;
   wire [k−1:0] Sum;
   wire Cout, Go;
   reg [k−1:0] C;

   ripple u1 (.carryin(0), .X(X), .Y(Result), .S(Sum), .carryout(Cout));
      defparam u1.n = k;
   regne u2 (.D(Sum), .Clock(Clock), .Resetn(Resetn), .E(Go), .Q(Result));
      defparam u2.n = 8;

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         C <= Y;
      else if (Go)
         C <= C − 1;

   assign Go = | C;

endmodule
```

Figure A.38 Code for a k-bit accumulator circuit.

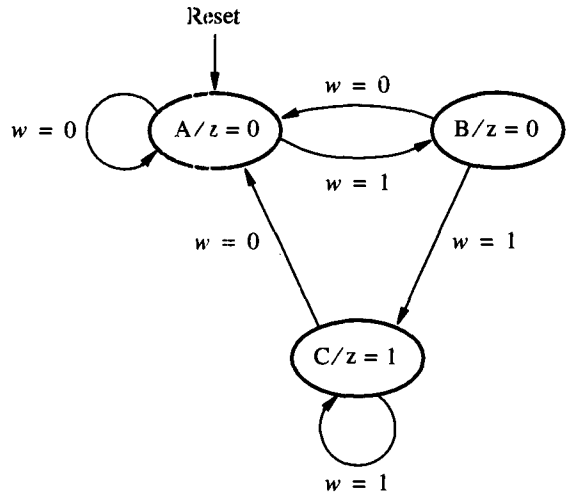


Figure A.39 State diagram of a simple Moore-type FSM.

machine, and the state codes are defined as parameters. Some CAD synthesis systems provide a means of requesting that the state assignment be chosen automatically, but we have specified the assignment manually in this example.

The present state signal *y* corresponds to the outputs of the state flip-flops, and the signal *Y* represents the inputs to the flip-flops, which define the next state. The code has two **always** blocks. The top one describes a combinational circuit and uses a **case** statement to specify the values that *Y* should have for each value of *y*. The other **always** block represents a sequential circuit, which specifies that *y* is assigned the value of *Y* on the positive clock edge. The **always** block also specifies that *y* should take the value *A* when *Resetn* is 0, which provides the asynchronous reset.

Since the machine is of the Moore type, the output *z* can be defined using the assignment statement z = (y == C) that depends only on the present state of the machine. This statement is provided as a continuous assignment at the end of the code, but it could alternatively have been given inside the top **always** block that represents the combinational part of the FSM. This assignment statement cannot be placed inside the bottom **always** block. Doing so would cause *z* to be the output of a separate flip-flip, rather than a combinational function of *y*. This circuit would set *z* to 1 one clock cycle later than required when the machine enters state *C*.

An alternative version of the code for the Moore machine is given in Figure A.41. This code uses a single **always** block to define both the combinational and sequential parts of the finite state machine. In practice, the code in Figure A.40 is used more commonly.

```
module moore (Clock, w, Resetn, z);
  input Clock, w, Resetn;
  output z;
  reg [1:0] y, Y;
  parameter A = 2'b00, B = 2'b01, C = 2'b10;

  always @(w or y)
  begin
    case (y)
      A:  if (w == 0)  Y = A;
          else         Y = B;
      B:  if (w == 0)  Y = A;
          else         Y = C;
      C:  if (w == 0)  Y = A;
          else         Y = C;
      default:         Y = 2'bxx;
    endcase
  end

  always @(posedge Clock or negedge Resetn)
  begin
    if (Resetn == 0)
      y <= A;
    else
      y <= Y;
  end

  assign z = (y == C);

endmodule
```

Figure A.40 Code for a Moore machine.

A.14.10 MEALY-TYPE FINITE STATE MACHINES

A state diagram for a simple Mealy machine is shown in Figure A.42, and the corresponding code is given in Figure A.43. The code has the same structure as in Figure A.40 except that the output z is defined within the top **always** block. The **case** statement specifies that, when the FSM is in state A, z should be 0, but when in state B, z should take the value of w. Since the top **always** block represents a combinational circuit, the output z can change value as soon as the input w changes, as required for the Mealy machine.

```
module moore (Clock, w, Resetn, z);
   input Clock, w, Resetn;
   output z;
   reg [1:0] y;
   parameter A = 2'b00, B = 2'b01, C = 2'b10;

   always @(posedge Clock or negedge Resetn)
   begin
      if (Resetn == 0)
         y <= A;
      else
         case (y)
            A:   if (w == 0)  y <= A;
                 else         y <= B;
            B:   if (w == 0)  y <= A;
                 else         y <= C;
            C:   if (w == 0)  y <= A;
                 else         y <= C;
            default:          y <= 2'bxx;
         endcase
   end

   assign z = (y == C);

endmodule
```

Figure A.41 Alternative version of the code for a Moore machine.

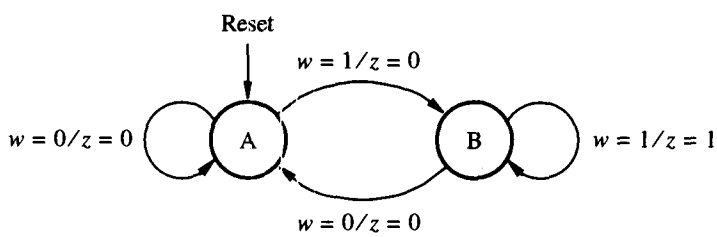


Figure A.42 State diagram of a Mealy-type FSM.

```
module mealy (Clock, w, Resetn, z);
   input Clock, w, Resetn ;
   output z ;
   reg z;
   reg y, Y;
   parameter A = 0, B = 1;

   always @(w or y)
      case (y)
         A: if (w == 0)
            begin
               Y = A;
               z = 0;
            end
            else
            begin
               Y = B;
               z = 0;
            end
         B: if (w == 0)
            begin
               Y = A;
               z = 0;
            end
            else
            begin
               Y = B;
               z = 1;
            end
      endcase

   always @(posedge Clock or negedge Resetn)
      if (Resetn == 0)
         y <= A;
      else
         y <= Y;

endmodule
```

Figure A.43 Code for a Mealy machine.

A.15 GUIDELINES FOR WRITING VERILOG CODE

Modern digital systems are large and complex. A good approach in designing such systems is to decompose them into smaller, manageable parts. Each part can then be designed by making use of the popular subcircuit building blocks that we describe in this book.

When using Verilog for synthesis, it is essential to write code such that the compiler will generate the intended circuit. For example, code for combinational circuits like adders, multiplexers, encoders, and decoders should be written as illustrated in Chapters 5 and 6. Flip-flops, registers, and counters should be described with the style of code presented in Chapter 7; and finite-state machine code should be expressed in the manner shown in Chapter 8. Section 7.14 and Chapter 10 give some examples of how larger circuits can be constructed by employing these common building blocks. This approach of writing HDL code such that the system is built by interconnecting common, relatively simple subcircuits is often referred to as the *register-transfer level* (RTL) style of code. It is the most popular design method used in practice. The rest of this section lists some common errors found in Verilog code and gives some useful guidelines.

Missing Begin-End

The **always** construct requires **begin** and **end** delimiters if there are multiple statements in the block. The compiler does not take indentation into consideration! For example the **always** block

```
always @(w0 or w1 or s)
   if ( s == 1 )
      f = w1;
   f = w0;
```

has one statement in it, not two.

Missing Semicolon

Every Verilog statement must end with a semicolon.

Missing { }

The replication operator requires a lot of brackets. A common mistake is to write {3{A}, 2{B}} instead of {{3{A}}, {2{B}}}.

Accidental Assignment

The statement

```
always @(w0 or w1 or s)
begin
   if ( s = 1 )
      f = w1;
   f = w0;
end
```

does not test the value of *s*. It sets *s* to the value 1.

Incomplete Sensitivity List

Consider the **always** block

```
always @(x)
begin
   s = x ^ y;
   c = x & y;
end
```

When using Verilog for synthesis, this code produces the desired circuit, which is a half-adder. However, if this code is used for simulation of circuits, the values of s and c are updated only as a result of changes in the value of x. Changes in y will have no effect, because y is not included in the sensitivity list. To avoid mismatches between simulation results and synthesized circuits, **always** blocks for combinational circuits should include all signals used on the right-hand side of assignments in the block.

Variables versus Nets

Only nets can serve as the targets of continuous assignment statements. Variables assigned values inside an **always** block have to be of type **reg** or **integer**. It is not possible to make an assignment to a signal both inside an **always** block and via a continuous assignment statement.

Assignments in Multiple always Blocks

In a module that has multiple **always** blocks, all the **always** blocks are concurrent with respect to one another. Therefore, a given variable should never be assigned a value in more than one **always** block. Doing so would mean that there exist multiple concurrent assignments to this variable, which makes no sense.

Blocking versus Non-blocking Assignments

When describing a combinational circuit in an **always** construct, it is best to use only blocking assignments (see section A.11.7). For sequential circuits, non-blocking assignments should be used (see section A.14.6). Blocking and non-blocking assignments should not be mixed in a single **always** block.

It is not possible to model both a combinational output and a sequential output in a single **always** block. The sequential output requires an edge-triggered event control, such as @(**posedge** Clock), and this means that all variables assigned a value in the **always** block will be implemented as the outputs of flip-flops.

Module Instantiation

A **defparam** statement must reference the instance name of a module, but not just the subcircuit's module name. The code

```
bit_count cbits (T, C);
   defparam bit_count.n = 8, bit_count.logn = 3;
```

is illegal, while the code

```
bit_count cbits (T, C);
  defparam cbits.n = 8, cbits.logn = 3;
```

is syntactically correct.

Label, Net, and Variable Names

It is illegal to use any Verilog keyword as a label, net, or variable name. For example, it is illegal to name a signal *input* or *output*.

Labeled Begin-End Blocks

It is legal to define a variable or parameter inside a **begin-end** block, but only if the block has a label. The code

```
always @(X)
begin
  integer k;
  Count = 0;
  for (k = 0; k < n; k = k+1)
    Count = Count + X[k];
end
```

is illegal, while the code

```
always @(X)
begin: label
  integer k;
  Count = 0;
  for (k = 0; k < n; k = k+1)
    Count = Count + X[k];
end
```

is syntactically correct.

Implied Memory

As shown in section A.14.1, implied memory is used to describe storage elements. Care must be taken to avoid unintentional implied memory. The code

```
always @(LA)
  if (LA == 1)
    EA = 1;
```

results in implied memory for the *EA* variable. If this is not desired, then the code can be fixed by writing

```
always @(LA)
  if (LA == 1)
    EA = 1;
  else
    EA = 0;
```

Implied memory also applies to **case** statements. The code

```
always @(W)
   case (W)
      2'b01:  EA = 1;
      2'b10:  EB = 1;
   endcase
```

does not specify the value of the *EA* variable when *W* is not equal to 01, and it does not specify the value of *EB* when *W* is not equal to 10. To avoid having implied memory for both *EA* and *EB*, these variables should be assigned default values, as in the code

```
always @(W)
begin
   EA = 0; EB = 0;
   case (W)
      2'b01:  EA = 1;
      2'b10:  EB = 1;
   endcase
end
```

A.16 MAX+PLUSII VERILOG SUPPORT

A useful source of information on Verilog is the MAX+plusII CAD system that accompanies the book. The on-line help included with the software describes how to use Verilog with MAX+plusII, and the *templates* provided with the Text Editor tool are a convenient guide to Verilog syntax. We describe how to access these features of the CAD tools in Appendix B.

A.16.1 LIMITATIONS IN MAX+PLUSII

Verilog support in MAX+PlusII has some limitations. The tool does not support memories (two-dimensional arrays) and Verilog **tasks**. Some other limitations follow.

Verilog Gates

Each logic gate can have up to 12 inputs, with the exception of XOR and XNOR, which can have only two inputs.

Module Instantiation

MAX+PlusII does not support the #(parameter_override) syntax for specifying the value of a subcircuit parameter. The **defparam** feature is supported, but has the following limitation: the **defparam** statement must appear immediately following the corresponding module instantiation statement. Also, parameters can be referred to by instance name only. For a subcircuit module *bit_count* with the instance name *cbits*, **defparam** bit_count.cbits.n = 8 is not allowed, but **defparam** cbits.n = 8 is supported.

Tri-state Logic

Although MAX+PlusII supports the **tri** net type, the software does not allow tri-state buffers to be described by using assignment statements. Instead, tri-state buffers have to be instantiated in the code with primitive tri-state buffer gates.

Verilog 2001 Support

MAX+PlusII does not support any of the new features in Verilog 2001.

A.17 CONCLUDING REMARKS

This appendix describes the important features of Verilog that are useful for the synthesis of logic circuits. As mentioned earlier, we do not discuss many features of Verilog which are useful only for simulation of circuits or for other purposes. A reader who wishes to learn more about using Verilog can refer to specialized books [1–8].

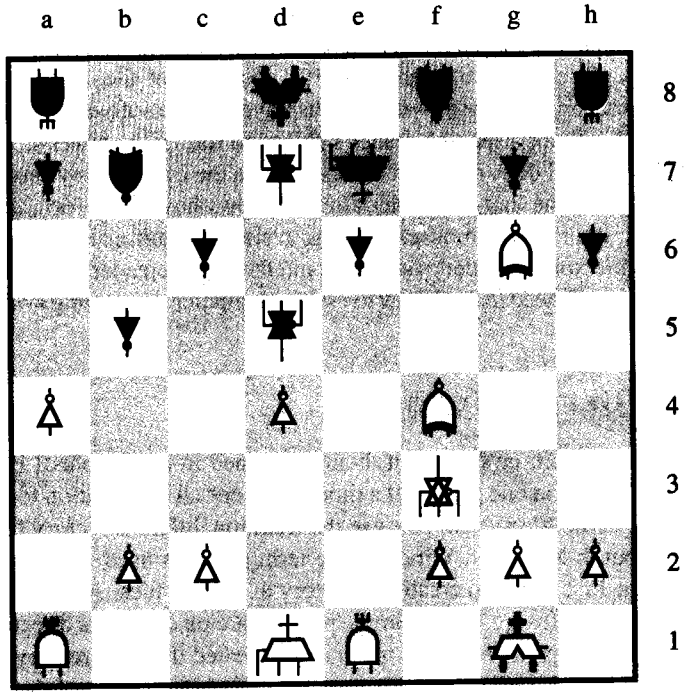
REFERENCES

1. Institute of Electrical and Electronics Engineers, *IEEE Standard Verilog Hardware Description Language Reference Manual*, (IEEE: Piscataway, NJ, 1995).
2. D. A. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*, 4th ed., (Kluwer: Norwell, MA, 1998).
3. S. Palnitkar, *Verilog HDL—A Guide to Digital Design and Synthesis*, (Prentice-Hall: Upper Saddle River, NJ, 1996).
4. D. R. Smith and P. D. Franzon, *Verilog Styles for Synthesis of Digital Systems*, (Prentice-Hall: Upper Saddle River, NJ, 2000).
5. Z. Navabi, *Verilog Digital System Design*, (McGraw-Hill: New York, 1999).
6. J. Bhasker, *Verilog HDL Synthesis—A Practical Primer*, (Star Galaxy Publishing: Allentown, PA, 1998).
7. D. J. Smith, *HDL Chip Design*, (Doone Publications: Madison, AL, 1996).
8. S. Sutherland, *Verilog 2001. A Guide to the New Features of the Verilog Hardware Description Language*, (Kluwer: Hingham, MA, 2001).

appendix

B

TUTORIAL 1



13. Rf1–e1, Nf6–d5

MAX+plusII is a sophisticated and easy to use CAD system. In this tutorial we introduce the design of logic circuits using MAX+plusII. Step-by-step instructions are presented for performing design entry with three methods: using schematic capture, writing Verilog code, and using a truth table. The tutorial also illustrates functional simulation.

B.1 INTRODUCTION

This tutorial assumes that the reader has access to a computer on which MAX+plusII is installed. Instructions for installing the copy of MAX+plusII provided with the book are included on the CD-ROM. The MAX+plusII software will run on several different types of computer systems. For this tutorial a computer running a Microsoft Windows operating systems is assumed. Although MAX+plusII operates similarly on all of the supported types of computers, there are some minor differences. A reader who is not using a Microsoft Windows operating system may experience some slight discrepancies from this tutorial. Examples of potential differences are the locations of files in the computer's file system and the exact appearance of windows displayed by the software. All such discrepancies are minor and will not affect the reader's ability to follow the tutorial.

This tutorial does not describe how to use the operating system provided on the computer. We assume that the reader already knows how to perform actions such as running programs, operating a mouse, moving, resizing, minimizing and maximizing windows, creating directories (folders) and files, and the like. A reader who is not familiar with these procedures will need to learn how to use the computer's operating system before proceeding.

B.1.1 GETTING STARTED

Each logic circuit, or subcircuit, being designed in MAX+plusII is called a *project*. The software works on one project at a time and keeps all information for that project in a single directory in the file system (we use the traditional term *directory* for a location in the file system, but in Microsoft Windows the term *folder* is used). To begin a new logic circuit design, the first step is to create a directory to hold its files. As part of the installation of the MAX+plusII software, a few sample projects are placed into a directory called *\max2work*. To hold the design files for this tutorial, we created the subdirectory *\max2work\tutorial1*. The location and name of the directory is not important; hence the reader may use any valid directory.

To create a directory to work in, use the normal utilities provided by the computer's operating system. MAX+plusII is not involved in this step. After the directory has been created, start the MAX+plusII software. You should see a window similar to the one in Figure B.1. This window is called the *MAX+plusII Manager*. It provides access to all the features of MAX+plusII, which the user selects with the computer mouse.

Most of the commands provided by MAX+plusII are accessed by using a set of menus that are located in the Manager window below the title bar. For example, in Figure B.1

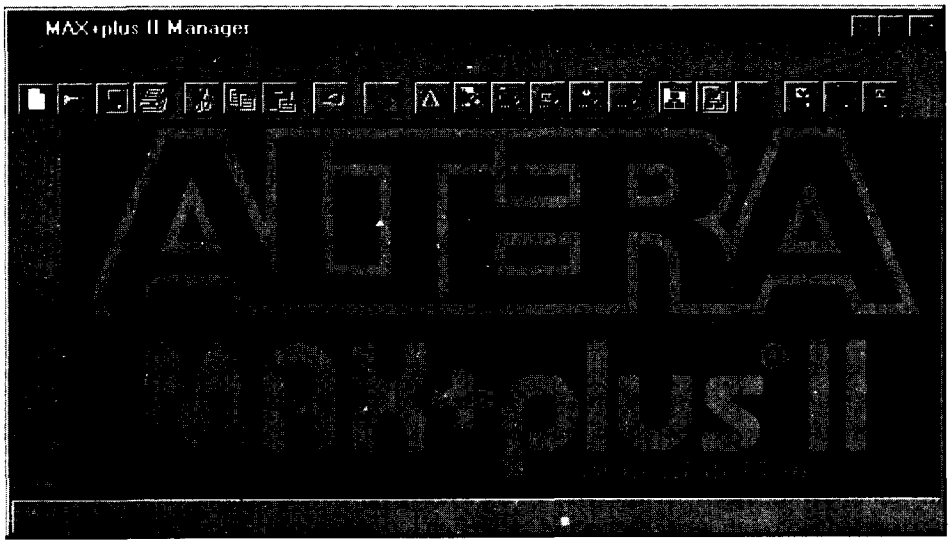


Figure B.1 The MAX+plusII Manager window.

clicking the left mouse button on the menu named File opens the menu shown in Figure B.2. Clicking the left mouse button on the entry Exit MAX+plusII Alt+F4 exits from MAX+plusII. In general, whenever the mouse is used to select something, the *left* button is used. Hence we will not normally specify which button to use. In the few cases when it is necessary to use the *right* mouse button, it will be specified explicitly. We should note that the Alt+F4 part of

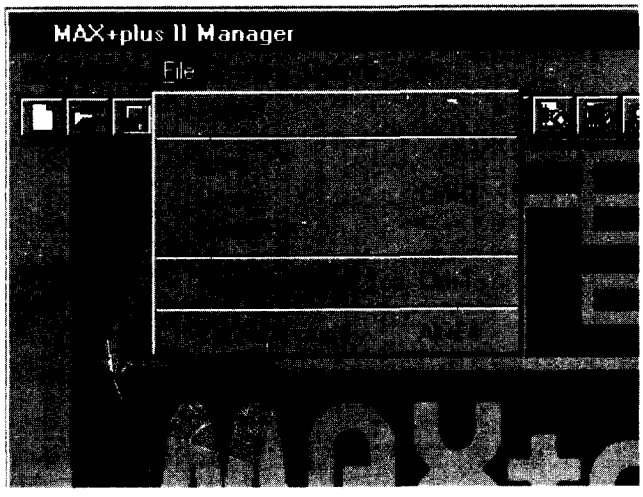


Figure B.2 The File menu in the Manager window.

the menu item indicates a keyboard shortcut; instead of using the mouse, the command can alternatively be invoked by the holding down the Alt key on the keyboard and pressing the F4 function key. Keyboard shortcuts are available for a few of the MAX+plusII commands, but commands are usually invoked using the mouse. For some commands it is necessary to access two or more menus in sequence. We use the convention Menu1 | Menu2 | Item to indicate that to select the desired command the user should first click the left mouse button on Menu1, then within this menu click on Menu2, and then within Menu2 click on Item. For example, File | Exit MAX+plusII describes how to use the mouse to exit from the MAX+plusII system.

The MAX+plusII system includes 11 main software modules, called *applications*. They can be accessed in two different ways. First, all the applications can be invoked via the MAX+plusII menu in the Manager window, as illustrated in Figure B.3. Second, some of the applications can be invoked using the small icons that appear below the Manager title bar. (If no icons are visible under the Manager title bar, select Options | Preferences to open the Preferences dialog box. Then use the mouse to place a check mark beside the entry for Show Toolbar and click OK.) To see which applications in Figure B.3 a particular icon is associated with, place the mouse pointer on top of the icon; the Manager displays a message near the bottom of the window that gives the name of the application.

The applications introduced in this tutorial include the Graphic Editor, Text Editor, Waveform Editor, Compiler, Simulator, Message Processor, and Hierarchy Display. The others are introduced in Tutorial 2.

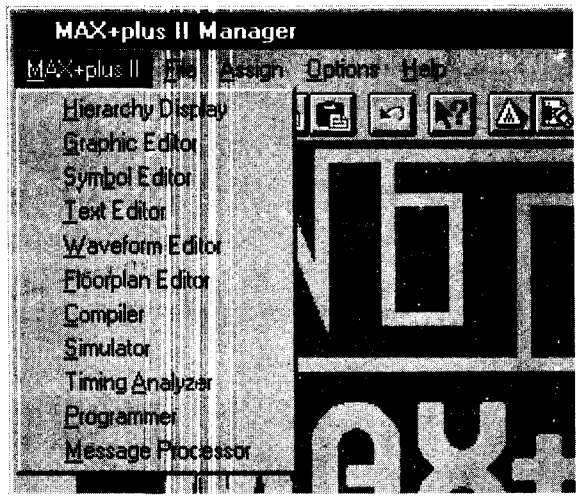


Figure B.3 The MAX+plus II menu in the Manager window.

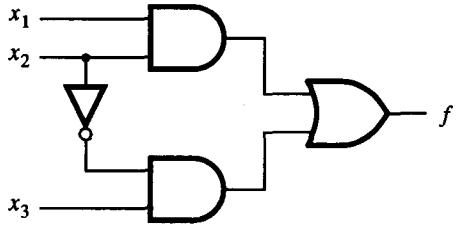
MAX+plusII On-Line Help

MAX+plusII provides comprehensive on-line documentation that answers most of the questions that may arise when using the software. The documentation is accessed from the Help menu in the Manager window. To get some idea of the extent of documentation provided, it is worthwhile for the reader to browse through the Help menu. For instance, selecting Help | MAX+plusII Table of Contents shows all the categories of documentation available.

The user can quickly search through the Help topics by selecting Help | Search for Help on, which opens a dialog box into which keywords can be entered. The available Help topics that match the keywords are automatically displayed. Two other methods are provided for quickly finding documentation for specific topics. First, while using any application, pressing the F1 function key on the keyboard opens a Help display that shows the commands available for that application. Second, in some instances holding down the Shift key and pressing the F1 key changes the mouse pointer into a *help* pointer. This feature is available when using the schematic capture tool provided in MAX+plusII. Clicking the help pointer on any circuit element in a schematic automatically displays any documentation that is available for that circuit element.

B.2 DESIGN ENTRY USING SCHEMATIC CAPTURE

In Chapter 2 we introduced three types of design entry methods: truth tables, schematic capture, and Verilog. This section illustrates the process of using the schematic capture tool provided in MAX+plusII, which is called the Graphic Editor. As a simple example, we will draw a schematic for the logic function $f = x_1x_2 + \bar{x}_2x_3$. A circuit diagram for f was shown in Figure 2.30 and is reproduced as Figure B.4*a*. The truth table for f is given in Figure B.4*b*. Chapter 2 also introduced functional simulation. After creating the schematic, we show how to use the functional simulator in MAX+plusII to verify the schematic's functionality.



(a) Circuit

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure B.4 The logic function of Figure 2.30.

B.2.1 SPECIFYING THE PROJECT NAME

As a first step we will specify the name of the design project. In the Manager window select File | Project | Name to open the pop-up box illustrated in Figure B.5. It is necessary to specify the location of the directory where MAX+plusII will store any files created for the project. For this example the directory used is named d:*max2work**tutorial1*. The disk drive designation, d:, is selected using the Drives pull-down menu shown in Figure B.5. The directory name is selected using the box labeled Directories. Use the mouse to double-click on the directory names displayed in the box until the proper directory is selected; the selected directory appears next to the words Directory is, as illustrated in the figure. In the box labeled Project Name, type *graphic1* as the name for this project and then click OK. Observe that the name of the project is displayed in the title bar of the Manager window.

B.2.2 USING THE GRAPHIC EDITOR

The next step is to draw the schematic. In the Manager window select MAX+plusII | Graphic Editor. The Graphic Editor window appears inside the Manager window. It may be helpful to move or resize the Graphic Editor window and to increase the size of the Manager window to provide more work space. In the screen capture in Figure B.6, the Graphic Editor window is maximized so that it fills the entire Manager window.

The title bar in Figure B.6 includes some menu names and icons that did not appear in Figure B.1. This is because the Manager window always indicates the features available in whatever application is currently being used. A number of icons that are used to invoke Graphic Editor features also appear along the left edge of the window. To see a description of the Graphic Editor feature associated with each icon, position the mouse on top of the icon; a message is displayed near the bottom of the window. Two of the most useful icons

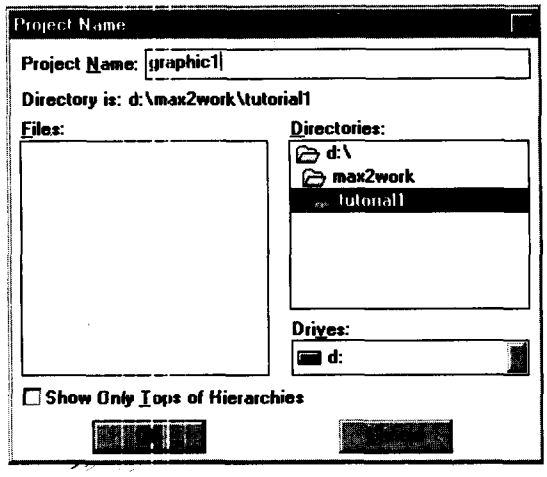


Figure B.5 Specifying the name and working directory for a project.

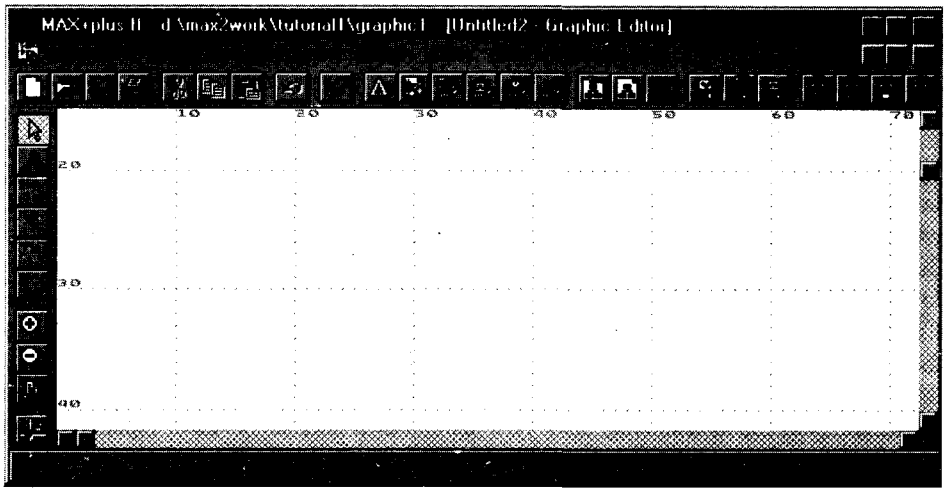


Figure B.6 The Graphic Editor display.

are the ones that look like a magnifying glass. These icons are used to see a larger or smaller view of the schematic.

Naming the Schematic

The schematic being created must be given a name. Select File | Save As to open the pop-up box depicted in Figure B.7. The directory that we chose for the project is already selected in the pop-up box. The Graphic Editor will create a separate file for the schematic and store it in the project's directory. In the box labeled File Name, type *graphic1.gdf*.

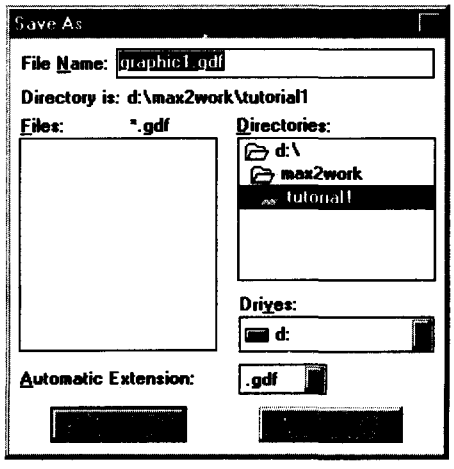


Figure B.7 Specifying the name of a schematic.

You must use exactly this name. The name *graphic1* must match the name of the project, and the filename extension *gdf*, which stands for *graphic design file*, must be used for all schematics. Click OK to return to the Graphic Editor.

Importing Logic-Gate Symbols

The Graphic Editor provides several libraries which contain circuit elements that can be imported into a schematic. For our simple example we will use a library called *Primitives*, which contains basic logic gates. To access the library, double-click on the blank space in the middle of the Graphic Editor display to open the pop-up box in Figure B.8 (another way to open this box is to select Symbol | Enter Symbol). The box labeled Symbol Libraries lists several available libraries, including the Primitives library. To open it, double-click on the line that ends with the word *prim*. A list of the logic gates in the library is automatically displayed in the Symbol Files box. Double-click on the *and2* symbol to import it into the schematic (you can alternatively click on *and2* and then click OK). A two-input AND-gate symbol now appears in the Graphic Editor window.

Any symbol in a schematic can be selected using the mouse. Position the mouse pointer on top of the AND-gate symbol in the schematic and click the mouse to select it. The symbol is highlighted in red. To move a symbol, select it and, while continuing to press the mouse button, drag the mouse to move the symbol. To make it easier to position the graphical symbols, a grid of guidelines can be displayed in the Graphic Editor window by selecting Options | Show Guidelines. Spacing between grid lines can be adjusted using Options | Guideline Spacing.

The logic function f requires a second two-input AND gate, a two-input OR gate, and a NOT gate. Use the following steps to import them into the schematic.

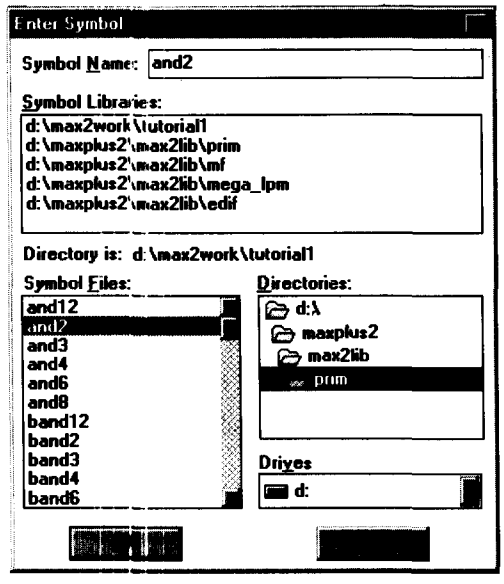


Figure B.8 Importing a logic gate from the Primitives library.

Position the mouse pointer over the AND-gate symbol that has already been imported. Press and hold down the Ctrl keyboard key and click and drag the mouse away from the AND-gate symbol. The Graphic Editor automatically imports a second instance of the AND-gate symbol. This shortcut procedure for making a copy of a circuit element is convenient when you need many instances of the same element in a schematic. Of course, an alternative approach is to import each instance of the symbol by opening the Primitives library as described above.

To import the OR-gate symbol, again double-click on a blank space in the Graphic Editor and then double-click on the Primitives library. In the box labeled **Symbol Files**, use the scroll bar to scroll down through the list of gates to find the symbol named *or2*. Import this symbol into the schematic. Next import the NOT gate using the same procedure. To orient the NOT gate so that it points downward, as depicted in Figure B.4*a*, select the NOT-gate symbol and then use the command **Edit | Rotate | 270** to rotate the symbol 270 degrees counterclockwise. The symbols in the schematic can be moved by selecting them and dragging the mouse, as explained above. More than one symbol can be selected at the same time by clicking the mouse and dragging an outline around the symbols. The selected symbols are moved together by clicking on any one of them and moving it. Experiment with this procedure. Arrange the symbols so that the schematic appears similar to the one in Figure B.9.

Importing Input and Output Symbols

Now that the logic-gate symbols have been entered, it is necessary to import symbols to represent the input and output ports of the circuit. Open the Primitives library again. Click the mouse anywhere in the box labeled **Symbol Files** and then type the letter "i" to jump ahead in the list of symbols to those whose names begin with *i*. This shortcut can be used in addition to the scroll bars provided on the **Symbol Files** box. Import the symbol

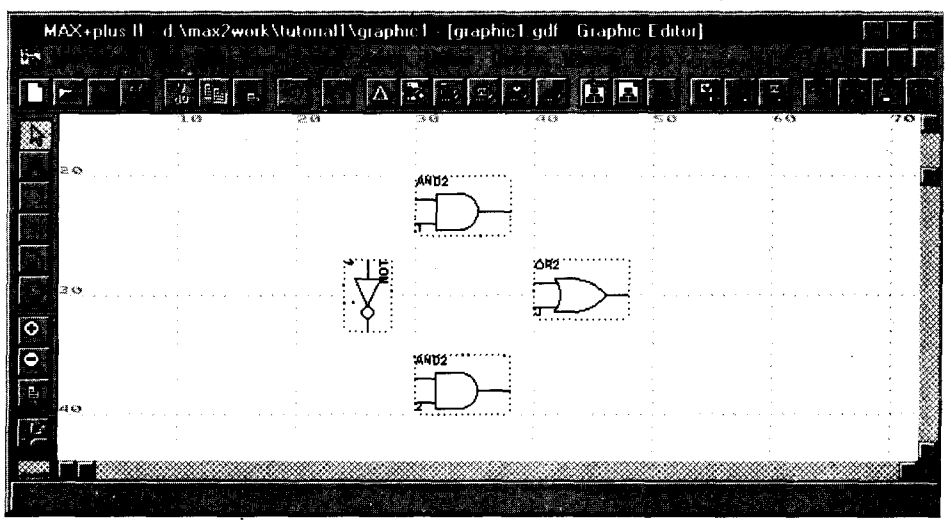


Figure B.9 A partially completed schematic for the circuit in Figure B.4.

named *input* into the schematic. Import two additional instances of the input symbol. To represent the output of the circuit, open the Primitives library and import the symbol named output. Arrange the symbols to appear as illustrated in Figure B.10.

Assigning Names to Input and Output Symbols

Point to the word PIN_NAME on the input pin symbol in the upper-left corner of the schematic and double-click the mouse. The pin name is selected, allowing a new pin name to be typed. Type $x1$ as the pin name. Hitting carriage return immediately after typing the pin name causes the mouse focus to move to the pin directly below the one currently being named. This method can be used to name any number of pins. Assign the names $x2$ and $x3$ to the middle and bottom input pins, respectively. Finally, assign the name f to the output pin.

Connecting Nodes with Wires

The next step is to draw lines (wires) to connect the symbols in the schematic together. Click on the icon that looks like an arrowhead along the left edge of the Manager window. This icon is called the Selection tool, and it allows the Graphic Editor to change automatically between the modes of selecting a symbol on the screen or drawing wires to interconnect symbols. The appropriate mode is chosen depending on where the mouse is pointing.

Move the mouse pointer on top of the $x1$ input symbol. The mouse pointer appears as an arrowhead when pointing anywhere on the symbol except at the right edge. The arrowhead means that the symbol will be selected if the mouse button is pressed. Move the mouse to point to the small line, called a *pinstub*, on the right edge of the $x1$ input symbol. The mouse pointer changes to a crosshair, which allows a wire to be drawn to connect the

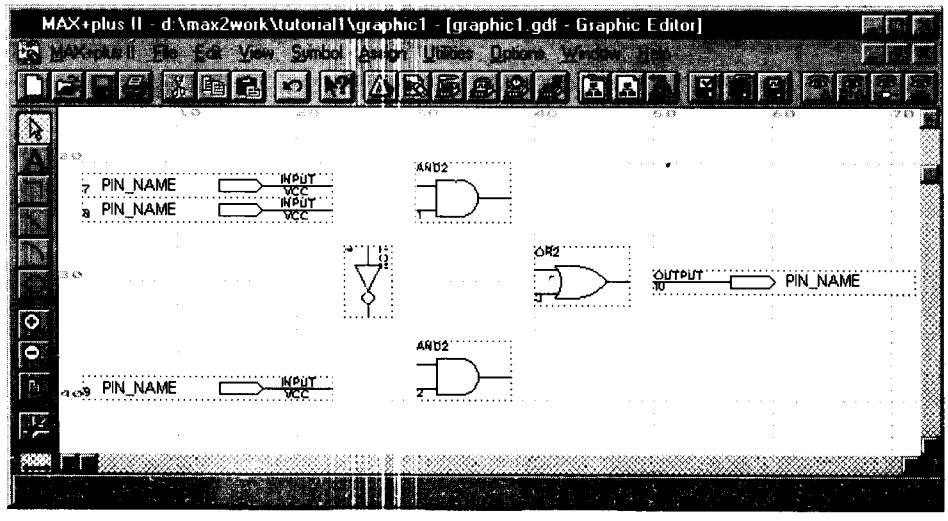


Figure B.10 Input and output symbols added to the schematic in Figure B.9.

pinstub to another location in the schematic. A connection between two or more pinstubs in a schematic is called a *node*. The name derives from electrical terminology, where the term *node* refers to any number of points in a circuit that are connected together by wires and thus have the same voltage.

Connect the input symbol for $x1$ to the AND gate at the top of the schematic as follows. While the mouse is pointing at the pinstub on the $x1$ symbol, click and hold the mouse button. Drag the mouse to the right until the line (wire) that is drawn reaches the pinstub on the top input of the AND gate; then release the button. The two pinstubs are now connected and represent a single node in the circuit.

Use the same procedure to draw a wire from the pinstub on the $x2$ input symbol to the other input on the AND gate. Then draw a wire from the pinstub on the input of the NOT gate upward until it reaches the wire connecting $x2$ to the AND gate. Release the mouse button and observe that a connecting dot is drawn automatically. The three pinstubs corresponding to the $x2$ input symbol, the AND-gate input, and the NOT-gate input now represent a single node in the circuit. Figure B.11 shows a magnified view of the part of the schematic that contains the connections drawn so far. To increase or decrease the portion of the schematic displayed on the screen, use the icons that look like magnifying glasses on the left side of the Manager window.

To complete the schematic, connect the output of the NOT gate to the lower AND gate and connect the input symbol for $x3$ to that AND gate as well. Connect the outputs of the two AND gates to the OR gate and connect the OR gate to the f output symbol. If any mistakes are made while connecting the symbols, erroneous wires can be selected with the mouse and then removed by pressing the Delete key or by selecting **Edit | Delete**. The finished schematic is depicted in Figure B.12. Save the schematic using **File | Save**.

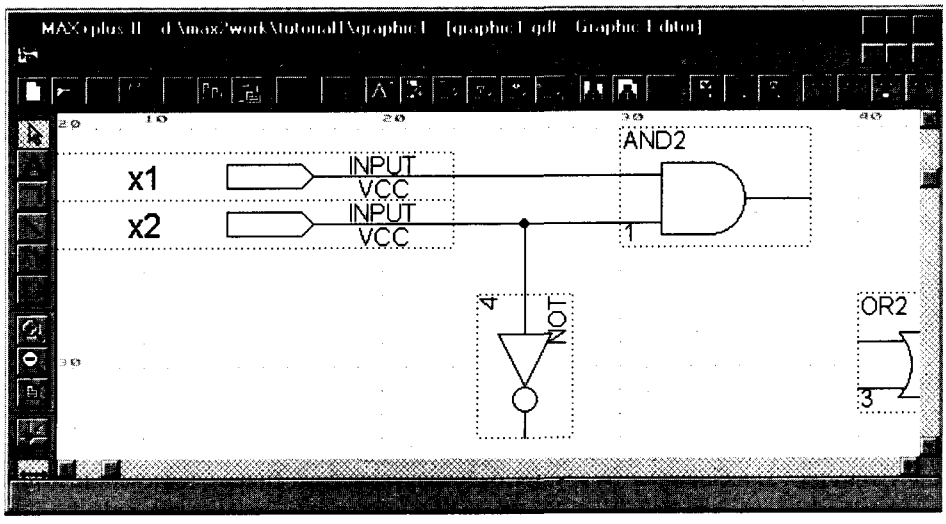


Figure B.11 Connecting the symbols in the schematic from Figure B.10.

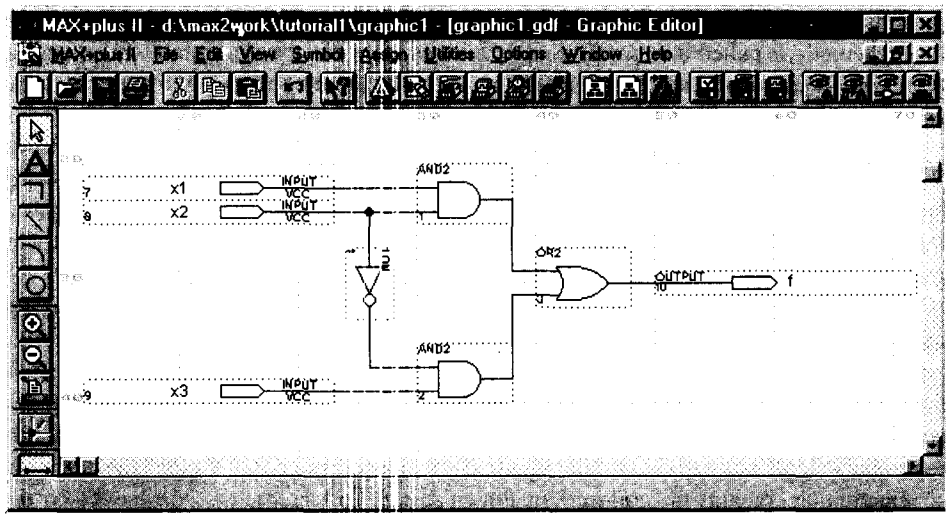


Figure B.12 The completed schematic for the circuit in Figure B.4.

Since our example schematic is quite simple, it is easy to draw all the wires in the circuit without producing a messy diagram. However, in larger schematics some nodes that have to be connected may be far apart, in which case it is awkward to draw wires between them. In such cases the nodes are connected by assigning labels to them, instead of drawing wires. We will illustrate this method of connecting nodes in section D.3.1.

B.2.3 SYNTHESIZING A CIRCUIT FROM THE SCHEMATIC

As we explained in section 2.9.2, after a schematic is entered into a CAD system, it is processed by initial synthesis tools. These tools analyze the schematic and generate a Boolean equation for each logic function in the circuit. In MAX+plusII the synthesis tools are controlled by the application program called the *Compiler*.

Using the Compiler

To open the Compiler window, click the mouse on the Compiler icon (it looks like a factory with a smoke stack) below the Manager window title bar or select MAX+plusII | Compiler.

For this tutorial we will use only the tools that are needed to allow us to perform a functional simulation of the schematic. To tell the Compiler to use these tools, select Processing | Functional SNF Extractor. The Compiler window should appear as shown in Figure B.13. The window shows three software modules that are invoked in sequence by the Compiler. The Compiler Netlist Extractor and Database Builder represent the initial synthesis tools. The module called Functional SNF Extractor creates a file, called a *simulator netlist file (SNF)*, which describes the functionality of the circuit and is used by the functional simulator.

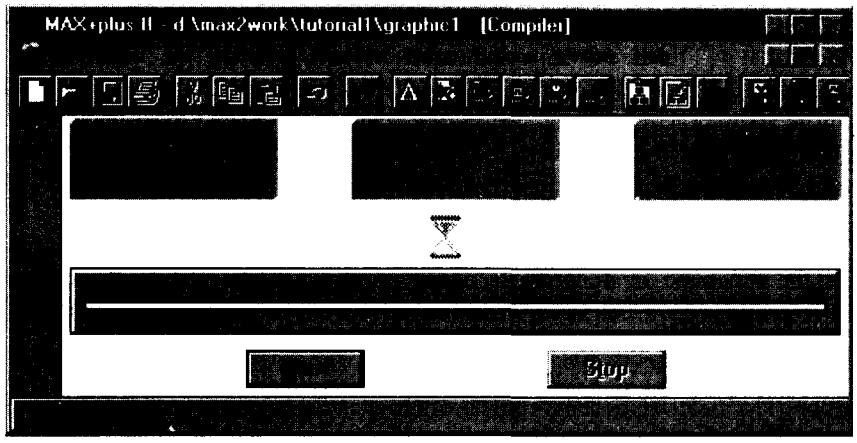


Figure B.13 The Compiler display.

Click the mouse on the Start button in the Compiler window. The Compiler indicates its progress by displaying a red progress bar and by placing an icon under each of the three software modules as they are executed. When the Compiler is finished, a window should be displayed that indicates zero warnings and zero errors. Click OK in this window to return to the Compiler window.

If the Compiler does not specify zero warnings and zero errors, then at least one mistake has been made when entering the schematic. In this case the Compiler opens a window called the Message Processor, which displays a message concerning each warning or error generated. An example showing how the Message Processor can be used to quickly locate and fix errors in a schematic is given in section B.2.5.

To close the Compiler window, use the *Close button* (it is an X) located in the top-right corner of its window.

B.2.4 PERFORMING FUNCTIONAL SIMULATION

Before the schematic can be simulated, it is necessary to create the desired waveforms, called *test vectors*, to represent the input signals. For this tutorial we will use the MAX+plusII Waveform Editor to draw test vectors, but it is also possible to use a text editor to create test vectors in a plain text (ASCII) file. Documentation pertaining to ASCII test vectors can be opened by selecting Help | MAX+plusII Table of Contents. Click on Simulator, then click on Basic Tools, and finally click on Vector File (.vec).

Using the Waveform Editor

Open the Waveform Editor window by selecting MAX+plusII | Waveform Editor. Because the Waveform Editor has many uses, it is necessary to indicate that we wish to enter test vectors for simulation purposes. Select File | Save As and type (if not already

there) *graphic1.scf* in the box labeled File Name. A file with *scf* extension stores the waveforms that will be used as simulation test vectors.

Select Node | Enter Nodes from SNF to open the pop-up box shown in Figure B.14. Click on the List button in the upper-right corner of this box to display the names of the nodes in the current project in the box labeled Available Nodes & Groups. Click the mouse on the name $x3$ to highlight it. Click on the button labeled => to copy $x3$ into the box labeled Selected Nodes & Groups. Use the same procedure to select each of the other signals and copy them into the Selected Nodes & Groups box. It is also possible to select multiple nodes at the same time, by dragging the mouse upward or downward inside the Available Nodes & Groups box. Click OK to return to the Waveform Editor. The nodes $x1$, $x2$, $x3$, and f are now shown in the waveform display.

We will now specify the logic values to be used for the input signals during functional simulation. The logic values at the output f will be generated automatically by the simulator.

Select File | End Time to specify the total amount of time for which the circuit will be simulated. In the box labeled Time, type *160ns* to set the total simulation time to 160 nanoseconds. This amount of time is rather arbitrary because functional simulation does not include any timing delays, as discussed in section 2.9.3. The concept of *simulation time* will become more significant in Tutorial 2 when timing simulation is introduced. Click OK to return to the Waveform Editor. Select View | Fit in Window so that the entire time range from 0 to 160 ns is visible in the Waveform Editor display. In the Options menu make sure that Show Grid has a check mark next to it so that the Waveform Editor displays light vertical guidelines in the waveform area of the display. The guidelines provide a visual aid for positioning the mouse when drawing waveforms. Select Options | Grid Size and type *20ns* in the box labeled Grid Size. Click the mouse when pointing to any of the guidelines and observe that a vertical reference line is drawn at that point. We will use the reference line in Tutorial 2. Figure B.15 shows how the Waveform Editor window should look at this

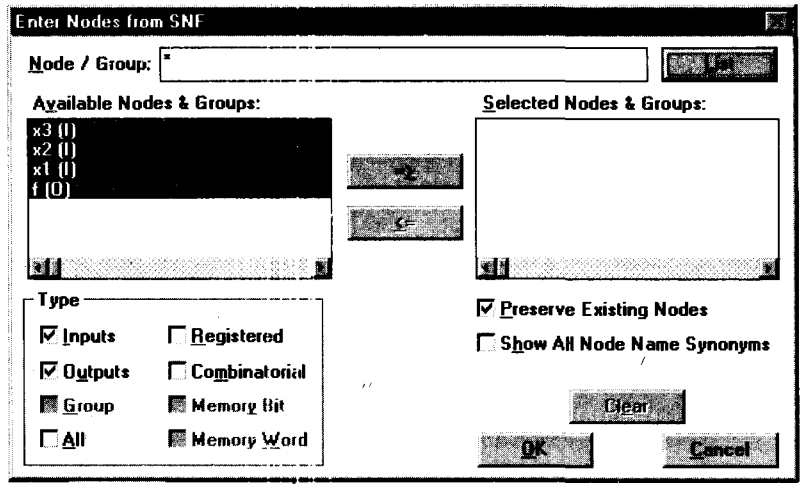


Figure B.14 Selecting nodes for simulation.

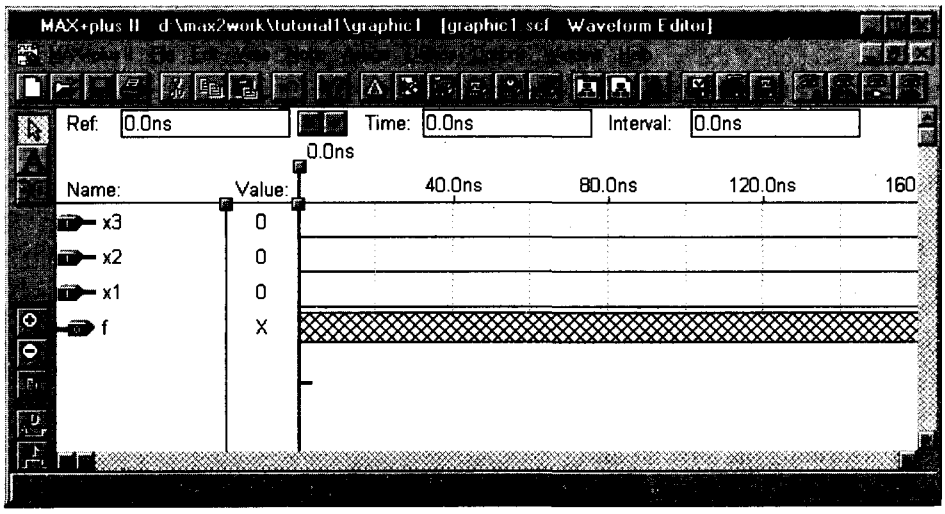


Figure B.15 The Waveform Editor display.

point. The input waveforms are set to logic value 0, and the output is shown as a hashed-line pattern that indicates that the logic value has not yet been determined.

To thoroughly test the circuit during simulation, it is desirable to use as many different values of the input signals as possible. For our small example, there are only eight different valuations, and so it is easy to include all of them. To make all eight valuations fit in the 160 ns simulation time, the signal valuations have to change every 20 ns. To create the waveforms for the input signals, do the following.

Activate the *Waveform Editing* tool by pressing its icon on the left edge of the window. The icon is shown in the top-left corner of Figure B.16; it looks like two arrows pointing left and right. Position the mouse pointer over the waveform for input $x3$ at the 20 ns grid line. Press and drag the mouse to the right to highlight the section of the $x3$ waveform from 20 ns to 40 ns, as illustrated in Figure B.16. The Waveform Editing Tool automatically

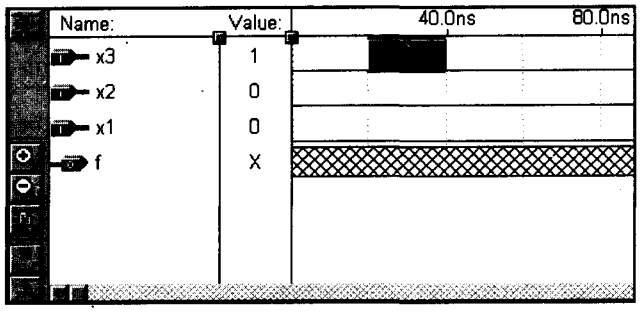


Figure B.16 Editing the waveform for $x3$ from Figure B.15.

changes the selected portion of the waveform from its present value 0 to the value 1. Next select the section of the waveform for $x3$ between 60 ns and 80 ns to set it to 1. Continue in this manner to set every second 20 ns section of $x3$ to 1.

An alternative way to draw waveforms is to use the Selection tool, which is activated by selecting the icon that looks like an arrowhead along the left edge of the window. Using the Selection tool, the procedure for drawing a waveform is to first select a section of the waveform by dragging the mouse over it. The highlighted section can be set to 1 by selecting Edit | Overwrite | High. The highlighted section can also be changed by using the buttons labeled 0 or 1 along the left edge of the window.

Use the Waveform Editing tool to set the waveform for $x2$ to 1 in the range from 40 ns to 80 ns, as well as from 120 ns to 160 ns. Also, set the waveform for $x1$ to 1 in the range from 80 ns to 160 ns. The waveforms drawn, as illustrated in Figure B.17, now include all eight input valuations. Select File | Save to save the waveforms in the *graphic1.scf* file.

Performing the Simulation

To open the Simulator window, shown in Figure B.18, click on its icon (it looks like a computer with a waveform on the screen) or Select MAX+plusII | Simulator. MAX+plusII provides both functional simulation and timing simulation. The type of simulation used by the Simulator application is determined automatically by the settings used in the Compiler application. The Simulator will perform a functional simulation in this case because we instructed the Compiler to generate information for functional simulation, as discussed for Figure B.13.

Observe in Figure B.18 that the Simulator specifies that it will use the file called *graphic1.scf* as the simulator input and will perform the simulation for the time range from 0 to 160 ns. Click the Start button to perform the simulation. The Simulator displays a message indicating that no errors were generated. Click OK to return to the Simulator window. The simulator stores the results of the simulation in the *graphic1.scf* file. To view the file, click on the Open SCF button in the simulator window, which automatically opens the Waveform Editor window and displays the file. As illustrated in Figure B.19, the Simulator creates a waveform for the output f. The reader should verify that the generated waveform corresponds to the truth table for f given in Figure B.4*b*. The Waveform Editor and Simulator windows can now be closed.

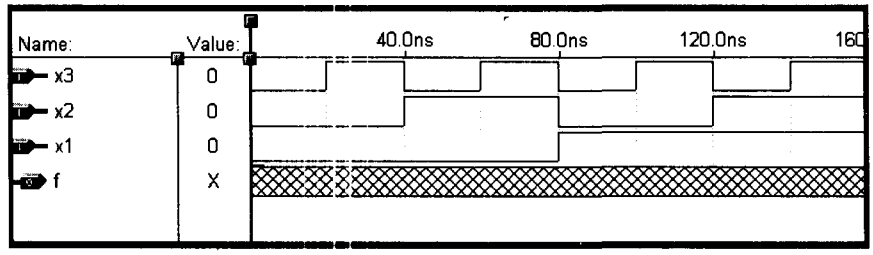


Figure B.17 The completed waveforms for $x1$, $x2$, and $x3$.

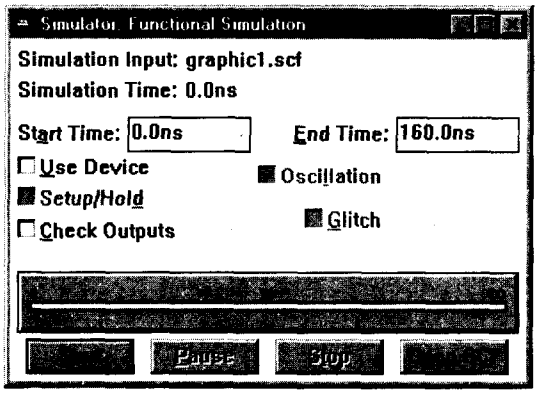


Figure B.18 The Simulator display.

B.2.5 USING THE MESSAGE PROCESSOR TO LOCATE AND FIX ERRORS

In the description in section B.2.3 of how the Compiler is used to synthesize a circuit from the schematic, we said that the Compiler should produce a message stating that no warnings or errors were generated. In this section we illustrate what happens when there is an error in the schematic. To insert an error in the schematic created for f, reopen the schematic by selecting File | Open to open the pop-up box shown in Figure B.20. In the box labeled Show in Files List, click on Graphic Editor Files. Then in the box labeled Files, click on the name

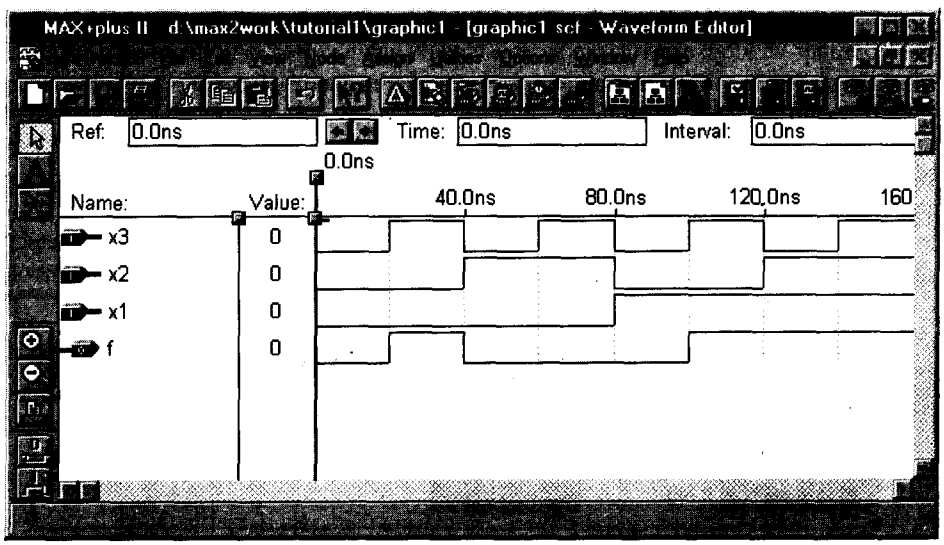


Figure B.19 Functional simulation results for the waveforms in Figure B.17.

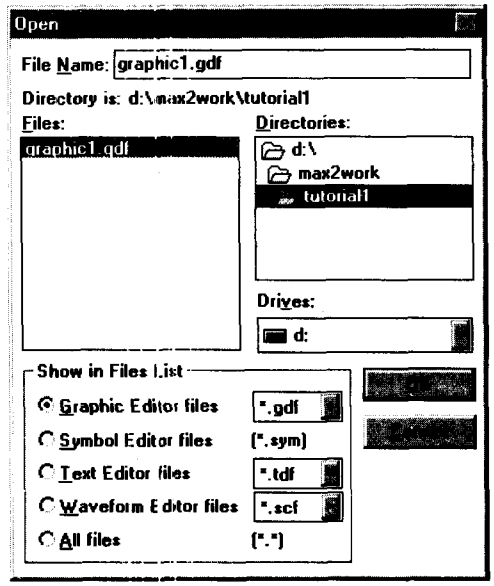


Figure B.20 The dialog box used to reopen the schematic.

graphic1.gdf to put this name in the box labeled File Name. Alternatively, *graphic1.gdf* can be typed into the box rather than using the mouse to select it from the list of files. Click OK to open the file inside the Graphic Editor.

Use the mouse to select the wire that connects the output of the OR gate to the *f* output symbol. Delete the wire by pressing the Delete key; then save the schematic file. Open the Compiler window and run the synthesis tools again. The Compiler should produce a message stating that one warning and one error were found. Click OK. A window, called the Message Processor, is automatically opened to display the messages generated by the Compiler, as illustrated in Figure B.21. If the Message Processor window is obscured by some other window, select MAX+plusII | Message Processor to bring the Message Processor window to the foreground.

The warning message is produced because the OR-gate output is not connected to any other node in the schematic. The error message states that the *f* output symbol is

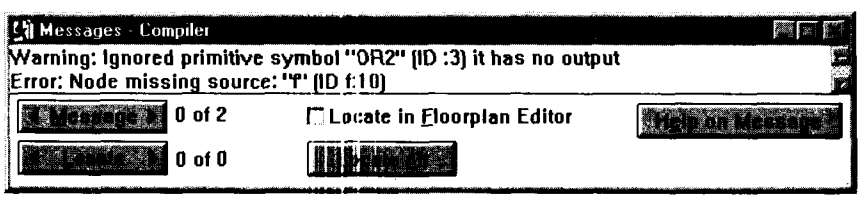


Figure B.21 The Message Processor display.

not connected to anything. Although it is clear how to fix the error, since we created it purposely, in general some of the messages displayed by the Compiler when synthesizing larger circuits may not be obvious. In such cases it is possible to select a message with the mouse and then click on the Help on Message button in the Message Processor window; documentation that explains the message is automatically opened. Experiment with this feature for both the warning and error messages in Figure B.21.

Another convenient feature of the Message Processor is the Locate button in the lower-left corner of the window. It can be used to automatically display the section of the schematic where the error exists. Select the warning message and then click the Locate button. Observe that the Graphic Editor is automatically displayed with the OR gate highlighted. Next select the error message in the Message Processor window and then click the Locate button again. The *f* output symbol becomes highlighted in the Graphic Editor.

Use the Graphic Editor to redraw the missing wire between the OR-gate output and the *f* output symbol. Save the schematic and then use the Compiler to run the synthesis tools to see that the error is fixed. We have now completed our introduction to design using schematic capture. If any application windows are still open, close them to return to the Manager window.

B.3 DESIGN ENTRY USING VERILOG

This section illustrates the process of using MAX+plusII to implement logic functions by writing Verilog code. We will implement the function *f* from section B.2, where we used schematic capture. After typing the Verilog code, it will be simulated with the Functional Simulator.

B.3.1 SPECIFYING THE PROJECT NAME

We need a new project name for the Verilog design. In the Manager window select File | Project | Name. We will store the design files for the project in the same directory that we used for the schematic capture design created earlier. In the box labeled Project Name, type *example1* as the name for the project and then click OK. The name of the project is displayed in the title bar of the Manager window.

B.3.2 USING THE TEXT EDITOR

MAX+plusII provides a text editor that can be used for typing Verilog code. Open the Text Editor window by selecting MAX+plusII | Text Editor. The first step is to specify a name for the file that will be created. Select File | Save As to open the pop-up box depicted in Figure B.22. Type *example1.v* in the box labeled File Name. You must use exactly this name. The name *example1* must match the name of the project, and the filename extension *v* must be used for all files that contain Verilog code. When File | Save As is selected, the Text Editor places the default name *example1.tdf* in the File Name box. The *tdf* extension stands for

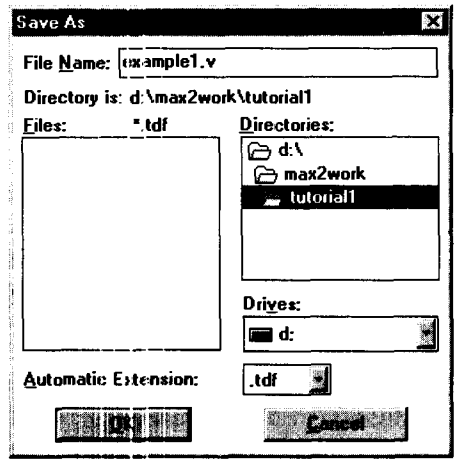


Figure B.22 Specifying a name for the Verilog design file.

text design file. It is used for files that contain source code written in the Altera Hardware Description Language (AHDL), which is another language supported by the MAX+plusII system. Make sure to change the filename extension from *tdf* to *v*. We should mention that it is not necessary to use the Text Editor provided in MAX+plusII. Any text editor can be used to create the file named *example1.v*, as long as the text editor can generate a plain text (ASCII) file.

The Verilog code for this example is shown in Figure 2.34. (Note the slight difference in the name of the example; in Figure 2.34 it is called *example3*, while here we are referring to it as *example1*.) Type the code into the Text Editor to obtain the display in Figure B.23. Most of the commands available in the Text Editor are self-explanatory. Text is entered at the *insertion point*, which is indicated by a thin vertical line. The insertion point can be moved either by using the keyboard arrow keys or by using the mouse. Two features of the Text Editor are especially convenient for typing Verilog code. First, the editor can optionally display different types of Verilog statements in different colors. To turn on this option, open the Options menu and place a check mark next to the item named Syntax Coloring. Second, the editor can automatically indent the text on a new line so that it matches the previous line. To turn on this option, place a check mark beside Options | Auto-indent. Save the file.

Using Verilog Templates

The syntax of Verilog code is sometimes difficult for a designer to remember. To help with this issue, the Text Editor provides a collection of *Verilog templates*. The templates provide examples of various types of Verilog statements, such as a **module** declaration, an **always** block, and assignment statements. It is worthwhile to browse through the templates by selecting Templates | Verilog Template to become familiar with this resource.

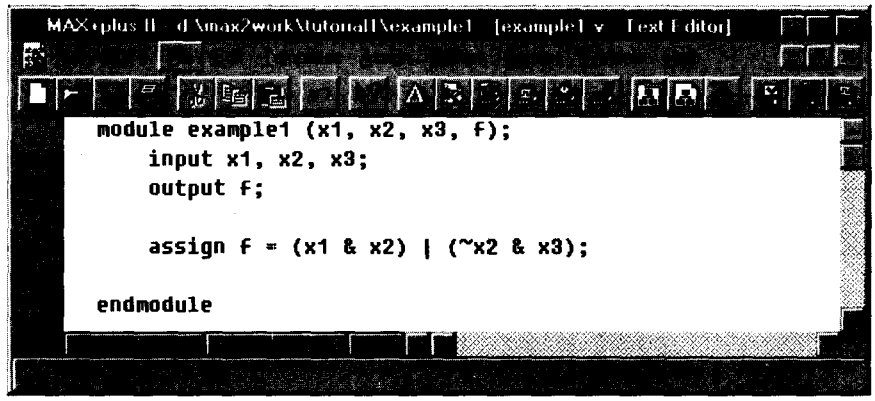


Figure B.23 The Text Editor display showing the Verilog code for *example1*.

B.3.3 SYNTHESIZING A CIRCUIT FROM THE VERILOG CODE

In section 2.9.2 we said that a Verilog compiler generates a logic circuit from Verilog code. The Verilog compiler provided by MAX+plusII is controlled by the Compiler application.

Using the Compiler

Open the Compiler window. As described for the design created with schematic capture earlier, select **Processing | Functional SNF Extractor** so that the Compiler will generate the information needed to perform functional simulation. Press the **Start** button in the Compiler window. If the Verilog code has been typed correctly, the Compiler will display a message that says that no errors or warnings were generated.

If the Compiler does not report zero warnings and zero errors, then at least one mistake was made when typing the Verilog code. In this case the **Message Processor** window is opened, and it displays a message corresponding to each warning or error found. An example showing how the Message Processor can be used to quickly locate and fix errors in Verilog code is given in section B.3.5. The Compiler window can now be closed.

B.3.4 PERFORMING FUNCTIONAL SIMULATION

Functional simulation of the Verilog code is done in exactly the same way as the simulation described earlier for the design created with schematic capture. Open the Waveform Editor and select **File | Save As** to save the file with the name *example1.scf*. Following the procedure given in section B.2.4, select **Node | Enter Nodes from SNF** and import the nodes in the project into the Waveform Editor. Draw the waveforms for inputs $x1$, $x2$, and $x3$ shown in Figure B.17. It is also possible to open the previously drawn waveform file *graphic1.scf* and then "copy and paste" the waveforms for $x1$, $x2$, and $x3$. The procedure for copying waveforms is described in **Help | MAX+plusII Table of Contents | Waveform Editor | Procedures | Copying, Cutting & Pasting Nodes and Groups**. Open the Simulator

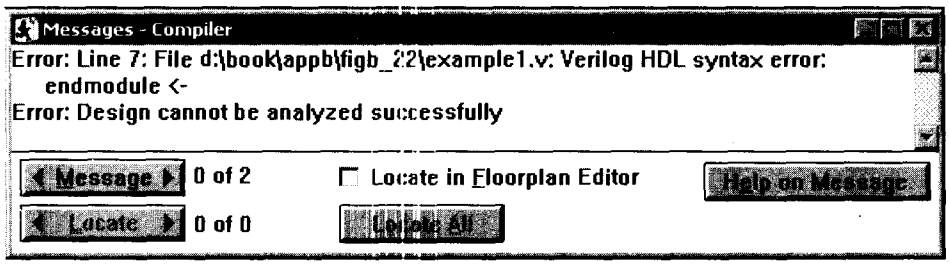


Figure B.24 The Message Processor window displaying an error in Verilog code.

and click on the Start button. The waveform generated by the Simulator for the output *f* should be the same as the waveform in Figure B.19.

B.3.5 USING THE MESSAGE PROCESSOR TO DEBUG VERILOG CODE

In section B.2.5 we showed that the Message Processor application can be used to quickly locate and fix errors in a schematic. A similar procedure is available for finding errors in Verilog code. To illustrate this, open the *example1.v* file with the Text Editor. In the fifth line, which is the **assign** statement, delete the semicolon at the end of the line. Save the *example1.v* file and then run the Compiler again. The Compiler generates one error, and the Message Processor window is opened, as illustrated in Figure B.24. The error message specifies that the problem was identified when processing line 7 in the Verilog source code file. Select the error message in the Message Processor window and then click the Locate button. The Text Editor window is automatically displayed with the insertion point at line 7.

Fix the error by reinserting the missing semicolon; then save the file and run the synthesis tools again, to confirm that the error is fixed. We have now completed the introduction to design using Verilog code. Close any open application windows to return to the Manager window.

B.4 DESIGN ENTRY USING TRUTH TABLES

This section describes the process of designing a logic circuit using a truth table. We will implement the truth table shown in Figure B.25. It will be entered into the CAD system by drawing a timing diagram with the Waveform Editor. We discuss the equivalence of truth tables and timing diagrams in section 2.4.1.

We need to specify a new project name for the truth table design. Using File | Project | Name, follow the procedure described in section B.3.1 to assign the name *timing1* to the project. Use the same directory as for the projects designed in the previous sections.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure B.25 A three-variable function.

B.4.1 USING THE WAVEFORM EDITOR

Open the Waveform Editor window by selecting MAX+plusII Waveform Editor. The Waveform Editor can be used for multiple purposes. In section B.2.4 the editor was used to create input files for simulation. In this section the Waveform Editor will be used to create a different type of file, called a *waveform design file*. To specify the type of file to be created, select File | Save As. In the box labeled File Name, type the name *timing1.wdf*. You must use exactly this name. The name *timing1* must match the name of the project, and the filename extension *wdf* indicates that the waveforms will be used to describe a logic function, instead of being used as simulation input.

B.4.2 CREATING THE TIMING DIAGRAM

To create a timing diagram, it is first necessary to specify the input and output signals for the circuit. Select Node | Insert Node to open the pop-up box shown in Figure B.26. In the box labeled Node Name in Figure B.26, type $x1$. Since $x1$ is an input to the circuit, make sure that Input Pin is selected in the box labeled I/O Type. Click OK. The input $x1$ appears in the Waveform Editor display. Use the same procedure to insert inputs $x2$ and $x3$ into the Waveform Editor display. Next, select Node | Insert Node again and type f in the Node Name box. Since f is the output for the circuit, make sure that Output Pin is selected in the box labeled I/O Type and then click OK. An alternative way to open the Insert Node pop-up box used above is to double-click in the Waveform Editor display in a blank space under the column labeled Name. The inserted node will be placed in the Waveform Editor window at the location where the mouse was double-clicked.

Having inserted the waveforms into the Waveform Editor, we will now draw a timing diagram to represent the truth table in Figure B.25. Since the truth table has eight rows, we will need to draw eight valuations of the inputs $x1$, $x2$, and $x3$. In section B.2.4 we set the size of the grid displayed in the Waveform Editor to 20 ns. If this same grid size is used, then the total time range needed in the Waveform Display is 160 ns. Select File | End Time and specify *160ns* as the total simulation time. To make the entire time range visible in

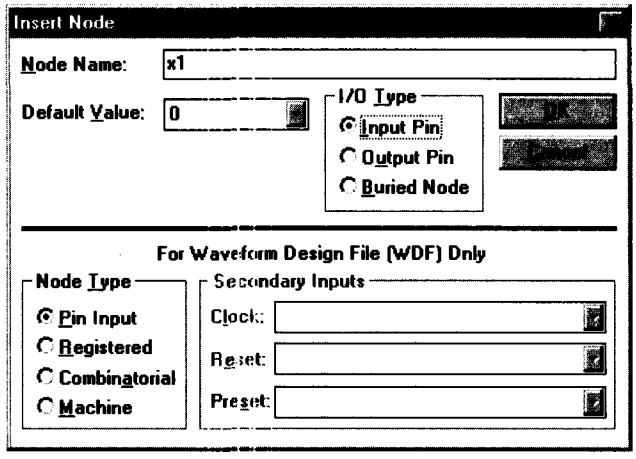


Figure B.26 Inserting a node into the Waveform Editor.

the waveform display, select View | Fit in Window or type the shortcut command Ctrl+w (while holding down the Ctrl key, press the w key). The Waveform Editor window should now appear as shown in Figure B.27.

Following the procedure described in section B.2.4, modify the waveform for signal $x3$ so that it is 1 for every second 20 ns time range. Also, edit the waveform for $x2$ so that it is 1 for the time ranges from 40 ns to 80 ns and from 120 ns to 160 ns. Finally, set the waveform for $x1$ to 1 in the time range from 80 ns to 160 ns. Previously, when using the Waveform Editor, we did not specify a waveform for the output of the circuit, because the output waveform was generated by the simulator. However, in this case we need to specify a waveform for output f that corresponds to its truth table. In Figure B.25 the function is 1 in the rows where $x1$, $x2$, and $x3$ have the valuations 001, 011, 101, 110, and 111. Use the Graphic Editor to change the waveform for f to 1 for the appropriate time ranges. For instance, f should be set to 1 in the time range from 20 ns to 40 ns because this represents the input valuation 001. After completing the waveform for f, the waveform display should appear as shown in Figure B.28. Notice that we have rearranged the waveforms, by moving f to the bottom, in comparison to Figure B.27. Waveforms can be moved by pointing the mouse at the small symbol, called the *node handle*, to the left of the signal name in the waveform display and then dragging the waveform upward or downward. Select File | Save to save the timing diagram in the *timing1.wdf* file.

B.4.3 SYNTHESIZING A CIRCUIT FROM THE WAVEFORMS

The next step is to use the MAX+plusII Compiler to perform the initial synthesis steps for the circuit. The Compiler will generate a Boolean expression to represent f, according to the truth table given by the timing diagram.

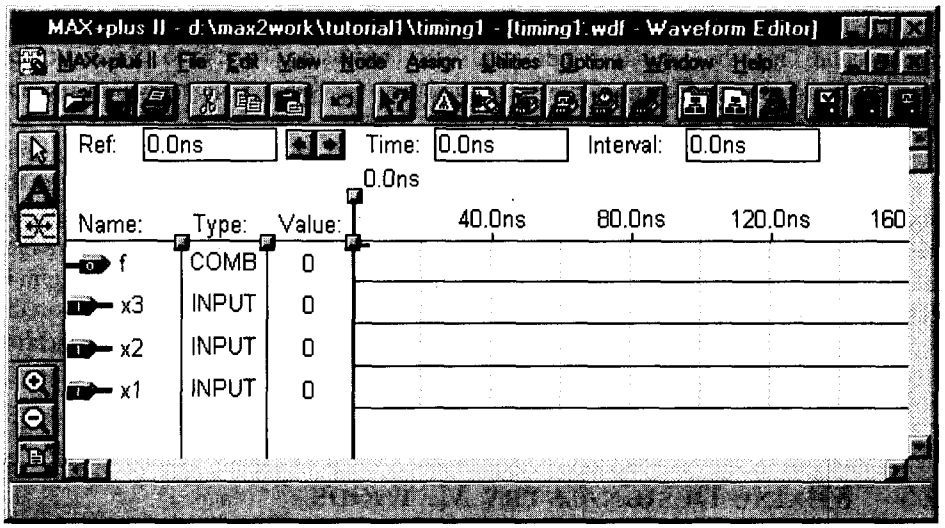


Figure B.27 The Waveform Editor display for the truth-table design.

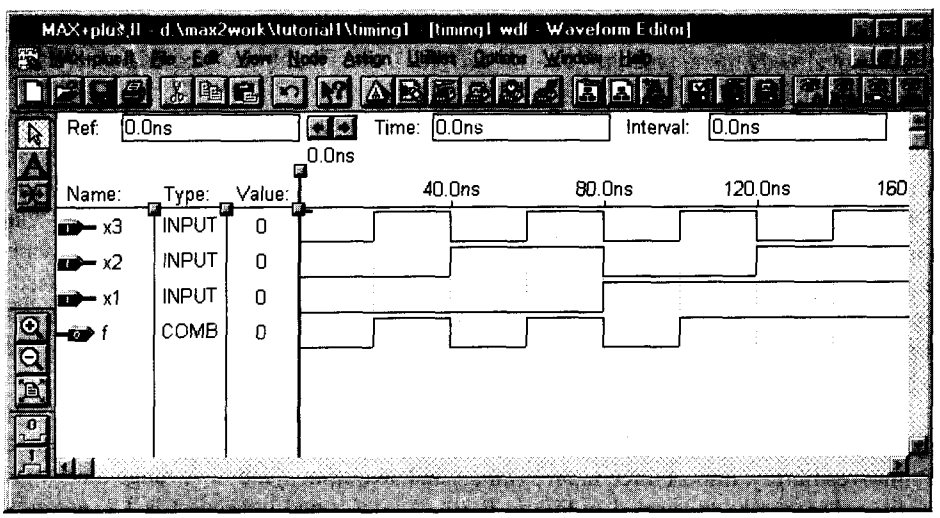


Figure B.28 The timing diagram representing the truth table in Figure B.25.

Use the same procedure described for the designs created with schematic capture and Verilog code. Open the Compiler window and select Processing | Functional SNF Extractor. Press the Start button in the Compiler window and then click OK in response to the Compiler message that says that no warnings or errors were found.

For the circuits designed in the previous sections, after logic synthesis was completed, the next step performed was functional simulation. It does not make sense to perform the functional simulation for the circuit designed in this section, because the waveforms that would be used as inputs to the simulator would be the same waveforms used to design the circuit! In the next section we will use the circuit synthesized from the timing diagram in this section as part of a larger circuit, and we will simulate the operation of the larger circuit.

The tutorial on design with truth tables is now complete, so close any open application windows to return to the Manager window.

B.5 MIXING DESIGN-ENTRY METHODS

It is possible to design a logic circuit using a mixture of design-entry methods. As an example, in this section we will create a schematic that includes the circuit designed using the truth table in the previous section.

We need to specify a new project name for the mixed design. Select File | Project | Name and assign the name *mixed1* to the project. Use the same directory as for the projects designed in the previous sections.

B.5.1 CREATING A SCHEMATIC THAT INCLUDES A TRUTH TABLE

Open the Graphic Editor by selecting MAX+plusII | Graphic Editor. Select File | Save As and, if not already there, type the name *mixed1.gdf* in the File Name box. Make sure to use exactly this name.

Double-click the blank space in the Graphic Editor to open the Enter Symbol pop-up box, as shown in Figure B.29. In the box labeled Symbol Name, type the name *timing1*, which is the name of the circuit designed using a truth table in the previous section. Click OK to import a graphical symbol for the *timing1* circuit into the Graphic Editor. Once the *timing1* symbol is imported into the Graphic Editor, double-clicking on the symbol automatically opens the Waveform Editor and displays the waveforms that were used to design the circuit. When the Waveform Editor is closed, the Graphic Editor is automatically reopened. This ability to move quickly from one design-entry tool to another is convenient when it is necessary to make changes to a schematic or the subcircuits in it.

Following the procedure described in section B.2.2, import a two-input AND-gate symbol and a NOT gate from the Primitives library into the Graphic Editor. Also from the Primitives library, import three input symbols and an output symbol. Arrange the symbols in the schematic as illustrated in Figure B.30. As described in section B.2.2, assign the names $x1$, $x2$, and $x3$ to the input symbols and assign the name f to the output symbol. The reader will observe that the name $x3$ is used twice in this design project: as an input to

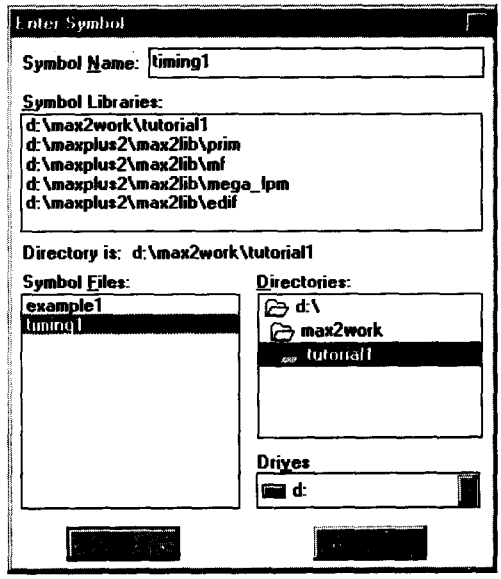


Figure B.29 Importing the truth-table design into the Graphic Editor.

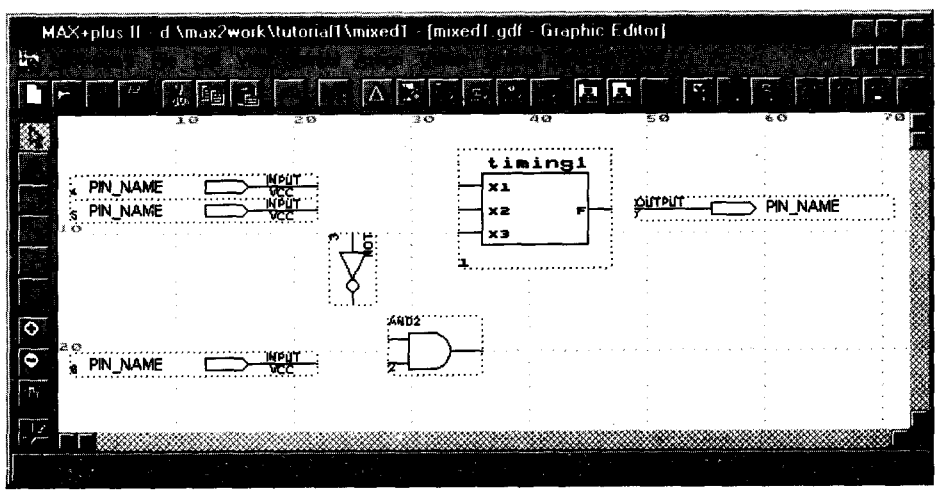


Figure B.30 A schematic including a truth table and logic gates.

the *timing1* subcircuit and as an input to the mixed schematic. The MAX+plusII compiler treats these two nodes named $x3$ as separate nodes because they appear in different levels of the design project hierarchy. Connect the symbols in the schematic together as shown in Figure B.31. Because a wire drawn with the Graphic Editor can be either straight or have a single bend, it is necessary to draw more than one wire for the connection shown in the figure from the AND-gate output to the input labeled $x3$ on the *timing1* subcircuit. Start drawing each wire so that it touches the end of the previously drawn wire; wires that touch are automatically connected by the Graphic Editor. Save the schematic.

B.5.2 SYNTHESIZING AND SIMULATING A CIRCUIT FROM THE SCHEMATIC

Use the procedure described for the designs created in the previous sections to synthesize a circuit from the schematic. The synthesis tools will create a single logic circuit by merging the *timing1* subcircuit with the other logic gates in the schematic. Open the Compiler window, select **Processing | Functional SNF Extractor**, and then run the Compiler.

Simulation of the *mixed1* project is done in exactly the same way as for the other projects created in this tutorial. Open the Waveform Editor and select **File | Save As** to create a new file named *mixed1.scf*. Following the procedure given in section B.2.2, import the input and output nodes $x1$, $x2$, $x3$, and f into the Waveform Editor. Draw the waveforms for inputs $x1$, $x2$, and $x3$ that are shown in Figure B.17. Open the Simulator and click on the **Start** button; then select **Open SCF** to see the results of the simulation. The waveform generated by the Simulator for the output f should be exactly the same as the waveform shown in Figure B.19. The *mixed1* schematic represents the logic function $f = x_1x_2 + \bar{x}_2x_3$ that was designed using both schematic capture and Verilog code in this tutorial. Techniques

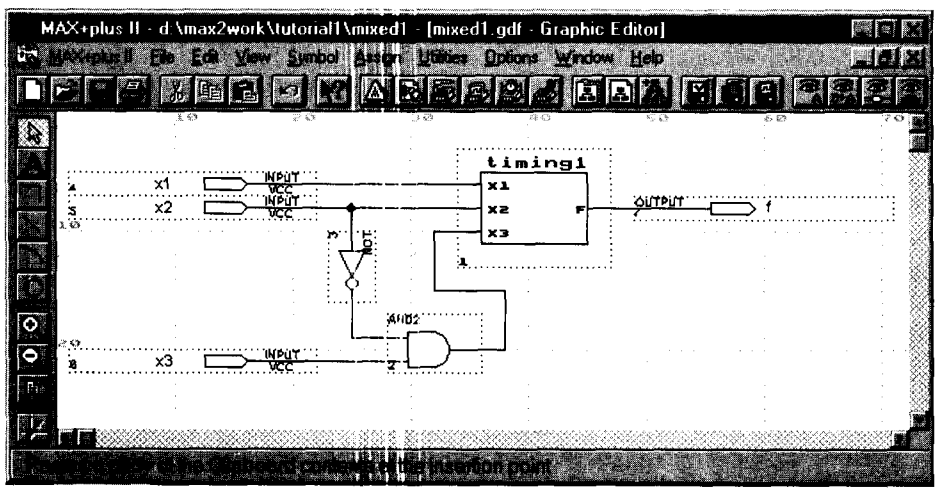


Figure B.31 The completed schematic corresponding to Figure B.30.

that can be used to synthesize the expression for f from the *mixed1* schematic are covered in Chapter 4.

In practice a designer would not use a mixture of design-entry methods for a circuit as simple as our example. The reason that we have created the *mixed1* schematic is simply to illustrate that MAX+plusII allows design-entry methods to be combined in a hierarchical manner. It is also possible, although not shown here, to create a schematic that includes a subcircuit designed using Verilog code. MAX+plusII provides a convenient feature, called the Hierarchy Display, for working with hierarchical design projects.

B.5.3 USING THE HIERARCHY DISPLAY

Select MAX+PlusII | Hierarchy Display to open the Hierarchy Display window shown in Figure B.32. The display shows that the design project consists of two hierarchical levels, with *mixed1* at the higher level and *timing1* at the lower level. The *mixed1* design project has an icon next to it, labeled *gdf*. It can be double-clicked to automatically open the *mixed1.gdf* file in the Graphic Editor. Similarly, *timing1* has an icon next to it, labeled *wdf*. If this icon is double-clicked, the file *timing1.wdf* is opened in the Waveform Editor. Experiment with this method of opening design files. Figure B.32 also shows a small icon labeled *acf*, which represents the *assignment & configuration file* for the project. The file contains settings for a large number of optional features of MAX+plusII that affect the way the design files are processed. These settings are saved automatically in the assignment & configuration file, and so we will not need to modify them manually. Although it is not necessary, the *acf* file can be opened in the Text Editor by double-clicking on its icon in the Hierarchy Display.

B.5.4 CONCLUDING REMARKS

This tutorial has introduced the basic use of the MAX+plusII CAD system. We have shown how to perform design entry by drawing a schematic, writing Verilog code, and drawing a timing diagram that represents a truth table. Each design was processed by the initial synthesis tools and then simulated with the functional simulator.

In the next tutorial we will show how the logic synthesis and physical design tools are used to implement circuits in PLDs. The timing characteristics of the implemented circuits will be examined using timing simulation.

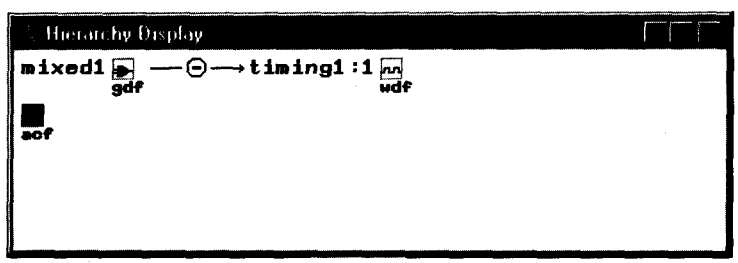
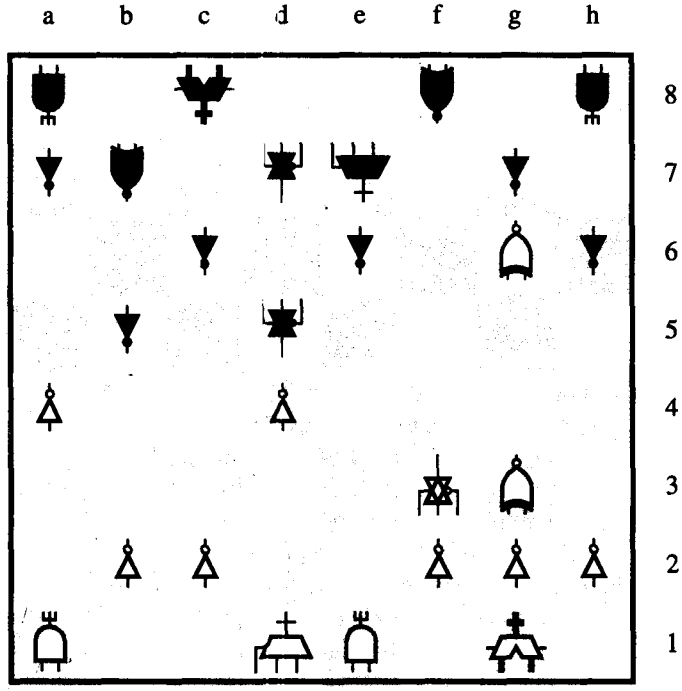


Figure B.32 The Hierarchy Display window for the *mixed1* design project.

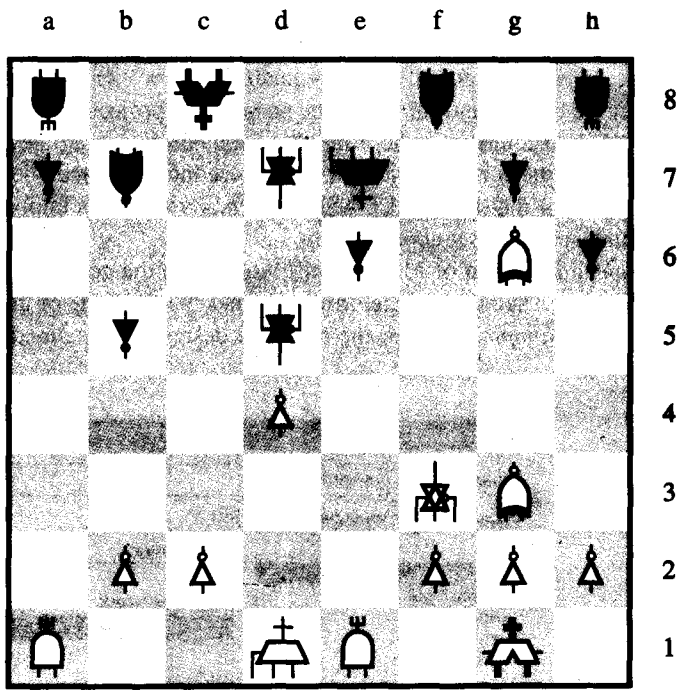


14 Bf4–g3, Kd8–c8

appendix

C

TUTORIAL 2



15. a4xb5, c6xb5

In this tutorial we describe how to use the logic synthesis and physical design tools in MAX+plusII. In addition to the modules used in Tutorial 1, the following modules are introduced: Logic Synthesizer, Fitter, Floorplan Editor, and Assembler. To illustrate the procedures involved, we will first implement the *example1* project created in Tutorial 1 in a MAX 7000 CPLD.

C.1 IMPLEMENTING A CIRCUIT IN A MAX 7000 CPLD

Select File | Project Name, which opens the window shown in Figure C.1. The box labeled Directories gives the contents of the d:*max2work**tutorial1* directory that was used for Tutorial 1. To select the *example1* project, in the box labeled Files double-click on *example1.v*. We will not modify *example1.v*, but the reader should review its contents. A convenient way to open the file is to select File | Hierarchy Project Top or type the shortcut for this command, which is Ctrl+t. Of course, the *example1.v* file can also be opened using File | Open.

As we showed in Tutorial 1, MAX+plusII always works in the context of one project at a time. For the current tutorial we set the name of the project before opening its design file. Alternatively, we could open the *example1.v* file before setting the project name. Then to set *example1* as the current project, we could use the command File | Project | Set Project to Current File or the shortcut Ctrl+Shift+j. In any case it is important to ensure that the project name is always set appropriately.

Another implication of MAX+plusII working in the context of one project at a time has to be noted. The name of the project defines the filename of the top-level design file. Hence for the *example1* project, the top-level design file is named *example1.v*. Because

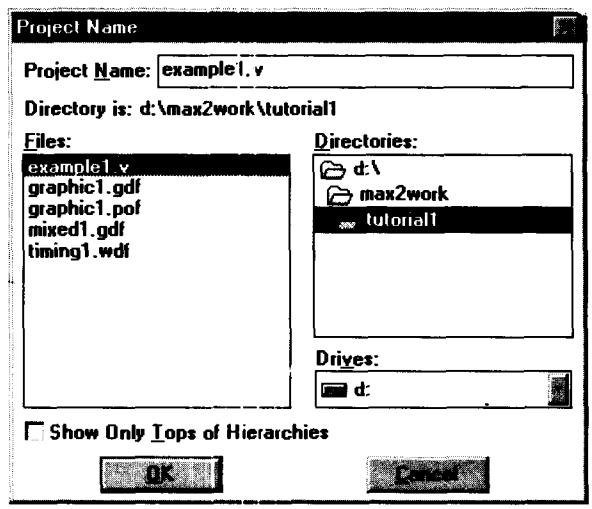


Figure C.1 Changing the project name to *example1*.

of the relationship between project names and filenames, the user should never create two separate projects with the same base filename and different filename extensions. For example, assume that two projects are created in the same directory: one project uses a schematic diagram for its top-level design file, and one uses Verilog code. If the user chooses the same base filename for both projects, such as *circuit.gdf* for the schematic and *circuit.v* for the Verilog code, then two different top-level design files will have the same project name. Such duplication creates confusion when trying to synthesize a circuit for each design file. To avoid this problem, always follow the methodology used in Tutorial 1 in which different base filenames are used for all projects, such as *graphic1.gdf* and *example1.v*.

C.1.1 USING THE COMPILER

To continue our discussion of how to implement *example1.v* in a MAX 7000 CPLD, open the Compiler window. In Tutorial 1 we used the Compiler to perform the initial synthesis that generated the information needed for functional simulation. In this tutorial we will implement the design in a CPLD and then use timing simulation. To specify that the Compiler should not generate the information for functional simulation, select the Processing menu. A check mark should appear beside Functional SNF Extractor because we turned *on* this setting in Tutorial 1. The operation of the check mark is that the feature is *toggled* on or off each time it is selected. Click on Functional SNF Extractor to turn it off. The Compiler window should now display several software modules that are invoked in sequence, as indicated in Figure C.2. (If the Timing SNF Extractor module does not appear in the Compiler window, select Processing | Timing SNF Extractor to activate it.) As we said in Tutorial 1, the first two modules perform the initial synthesis. The Logic Synthesizer performs logic synthesis and optimization. The Fitter performs the physical mapping, and the Timing SNF Extractor generates the information that indicates the timing delays in the circuit when implemented in the target chip. This information is stored in a file called the *simulator netlist file (SNF)*, which is used by the Timing Simulator. The last

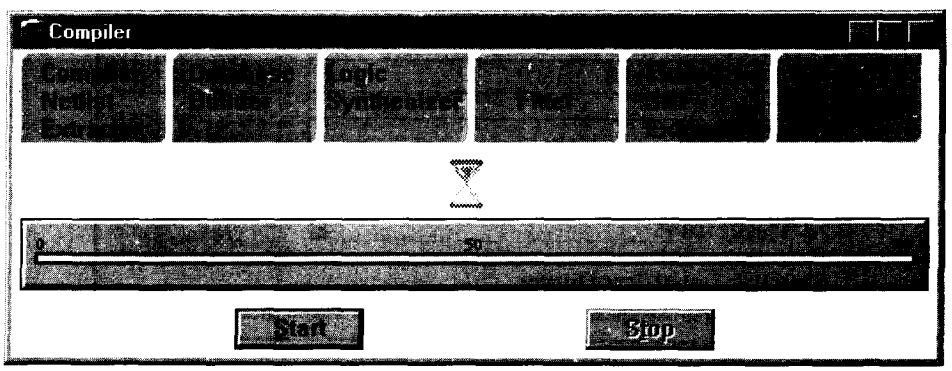


Figure C.2 The Compiler display.

module executed, the Assembler, produces the information that specifies how to configure the selected chip to implement the circuit.

C.1.2 SELECTING A CHIP

Before starting the Compiler, it is necessary to specify which chip to use. Select Assign | Device to open the window shown in Figure C.3. To select the MAX 7000 CPLD family, click on the pull-down menu in the box labeled Device Family and select MAX7000S. The S at the end of the name refers to the members of the MAX 7000 family that are in-system programmable. Methods of CPLD programming are discussed in section 3.6.4. The available chips in the MAX 7000S family are displayed in the box labeled Devices. One available chip is the EPM7128SLC84-7. If this chip is not listed in the Device dialog box, click on the option at the bottom of Figure C.3 that specifies Show Only Fastest Speed Grades. Turning off this option by clicking on it results in all speed grades of the chips available to the designer being shown in the Devices box (the copy of MAX+plusII included with the book allows the user to select only a limited subset of the chips available from Altera). The meaning of the chip name is as follows: The EPM7 means that the chip is a member of the MAX 7000 family, and the 128 gives the number of macrocells in the chip. The designator LC84 indicates an 84-pin PLCC package; this type of package is described in section 3.6.3. The –7 gives the *speed grade*. We discuss speed grades in Appendix E.

In the Devices box it is possible to select the word AUTO instead of a specific chip. Choosing this option means that the Fitter module will automatically select an appropriate chip from the list of chips shown in the Devices box. The ability to have a chip chosen automatically, based on the complexity of the circuit that has to be implemented, is sometimes convenient for the designer. In this case we wish to select a specific chip, so click on EPM7128SLC84-7 as indicated in the figure. Click OK to return to the Compiler window.

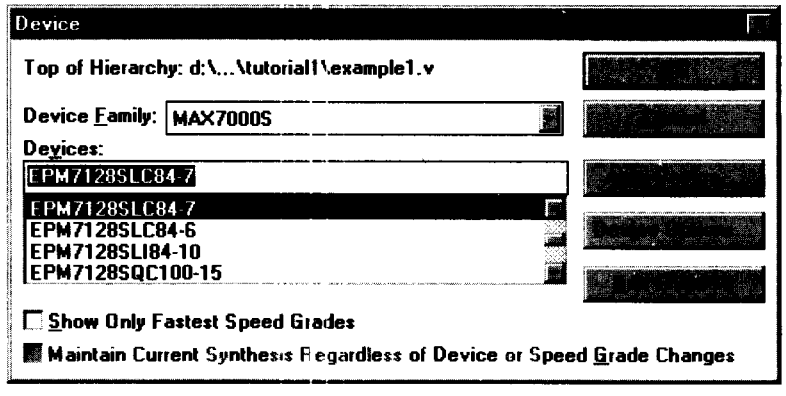


Figure C.3 Selecting a device for implementation.

C.1.3 VIEWING THE LOGIC SYNTHESIS OPTIONS

MAX+plusII provides a number of options that affect the results produced by logic synthesis. These options can be accessed by selecting Assign | Global Project Logic Synthesis to open the window that appears in Figure C.4. To see a description of the purpose of each area in the displayed window, use the MAX+plusII context-sensitive help. Recall that this feature is invoked by pressing the F1 key. For the *example1* project we do not need to change any of the logic synthesis options. Click Cancel to return to the Compiler window.

In our previous examples we selected the Compiler's Start button to run the synthesis tools. Another way to start the Compiler is to select File | Project | Save and Compile. A shortcut for this sequence is to type Ctrl+l. This sequence causes MAX+plusII to save open design files and then run the synthesis tools in the Compiler module. There is also an icon for this command along the top of the Compiler window. It looks like a factory smoke stack on top of a floppy disk. Starting the Compiler in this way has an important benefit. It checks whether the name of the design file that the user is currently working with matches the name of the current project. If not, the system provides a warning message. This avoids the possibly confusing situation that arises if the user forgets to change the project name before working on a design file. Use the Ctrl+l shortcut to start the Compiler. After all the modules have been executed, click OK in response to the prompt from the Compiler.

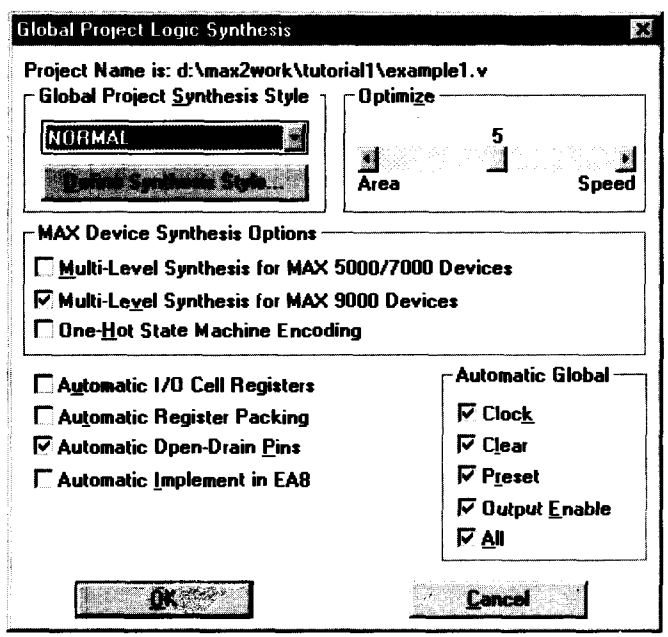


Figure C.4 Logic synthesis options.

C.1.4 EXAMINING THE IMPLEMENTED CIRCUIT

The Compiler produces a report file that documents the results of the implementation. To open the report file in the Text Editor, double-click on the icon labeled *rpt* displayed under the Fitter module in the Compiler window. Alternatively, the report file can be opened using File | Open and specifying *example1.rpt*. A part of the report file is illustrated in Figure C.5. It indicates that the project is successfully implemented in the selected device, and lists some of the chip's resources used for the circuit. Our small example uses three input pins, one output pin, and one macrocell (LC). The percentage of the chip consumed by the circuit is also shown. Since the chip has 128 macrocells, less than one percent is used for this example.

Use the scroll bars on the right side and bottom of the Text Editor to examine more of the report file. Observe that a diagram is included in the file that shows which pins on the chip package the Fitter has chosen for the input and output signals in the circuit. The file specifies that inputs $x1$, $x2$, and $x3$ are assigned to the pins numbered 12, 11, and 10, respectively. The output f is assigned to pin 73. Rather than having the pins selected automatically by the Fitter module, it is also possible to choose the pins manually. The manual selection procedure is described in section C.4.

Scroll through the report file to find the section labeled Equations, which shows the logic equation generated by the Logic Synthesis module for each output in the circuit. (If this section is not included in the report file, then in the Compiler window select Options | Report File, click on Equations to select it, and then run the Compiler again.) Figure C.6 shows the equation generated for f in the *example1* project. The equation has an unusual syntax. It states that f is the output of a logic cell, or *LCELL*, whose inputs are the signals *_EQ*001 and *Gnd*. The LCELL represents a macrocell in the MAX 7000 chip. The $ symbol represents XOR. Hence the LCELL produces the XOR of the signal *_EQ*001 with *Gnd*. This XOR is included in the circuit because the macrocell in the MAX 7000 device includes an XOR gate in the output path, as indicated in Appendix E. Since one input to this gate is 0 (*Gnd*), then the output of the macrocell is defined by *_EQ*001. The expression shown for *_EQ*001 corresponds to the Boolean equation that we specified for f in *example1.v*. Note that in the report file, the symbol & indicates AND, # is OR, and ! is NOT. Close the report file.

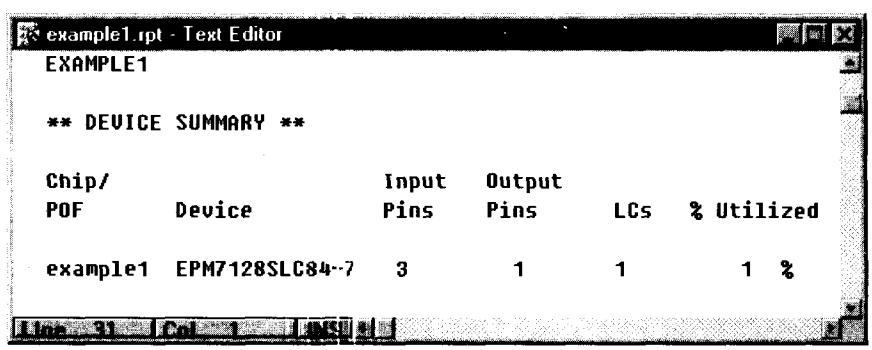


Figure C.5 Compiler messages generated for the *example1* project.

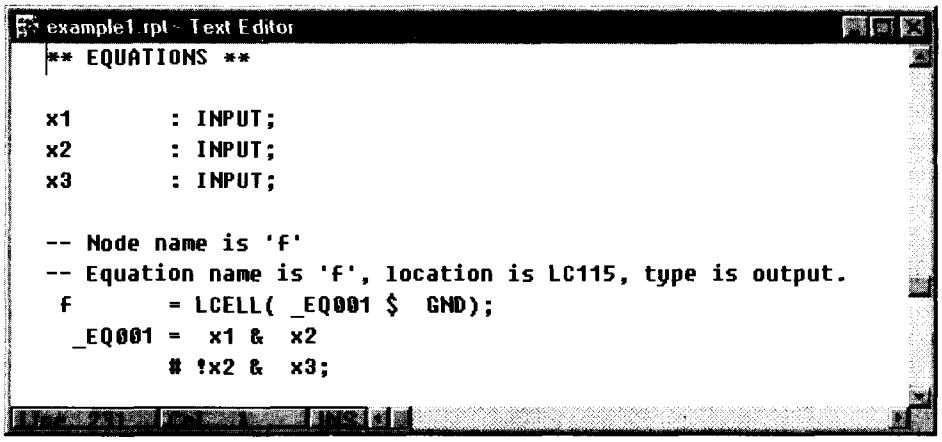


Figure C.6 The Equations section of the report file for *example1*.

C.1.5 RUNNING THE TIMING SIMULATOR

The Timing Simulator is invoked using the same procedure that we described in Tutorial 1 for the Functional Simulator. Select MAX+plusII | Simulator. The input waveforms for $x1$, $x2$, and $x3$ that were drawn in Tutorial 1 using the Waveform Editor can be used as inputs for the Timing Simulator. Click the Start button in the Simulator window to run the timing simulation. When the simulation is completed, click Open SCF. Select View | Fit in Window to display the waveforms shown in Figure C.7. Compare these waveforms to those shown in Figure B.19. The timing simulation produces the same results as the functional simulation in Tutorial 1 except that the times at which changes in f occur are now determined by the timing characteristics of the EPM7128SLC84-7 chip.

Align the mouse pointer over the 20 ns grid line and click on the waveform for f. This is the time when the input $x3$ changes to 1, which causes f to become 1. The Waveform

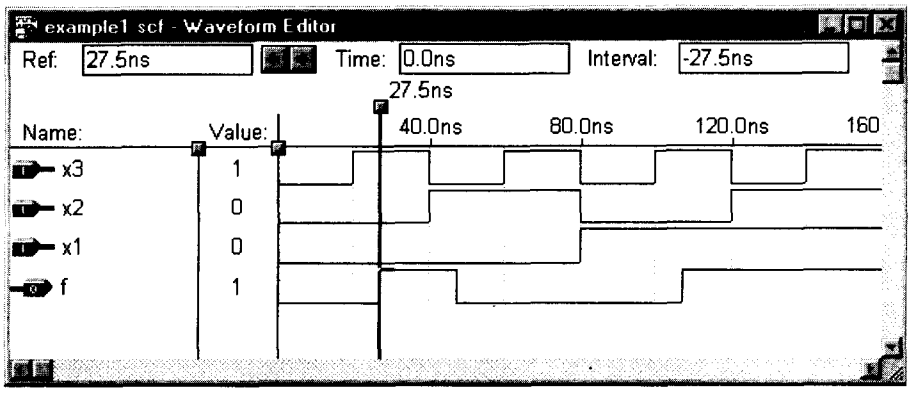


Figure C.7 Timing simulation results for *example1*.

Editor places the vertical reference line at 20 ns, and 20 ns is displayed in the box labeled Ref. Next click on the right arrow icon in the Waveform Editor. (This icon is situated to the right of the box labeled Ref.) The reference line moves to the time when f makes a transition from 0 to 1. The Ref box now displays 27.5 ns, meaning that it takes 7.5 ns for the change in $x3$ to cause a change in f. This result is a reflection of the -7 speed grade of the chip, which is specified as having a delay from an input to an output pin of 7.5 ns.

C.1.6 USING THE FLOORPLAN EDITOR

In section C.1.4 we showed that the implementation results produced by the Compiler can be examined by opening the report file. Another way to view the implementation results is to use the Floorplan Editor. Select MAX+plusII | Floorplan Editor to open the window shown in Figure C.8. To make the window look like the one in the figure, it may be necessary to change some of the settings in the Layout menu. Click on Layout and make the settings indicated in Figure C.9. Figure C.8 shows some of the macrocells in the EPM7128SLC84-7 chip. As we describe in Appendix E, the macrocells are organized into logic array blocks (LABs), where each LAB contains 16 macrocells. To see larger or smaller views of the LABs, click on the magnify buttons on the left edge of the window. To display different sections of the chip, use the window scroll bars.

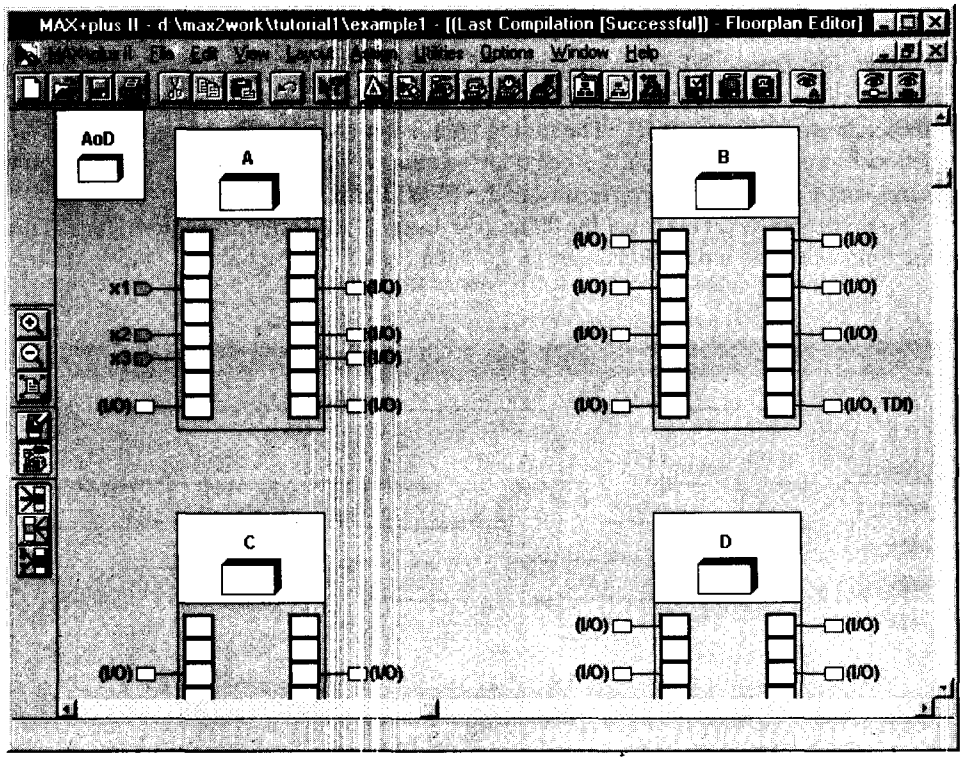


Figure C.8 The Floorplan Editor display.

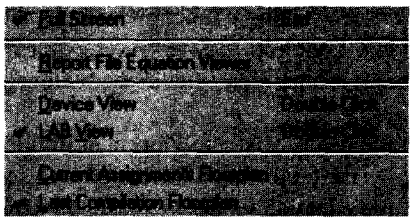


Figure C.9 The Layout menu in the Floorplan Editor.

The Floorplan Editor uses different colors to indicate macrocells that are used in a circuit and macrocells that are unused. For our small example three macrocells are used for the three inputs to the circuit, and one macrocell provides the circuit output. Orient the display so that the macrocell that produces the output f is visible, as depicted in Figure C.10. Click on this macrocell to select it. The Floorplan Editor can draw lines that indicate which other macrocells the selected macrocell is connected to by selecting **Options | Show Node**

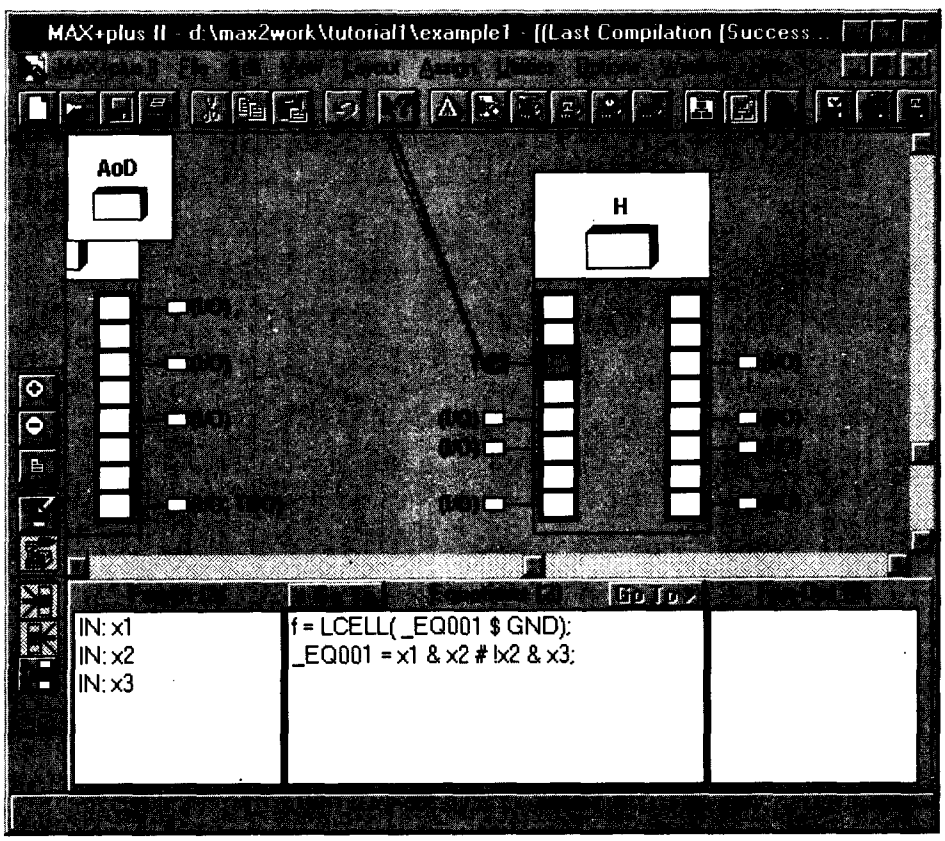


Figure C.10 Viewing the implementation of a function in the Floorplan Editor.

Fan-In. It is also possible to see what logic function is implemented in the selected node by selecting Layout | Report File Equation Viewer. As illustrated in the figure, this displays the logic equation from the Compiler report file in the bottom part of the Floorplan Editor window.

Instead of displaying the macrocells, the Floorplan Editor can alternatively display a picture of the package pins on the target chip. To change to this view, select Layout | Device View. To close the report file equation viewer, select again Layout | Report File Equation Viewer to toggle off this feature. To increase the available window area for viewing the device pin-out, select Layout | Full Screen. It is possible to make visible the entire device by clicking on the "negative" magnify button a few times. The "positive" magnify button and scroll bars can be used to change the view until the signal names assigned to the pins are large enough to read. Figure C.11 gives an example of the display oriented so that the pins used for inputs $x1$, $x2$, and $x3$ are visible. When the window is oriented in this way, scrolling to the right displays the pin assigned to the output f.

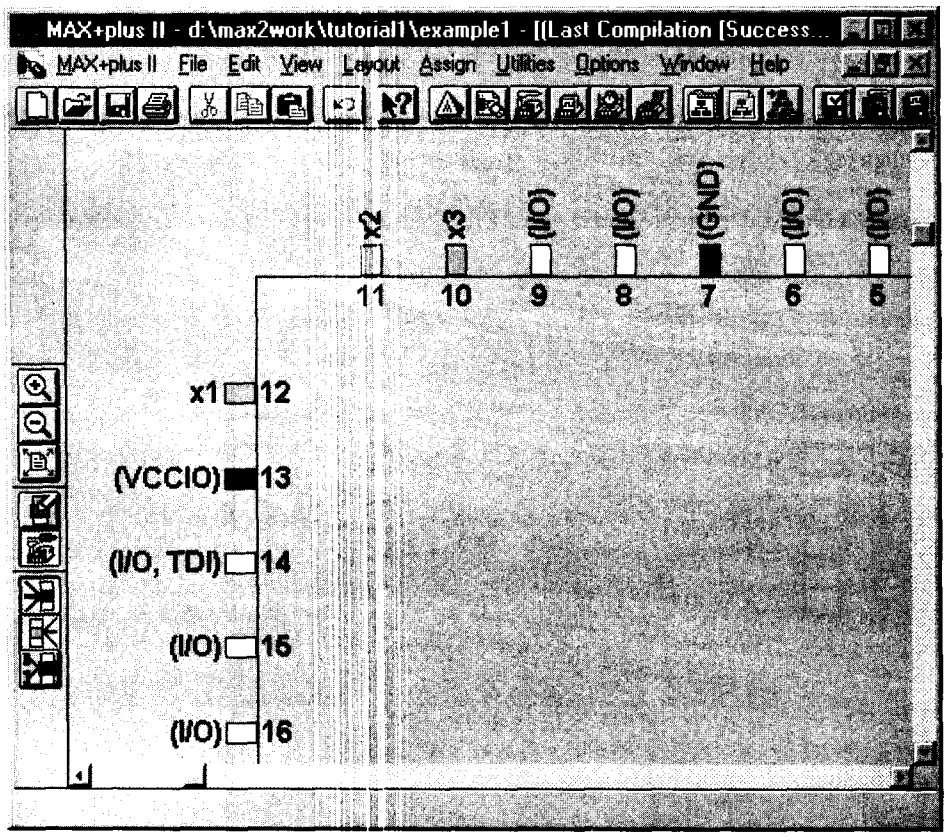


Figure C.11 Viewing the device pin-out in the Floorplan Editor.

The Floorplan Editor is not essential in the CAD flow described above. It just provides a graphical view of the information contained in the Compiler report file. We will describe a different use of the Floorplan Editor in section C.4.1, in which it will be used to modify the implementation results produced by the Compiler, instead of just displaying them.

We have now completed the implementation of the *example1* project in a MAX 7000 chip. Close all open windows to return to the Manager window.

C.2 IMPLEMENTING A CIRCUIT IN A FLEX 10K FPGA

The CAD flow used to implement a circuit in a FLEX 10K FPGA is the same as that used for the MAX 7000 CPLD. We showed in Chapter 4 that multilevel logic synthesis is an effective optimization strategy when targeting designs to lookup table–based FPGAs. Figure 4.54 gives Verilog code for a seven-variable logic function used to illustrate the benefits of multilevel synthesis. In this section we will create a new design project, named *example2*, which represents the Verilog code in that figure.

Select File | New to open the dialog box in Figure C.12. Click the circle beside the entry *Text Editor file* to select it and then click OK. The Text Editor module is automatically opened, editing an empty file. Type the Verilog code in Figure 4.54 into the Text Editor and then use File | Save As to save the file with the name *example2.v*. To make this file the current project, use the shortcut command Ctrl+Shift+j. The Verilog code is displayed in Figure C.13.

Open the Compiler window. Select Assign | Device to open the Device dialog box. In the drop-down menu labeled Device Family, select FLEX10K and then click on OK. Run the Compiler. If there are no errors in the Verilog code, the Message Processor reports which member of the FLEX 10K family the Fitter has chosen for implementation of the project. To see the results produced by the Logic Synthesis and Fitter modules, open the Compiler report file by double-clicking on the icon labeled rpt, which is displayed below the box that represents the Fitter module. The report file gives the assignment of signals to pins on the FLEX 10K device and reports the percentage of the chip's various resources that are used

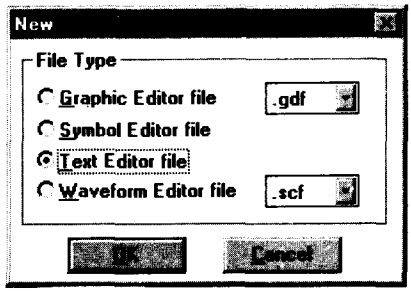


Figure C.12 Creating a new Text Editor file.

```
module example2 (x1, x2, x3, x4, x5, x6, x7, f);
  input x1, x2, x3, x4, x5, x6, x7;
  output f;

  assign f = (x1 & x3 & ~x6) | (x1 & x4 & x5 & ~x6) |
    (x2 & x3 & x7) | (x2 & x4 & x5 & x7);

endmodule
```

Figure C.13 The Verilog code for *example2*.

for our sample circuit. Scroll down in the report file until the logic equations depicted in Figure C.14 are visible.

The output of the circuit, f, is implemented as the output of the logic cell (lookup table) designated as $LC1_A1$. This logic cell is in the top row (row 1) and left-most column (column 1) of the FPGA. The reader may wish to use the Floorplan Editor to see where the logic cell is located in the chip, but we will not do so here. The bottom equation in Figure C.14 specifies that the logic cell $LC1_A1$ implements the function given by

$$LC1_A1 = LC2_A1 \cdot x_3 + LC2_A1 \cdot x_4 x_5$$

where $LC2_A1$, which is the logic cell in column 1 of row 2, implements the function

$$LC2_A1 = x_2 x_7 + x_1 \bar{x}_6$$

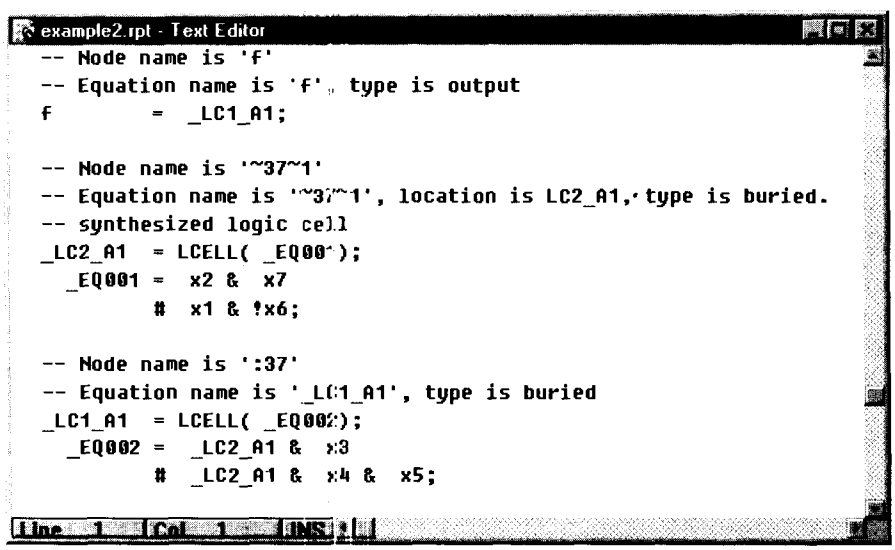


Figure C.14 The logic equations produced for the code in Figure C.13.

This is the same expression that we showed in Example 4.23, in section 4.12, except that earlier we used the variable name *S* instead of the name *LC2_A1*.

Having implemented the design in the FLEX 10K device, the next step is to perform a timing simulation. Since the steps involved are the same as in our previous examples, we will not show them here. The reader should perform the timing simulation to gain a feeling for the timing characteristics of the FLEX 10K device.

C.3 DOWNLOADING A CIRCUIT INTO A DEVICE

Once a circuit has been synthesized for a design project, the circuit can be downloaded into the selected device. *Downloading* involves programming the appropriate switches in the device to implement the desired circuit. To illustrate the steps involved, we will describe how a circuit can be downloaded into the laboratory development board that is available from the Altera Corporation. The board is called the UP–1 Education Board and includes both a MAX 7000 CPLD and a FLEX 10K FPGA. The UP–1 board can be obtained by following the instructions in the University Program section of Altera's Web site at http://www.altera.com.

We will describe how the *example1* project that we implemented in a MAX 7000 CPLD can be downloaded into the UP–1 board, assuming that it is connected to the reader's computer. A reader who does not have access to the UP–1 board will not be able to download the circuit, but the steps involved are still easy to follow. The UP–1 board is connected to the computer using one of two types of cables that are available from Altera. The ByteBlaster cable provides a connection to a parallel port on the computer, and the BitBlaster cable provides a connection to a serial port. In the following instructions we assume that the ByteBlaster cable is used.

The UP–1 board contains an EPM7128SLC84-7 chip. There is a socket that connects this chip to the ByteBlaster cable. Plug the ByteBlaster cable into this socket and plug the other end of the cable into the parallel port on the computer. Ensure that the UP–1 board is plugged into a power supply and that the green "power LED" is lit.

Set the name of the current project in MAX+plusII to be *example1*. Select MAX+plusII | Programmer to open the Programmer module window shown in Figure C.15. Observe that the programming file for the *example1* project, which is called *example1.pof*, is automatically selected by the Programmer. To specify that the ByteBlaster is to be used as the programming hardware, select Options | Hardware Setup. In the window shown in Figure C.16, use the Hardware Type drop-down menu to select ByteBlaster. Also, use the drop-down menu labeled Parallel Port to select the appropriate parallel port (usually either LPT1 or LPT2).

To program the EPM7128SLC84-7 chip, click Program in the Programmer window. The Programmer module automatically downloads the *example1.pof* file through the ByteBlaster cable into the device and then verifies that the programming has been performed correctly. The Programmer module can now be closed. The designer can test the circuit implemented in the chip by using appropriate test equipment.

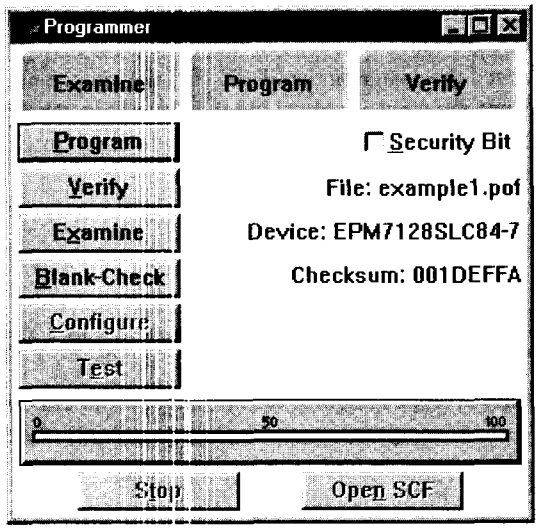


Figure C.15 The Programmer display.

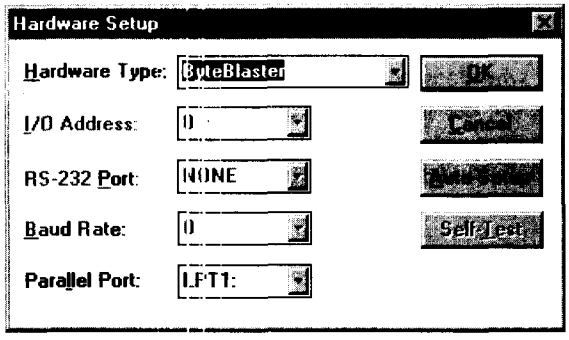


Figure C.16 Selecting the programming hardware.

The UP–1 board also contains a FLEX 10K chip. The procedure used to download a circuit into this chip is similar to the one described for the MAX 7000 device, but a few extra steps are needed. The reader who tries using the FLEX 10K chip should refer to the documentation that accompanies the UP–1 board for detailed instructions.

C.4 MAKING PIN ASSIGNMENTS

In the examples given in this tutorial, the assignment of signals to device pins is done automatically. In some cases the designer needs to be able to manually specify which pins to use for some of the signals in the circuit. For example, the circuit board that contains

the chip(s) being used may have hardwired connections from some of the device pins to other components, such as switches or LEDs. To make use of the hardwired connections, the designer has to be able to specify which device pins signals should be assigned to. This procedure is described below.

Selecting which pin a signal should be assigned to can be thought of as either part of design entry or part of logic synthesis. MAX+plusII allows pin assignments to be made using any of the design-entry tools or using the Compiler or Floorplan Editor. The same procedure can be used in all modules. We will illustrate the steps involved using the Text Editor. Set the name of the current project to be *example1*. Use the Ctrl+t shortcut to open *example1.v* in the Text Editor. Before pins can be assigned manually, it is necessary to first specify which chip to use. This was already done in section C.1.2, when we selected the EPM7128SLC84-7 as shown in Figure C.3. To assign a signal to a pin, select Assign | Pin/Location/Chip to open the window in Figure C.17. Click on Search to open the window in Figure C.18 and then in this window click on List. In the box labeled Names in Database, click on $x1$ to select it and then click on OK. Control now returns to the window in Figure C.17, with $x1$ displayed in the box labeled Node Name. In the box labeled Chip Resource, click on the pull-down menu called Pin. A list of all legal pins on the EPM7128SLC84-7 chip that can be used for the $x1$ signal is displayed. Select a pin, such as pin 4, by clicking on it and then click OK.

The pin assignment is saved in the project's assignment & configuration file (acf), which was introduced in section B.5.3. Open this file in the Text Editor, either by using the Hierarchy Display and double-clicking on the icon labeled *acf*, or by using File | Open and

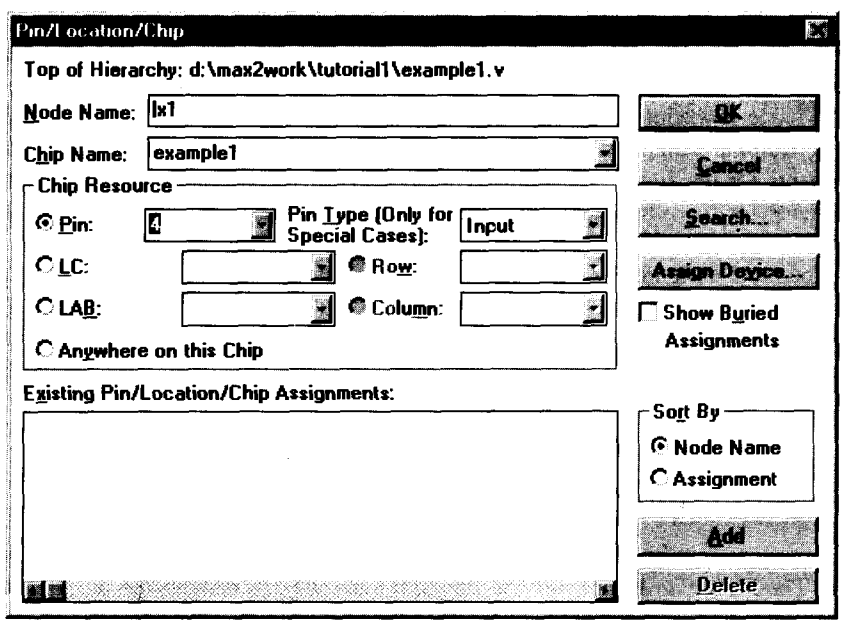


Figure C.17 Manual assignment of a signal to a device pin.

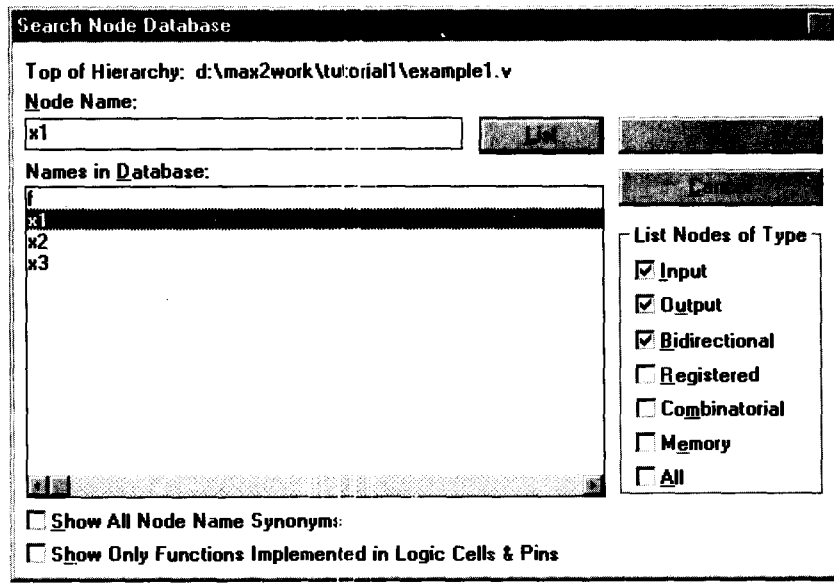


Figure C.18 Selecting a signal name by searching for it.

specifying the file *example1.acf*. The line in the file that specifies the pin assignment is

```
|x1 : INPUT_PIN = 4;
```

An easy way to locate this line in the file is to select Utilities | Find Text and then type $x1$ in the box labeled Search For. Close the *acf* file.

The same procedure can be used to assign pins for $x2$, $x3$, and f, but we will not do so here. Run the Compiler. During the synthesis process the Fitter module uses the pin assignments for the signals that have been specified manually and makes automatic pin assignments for the other signals. Open the report file, or use the Floorplan Editor, to confirm that the manual pin assignment is used in the implemented circuit.

Manual pin assignments can be removed by selecting Assign | Clear Project Assignments in the Text Editor (or Compiler) window. Do this and then click on All in the window that appears; then click OK. Reopen the *acf* file and confirm that it no longer specifies any pin assignments. Run the Compiler again to resynthesize the circuit without the manual pin assignment.

C.4.1 ASSIGNING SIGNALS TO PINS IN THE FLOORPLAN EDITOR

The pin assignment procedure described above can be used in modules other than the Text Editor. The Floorplan Editor provides an additional way of making the pin assignment, which is described below.

The Floorplan Editor can display two main types of information. In section C.1.6 we used it to show the implementation results produced by the Compiler. This mode of display

is activated via the Layout menu by selecting the item called Last Compilation Floorplan. The Last Compilation mode is provided to allow the user to view the implemented circuit, but not to make changes to it. The other possible mode of display is activated by selecting Layout | Current Assignments Floorplan. In this mode the user is able to make manual assignments of signals to pins. Figure C.19 depicts the Floorplan Editor in the mode where pin assignment can be done. The settings from the Layout menu used for the figure are given in Figure C.20. Select Layout | Current Assignments Floorplan. If a message is displayed indicating that a floorplan cannot be displayed because the AUTO chip setting is

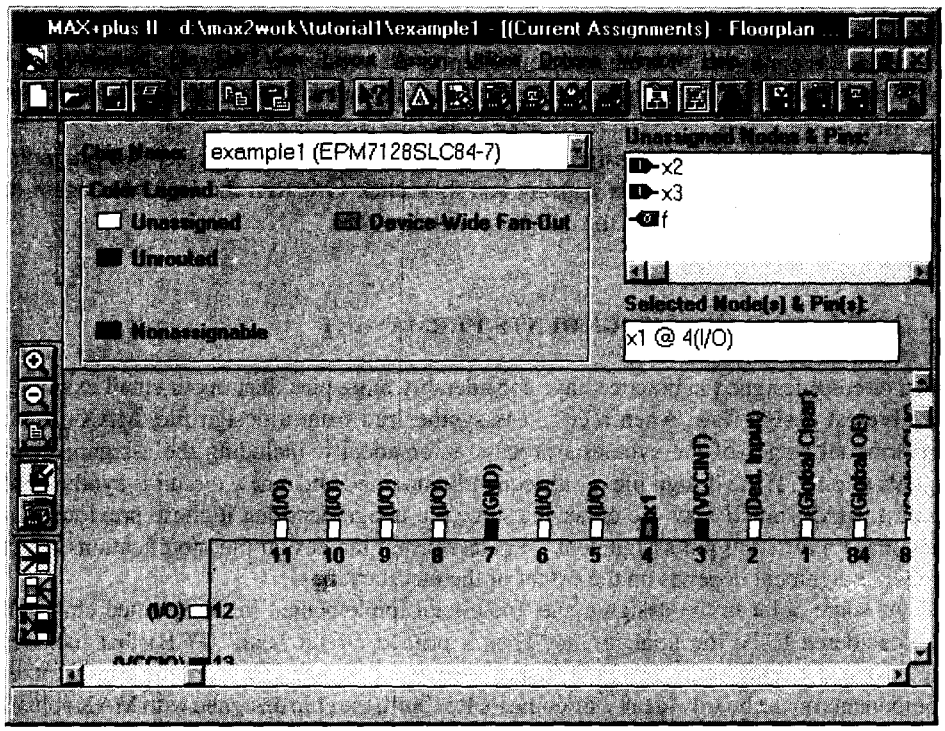


Figure C.19 Using the Current Assignments mode of the Floorplan Editor.

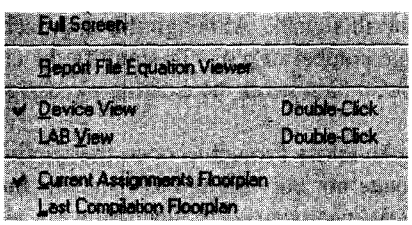


Figure C.20 The Layout menu settings used for Figure C.19.

in effect, then select Assign | Device and click on EPM7128SLC84-7 in the Devices box. Use the scroll bars and magnify buttons to orient the screen as depicted in Figure C.19.

In the Current Assignments Floorplan mode, a signal can be assigned to a pin by clicking on the signal name in the box labeled Unassigned Nodes and Pins and then "dragging and dropping" it on a pin. To create the signal assignment for $x1$ shown in Figure C.19, point the mouse to the $x1$ signal name in the Unassigned Nodes and Pins box. Click and hold down the mouse button. Drag the mouse until the pointer is over pin 4 on the picture of the EPM7128SLC84-7 chip and then release the mouse button. This technique can be used to assign the signals $x2$, $x3$, and f to pins as well. Any signals left unassigned will be assigned to pins automatically by the Fitter. When finished making pin assignments, type the Ctrl+l shortcut to run the Compiler to synthesize a new circuit with the manual assignments. Then examine the report file or use the Floorplan Editor (in the Last Compilation Floorplan mode) to confirm that the manually specified pin assignment was used.

To remove all manually assigned pins, select Assign | Clear Project Assignments. Click on All in the window that appears and then click OK. Run the Compiler again to resynthesize the circuit without the manual pin assignment.

C.4.2 MAKING PIN ASSIGNMENTS PERMANENT

Most circuits designed in practice have considerably more pins than in our small example. In a typical design flow, when a circuit is synthesized from a design file, MAX+plusII performs all stages of the synthesis process automatically, including the assignment of signals to pins. If the design file is subsequently modified and a new circuit is synthesized from it, MAX+plusII may use either the same pin assignment that it chose previously or may make a different pin assignment. Whether or not the design file modifications affect the pin assignment depends on the extent of the modifications.

At some point in the design cycle, the circuit implemented in the selected chip will be considered to be the final result. Then a printed circuit board (PCB) that includes the designed chip is built. The PCB makes permanent connections from the chip to other components on the board. Ideally, after the PCB is built, the chip designed with MAX+plusII will not require changes. However, in practice it is often necessary to make modifications, either because errors are discovered or because of updates to the design specifications. Since the PCB has hardwired connections to the chip that implements the designed circuit, we need to ensure that changes can be made to the design file(s) without changing the existing pin assignment. One way to solve this problem is to assign pins manually, as described in the previous sections, for all signals. Although this approach solves the problem, it is tedious when there is a large number of pins. To simplify the task, MAX+plusII allows a circuit to be *back-annotated*, which saves the existing pin assignment to the project's *acf* file.

Select Assign | Back-Annotate Project to open the window shown in Figure C.21. It is possible to back-annotate not only the pin assignments but also the assignments of logic functions to logic cells (macrocells). In this case we wish to back-annotate the pin assignments, so click on Chips, Pins & Devices. The pin assignments are saved in the project's *acf* file, as we described in section C.4, as if the pins were assigned manually. Open the *acf* file and find the section depicted in Figure C.22.

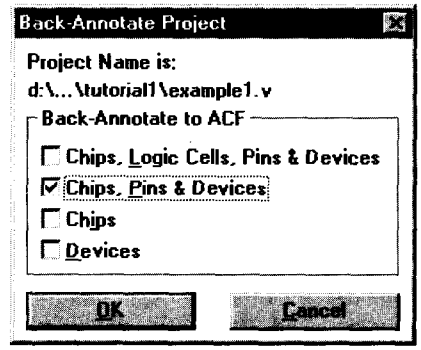


Figure C.21 The back-annotation window.

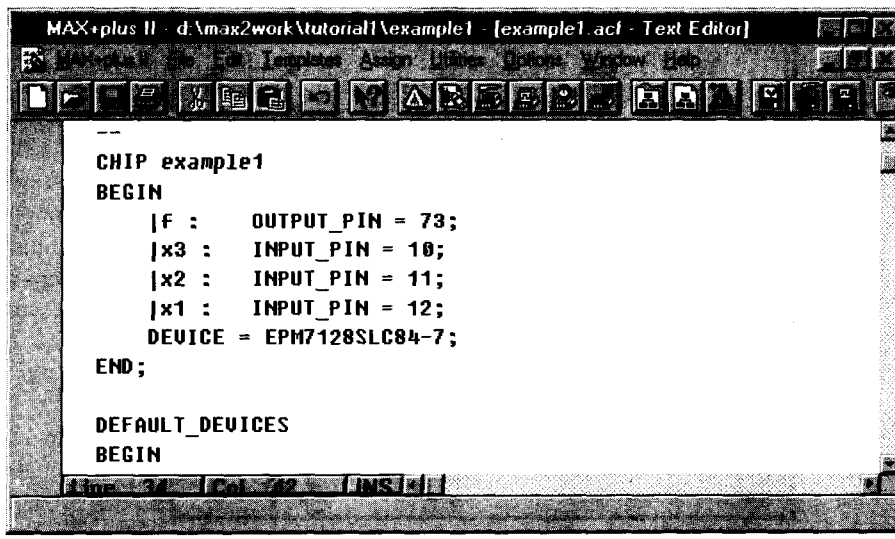
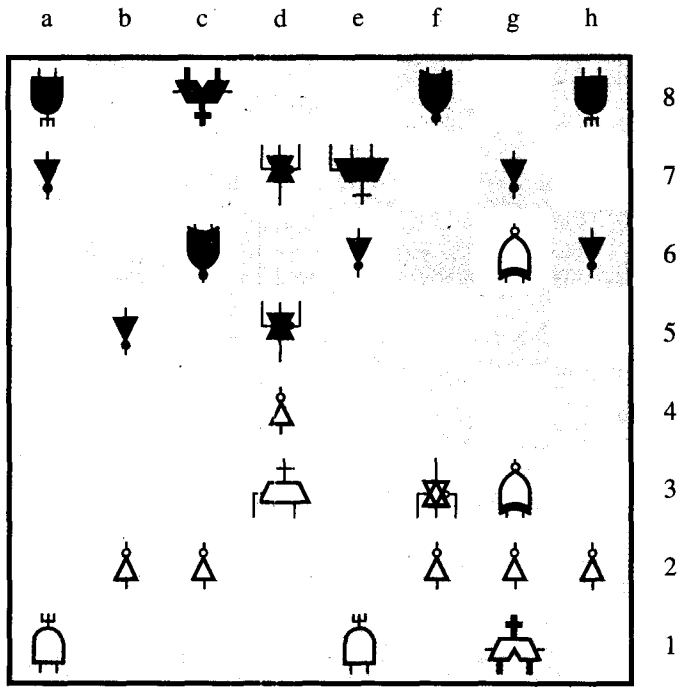


Figure C.22 Back-annotated pin assignments in the assignment & configuration file.

Close the *acf* file. Since for our purposes we do not actually need to back-annotate the circuit, clear the assignments by selecting Assign | Clear Project Assignments. Reopen the *acf* file and confirm that it no longer includes the lines shown in Figure C.22.

C.5 CONCLUDING REMARKS

Having completed this and the preceding tutorial, the reader is familiar with many of the most important features of MAX+plusII. In the next tutorial we will introduce some additional features that are useful for larger circuits, especially those that contain storage elements.

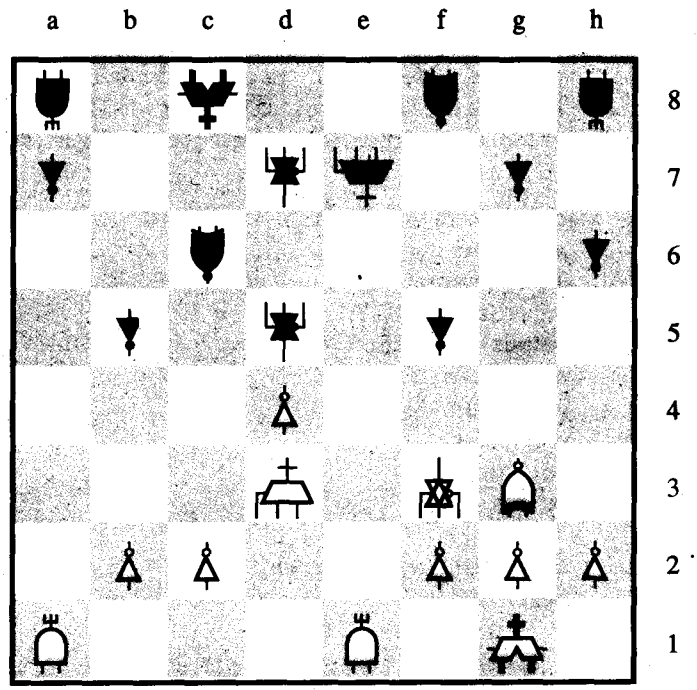


16. Qd1–d3, Bb7–c6

appendix

D

TUTORIAL 3



17. Bg6–f5, e6xf5

This tutorial introduces more advanced capabilities of the MAX+plusII system. We show how Verilog code is organized and compiled and illustrate how multibit signals are represented using the CAD tools. Examples using the building blocks in the library of parameterized modules (LPM) are presented, as well as examples of sequential circuits. In addition to the CAD tool applications that are used in Tutorials 1 and 2, this tutorial introduces the Timing Analyzer application.

D.1 DESIGN USING VERILOG CODE

In section 5.5 we show how an n-bit ripple-carry adder can be specified in Verilog code. In this section we show how the ripple-carry adder can be implemented using the MAX+plusII system.

D.1.1 THE RIPPLE-CARRY ADDER CODE

For storing the files used in this tutorial, we created the directory d:*max2work**tutorial3*. To enter the Verilog code for the adder, in the Manager window select File | New and create a new Text Editor file. Code for the n-bit adder is given in Figure D.1. It takes the carry-in

```
module addern (carryin, X, Y, S, carryout);
   parameter n = 16;
   input carryin;
   input [n-1:0] X, Y;
   output [n-1:0] S;
   output carryout;
   reg [n-1:0] S;
   reg [n:0] C;
   reg carryout;
   integer k;

   always @(X or Y or carryin)
   begin
      C[0] = carryin;
      for (k = 0; k <= n-1; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
      end
      carryout = C[n];
   end

endmodule
```

Figure D.1 Verilog code for a ripple-carry adder.

signal, *carryin*, plus two n-bit numbers, X and Y, as inputs and produces the n-bit output sum, S, and carry-out signal, *carryout*. The code uses the parameter n, so that the adder can be parameterized to work for any value of n. In this example, n is set to 16. In the code the vector C is used to represent the intermediate carries between the stages in the adder. A **for** loop is used to create n full-adders that comprise the ripple-carry adder.

Type the code in Figure D.1 into the Text Editor and save the file in the *tutorial3* directory using the name *addern.v*. Use the Ctrl+Shift+j shortcut command to set *addern* as the name of the current project. We will synthesize a circuit to implement this project in a MAX 7000 CPLD. Select Assign | Device to open the Device dialog box. In the drop-down menu labeled Device Family, select MAX7000S. For this project we selected the EPM7128SLC84-7 chip because this chip is provided on the Altera development board, which is discussed in section C.3. If the EPM7128SLC84-7 chip is not listed in the Device dialog box, click on the option at the bottom of the dialog box that specifies Show only fastest speed grades. Turning this option off by clicking on it results in all speed grades of the chips available to the designer being shown in the Devices box (the copy of MAX+plusII included with the book allows the user to select only a limited subset of the chips that are available from Altera).

Click OK to return to the Text Editor and then open the Compiler module. Since we will use timing simulation for this project, make sure that Timing SNF Extractor is turned on in the Processing menu. Run the Compiler. When analyzing the code, if the Compiler reports errors in your code, fix them.

We will perform a timing simulation to determine the speed of operation of the ripple-carry adder in the chosen device. Open the Waveform Editor and then select Node | Enter Nodes from SNF to open the window shown in Figure D.2. Click on the List button and scroll down in the Available Nodes & Groups box until the node *carryin* is visible. Click on the button marked => to copy *carryin* into the Selected Nodes & Groups box. Scroll

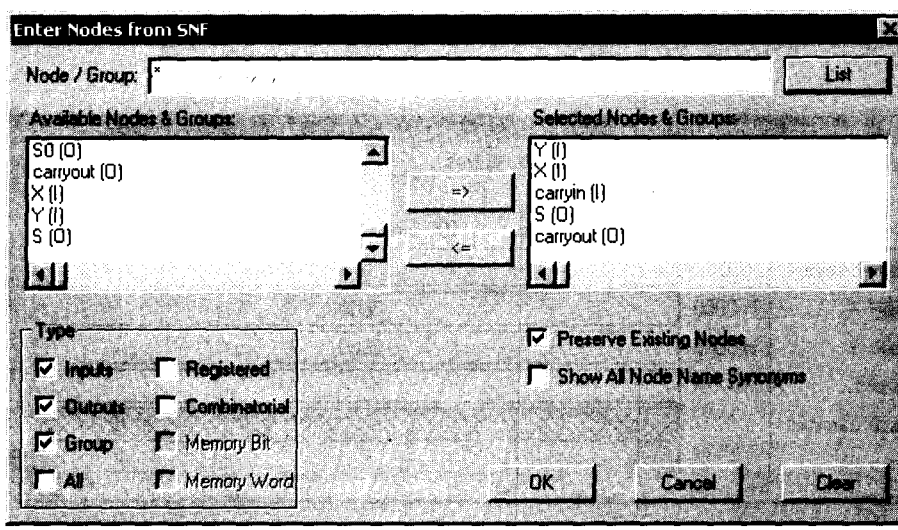


Figure D.2 Importing single-bit and multibit node names.

further down in the Available Nodes & Groups box and select *carryout*. Finally, select the nodes *X*, *Y*, and *S*, which are displayed at the bottom of the list of Available Nodes & Groups. These nodes represent the 16-bit vectors in the Verilog code in Figure D.1. MAX+plusII uses the term *Group*, or *Bus*, for multibit nodes. Click OK to return to the Waveform Editor.

Select File | End Time and set the total simulation time to 250 ns. Select Options | Grid Size to set the guideline spacing to be 25 ns. Save the file with the name *addern.scf*. Use the Ctrl+w shortcut command so that the entire time range, from 0 to 250 ns, is displayed in the Waveform Editor window, as depicted in Figure D.3.

Below the label Value the Waveform Editor displays the value of the signal waveforms at the point where the reference line is currently situated (the value X for an output signal means that the signal value is unknown, because the simulation has not yet been performed). For multibit signals the displayed signal values have an associated number base, which is indicated by a letter. In Figure D.3 the number base used for the multibit signals *X*, *Y*, and *S* is hexadecimal, denoted by the letter H. It is possible to change the number base (to binary, octal, or decimal) by double-clicking on the H, but we do not need to do so here. If not already done, activate the Waveform Editing tool by selecting its icon on the left side of the Manager window (the icon is labeled <–|–>). Click the mouse at the 100 ns point on the *X* waveform, drag to 175 ns, and then release the mouse. The Overwrite Group Value dialog box shown in Figure D.4 appears. Type 3FFF into this box and click OK. Next set the value of *X* to 7FFF in the time range from 175 ns to 250 ns and set *Y* to the value 0001 in the time range from 50 ns to 250 ns. Save the *addern.scf* file.

Open the Simulator module and click Start to run the timing simulation. The results of the simulation are shown in Figure D.5. Move the reference line in the Waveform Editor to the point in the figure where *S* takes the value 4000. Since this sum is produced at 137.5 ns, and *X* changes to 3FFF at 100 ns, the adder requires 37.5 ns to generate the sum.

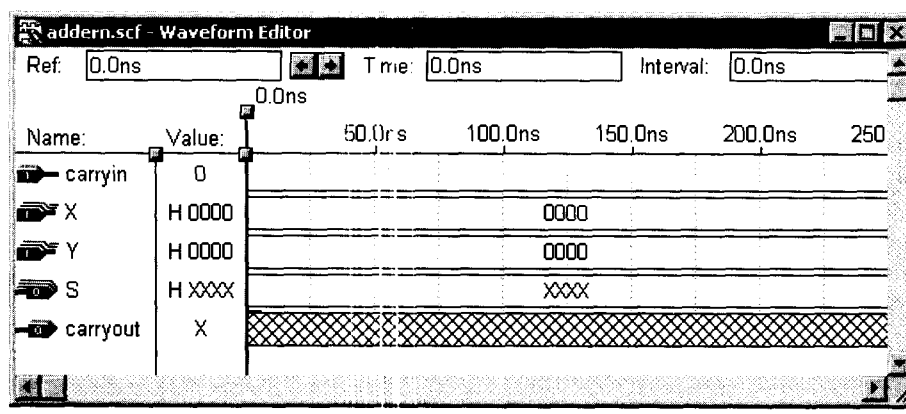


Figure D.3 The waveform display for the 16-bit adder.

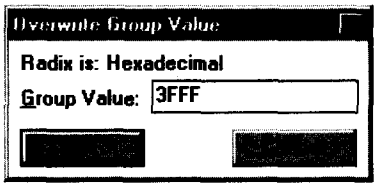


Figure D.4 Specifying the value of a multibit signal.

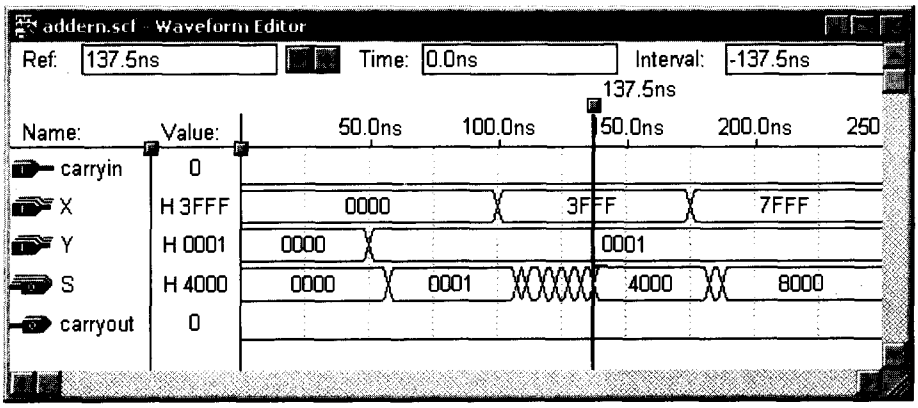


Figure D.5 Timing simulation results for the ripple-carry adder.

D.1.2 USING THE TIMING ANALYZER MODULE

The only MAX+plusII module that we have not yet used is the Timing Analyzer. It shows detailed timing information for the circuit synthesized by the Compiler. Select MAX+plusII | Timing Analyzer. Three types of analyses are available, which are listed in the Analysis menu. The type currently selected should be the Delay Matrix. It reports the propagation delays in the circuit from each primary input to each primary output. The other types of analysis are applicable only to circuits that contain storage elements. In the Timing Analyzer window, click the Start button. Figure D.6 shows part of the results produced. In the figure we have sized the window so that four columns of data are visible, and the data has been scrolled to the right to show the propagation delays for nodes *S*12 to *S*15. The values in each square of the matrix indicate the minimum and maximum delays in the circuit through all paths from each input node to each output node. For instance, the minimum delay from node *X*0 to node *S*14 is 32.5 ns, and the maximum delay is 41.5 ns. Observe that the propagation delays increase for more significant stages of the adder, as we expect in the ripple-carry structure. The longest delay from any input to any output in the circuit is 42.5 ns. The Timing Analyzer can be set to report only the longest delay by selecting Options | Time Restrictions. Other features of the Timing Analyzer will be discussed in section D.3.3.

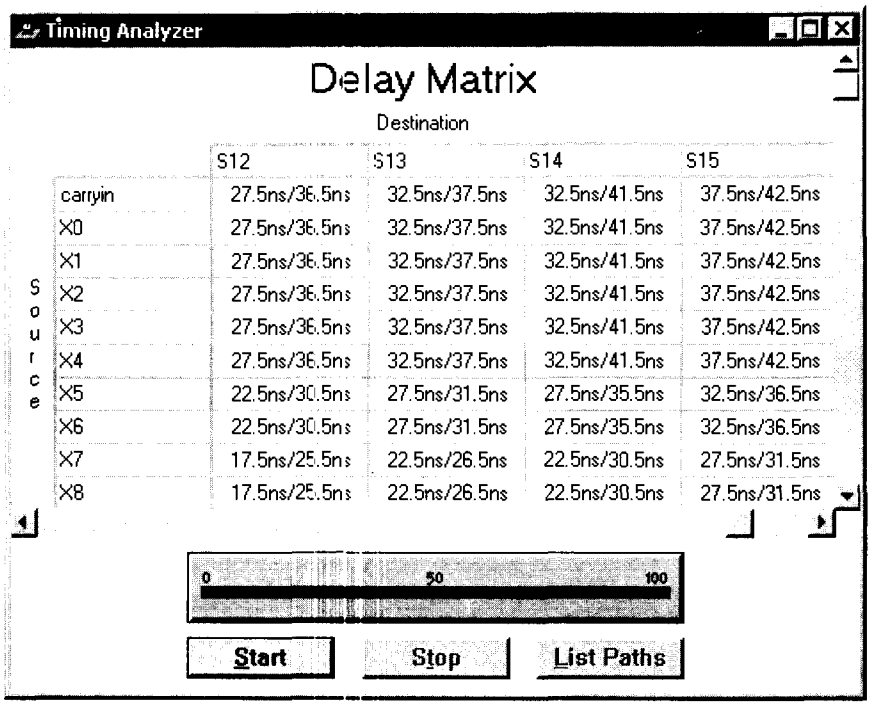


Figure D.6 Using the delay matrix in the Timing Analyzer.

We have finished working on the *addern* project, so close all open windows to return to the Manager window.

D.2 USING AN LPM MODULE

In section 5.5.1 we show how to create an adder circuit using the *lpm_add_sub* module in the library of parameterized modules (LPM). In this section we compare the adder circuit produced by the *lpm_add_sub* module to the ripple-carry adder implemented in section D.1.

Create a new Text Editor file and enter the code shown in Figure D.7. It instantiates a 16-bit version of the *lpm_add_sub* module with the same input and output signals used in Figure D.1. Save the file with the name *adderlpm.v* and set this file as the current project. Using the Assign menu, select the same device used in the previous example. Run the Compiler to synthesize a circuit that implements the adder.

Open the Waveform Editor and create the same simulation vectors that we used in section D.1.1, shown in Figure D.5. Save the waveform file, open the Simulator, and run the timing simulation. The results of the simulation should be as illustrated in Figure D.8. Position the reference line so that it shows the delay incurred by the adder to produce the

```
module adderlpm (carryin, X, Y, S, carryout);
   input carryin;
   input [15:0] X, Y;
   output [15:0] S;
   output carryout;

   lpm_add_sub instance (.cin(carryin), .dataa(X), .datab(Y),
      .result(S), .cout(carryout));
      defparam instance.lpm_width = 16;

endmodule
```

Figure D.7 A 16-bit adder built using the lpm_add_sub module.

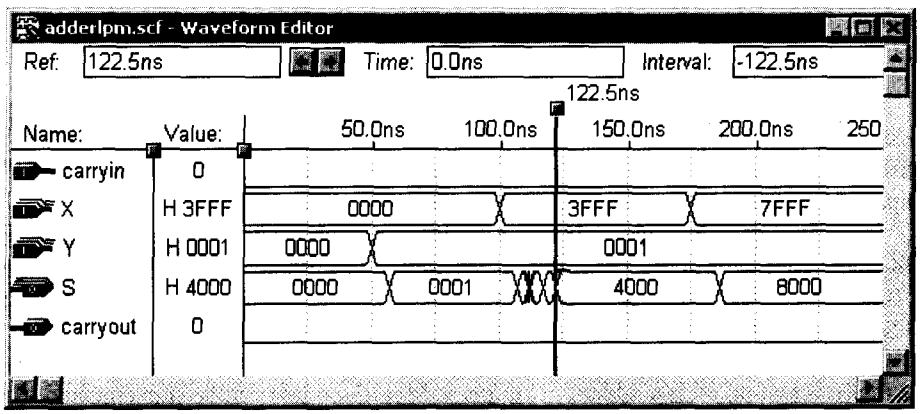


Figure D.8 Timing simulation results for the code in Figure D.7.

sum 4000. This sum is produced 22.5 ns after *X* changes to 3FFF. Comparing this delay to the one in Figure D.5, which is 37.5 ns, we see that the *lpm_add_sub* module generates a faster adder circuit.

To view the source code file that defines the *lpm_add_sub* module, open the Hierarchy Display module. A simplified picture of the window that appears is given in Figure D.9. It shows that the *lpm_add_sub* module is instantiated using a building block named *addcore*. Double-click on the small icon labeled tdf, which stands for *text design file*, next to addcore:adder. The source code file named *addcore.tdf* is opened in the Text Editor. The code is written using the Altera Hardware Description Language (AHDL), which is another language supported by MAX+plusII. Although AHDL uses a different syntax than Verilog uses, the two languages have enough similarity to enable the reader to understand some of the code. The file includes comments that specify how the *addcore* module is implemented

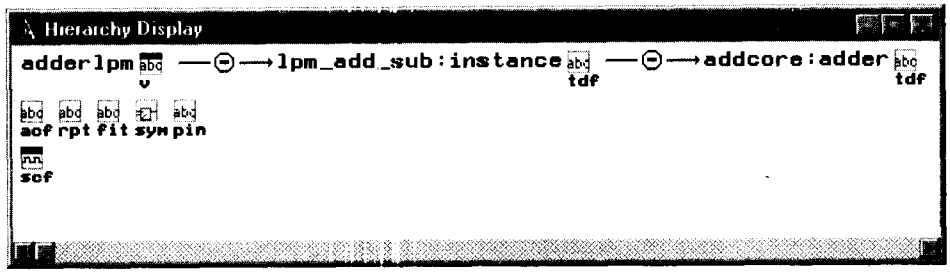


Figure D.9 The Hierarchy display for the code in Figure D.7.

in different types of devices. For devices designated as MAX, such as MAX 7000, *addcore* is implemented as blocks of eight-bit carry-lookahead adders, with ripple-carry between the adder blocks. Hence our 16-bit adder in Figure D.7 is implemented using two 8-bit carry-lookahead adders, with the carry-out of one adder connected to the carry-in of the other. A diagram of this type of adder circuit is shown in Figure 5.17.

Implementation in a FLEX 10K Chip

Close *addcore.tdf* and close the Hierarchy Display. We will now implement the code in Figure D.7 in a FLEX 10K device. Select Assign | Device and choose FLEX10K in the Device Family drop-down menu. For the results shown here, we selected the EPF10K20RC240-4 chip, which is included on the Altera development board described in section C.3.

Open the Compiler and synthesize a circuit that implements the project in the FLEX 10K chip. Run the Timing Simulator to generate the simulation results shown in Figure D.10. Positioning the reference line in the Waveform Editor at the point where *S* changes to 4000 shows that 56 ns are needed to generate the sum in the FLEX 10K device.

In Appendix E we show that FLEX 10K devices include special-purpose carry logic for implementation of fast adders. The *lpm_add_sub* module can be implemented using this resource by directing the logic synthesis algorithms to optimize the generated circuit

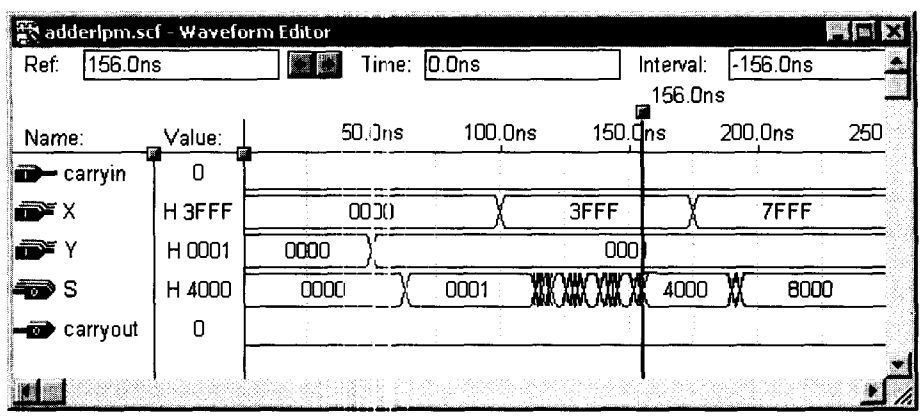


Figure D.10 Timing results when optimized for area in a FLEX 10K device.

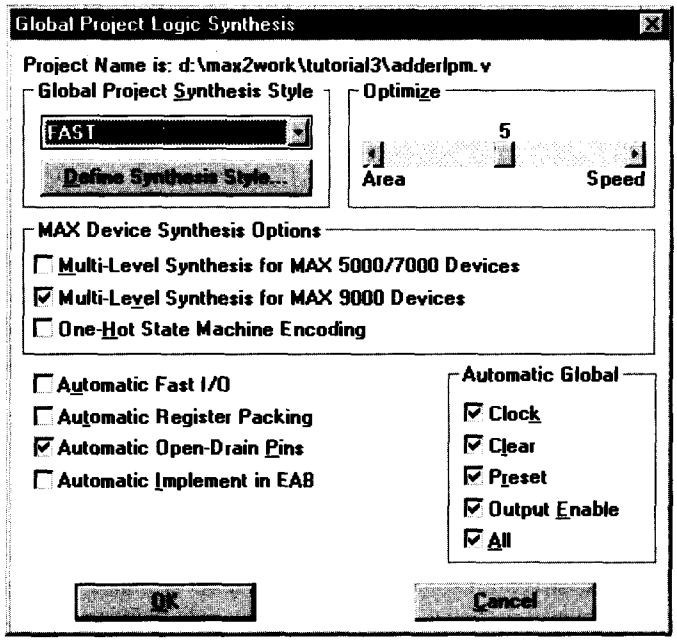


Figure D.11 Setting logic synthesis options.

for speed. Select Assign | Global Project Logic Synthesis to open the window shown in Figure D.11. In the Global Project Synthesis Style drop-down menu, select FAST. Run the Compiler to synthesize a circuit optimized for speed and then run the Timing Simulator again. The results, illustrated in Figure D.12, show that only 20.9 ns are needed to generate $S = 4000$ when the dedicated carry-logic resources are used.

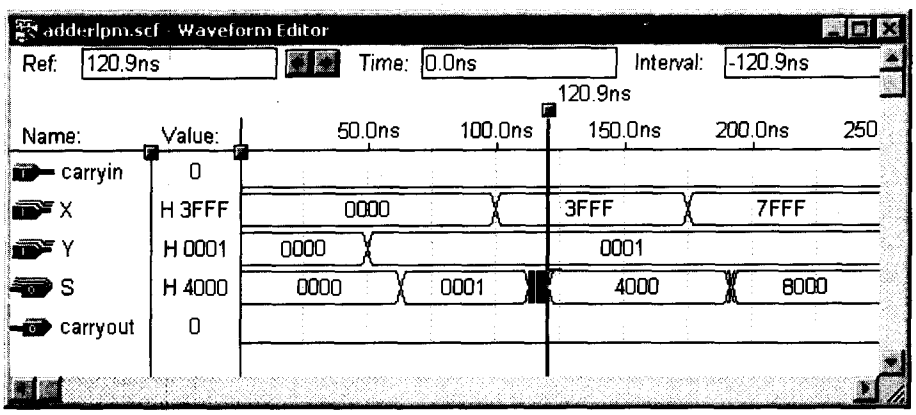


Figure D.12 Timing results when optimized for speed in a FLEX 10K device.

We have finished working on the *adderlpm* project. Close all open windows to return to the Manager window.

D.3 DESIGN OF A SEQUENTIAL CIRCUIT

This example shows how to implement a sequential circuit using MAX+plusII. The presentation assumes that the reader is familiar with the material in Chapter 7. Figure 7.48 depicts a circuit with a four-bit adder connected to a register that feeds back to the adder. This section shows how to implement the circuit using modules from the LPM library. We will first describe the circuit using a schematic and then give an equivalent design using Verilog code.

D.3.1 USING THE GRAPHIC EDITOR

Select File | New and create a new Graphic Editor file. Save the file with the name *graphic2.gdf*. Use the Ctrl+Shift+j shortcut command to set *graphic2* as the current project. Double-click on the blank space in the Graphic Editor window. Open the *primitives* library by double-clicking on the line in the Symbol Libraries box that ends in prim. Import an *input* symbol using the procedure described in section B.2.2. Import two more *input* symbols and two *output* symbols.

Next we need to import two symbols, *lpm_ff* and *lpm_add_sub*, from the LPM library. The *lpm_ff* module is an *n*-bit register, and the *lpm_add_sub* module is an adder/subtractor subcircuit. One way to import the modules is to open the LPM library in the same way as we opened the *primitives* library and then select the appropriate symbols to be imported into the schematic. However, each LPM module has various parameters that have to be set to configure the module. MAX+plusII provides a tool called the *MegaWizard Plug-In Manager* to help configure the modules.

Double-click on a blank space in the Graphic Editor to open the Enter Symbol dialog box. Click on the MegaWizard Plug-In Manager button to open the window shown in Figure D.13. Another way to start this tool is to select File | MegaWizard Plug-In Manager. Click on Next to create a new instance of an LPM module to be imported into the schematic.

In the next page of the MegaWizard Plug-In Manager, shown in Figure D.14, click on the + symbol next to the LPM storage item in the Available Megafunctions box. Then click on LPM_FF to select this module. In addition to creating a symbol for use in a schematic, the MegaWizard tool creates a design file for use with a hardware description language. In Figure D.14 we clicked on Verilog, but we will not use the resulting Verilog file here. It is necessary to provide a name that will be used for the symbol file. We chose the name *Reg4*, as shown in the figure.

Click Next to open the window in Figure D.15. Click on the drop-down menu and set the number of flip-flops in the *lpm_ff* module to 4. The default flip-flop type is now set to D, and the MegaWizard displays the graphical symbol for the *lpm_ff* that will be created for use in the schematic. Click Next to open the window in Figure D.16. Under

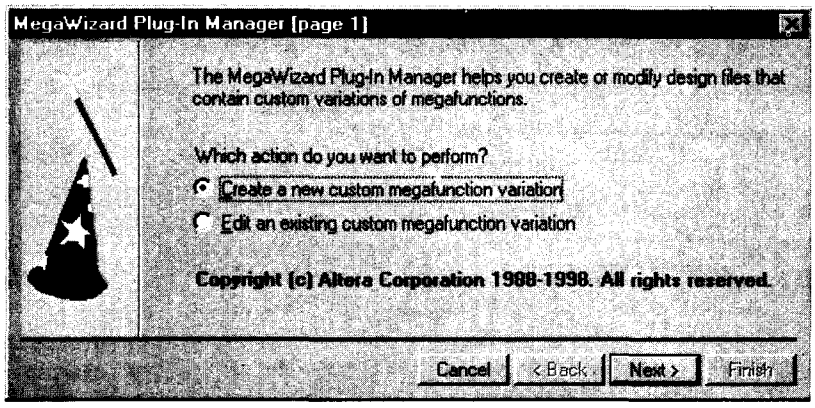


Figure D.13 Using the MegaWizard Plug-In Manager.

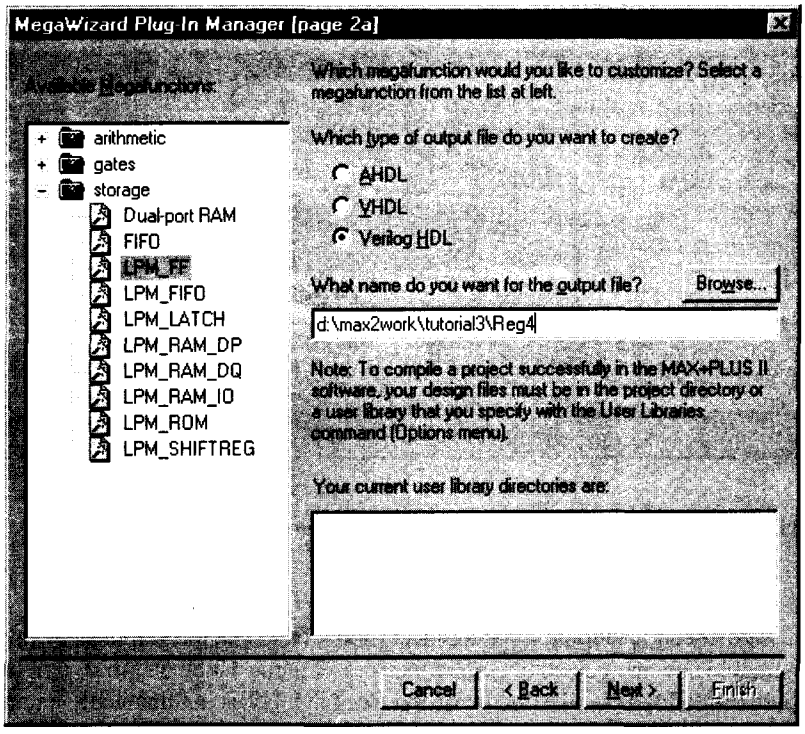


Figure D.14 Selecting the *lpm_ff* module.

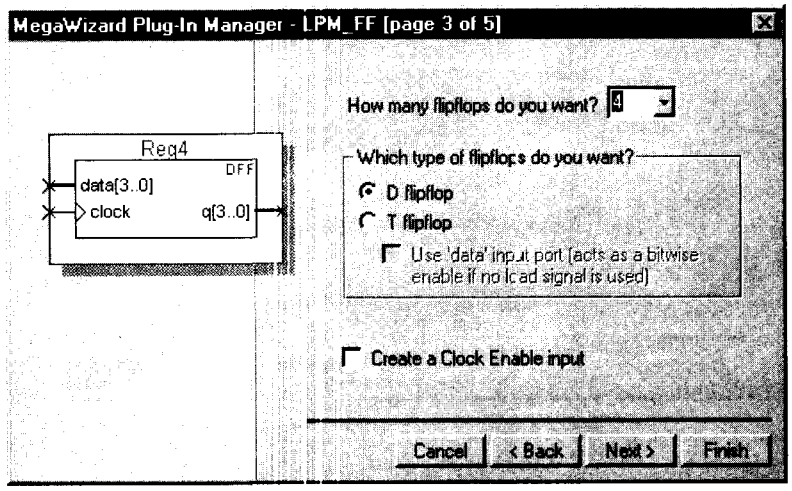


Figure D.15 Configuring the *lpm_ff* module.

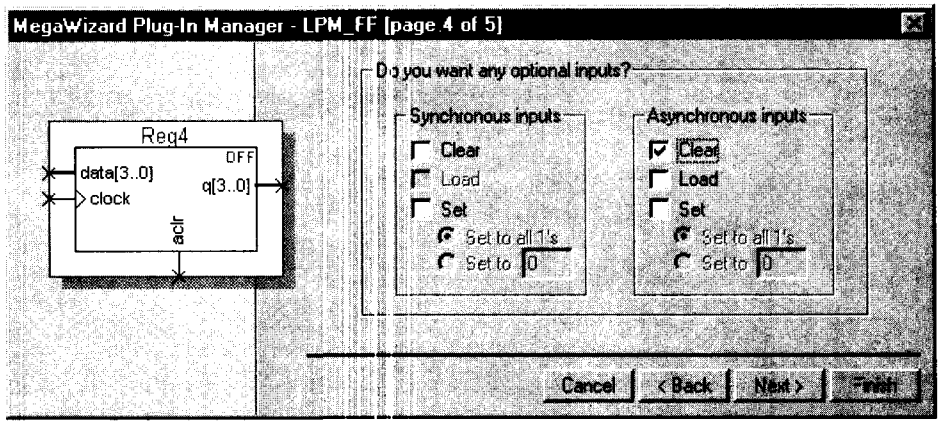


Figure D.16 Adding an clear input on the *lpm_ff* module.

the item Asynchronous inputs, click to select Clear. Note that the graphical symbol shown for the module now includes the asynchronous reset input. Click on Finish to return to the Enter Symbol dialog box. The name of the *Reg4* symbol is automatically entered in the box labeled Symbol Name. Click OK to return to the Graphic Editor.

Start the MegaWizard tool again to import an instance of the *lpm_add_sub* module. In the screen shown in Figure D.17, click on the + symbol beside LPM arithmetic and then

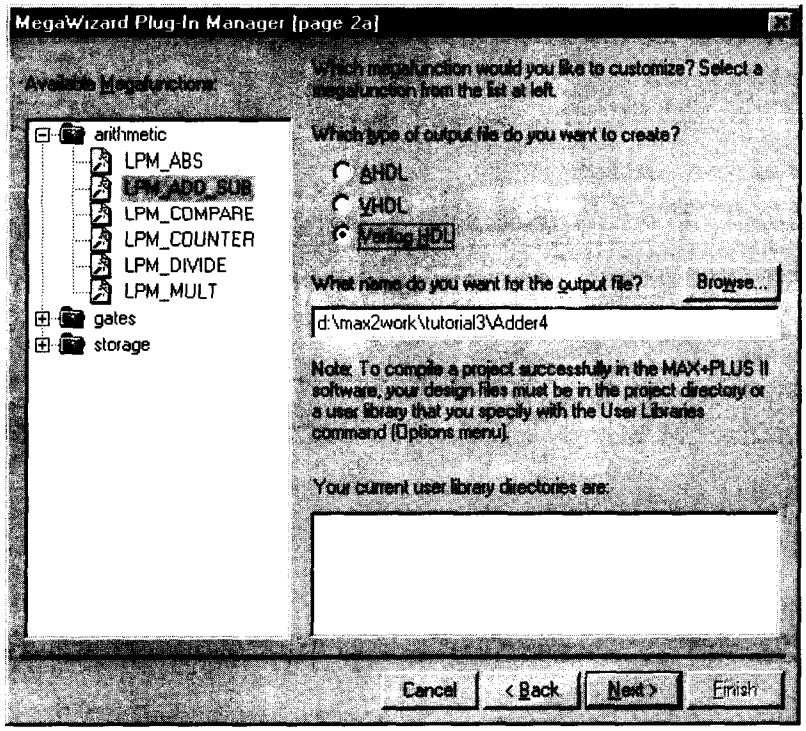


Figure D.17 Importing the *lpm_add_sub* module.

click on LPM_ADD_SUB. Select Verilog for the type of HDL source code file that should be generated. Although we will not make use of the generated Verilog file, it specifies how the *lpm_add_sub* module can be instantiated in Verilog code with our chosen parameter values. As shown in the figure, use the name *Adder4* for the symbol. Click Next to open the window in Figure D.18. Use the drop-down menu to configure the module as a four-bit adder. Click Finish to return to the Enter Symbol box and then click OK.

The symbols that have been imported into the schematic are given in Figure D.19. As indicated in the figure, assign the name *Reset* to the input symbol in the lower-left corner of the schematic and assign the name *Clock* to the input symbol above that. The third input symbol is used for the circuit's four-bit input, named *Data*. Use the syntax *Data*[3..0] for the four-bit signal. Assign the name *Sum*[3..0] to the output symbol in the top-right corner of the schematic; we use this symbol to show the sum produced by the adder in a timing simulation. Assign the name *RegSum*[3..0] to the other output symbol.

Connecting Nodes with Wires and Names

If not already done, activate the Selection tool in the Graphic Editor by clicking on the icon that looks like an arrowhead on the left edge of the window. As explained in section

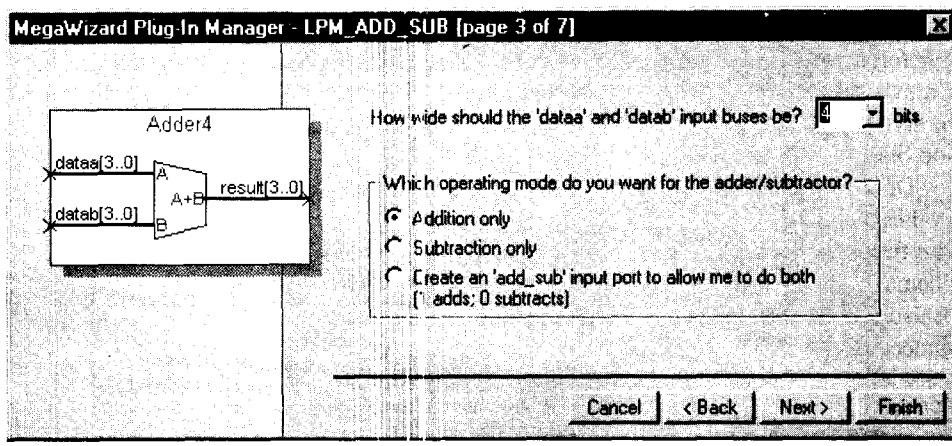


Figure D.18 Configuring the *lpm_add_sub* module.

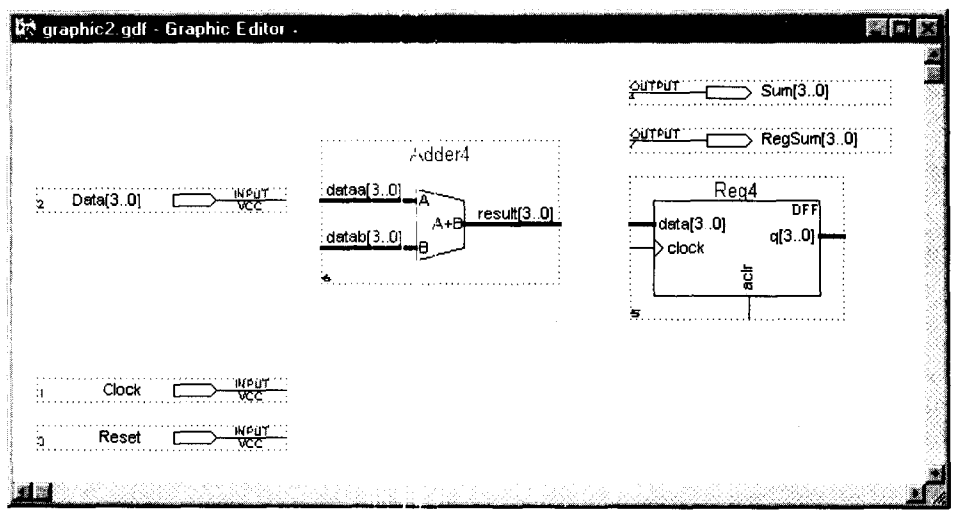


Figure D.19 The symbols imported into the schematic.

B.2.2, the Selection tool allows the Graphic Editor to change automatically between the modes of selecting a symbol or drawing wires to interconnect symbols.

Draw a wire from the *Clock* input symbol to the clock node on the *Reg4* symbol and another wire from *Reset* to the *aclr* node on *Reg4*. Next place the mouse on top of the pinstub for the node *dataa* [3..0] on the *Adder4* symbol. Draw a wire from this pinstub to

the left until it reaches the pinstub on the *Data* [3..0] input symbol. The wire is drawn as a bold line to indicate that it represents a multibit signal. MAX+plusII uses the term *bus wire* for multibit wires. The Graphic Editor automatically creates a bus wire when the wire is drawn starting at a multibit node such as *dataa* [3..0]. It is possible to manually select the style of line that should be drawn by clicking the right mouse button and selecting the Line Style menu item.

Draw a bus wire from the output of *Adder4* to the *data* [3..0] node on *Reg4*. Draw another wire to connect this node to the *Sum* [3..0] output symbol. Finally, draw a bus wire from the output of *Reg4* back to the *datab* [3..0] node on *Adder4*. To complete the schematic, we need to connect the output of *Reg4* to the *RegSum* [3..0] output symbol. Instead of drawing a wire to make this connection, we will illustrate a different way of connecting the nodes in the schematic. The Graphic Editor allows the user to attach a label to a wire. To attach a label to the bus wire between the output on *Reg4* and the input on *Adder4*, click the mouse somewhere on the wire. The wire is highlighted to show that it is selected, and a small cursor appears on the wire. Type the label *RegSum* [3..0] and observe that this label appears on the wire. In Figure D.20 we attached the label at a point below the *Adder4* symbol, but the label can be placed anywhere on the wire. The Graphic Editor now considers the bus wire to be physically connected to the *RegSum* [3..0] output symbol, just as if a wire were drawn between them. We say that the nodes are interconnected *by name*, rather than by drawing a wire. Most schematic capture tools allow nodes to be interconnected in this manner. In large schematics, connecting nodes by name allows the schematic diagram to appear less cluttered because it means that fewer wires have to be drawn.

For completeness we should also mention an alternative way to specify the name (label) of a bus wire. It can be specified using individual signal names separated by commas. In this

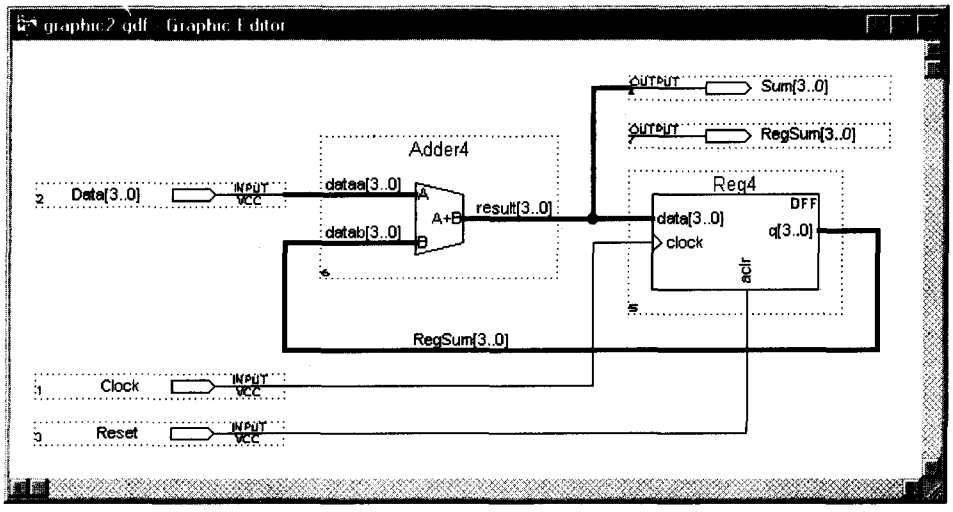


Figure D.20 The completed schematic.

example we could type the label as follows: [*RegSum3*, *RegSum2*, *RegSum1*, *RegSum0*]. This style of label allows a bus wire to be composed of any group of node names. For instance, if nodes *x*, *y*, and *z* exist in a design, then a three-bit bus could have the label [*x*, *y*, *z*].

In some schematics it is necessary to connect the individual signals within a bus to other nodes. For example, if a bus exists called *C* [3..0], an individual signal in this bus can be accessed by assigning an appropriate label to a node. For instance, to connect a node to the right-most bit in *C* [3..0], the node would be given the label *C* [0]. An example of a schematic that uses labels in this manner appears in Figure 10.33 in section 10.2.5.

D.3.2 SYNTHESIZING A CIRCUIT AND USING THE TIMING SIMULATOR

Save the completed schematic. Use the Assign menu to choose a device from the MAX 7000S family. For the results shown here, we selected the EPM7128SLC84-7 device. Compile the design. Open the Waveform Editor and use Node | Enter Nodes from SNF to select the nodes shown in Figure D.21. Set the total simulation time to 500 ns and set the grid size to 25 ns. Set *Reset* to 1 for the first 50 ns of the simulation time and then leave *Reset* at 0 for the rest of the time. To enter the waveform for the clock signal, point the mouse at the name of the *Clock* waveform in the Waveform Editor display and click the right mouse button. In the pop-up menu, select Overwrite | Clock to open the dialog box shown in Figure D.22. The box specifies that the *Clock* signal has the initial value 0 and its period is equal to twice the grid size (50 ns). Click OK, and the Waveform Editor automatically draws the periodic clock signal.

The next step is to draw the waveform for the *Data* [3..0] input. In a typical digital system, input signals change values shortly after the active clock edge because they are the outputs of a register controlled by the same clock. In this example all changes in the

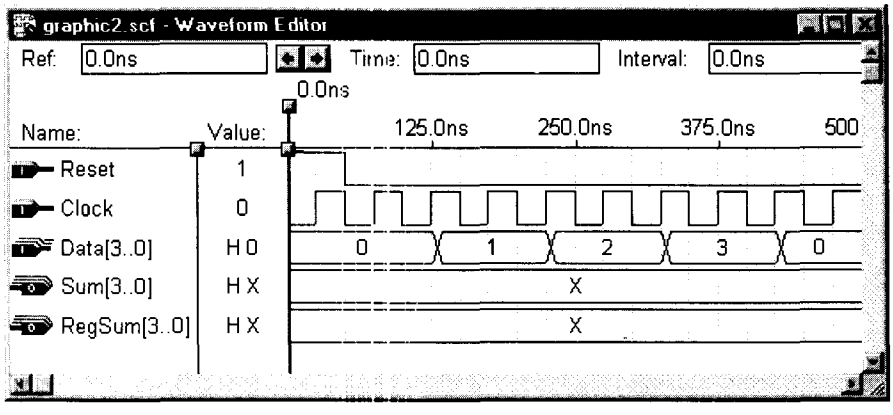


Figure D.21 The input waveforms.

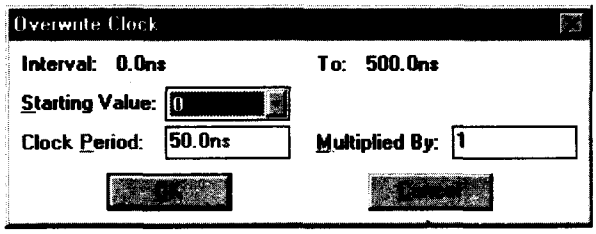


Figure D.22 Creating a clock waveform.

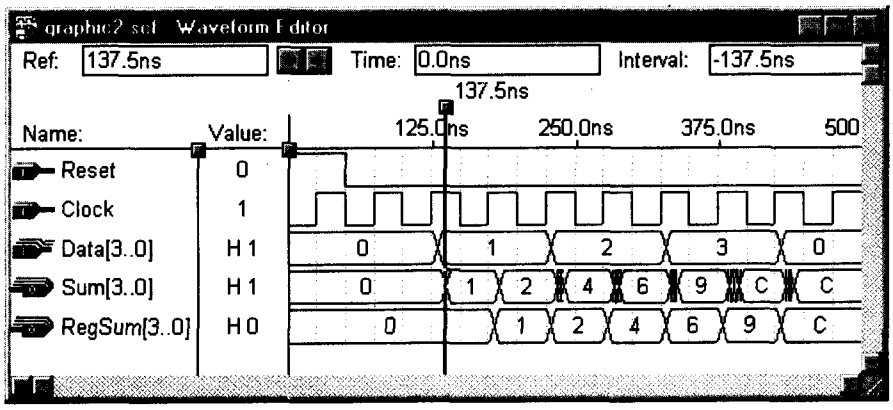


Figure D.23 The results produced by timing simulation.

Data [3..0] input will occur 5 ns after a positive clock edge. Change the grid size to 5 ns. Use the magnifying glass button to zoom in on the Waveform Editor display and draw the waveform shown in Figure D.21 for *Data* [3..0]. In this waveform changes to the *Data* [3..0] signal occur at 130 ns, 230 ns, 330 ns, and 430 ns. Save the waveform file and run the Timing Simulator to generate the simulation results given in Figure D.23. Observe that 7.5 ns are needed to generate a sum from the adder, which is then clocked into the register at the next active clock edge.

D.3.3 USING THE TIMING ANALYZER

We introduced the Timing Analyzer in section D.1.2 and showed that it can display propagation delays in a circuit. Another use of the Timing Analyzer is to calculate the minimum clock period for which a sequential circuit will operate correctly. Open the Timing Analyzer and select Analysis | Registered Performance to open the window in Figure D.24.

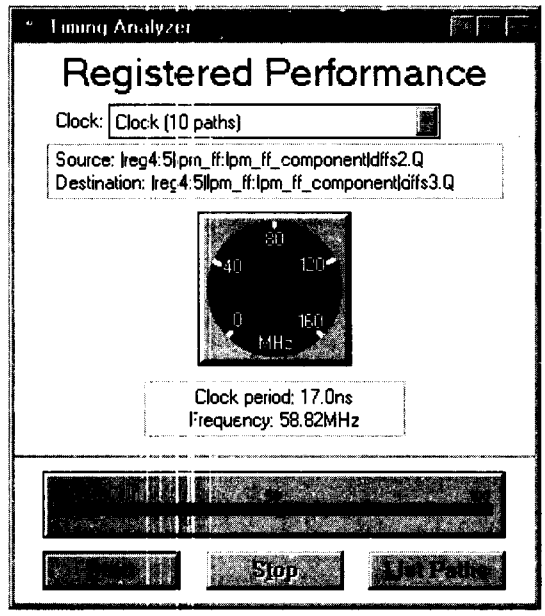


Figure D.24 Maximum clock frequency estimate.

Click the Start button. The Timing Analyzer reports that the circuit operates correctly with a minimum clock period of 17 ns. This clock period accounts for all propagation delays in the circuit and for the setup time at the data input on *Reg4*. The Timing Analyzer can also be used to check for setup and hold time violations in a circuit by selecting Analysis | Setup/Hold Matrix.

We have now finished working with the *graphic2* project.

D.3.4 USING VERILOG CODE

Figure D.25 shows Verilog code that is equivalent to the *graphic2* schematic. The register is specified as a separate module, *reg4*. The adder in the circuit is defined using the + operator in the **assign** statement.

The reader should type the code in Figure D.25 into a Text Editor file, compile it, and simulate the resulting circuit. The results should be identical to those produced for the *graphic2* project.

```
module example3 (Data, Clock, Reset, RegSum, Sum);
   input Clock, Reset;
   input [3:0] Data;
   output [3:0] Sum, RegSum;
   wire [3:0] Sum, RegSum;

   reg4 R1 (Sum, Clock, Reset, RegSum);
   assign Sum = Data + RegSum;

endmodule

module reg4 (D, Clock, Reset, Q);
   input [3:0] D;
   input Clock, Reset;
   output [3:0] Q;
   reg [3:0] Q;

   always @(posedge Reset or posedge Clock)
      if (Reset)
         Q <= 0;
      else
         Q <= D;

endmodule
```

Figure D.25 Verilog code equivalent to the *graphic2* project.

D.4 DESIGN OF A FINITE STATE MACHINE

In section 8.1 we show a simple Moore-type finite state machine that has one input, w, and one output, z. Whenever w is 1 for two successive clock cycles, z is set to 1. The state diagram for the FSM is given in Figure 8.3; it is reproduced in Figure D.26. Verilog code that describes the machine appears in Figure 8.29; it is reproduced in Figure D.27. Create a new Text Editor file and enter the code shown in Figure D.27. Save the file with the name *simple.v* and set this as the name of the current project.

D.4.1 IMPLEMENTATION IN A CPLD

Use the Assign menu to select a MAX 7000S device. We chose the EPM7128SLC84-7 chip for the results presented here. Run the Compiler to synthesize a circuit for the FSM. Open the Compiler report file and scroll down in the file to see the logic expressions shown in Figure D.28. These expressions can be derived by hand by considering the state diagram

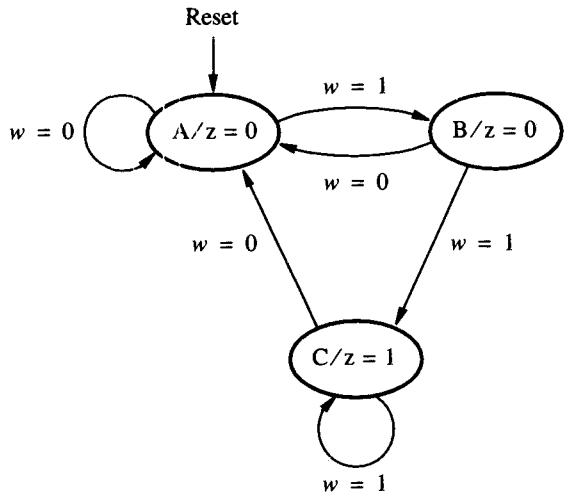


Figure D.26 State diagram of a Moore-type FSM.

and state assignment. For example, state variable y_1 has the value 1 only in state B, and the state diagram specifies that the machine changes to state B only if it is currently in state A and $w = 1$. Since A has the code $y_2y_1 = 00$, then y_1 should be set to 1 if $y_2 = y_1 = 0$ and $w = 1$. Hence the expression for y_1 in Figure D.28 is

$$y_1 = w\bar{y}_1\bar{y}_2$$

The expressions for y_2 and z are derived similarly.

Open the Waveform Editor and import the nodes *Resetn*, *Clock*, *w*, *z*, and *y*, as shown in Figure D.29. Set the total simulation time to 650 ns and set the grid size to 25 ns. Set *Resetn* = 0 during the first 50 ns, and then set *Resetn* = 1. Create the periodic *Clock* signal as described in section D.3.2. Draw the waveform for *w* shown in the figure. Each change in *w* occurs 5 ns after a positive clock edge; use a grid size of 5 ns to draw this waveform. Save the file and run the Timing Simulator to generate the results shown. The FSM behaves correctly, setting $z = 1$ in each clock cycle for which $w = 1$ in the preceding two clock cycles. Examine the timing delays in the circuit, using the reference line in the Waveform Editor. It shows that changes in the FSM's state occur 2.5 ns after an active clock edge and that an additional 7 ns are needed to change the value of the z output.

Open the Timing Analyzer. Select Analysis | Registered Performance and click the Start button. The analysis reports that the FSM operates correctly with a maximum clock frequency of 125 MHz.

D.4.2 IMPLEMENTATION IN AN FPGA

To see how the FSM is implemented in an FPGA, use the Assign menu to select a device in the FLEX 10K family, such as the EPF10K20RC240-4.

```
module simple (Clock, Resetn, w, z);
   input Clock, Resetn, w;
   output z;
   reg [2:1] y, Y;
   parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10;

   // Define the next state combinational circuit
   always @(w or y)
      case (y)
         A: if (w)    Y = B;
            else      Y = A;
         B: if (w)    Y = C;
            else      Y = A;
         C: if (w)    Y = C;
            else      Y = A;
         default:     Y = 2'bxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn or posedge Clock)
      if (Resetn == 0)  y <= A;
      else  y <= Y;

   // Define output
   assign  z = (y == C);

endmodule
```

Figure D.27 Verilog code for the FSM in Figure D.26.

In section 8.8 we said that when implementing an FSM in an FPGA, a good strategy is to use one-hot encoding, with one state variable assigned to each state. Although our simple FSM has only three states we can use it to illustrate the general idea. Using a one-hot code would give $A = 100$, $B = 010$, and $C = 001$. In PLDs, flip-flops usually have clear inputs but they do not have preset inputs. It is convenient to invert the left-most bit in the state codes so that the reset state is $A = 000$, while the other two states are $B = 110$, and $C = 101$. The reader is encouraged to modify the Verilog code in Figure D.27 to extend the signals y and Y to have three bits and to change the state codes as described above. Compile the code, examine the equations generated in the report file, and perform a timing simulation for the implementation in the FPGA chip.

simple.rpt - Text Editor

```
-- Node name is ':50' = 'y1'
-- Equation name is 'y1', location is LC113, type is buried.
y1       = DFFE( _EQ001 $  GND, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ001 =  w & !y1 & !y2;

-- Node name is ':49' = 'y2'
-- Equation name is 'y2', location is LC114, type is buried.
y2       = DFFE( _EQ002 $  GND, GLOBAL( Clock), GLOBAL( Resetn),
  _EQ002 =  w & !y1 &  y2
         #  w &  y1 & !y2;

-- Node name is 'z'
-- Equation name is 'z', location is LC115, type is output.
 z       = LCELL( _EQ003 $  GND);
  _EQ003 = !y1 &  y2;
```

Figure D.28 Logic expressions for the code in Figure D.27.

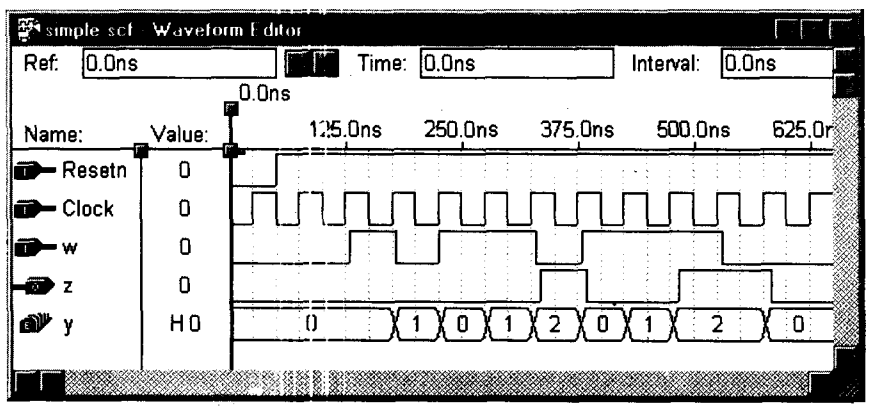
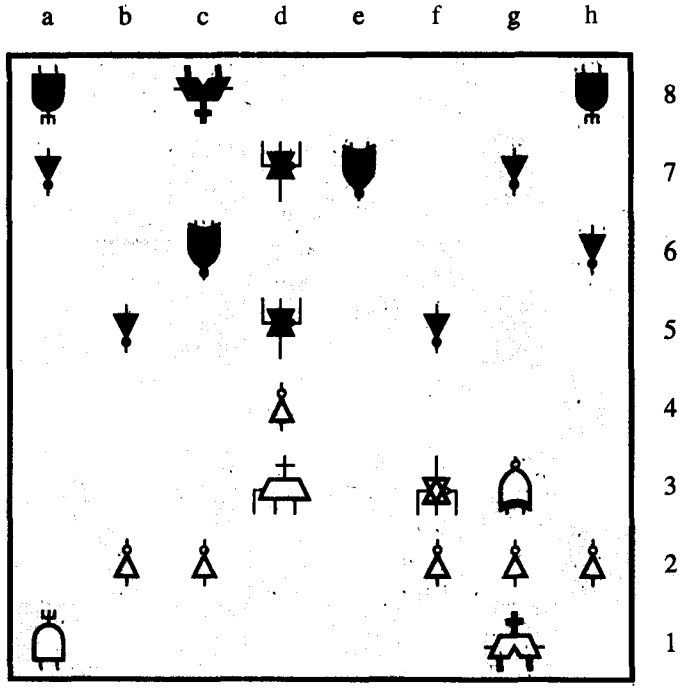


Figure D.29 Timing simulation for the code in Figure D.27.

D.5 CONCLUDING REMARKS

In Tutorials 1, 2, and 3, we have introduced many of the most important features of MAX+plusII. However, many other features are available. The reader is encouraged to learn about the more advanced capabilities of the CAD system by exploring the various commands and on-line help provided in each application.

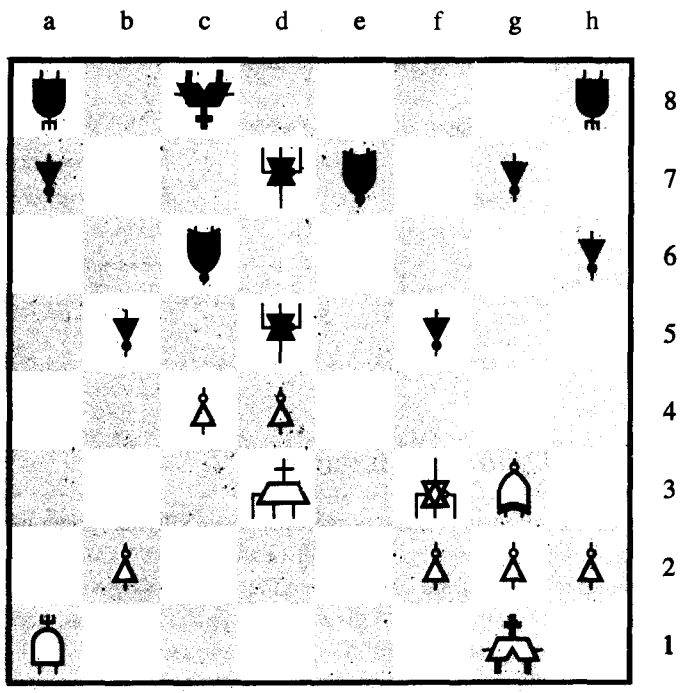


18. Re1xe7, Bf8xe7

appendix

E

COMMERCIAL DEVICES



19. c2–c4, Black resigns

In Chapter 3 we described the three main types of programmable logic devices (PLDs): simple PLDs, complex PLDs, and field-programmable gate arrays (FPGAs). This appendix describes some examples of commercial PLD products.

E.1 SIMPLE PLDs

Simple PLDs (SPLDs) include PLAs, PALs, and other similar types of devices. The major manufacturers of SPLD products are listed in Table E.1. The first and second columns show the company name and some of the SPLD products it offers. Data sheets that describe each product can be obtained from the World Wide Web (WWW), using the locator given in the third column in the table.

Table E.1 Commercial SPLD products.

Manufacturer	SPLD Products	WWW Locator
Altera	Classic	http://www.altera.com
Atmel	PAL	http://www.atmel.com
Cypress	PAL	http://www.cypress.com
Lattice	ispGAL	http://www.latticesemi.com

E.1.1 THE 22V10 PAL DEVICE

PAL devices are among the most commonly used SPLDs. They are offered in a range of sizes and are identified by a part number of the form *NNXMM–S*. The digits *NN* specify the total number of input and output pins; the digits *MM* give the number of pins that can be used as outputs. The letter *X* gives additional information, such as whether the PAL contains flip-flops. The final digit, *S*, specifies the *speed grade*. This value represents the propagation delay from an input pin on the PAL to an output pin, assuming that the flip-flop, if present, is bypassed.

An example of a commonly used PAL is the 22V10 [1], which is depicted in Figure E.1. There are 11 input pins that feed the AND plane, and an additional input that can also serve as a clock input. The OR gates are of variable size, ranging from 8 to 16 inputs. Each output pin has a tri-state buffer, which allows the pin to optionally be used as an input pin.

We said in section 3.6.2 that the circuitry between an OR gate and an output in a PAL is usually called a macrocell. Figure E.2 shows one of the macrocells in the 22V10 PAL. It connects the OR gate shown to one input on an XOR gate, which feeds a D flip-flop.

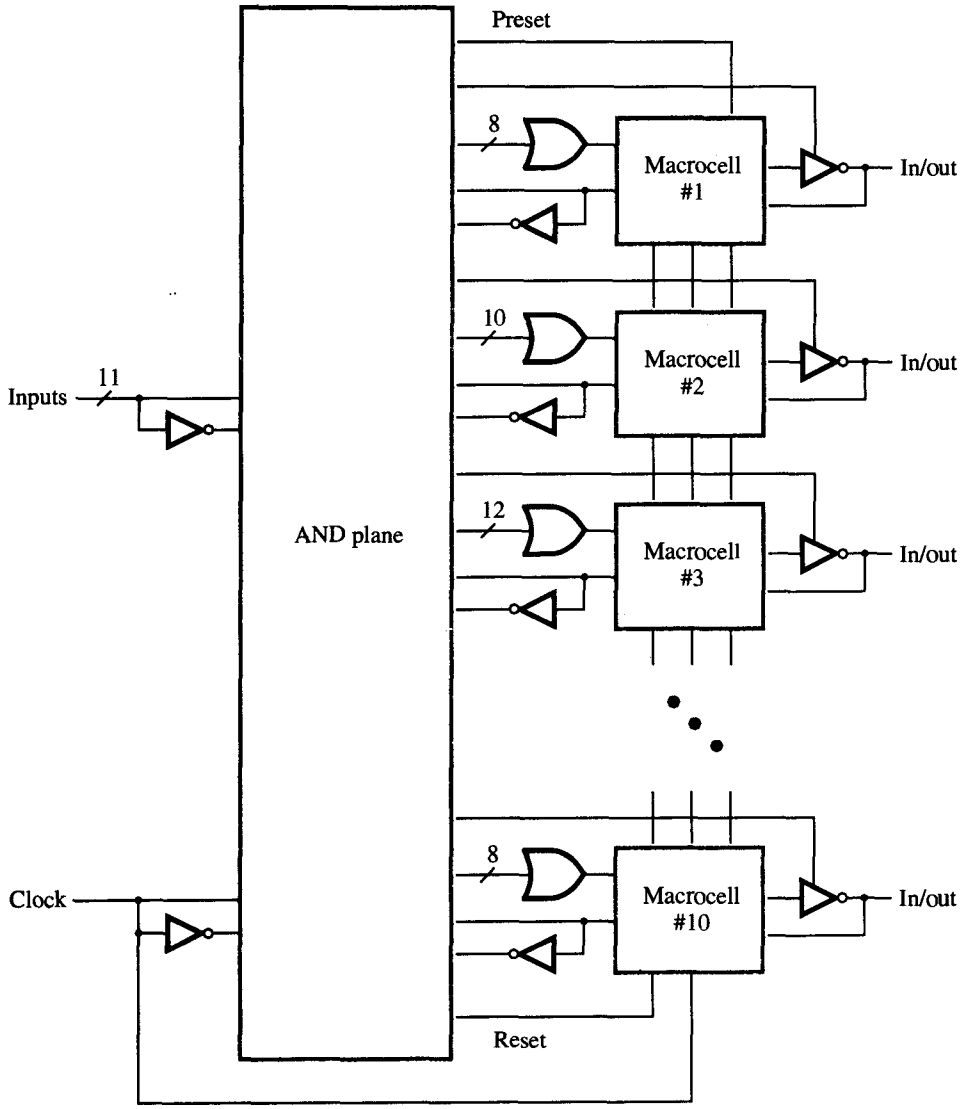


Figure E.1 The 22V10 PAL device.

Since the other input to the XOR gate can be programmed to be 0 or 1, it can be used to complement the OR-gate output. A 2-to-1 multiplexer allows bypassing of the flip-flop, and the tri-state buffer can be either permanently enabled or connected to a product term from the AND plane. Either the $\overline{Q}$ output from the flip-flop or the output of the tri-state buffer can be connected to the AND plane. If the tri-state buffer is disabled, the corresponding pin can be used as an input.

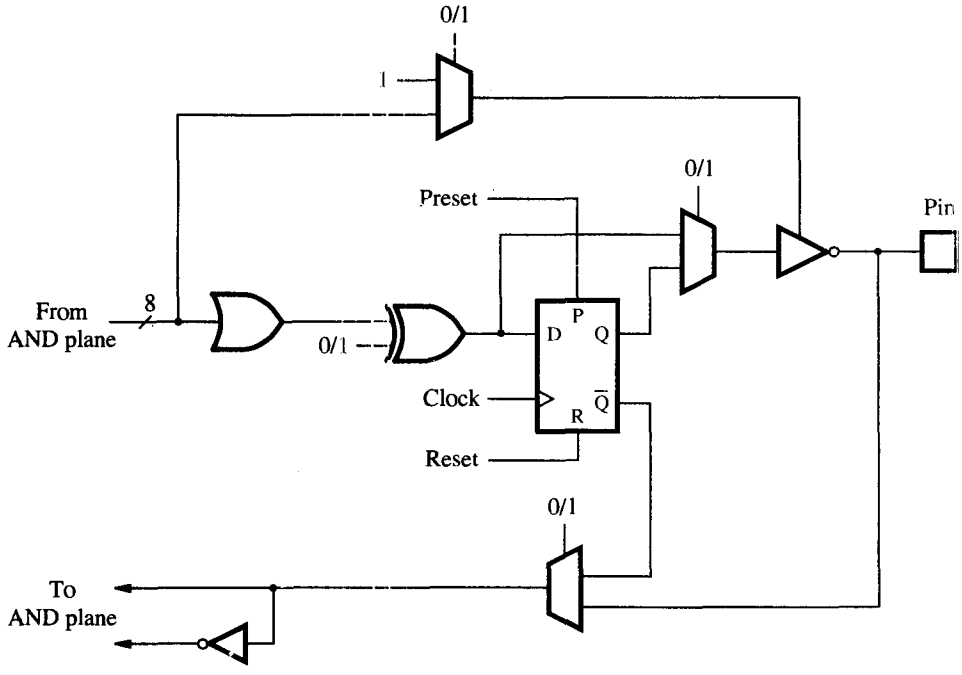


Figure E.2 The 22V10 macrocell.

E.2 COMPLEX PLDS

The names of several manufacturers of Complex PLDs (CPLDs), products they offer, and the corresponding WWW locators are listed in Table E.2. An example of a widely used CPLD family, the Altera MAX 7000 [2], is described in the next section.

Table E.2 Commercial CPLD products.

Manufacturer	CPLD Products	WWW Locator
Altera	MAX 5000, 7000 and 9000	http://www.altera.com
Atmel	ATF, ATV	http://www.atmel.com
Cypress	Delta39K, FLASH370	http://www.cypress.com
Lattice	ispLSI, ispMACH	http://www.latticesemi.com
Xilinx	XC9500, CoolRunner	http://www.xilinx.com

E.2.1 ALTERA MAX 7000

The MAX 7000 CPLD family includes chips that range in size from the 7032, which has 32 macrocells, to the 7512, which has 512 macrocells. There are two main variants of these chips, identified by the suffix *S*. If this letter is present in the chip name, as in 7128S, then the chip is in-system programmable. But if the suffix is absent, as in 7128, then the chip has to be programmed in a programming unit.

The overall structure of a MAX 7000 chip is illustrated in Figure E.3. There are four dedicated input pins; two of these can be used as global clock inputs, and one can be used as a global reset for all flip-flops. Each shaded box in the figure is called a *logic array block* (*LAB*), which contains 16 macrocells. Each LAB is connected to an *I/O control block*, which contains tri-state buffers that are connected to pins on the chip package; each of these pins can be used as an input or output pin. Each LAB is also connected to the *programmable interconnect array* (*PIA*). The PIA consists of a set of wires that span the entire device. All connections between macrocells are made using the PIA.

Figure E.4 shows the structure of a MAX 7000 macrocell. There are five product terms that can be connected through the *product term select matrix* to an OR gate. This OR gate can be configured to use only the product terms needed for the logic function being implemented in the macrocell. If more than five product terms are required, additional product terms can be "shared" from other macrocells, as described below. The OR gate is connected through an XOR gate to a flip-flop, which can be bypassed.

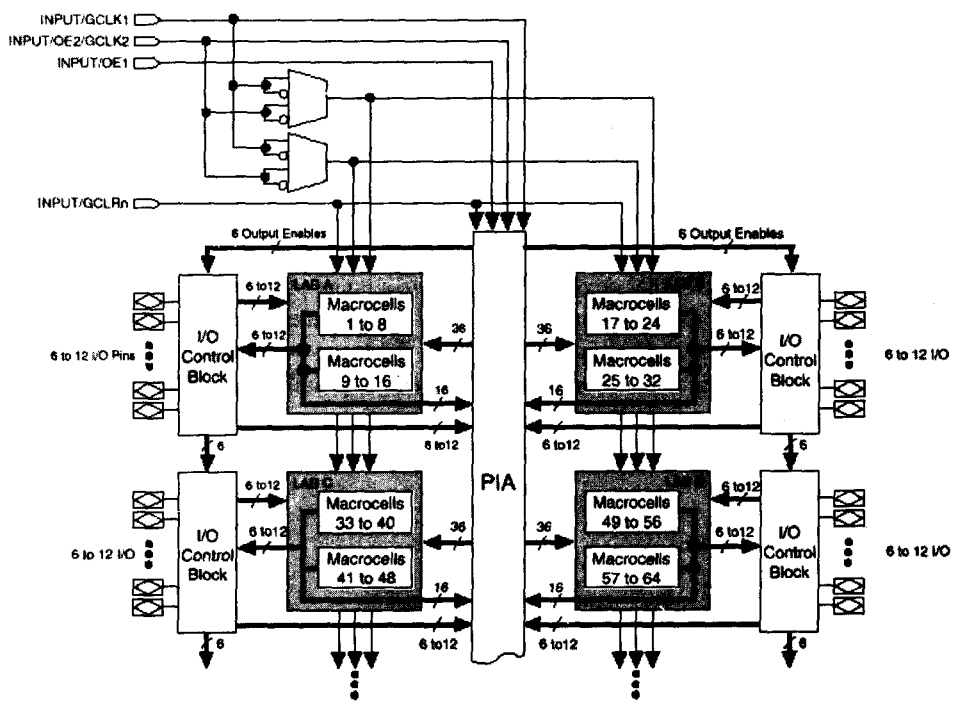


Figure E.3 MAX 7000 CPLD (courtesy of Altera).

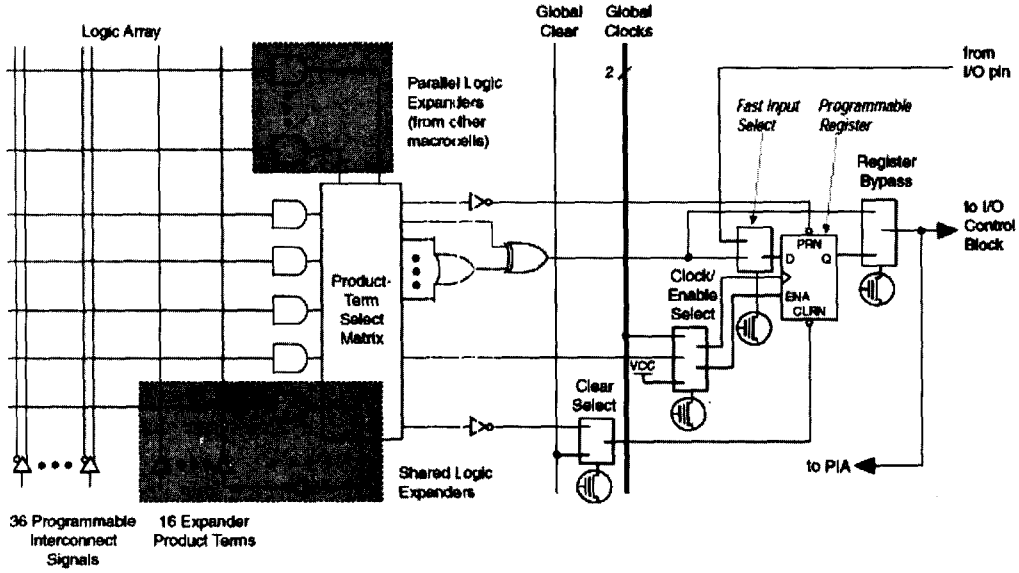


Figure E.4 MAX 7000 macrocell (courtesy of Altera).

Figure E.5 shows how product terms can be shared between macrocells. The OR gate in a macrocell includes an extra input that can be connected to the output of the OR gate in the macrocell above it. This feature is called *parallel expanders* and is used for logic functions with up to 20 product terms. If even more product terms are needed, then a feature called *shared expanders* is used. As shown in the lower shaded box in Figure E.4, one of the product terms in a macrocell is inverted and fed back to the product term array. If the inputs to this product terms are used in their complemented form, then using DeMorgan's theorem, a sum term is produced. A shared expander can be used by any macrocell in the same LAB.

Each specific MAX 7000 device is available in a range of speed grades. These grades specify the propagation delay from an input pin through the PIA and a macrocell to an output pin. For example, the chip named 7128S-7 has a propagation delay of 7.5 ns. If the logic function implemented uses parallel or shared expanders, the propagation delay is increased.

E.3 FIELD-PROGRAMMABLE GATE ARRAYS

Table E.3 lists the names of FPGA manufacturers, their products, and their WWW locators. This section describes examples of FPGAs produced by Altera and Xilinx.

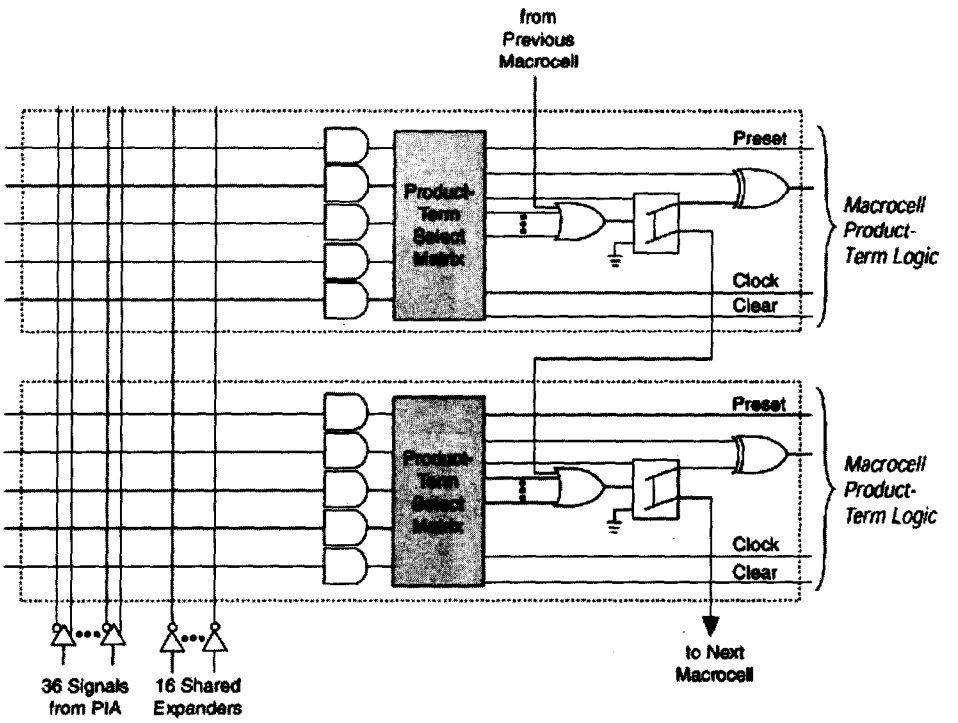


Figure E.5 Parallel Expanders (courtesy of Altera).

Table E.3 Commercial FPGA products.

Manufacturer	FPGA Products	WWW Locator
Actel	Act 1, 2 and 3, MX, SX	http://www.actel.com
Altera	FLEX 6000, 8000 and 10K, Mercury APEX 20K, APEX II, Excalibur, Stratix	http://www.altera.com
Atmel	AT6000, AT40K	http://www.atmel.com
Lattice	ORCA	http://www.latticesemi.com
QuickLogic	pASIC	http://www.quicklogic.com
Xilinx	XC3000, XC4000, Spartan Virtex, Virtex II (Pro)	http://www.xilinx.com

E.3.1 ALTERA FLEX 10K

Figure E.6 shows the structure of the FLEX 10K chip [3]. It contains a collection of logic array blocks (LABs), where each LAB comprises eight *logic elements* based on lookup tables (LUTs). In addition to LABs, the chip also contains embedded array blocks (EABs),

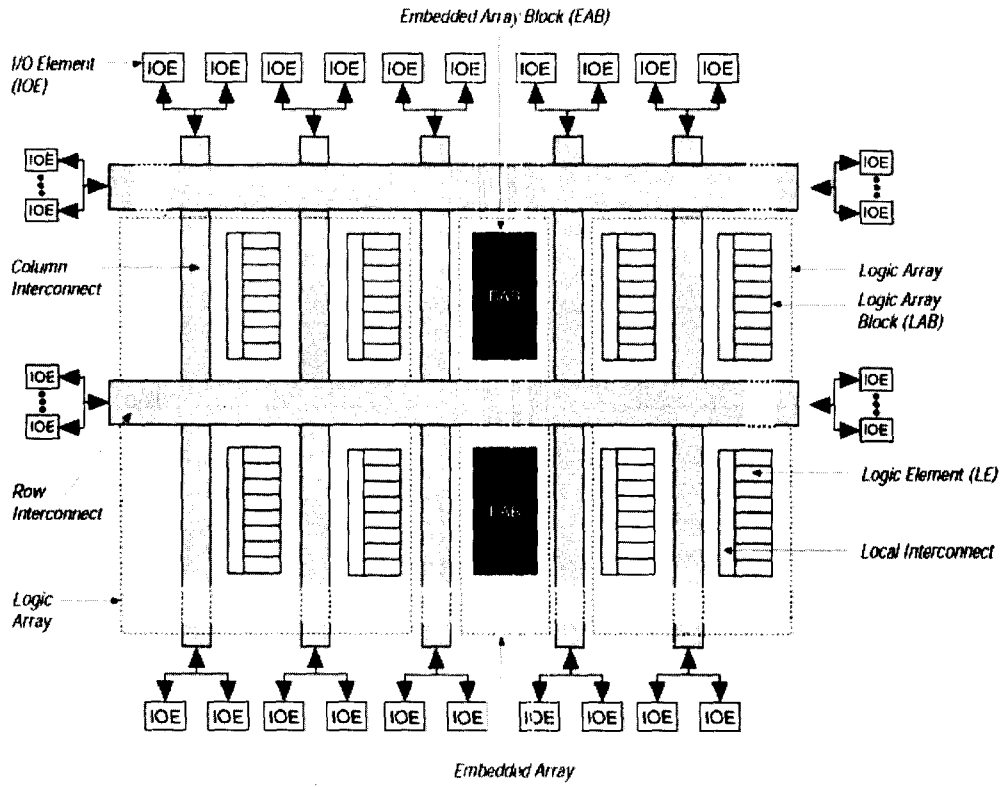


Figure E.6 FLEX 10K FPGA (courtesy of Altera).

which are SRAM blocks that can be configured to provide memory blocks of various aspect ratios (see section 10.1.3). The LABs and EABs can be interconnected using the row and column *interconnect wires*. These wires also provide connections to the input and output pins on the chip package.

Figure E.7 shows the contents of a LAB. It has a number of inputs that are provided from the adjacent row interconnect wires to a set of local interconnect wires inside the LAB. These local wires are used to make connections to the inputs of the logic elements, and the logic element outputs also feed back to the local wires. Logic element outputs also connect to the adjacent row and column wires. The structure of a logic element is depicted in Figure E.8. The element has a four-input LUT and a flip-flop that can be bypassed. For implementation of arithmetic adders, the four-input LUT can be used to implement 2 three-input functions, namely, the sum and carry functions in a full-adder.

The structure of an EAB is depicted in Figure E.9. It contains 2048 SRAM cells, which can be used to provide memory blocks that have a range of aspect ratios: 256 × 8, 512 × 4, 1024 × 2, and 2048 ×1 bits. The address and data inputs to the memory block are provided from a set of local interconnect wires. These inputs, as well as a write enable for the memory block, can optionally be stored in flip-flops. Figure E.9 shows that the number of address and data inputs connected to the memory block varies depending on the aspect

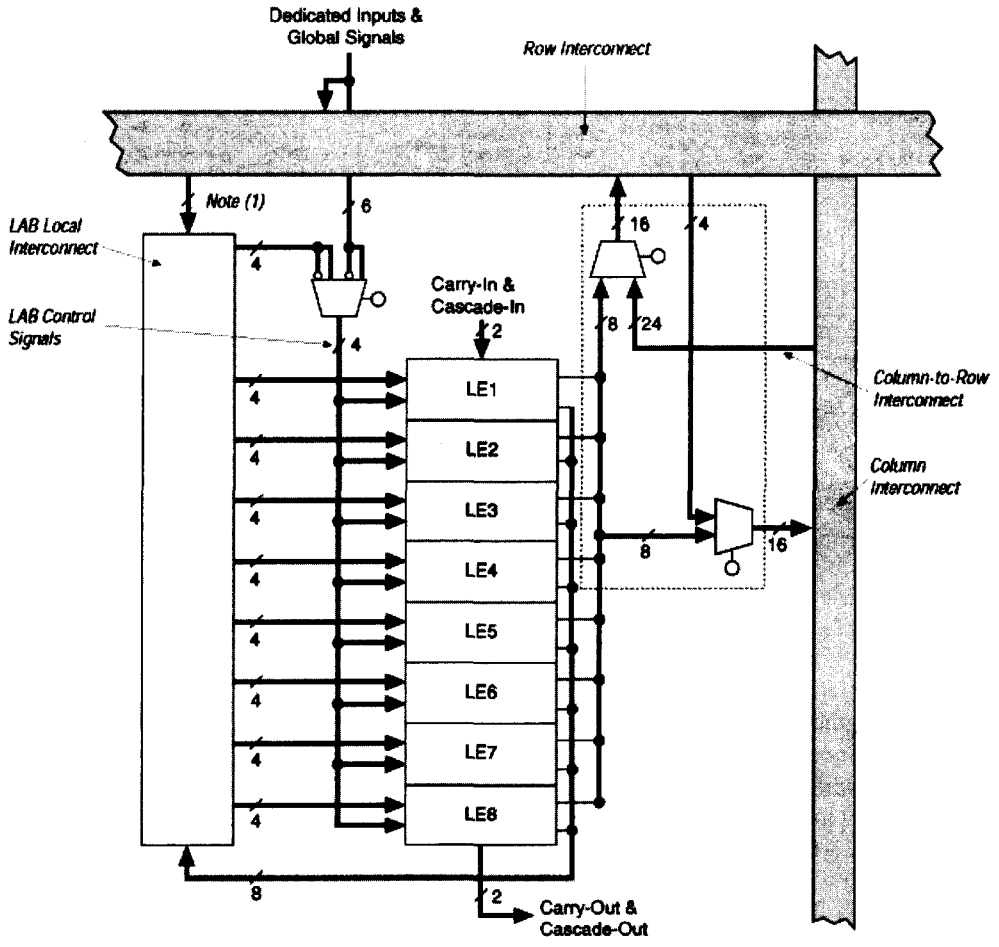


Figure E.7 FLEX 10K logic array block (courtesy of Altera).

ratio being used. The data outputs can also optionally be stored in flip-flops. For large memory blocks it is possible to combine multiple EABs.

Configuration of EABs is done using predesigned modules, such as those in the LPM library. For example, the module named *lpm_ram_dq* can be used to specify an SRAM block, and *lpm_rom* can be used for a ROM block. These modules can be imported into a schematic or instantiated in code using a language such as Verilog. It is possible to specify initial data to be loaded into the memory block when the FPGA chip is programmed. This is done by creating a special type of file, called a *memory initialization file*, that is associated with the *lpm_ram_dq* or *lpm_rom* module. Complete details on using these modules can be found in the MAX+plusII documentation.

FLEX 10K chips are available in sizes ranging from the 10K10 to 10K250, which offer about 10,000 and 250,000 equivalent logic gates, respectively. Specific chips are available in various speeds, indicated using a suffix letter, such as *A*, as in 10K10A, and a speed

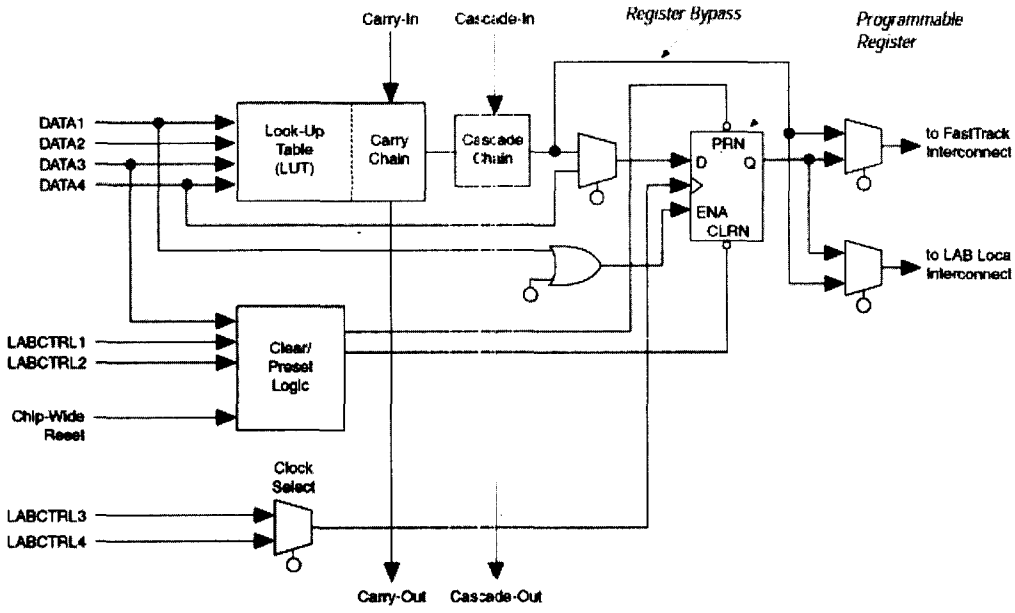


Figure E.8 FLEX 10K logic element (courtesy of Altera).

grade, as in 10K10A-1. Unlike PALs and CPLDs, the speed grade for an FPGA does not specify an actual propagation delay in nanoseconds. Instead, it represents a relative speed within the device family. For instance, the 10K10-1 is a faster chip than the 10K10-2. The actual propagation delays in implemented circuits can be examined using a timing simulator CAD tool.

E.3.2 XILINX XC4000

The structure of a Xilinx XC4000 chip [4] is similar to the FPGA structure shown in Figure 3.35. It has a two-dimensional array of *configurable logic blocks* (*CLBs*) that can be interconnected using the vertical and horizontal routing channels. Chips range in size from the XC4002 to XC40250, which have about 2000 and 250,000 equivalent logic gates, respectively. As shown in Figure E.10, a CLB contains 2 four-input LUTs; hence it can implement any two logic functions of up to four variables. The output of each of these LUTs can optionally be stored in a flip-flop. The CLB also contains a three-input LUT connected to the 2 four-input LUTs, which allows implementation of functions with five or more variables.

Similar to the logic elements in the FLEX 10K FPGAs described in section E.3.1, the CLB can be configured for efficient implementation of adder modules. In this mode each four-input LUT in the CLB implements both the sum and carry functions of a full-adder. Also, instead of implementing logic functions, the CLB can be used as a memory module. Each four-input LUT can serve as a 16 × 1 memory block, or both four-LUTs can be combined into a 32 × 1 memory block. Multiple CLBs can be combined to form larger memory blocks.

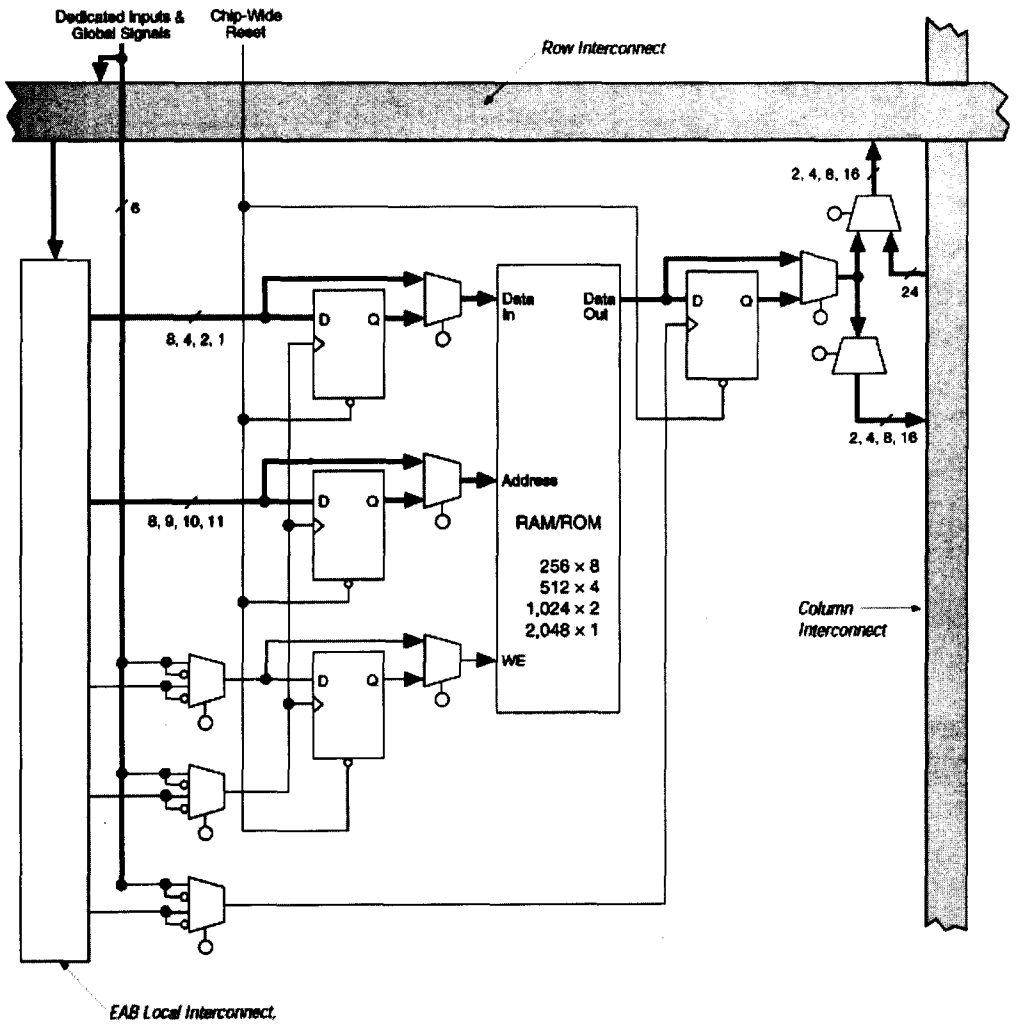


Figure E.9 Embedded array block (courtesy of Altera).

The CLBs are interconnected using the wires in the routing channels. Wires of various lengths are provided, from wires that span a single CLB to wires that span the entire device. The number of wires in a routing channel varies for each specific chip.

E.3.3 ALTERA APEX 20K

The Altera APEX 20K [5] family is the next generation product following the FLEX 10K. The logic element (LE), which is an optimized version of the one depicted in Figure E.8, contains a four-input LUT and a flip-flop. Chips range in sizes from 1200 to 51,840 LEs.

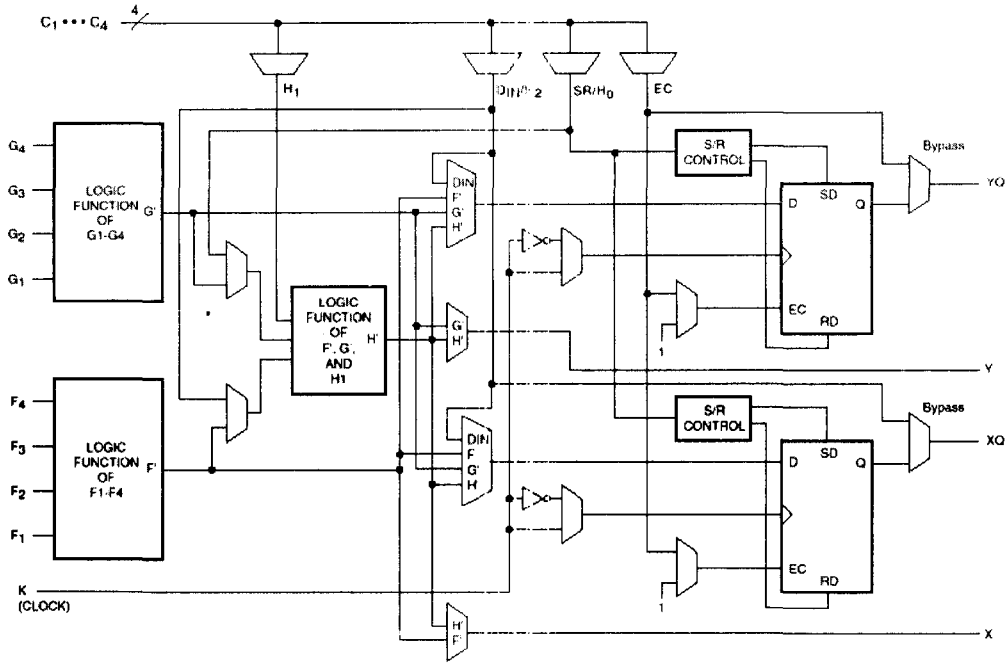


Figure E.10 XC4000 configurable logic block (courtesy of Xilinx).

Each APEX device contains logic elements (LUTs), memory blocks, and IO cells. The LEs are arranged into LABs similar, to the structure depicted in Figure E.7, with ten LEs per LAB. The LABs are further grouped into MegaLABs, with up to 24 LABs in a MegaLAB. As shown in Figure E.11, the MegaLAB contains wires to interconect the LABs, and it also contains a memory block, called the embedded system block (ESB). Similar to the EAB shown in Figure E.9, the ESB supports memory blocks with various aspect ratios. An APEX device comprises either two or four columns of MegaLABs; the number of MegaLABs per column varies for each device.

E.3.4 ALTERA STRATIX

Stratix [6] is Altera's FPGA product that supercedes the APEX family. Figure E.12 shows the architecture of a Stratix device. Each chip comprises columns of resources of various types. The LAB columns house logic elements arranged into LABs that have ten LEs per LAB. Each LE contains a four-input LUT and a register, and can be configured in a variety of modes, including a fast arithmetic mode. There are a number of types of wiring resources in a Stratix chip. Connections within a LAB are made using fast local resources, such as a carry chain that runs downward in each column. For connections from one LAB to other resources there exist short nearest-neighbour connections, wires that span four columns or rows, and longer wires.

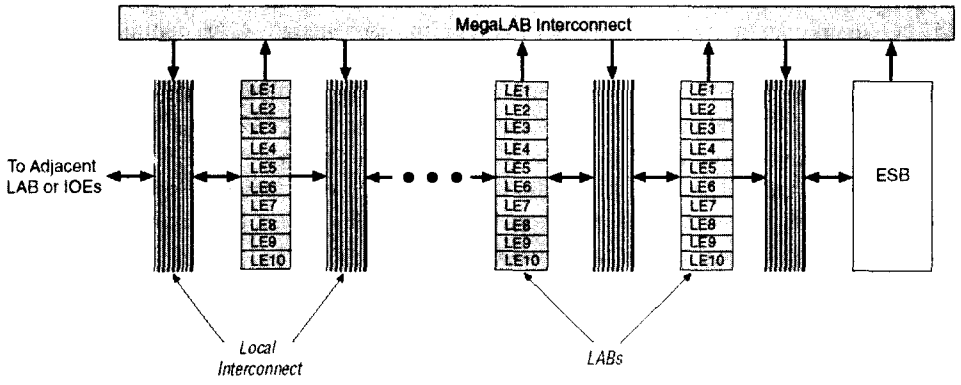


Figure E.11 APEX 20K MegaLAB (courtesy of Altera).

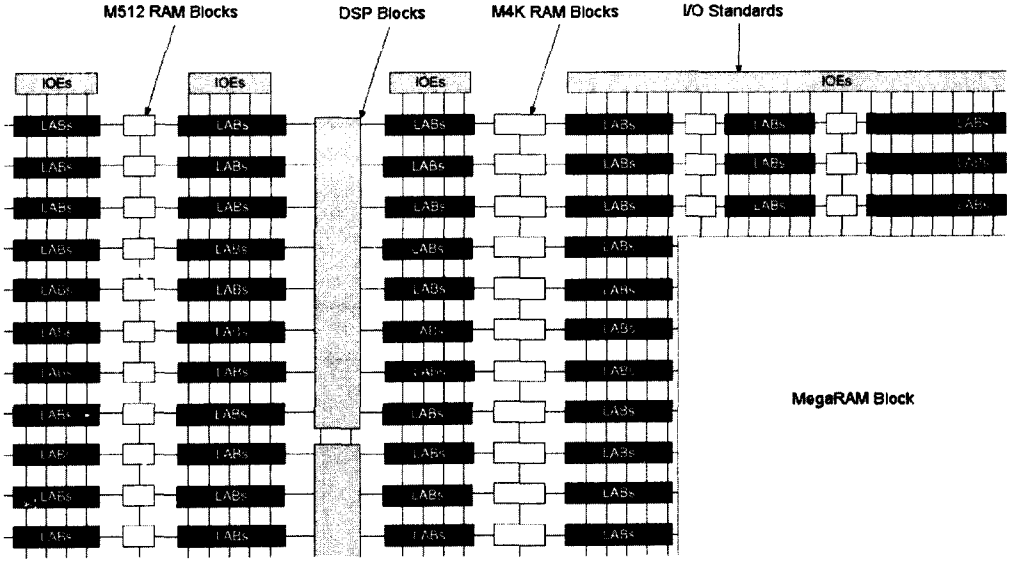


Figure E.12 Stratix LAB, DSP, and memory blocks (courtesy of Altera).

In addition to LAB columns, Stratix devices contain three other types of columns. The M512 columns consist of memory blocks with 512 bits each, and the M4K columns contain larger memory blocks with 4K bits per block. Each of the M512 and M4K blocks support implementations of memories with various aspect ratios. Stratix devices also include very large memory blocks called MegaRAMs, each of which contains 512K bits of memory.

Finally, there are columns that comprise Digital Signal Processing (DSP) blocks. Each of these blocks includes hardware multiplier and adder circuits that allow fast multiplication and accumulation (summing) of data. These blocks provide efficient implementation of the types of circuits used in digital signal processing applications.

E.3.5 XILINX VIRTEX

The Xilinx Virtex [7] FPGAs are the next generation family following the XC4000. As indicated in Figure E.13, each Virtex chip comprises logic resources called CLBs, and memory resources called Block RAMs (BRAMs). The CLB is an enhanced version of the XC4000 CLB shown in Figure E.10. As indicated in Figure E.14, the Virtex CLB is divided into two halves; each half is called a *slice*. Each slice contains two four-input LUTs, two registers, and dedicated arithmetic (carry chain) logic.

The BRAM blocks contain 4K bits of memory, and can be configured to support aspect ratios from 4096 × 1 to 256 × 16. The CLB and BRAM blocks can be interconnected by wires that span a single CLB, or longer distances. Virtex devices are available in sizes from 256 to 46,592 CLB slices.

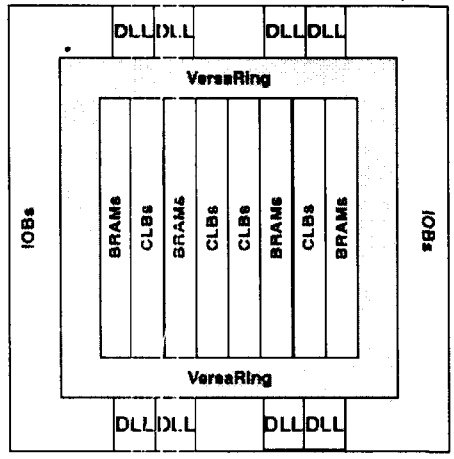


Figure E.13 Virtex FPGA (courtesy of Xilinx).

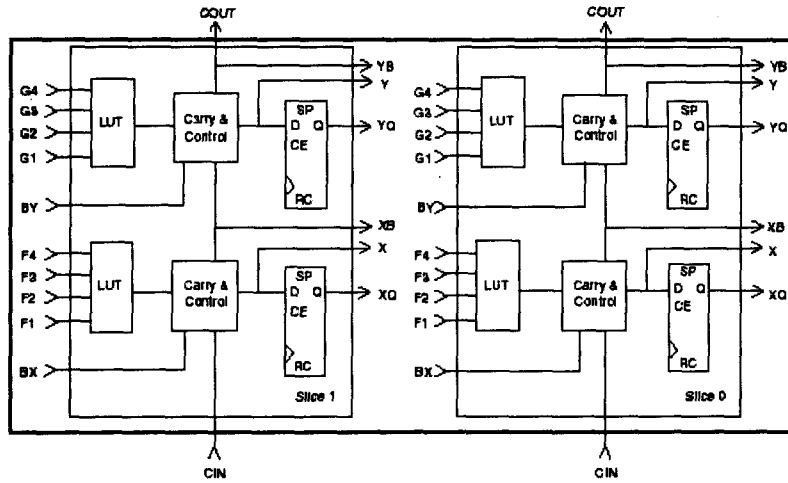


Figure E.14 Virtex logic block (courtesy of Xilinx).

E.4 TRANSISTOR-TRANSISTOR LOGIC

Before the emergence of CMOS, the dominant technology was *transistor-transistor logic*, commonly referred to as *TTL*. Most digital systems built in the 1970s and 1980s were based on this technology. TTL circuits are available in relatively small sizes, known as small-scale integration (SSI) and medium-scale integration (MSI), as explained in section 3.5. A typical SSI chip contains just a few logic gates, with their inputs and outputs available on the pins of the package. An MSI chip may comprise a somewhat larger circuit, such as a four-bit arithmetic and logic unit (ALU).

TTL technology is not as suitable for large-scale integration as CMOS technology, which has led to TTL's demise. However, its impact was so large that some aspects are still important today. In this section we consider these aspects.

Voltage Levels

TTL circuits use a 5-volt power supply. Any voltage in the range 0 to 0.8 V is interpreted as a logic 0 when applied to an input pin. A voltage in the range 2 to 5 volts is interpreted as a logic 1. Using the terminology from section 3.8, $V_{IL} = 0.8$ V and $V_{IH} = 2$ V. The maximum output voltage produced for logic 0 is $V_{OL} = 0.4$ V, and the minimum voltage produced for logic 1 is $V_{OH} = 2.4$ V. These parameters lead to the noise margins $NM_L = NM_H = 0.4$ V. Typical output voltages generated by a TTL circuit are 0.2 V for logic 0 and 3.6 V for logic 1.

When a new digital circuit is designed, it is often intended for use in an existing digital system. If different technologies are used to implement different parts of a system, it is essential to ensure that compatible voltage levels are used for signals in the interfaces

between the different parts. While CMOS voltage levels are normally different from TTL levels, some CMOS chips, such as PLDs, can be configured to use TTL-compatible voltage levels on their input and output pins.

Input Connections

In CMOS circuits all inputs to a gate must always be driven to either logic value 0 or 1. Otherwise, the gate's output will have an unknown (usually tri-state) value. In the case of TTL circuits, an unconnected input behaves as if it were connected to a constant 1.

E.4.1 TTL Circuit Families

TTL circuits are available in several designs that have different propagation speeds and power consumption. They have the same functional characteristics, defined by the specifications for the type of circuits known as the 7400 series, which is introduced in section 3.5. Actually, the 7400 label denotes a chip that comprises 4 two-input NAND gates. Other chips that contain different logic elements have the same prefix 74, but are identified by additional digits. For example, 7421 denotes a chip that comprises 2 four-input AND gates. Table E.4 presents the propagation delay and power dissipation characteristics of the various TTL families.

Table E.4 TTL logic families.

Name	Designation	Propagation Delay (ns)	Power Dissipation (mW)
Standard	7400	9	10
Low power	74L00	33	1
High speed	74H00	6	22
Schottky	74S00	3	20
Low-power Schottky	74LS00	9	2
Advanced Schottky	74AS00	1.5	20
Advanced low-power Schottky	74ALS00	4	1
Fast	74F00	3	4

Standard TTL is based on the original specifications, and it was the first type of such circuits introduced in the 1960s. Subsequent versions provided various improvements. Faster circuits were developed, trading off increased power consumption for shorter propagation delays. Conversely, low-power circuits were developed, at the cost of longer propagation delays. Table E.4 gives the typical values that can be expected under normal operating conditions.

The maximum fan-out in TTL circuits is 10 in most cases, but it can be as high as 20 for the low-power types. The fan-in is determined by the number of inputs provided on a given chip.

TTL gates can have different output configurations. In addition to the normal output configuration, there exist gates that have tri-state outputs or open-collector outputs. The purpose of a tri-state output is discussed in section 3.8.8. Gates with open-collector outputs are used when it is desirable to connect the outputs of two or more gates together directly. These gates are not damaged by such a connection, because each gate either drives the output to 0 or does not affect it at all. Connecting the outputs of several open-collector gates through a pull-up resistor to +5 V results in a circuit where the voltage at the output point is equal to +5 V if none of the gates produces an output of 0 and is equal to 0 if one or more gates produce the output of 0. A similar approach can be used with CMOS technology, resulting in open-drain gates.

We have not pursued TTL technology in any detail because of its diminished importance in today's design environment. An interested reader may consult numerous books that provide a detailed explanation. A particularly thorough reference is [8].

REFERENCES

1. Lattice Semiconductor, Simple PLDs Data Sheets, http://www.latticesemi.com
2. Altera Corporation, MAX 7000 CPLD Data Sheets, http://www.altera.com
3. Altera Corporation, FLEX 10K Data Sheets, http://www.altera.com
4. Xilinx Corporation, XC4000 FPGA Data Sheets, http://www.xilinx.com
5. Altera Corporation, APEX 20K Data Sheets, http://www.altera.com
6. Altera Corporation, Stratix FPGA Data Sheets, http://www.altera.com
7. Xilinx Corporation, Virtex FPGA Data Sheets, http://www.xilinx.com
8. A. S. Sedra and K. C. Smith, *Microelectronic Circuits*, 4th ed. (Oxford University Press: New York, 1998).

INDEX

A

Absorption property, 28
Accumulator, 398, 736
Actel Corporation, 344
  Act 1 block, 344
Active clock edge, 446
Active-low signal, 124
Adder:
  BCD, 287
  carry lookahead, 255–262
  full-adder, 237, 275, 705
  half-adder, 234
  in Verilog code, 265–275, 710, 716
  propagation delay, 254, 259
  ripple-carry, 239
  serial, 477
Adder/subtractor, 248, 626
Addition, 234–239, 244–246
  BCD, 285
  carry, 234
  generate function, 255
  overflow, 253
  propagate function, 255
  sum, 234
  Verilog, 271–275
Address, 316, 610
Aliasing problem in testing, 685
Algorithm, 612
Algorithmic state machine (ASM):
  ASM charts, 516–519
  ASM block, 519
  conditional output box, 517
  decision box, 516
  implied timing, 613
  state box, 516
Alphanumeric characters, 289
Altera Corporation, 13
Altera APEX 20K, 5, 831
Altera FLEX 10K, 827
Altera Hardware Description Language (AHDL), 766, 803
Altera MAX 7000 CPLD, 825
Altera Stratix FPGA, 832
Altera UP-1 board, 789
Alternative (in Verilog **case** statement), 715
always block (Verilog), 59, 389, 711, 743
Analog circuit, 18
Analysis, 24, 184, 512, 531
AND gate (*see* Gates)
AND operator (Verilog), 704
AND plane (also AND array), 88, 133–135
Anode terminal, 429
Arbiter circuit, 505, 547
Arithmetic:
  floating-point (*see* Floating point)
  overflow, 253
  (*See also* Addition; Division; Multiplication; Subtraction)
Arithmetic and logic unit (ALU), 330
Arithmetic assignment (Verilog), 271, 704, 709
Array multiplier (*see* Multiplication)
ASCII code, 289
ASIC, 6, 104
ASM block, 519
ASM chart (*see* Algorithmic state machine)
Aspect ratio, 610
assign (Verilog), 709
Assign menu, 780
Assignment & Configuration File (.acf), 775, 791
Associative property, 28
Asynchronous clear (reset), 364, 381
Asynchronous clear (in Verilog), 395
Asynchronous counter, 373
Asynchronous inputs, 655
Asynchronous sequential circuit (*see* Sequential circuits)
Axioms of Boolean algebra, 27

B

Back-annotate project, 794
Barrel shifter, 623
Basic latch, 351, 368, 528
BCD (*see* Binary-coded decimal)
BCD-to-7-segment decoder, 319, 329
begin (Verilog), 712
begin-end block (Verilog), 712
Behavioral Verilog code, 58, 433, 712
BGA package, 100
BILBO (Built-in Logic Block Observer), 685
Binary-coded decimal (BCD), 284–289
  addition, 285
  counter, 382
  digits, 285
Binary decoder (*see* Decoder)
Binary encoder (*see* Encoder)
Binary numbers, 231
  in Verilog code, 275
Binary variable, 18
BIST (Built-in Self Test), 681–687
Bit, 231
Bit-counting circuit, 612
Bit-select (Verilog), 701
Bitwise operators (Verilog), 333, 704
Blocking assignment (Verilog), 390, 440, 712, 721, 734, 743
Body effect, 119
Boolean algebra, 27–34
Boundary scan, 688
Branching heuristic, 199, 206
Buffer, 85, 122, 431
  tri-state, 85, 124, 406
Bus, 406, 650
Bus wires (MAX+plusII), 800, 811
Bypass capacitor, 689
Byte, 231
ByteBlaster, 789

C

CAD (*see* Computer aided design)
Canonical expressions:
  canonical product-of-sums, 41
  canonical sum-of-products, 38
Capacitance, 124
Carry, 234
  carry-in, 234
  carry-out, 236
Carry chain, 377
Carry lookahead adder, 255–262
Cascade chain, 830
case statement, 326, 714, 745
casex statement, 331, 717
casez statement, 331, 717
Cathode terminal, 429

Channel (in MOSFET), 108
Character codes, 289
Characteristic impedance, 690
Chip configuration, 210
Clear input (*see* Reset input)
Clock, 355
Clock divider, 429
Clock enable, 606, 733
Clock network, 653
Clock skew, 408, 470, 653
Clock synchronization, 657
Clock-to-output time (t_{co}), 388, 655
CMOS technology, 74–81, 117–120
Code:
  BCD (*see* Binary-coded decimal)
  binary, 231–234
  converter, 318
  decimal, 230
  error-detecting, 289
  Gray, 154
Cofactor, 307
Coincidence operation, 238
Column dominance, 197
Combinational circuits, 297–343
Combining property, 28,
Comment (Verilog), 57
Commutative property, 28
Comparator, 320, 337
Compatible states, 557
Compiler (MAX+plusII), 758
Complement:
  diminished radix, 252
  of a logic variable, 21
  1's, 242
  radix, 249
  2's, 243
Complementary metal-oxide semiconductor (*see* CMOS technology)
Completely specified FSM, 493
Complex gate (CMOS), 79
Complex programmable logic device (CPLD), 13, 94–98, 129–135, 824
Compressor circuit, 683
Computer-aided design (CAD), 2, 48–54, 209–215
  chip configuration, 210
  design entry, 48, 210
  functional simulation, 52, 210
  initial synthesis, 210
  layout synthesis (physical design), 52, 211
  logic synthesis (optimization), 51, 210
  technology mapping, 52, 209
  timing simulation, 52, 213
Concatenation (Verilog), 272, 338, 704
Concurrent statement, 708, 712, 743
Conditional operator (Verilog), 321, 704
Configurable logic block (CLB), 830
Consistency check, 671
Constant (in Verilog), 275
Context sensitive help, 751
Continuous assignment (Verilog), 58, 709
Control circuit, 606, 615
Controlling expression (Verilog), 715
Cost, 38, 160
Counter:
  asynchronous, 373
  asynchronous circuit design, 544
  BCD, 382, 431
  down, 373, 404
  enable and clear capability, 375
  Johnson, 384, 437
  modulo-n, 379
  parallel load of, 378, 403
  reset of, 378, 380
  ring, 383, 437
  ripple, 373
  synchronous, 495, 374, 376
  up, 372, 402
  up/down, 373, 405
  Verilog code, 372, 735
Cover, 160
  fault, 669
  minimal, 161, 195, 206
  table, 195
Critical path, 255
Crossbar, 301
Cross-coupled gates, 351
Crosstalk, 689
Cubical representation, 189–193
Current flow:
  dynamic, 117
  I_{OL}, 429
  short circuit, 117
  static, 111
Custom chips, 6, 103
Cut-off region, 108
Cut set, 540

D

D flip-flop (*see* Flip-flop)
D-algorithm, 673
Data, 316
Datapath, 606
DC-set, 206
Debouncing, 657
Debugging, 763, 768
Decimal numbers, 230
Decoder, 311, 327, 333
  tree, 311
Decomposition (*see* Functional decomposition)
default case alternative (Verilog), 715
Default value (Verilog), 326
defparam (Verilog), 399, 414, 724, 743, 745
Delay, 115, 116, 255
Delay (in Verilog), 710
DeMorgan's theorem, 28, 42, 165
Demultiplexer, 314
Design entry, 46, 210
Design for testability, 677–681
Design process, 6–13, 210
Development process, 7
Digital circuit, 18
Digital system, 606
Diminished radix complement, 252
DIP package, 85
Directory, 748
Disjoint decomposition, 179
Distributive property, 28, 32
Division, 282, 623
Don't-care condition, 166
  in Verilog code, 276, 329, 700
Double precision (*see* Floating point)
Down-counter, 373, 404
Download, 789
Drain (in MOSFET transistor), 69
Dual networks (CMOS), 75
Duality, 28, 75
Duty cycle, 662
Dynamic hazard, 585, 590

E

Edge (in signals), 359
Edge-triggered, 358, 362
Electrically-erasable programmable read-only memory (EEPROM), 131
Embedded Array Blocks (EABs), 827
Enable input, 311, 353, 482, 606, 733
Encoder, 316, 414
  binary, 316
  priority, 317, 332
end (Verilog), 712
Enter Nodes from SNF, 760
Equivalence:
  of logic expressions, 26
  of states, 486
Equivalent-gates metric, 98
Erasable programmable read-only memory (EPROM), 132
Errors in Verilog code, 742
Escaped identifier, 700
Espresso, 208
Essential prime implicant, 161, 195, 204

Event control (Verilog), 712
Event expression (Verilog), 712
Excess-127 format, 283
Excess-1023 format, 284
Excitation table, 500, 531
Exclusive-NOR (XNOR) gate (*see* Gates)
Exclusive-OR (XOR) gate (*see* Gates)
Expansion theorem (Shannon's), 304

F

Factoring, 172–175
Fall time, 115
Fan-in, 120, 172
Fan-out, 122, 837
Fault:
  detection, 668
  model, 666
  propagation, 671
  stuck-at, 666
Feedback, 351
Field-programmable gate array (FPGA), 5, 13, 98–103, 135, 826
Finite state machine (FSM), 446
  incompletely specified, 493
  summary of design procedure, 454
555 programmable timer chip, 662
Fixed-point numbers, 282
Flip-flop, 359, 368
Flip-flops:
  clear and preset inputs, 362
  configurable (in PLDs), 367
  D, 359, 361, 389
  edge-triggered, 361
  JK, 498, 367
  master-slave, 359, 532
  SR, 435
  T, 364
  Verilog code for, 389, 730
Floating gate, 131
Floating point, 282
  double precision, 284
  exponent, 283
  format, 283
  IEEE standard, 283
  mantissa, 283
  normalized, 283
  representation, 283
  single precision, 283
Floorplan Editor, 212
Flowchart, 613
Flow table, 531
  primitive, 555
  state reduction, 553–568
Folder, 748
for loop statement, 269, 331, 717
forever loop statement, 717, 720
Fowler-Nordheim tunneling, 131
FPLA (*see* PLA)
FSM (*see* Finite state machine)
Full-adder, 237, 275, 705
function (Verilog), 727
Functional decomposition, 175–181
Functional equivalence, 26
Functional simulation, 12, 52, 210
Fundamental mode, 523
Fuse map, 93

G

Gate (in MOSFET transistor), 69
Gate array, 105, 145
Gate delay (*see* Propagation delay)
Gated D latch, 356, 358, 386, 388, 532, 730
Gated latch, 355, 368
Gated SR latch, 353, 368
Gates, 23
  AND, 24, 76, 79, 120
  NAND, 42, 73, 78
  NOR, 42, 74, 79
  NOT, 24, 72, 78, 111
  OR, 24, 76
  XNOR, 238
  XOR, 127, 180, 235
Generate construct (Verilog), 238, 726
genvar (Verilog), 726
Glitch, 538, 584
Global signals, 654
Graphic Design File (.gdf), 754
Graphic Editor, 50
Gray code, 154
Group (MAX+plusII), 300

H

H tree, 654
Half-adder, 234, 704
Hamming distance, 568
Handshake signaling, 547
Hardware, 2
Hardware description language (HDL), 50
Hazard-free design, 587
Hazards, 532, 584–592
  dynamic, 585, 590
  static, 386, 585
Help (MAX+plusII), 751
Heuristic approach, 162
Hexadecimal numbers, 232
Hierarchical design, 49, 258
Hierarchical Verilog code, 723
Hierarchy Display, 775
High-level behavioral Verilog code, 433
High-impedance output, 124
Hold time, 358, 655
Huntington's postulates, 28
Hypercube, 193

I

Identifier, 699
if-else statement, 59, 323, 713
IEEE, 50
IEEE standards (*see* Standards)
Implicant, 159
Implied memory (Verilog), 388, 715, 744
Incompletely specified FSM, 493
Incompletely specified functions, 166
Infer (Verilog), 709
initial block (Verilog), 711
Initial synthesis, 210
Input variable, 19
Instantiation (of Verilog gates), 706, 745
Instantiation (of Verilog modules), 267, 721, 743, 745
Instrumentation, 691
In-system programming (ISP), 94
Integer:
  in Verilog, 702
  signed, 240
  unsigned, 230
Integrated circuit (IC), 2
Intersection, 32
Inversion, 21
Inverter, 73, 111, 112

J

JK flip-flop, 498, 367
Johnson counter, 384
JTAG port, 97

K

Karnaugh map, 150–158
k-cube, 193
k-successor, 486
Keyboard short-cuts, 750
Keywords (Verilog), 699

L

Label (in Verilog code), 718, 744
Large scale integration (LSI), 86
Latch:
  basic SR, 368, 528
  D, 368
  gated D, 368, 386, 532
  gated SR, 368
  set-dominant SR, 435
  Verilog code, 388, 730

Layout synthesis, 52, 211
Leakage current, 112
Least-significant bit, 231
LED (Light emitting diode), 117, 429
Level-sensitive element, 358, 362
Level-sensitive scan design, 681
Libraries, 49
Library of Parameterized Modules (LPM), 263
lpm_add_sub, 263, 725, 803
lpm_counter, 437
lpm_ff, 396, 732
lpm_ram_dq, 637, 829
lpm_rom, 829
lpm_shiftreg, 399
Line inductance, 689
Linear feedback shift register (LFSR), 440, 682
Linear region (*see* Triode region)
Literal, 159
Logic analyzer, 691
Logic array block (LAB), 828
Logic circuit, 23
Logic expression, 19
Logic functions, 19
AND, 19, 32
minimization, 158–165, 193–208
NAND, 42
NOR, 42
NOT, 20
OR, 20, 32
synthesis, 35–41, 171–184, 287
XNOR, 238
XOR, 127
Logic gates:
drive capability, 122, 837
dynamic operation, 122
fall time, 115
fan-in, 120
fan-out, 122, 837
noise margin, 113
power dissipation, 117
propagation delay, 115
rise time, 114
transfer characteristic, 112
Verilog gates, 706, 745
Logic network, 23
Logic values, 18–19, 68
Logical operators (Verilog), 704
Logical product (AND), 34
Logical sum (OR), 34
Lookup table, 100
Loop statement (*see* **for** loop)

M

Macrocell, 91, 95, 822
Macrofunction, 262
Magnitude, 240
Majority function, 221, 305
Master (*see* Flip-flop, master-slave)
Master-slave (*see* Flip-flop)
MAX+plusII design tools, 13, 745
Maxterm, 40
Mealy FSM, 446, 462
Verilog code, 475, 739
Mealy output, 517
Mean operation, 631
Medium-scale integration (MSI), 86
Megafunction, 262
MegaWizard Plug-in Manager, 806
Memory, 135, 828
implied memory (Verilog), 388, 744
in Verilog, 703, 745
memory element, 350
Memory initialization file, 829
Merger diagram, 557
Merging, 555
procedure, 557
Metal-oxide semiconductor (*see* MOSFET)
Metastability, 657
Minimization:
of logic functions, 158–165, 193–208
of states, 486
Minterm, 37, 152
Mixed logic, 83
Module (Verilog), 705
Moore FSM, 446
Verilog code, 467, 737
Moore output, 516
Moore's law, 3
MOSFET transistor, 69–71, 106–110
on-resistance, 110
Most-significant bit, 231, 240
Multibit assignment (in Verilog), 709
Multilevel circuits, 171–189
Multiple-output circuits, 167–171
Multiplex (definition), 314
Multiplexer, 45, 125, 126, 128, 298–311, 322–324, 410
Multiplexer (Verilog code), 714, 716
Multiplication, 277–282
array implementation, 279
partial product, 277
sequential implementation, 278, 618
signed-operand, 279
Mutual exclusion element (ME), 553
Muxdff, 401

N

Named port connection, 399, 722
Names (Verilog), 699
NAND circuits, 41–43, 181
NAND gate (*see* Gates)
n-cube, 192
Negative edge, 359
Negative logic, 68, 82
Negative numbers, 240
negedge (Verilog), 389
Net (in Verilog), 699, 743
Network, 23
Next state, 449, 529
Nibble, 231
9's complement, 249
NMOS technology, 71–76
NMOS transistor, 69
Node handle, 770
Node (MAX+plusII), 757
connecting by name, 809
connecting with wires, 756
naming, 809
Noise, 113
immunity, 113
margin, 113
power supply, 689
Non-blocking assignment (Verilog), 391, 394, 712, 721, 734, 743
Non-disjoint decomposition, 179
Non-volatile programming, 98
NOR gate (*see* Gates)
NOR circuits, 41–44, 181
NOR plane, 129
NOT operator (Verilog), 704
NOT gate (*see* Gates)
NUMAchine, 10
Number conversion, 231
Number representation:
binary coded decimal, 284
fixed-point, 282
floating-point, 282
hexadecimal, 232
octal, 232
1's-complement, 242
positional notation, 230
sign and magnitude, 242
signed integer, 240
10's-complement, 249
2's-complement, 243, 704
in Verilog, 275, 702
Numbers (in Verilog), 700

O

Octal numbers, 232
Odd function, 236
One-hot encoding, 311, 383, 460, 582
1's-complement representation, 242, 704
1149.1 Standard, 688
On-resistance, 110
ON-set, 206

Open-collector, 837
Open-drain, 837
Operations (*see* Logic functions)
Operators (Verilog), 333, 703
Optimization (*see* Minimization)
OR gate (*see* Gates)
OR operator (Verilog), 704
OR plane, 88
Ordered port connection, 722
Ordering of statements, 714
Oscillations, 353
Oscilloscope, 691
Output delay time (t_{od}), 655
Overflow (*see* Arithmetic overflow)
Override (in Verilog), 722, 745

P

Packages (physical):
  ball grid array (BGA), 100
  dual inline (DIP), 85
  pin grid array (PGA), 99
  plastic-leaded chip carrier (PLCC), 94
  quad flat pack, 97
  small-outline integrated circuit (SOIC), 85
PAL, 90–94, 135, 822
Parallel expanders, 826
Parallel-to-series converter, 370
Parallel transfer, 370
parameter (Verilog), 269, 701, 710, 723
Parasitic capacitance, 114
Parity, 289
Part-select, 701
Partial product, 277
Pass transistor, 136
Path sensitizing, 669
PGA package, 99
Physical design, 12, 52, 211
Pin assignments, 790
Pinstub, 756
PLA, 87–90, 130–134
Placement, 211, 688
PLCC package, 94
PLD, 5, 87–103
PMOS transistor, 70, 110
Polysilicon, 108
posedge (Verilog), 389, 730, 743
Port (Verilog), 55, 705
Portability, 51
Positional number representation, 230
Positive logic, 68
Power dissipation, 117
  dynamic, 118
  in CMOS circuits, 117
  in NMOS circuits, 117
  static, 117
PRBSG, 683
Precedence of operations, 34, 704
Present state, 449, 529
Preset input, 355
Price/performance ratio, 254
Prime implicant, 159
Primitive flow table, 555
Primitives library, 754
Printed circuit board (PCB), 3, 10, 13, 688–692
Priority, 317
  encoder, 317, 332
  in Verilog code, 332, 334, 717
Procedural statements (Verilog), 59, 711–721
Process transconductance parameter, 109–110
Processor, 417
Product-of-sums form (POS), 41, 164
Programmable array logic (*see* PAL)
Programmable logic array (*see* PLA)
Programmable logic device (*see* PLD)
Programmable ROM (PROM), 316
Programmable switch, 5, 89, 131
Programming file, 93
Programming unit, 93
Project (MAX+plusII), 748
Project Name, 752
Project Save & Compile, 781
Propagation delay, 115–116, 254, 437, 529
Properties of Boolean algebra, 28
Pseudo-NMOS technology, 111, 142
Pseudorandom tests, 682
Pseudorandom binary sequence generator (PRBSG), 683
Pull-down network, 75
Pull-up network, 75
Pull-up resistor, 72
Pulse mode, 528

Q

QFP package, 97
Quine-McCluskey method, 193–201

R

Race condition, 542
Radix, 230
Radix (in Verilog), 700
Radix complement, 249
RAM (*see* Static random access memory)
Random testing, 674–677
Range (Verilog), 699
Read-only memory (ROM), 315
Reduction operator (Verilog), 336, 704
Reflections, 689
reg (Verilog), 702
Register, 368, 400, 607, 733
  used in Verilog texts, 702
Register delay time (t_{rd}), 655
Register-Transfer Level (RTL) code, 434, 742
Register transfers, 408
Relational operator (Verilog), 337, 704
Reliability, 691
repeat loop (Verilog), 717, 720
Replication operator (Verilog), 277, 338, 704, 742
Report file, 210
Reset input, 355
Reset state, 447, 690
Reset synchronization, 614
Resistance (transistor channel), 110
Ring counter, 383
Ring Oscillator, 438
Ripple-carry adder, 239
Ripple-carry adder (Verilog), 718
Ripple counter, 373
Rise time, 114
ROM (*see* Read only memory)
Routing, 212, 688
  channel, 98, 104
Row dominance, 195

S

Saturation region, 109
Scalar (Verilog), 699
Scan path, 678
Schematic, 23
Schematic capture, 49
Sea-of-gates technology, 106
Selection tool, 756
Semi-custom chips, 6, 106
Sequence detector, 513
Sequential circuits, 446
  analysis, 512, 531
  asynchronous, 528–599
  definition of, 350
  finite state machine, 446
  flow table, 531
  formal model, 519
  merger diagram, 557
  state assignment, 449, 458, 496, 568–584
  state assignment in Verilog, 474
  state diagram, 448
  state reduction, 486–493, 553–568
  state table, 449
  synchronous, 446–521
  testing, 677–688
  transition diagram, 571
Sequential statement (Verilog), 712
Sensitivity list (Verilog), 59, 395, 743

Serial adder, 477
Serial parity generator, 540
Serial transfer, 370
Series-to-parallel converter, 370
Set input, 355
Setup time, 358, 399, 655
7400-series chips, 83–86
7-segment display, 319
  BCD-to-7-segment decoder, 319
  Verilog code, 722
Shannon's expansion, 304
Sharp-operation (#-operation), 204
Shift operators (Verilog), 704
Shift register, 369
  Verilog code, 401, 607, 734
SIA roadmap, 3
Sign bit, 240
Sign-and-magnitude representation, 242
Signal (in Verilog), 699, 700
Signature, 683
Signature analysis, 687
Sign extension, 277
Sign extension (in Verilog), 710
Signed numbers, 240
Silicon wafer, 2,
Simplification (*see* Minimization)
Simulation, 12, 52
Simulator, 759, 783
Single-pole single-throw switch, 657
Single-pole double-throw switch, 657
Single-precision (*see* Floating point)
SIS (Sequential Interactive Synthesis), 209
Skew (*see* Clock skew)
Slave (*see* Flip-flop, master-slave)
Small-scale integration (SSI), 86
Socket, 93
Sort operation, 641
Source (in MOSFET transistor), 69
Speed grade, 780, 826
SR latch (*see* Latch)
Stable state, 528
Standard cells, 104
Standard chips, 4, 83–86
Standards:
  IEEE floating-point, 283
  1149.1 (Testing), 688
  Verilog 1364–1995, 54
  Verilog 1364–2001, 54
  VHDL, 50
Star-operation (*-operation), 201
Starvation, 512
Starting state, 447
State, 446
  assignment, 449, 458, 568–584
  assignment in Verilog, 474
  compatibility, 557
  diagram, 448
  equivalence, 486
  minimization, 486–493, 553–568
  definition of, 350
  table, 449
  variables, 449
State-adjacency diagram, 571
State-assigned table, 450
State machine (*see* Finite state machine)
Statement ordering, 714
Static hazard, 386, 585
Static random access memory (SRAM), 135, 609, 637, 828
Storing a bit value, 356
Structural Verilog code, 55
Stuck-at fault, 666
Subcircuit (in Verilog code), 721
Substrate, 69
Subtraction, 246, 626
Sum, 234
Sum-of-products form (SOP), 38
Swap, 408
Switch, 18, 69, 431
Synchronous clear (reset), 364, 380, 396
Synchronous counter, 374, 376
Synchronous sequential circuits (*see* Sequential circuits)
Syntax (Verilog), 56
Synthesis, 24, 51
  layout, 52
  logic, 35–41, 51, 210, 303–310
  multilevel, 171–184
  technology independent, 209

T

T flip-flop, 364
task (Verilog), 729, 745
Technology mapping, 52, 209
10's complement, 249
Template (Verilog), 745
Terminations, 690
Test, 666
Test generation, 667–677
Test set, 667
Test vectors (*see* Test generation)
Testing, 470, 666–692
Text Editor, 765
Theorems of Boolean algebra, 27
Three-state output (*see* Tri-state)
Three-way light control, 44
Threshold voltage, 63, 108, 119
Timer, 662
Timing Analyzer, 801, 813
Timing diagram, 26, 453
Timing simulation, 12, 52, 213
Toggling, 366
Traffic light, 662
Transfer characteristic, 112
Transistor:
  EEPROM, 131
  EPROM, 132
  MOSFET, 69–71, 106–110
  size, 114
Transistor-transistor logic (TTL), 836
Transition diagram, 571
Transition table (*see* Excitation table)
Transmission gate, 126
Transmission line effects, 690
Tree structure, 311, 673
Triode region, 108
tri (Verilog), 413, 702
Tri-state:
  buffer, 85, 122, 406
  Verilog code, 412, 707, 745
Truth table, 22
Truth table (in Verilog code), 716
Tunneling, 131
2's-complement representation, 243
22V10 PAL, 822

U

Unary plus (+), 704
Union, 32
Unsigned numbers, 230
Unstable state, 538
Up-counter, 373, 402
Up/down-counter, 373, 405
User-programmable device (*see* PLD)
Universal shift register, 441

V

Valuation, 22
Variable, 19
  in Verilog, 59, 271, 699, 743
Vector (Verilog), 268, 699
Vending machine controller, 489, 592
Venn diagram, 30–33
Verilog 2001, 712, 745
Verilog HDL, 54–60, 265–277, 320–342, 467–477, 698–746
  assign statement, 709
  asynchronous clear, 395, 731
  always block, 59, 389, 711, 743
  arithmetic assignment, 271
  blocking assignment, 390, 440, 712, 721, 734, 743
  begin, 712
  begin-end block, 712, 742
  bit-select, 701
  case sensitivity, 700
  case statement, 326, 714, 745
  casex statement, 331, 717
  casez statement, 331, 717

Verilog HDL—*Cont.*
comment, 699
concatenation, 272, 338, 623, 704
concurrent statement, 708, 743
continuous assignment, 58, 709
default case alternative, 715
defparam, 399, 414, 724, 743, 745
don't care, 700
end, 712
escaped identifier, 700
for loop, 269, 331, 717
forever loop, 717, 720
function, 340, 727
gate level primitives, 55, 265
generate, 726
genvar, 726
hierarchical code, 723
identifier, 699
if-else statement, 59, 323, 713
implied memory, 388, 744
initial block, 711
integer type, 271, 702
instantiation of gates, 706
instantiation of modules, 267, 721, 743, 745
label, 718, 744
logic gates, 706
logical operators (AND, OR, XOR), 704
memories, 703, 745
module, 55, 705
multibit assignment, 709
named port connection, 722
names, 699
negedge, 730
net, 271, 699, 743
non-blocking assignment, 391, 394, 712, 721, 734, 743
number representation, 275
operators, 333, 703
ordered port connection, 722
parameter, 269, 701, 710, 723
parameter override, 722, 745
part-select, 701
posedge, 730, 743
precedence, 704
procedural statements, 59, 711–721
port, 55, 705, 722
reg type, 271
radix, 700
range, 699
reg, 702
reduction operator, 704
repeat loop, 717, 720
replication, 277, 338, 704, 742
scalar, 271, 699
sensitivity list, 59, 395, 743
shift operators, 704
sign extension, 710
signal, 699
style, 698
synchronous clear, 396, 732
syntax, 56
task, 339, 729, 745
tri type, 271, 702
truth tables, 717
variable, 59, 271, 699, 743
vector, 268, 699
while loop, 717, 720
white space, 699
wire type, 271, 701
Vertex, 189
Very large-scale integration (VLSI), 87
VHDL, 50
Via, 104
Volatile programming, 102
Voltage levels, 68, 111
substrate bias, 119
V_{OH} and V_{OL}, 111–114
V_{IH} and V_{IL}, 112–114
Voltage spikes, 689
Voltage transfer characteristic (VTC), 112

W

Waveform Editor, 49
while loop (Verilog), 717, 720
wire (Verilog), 701
Wiring complexity, 175
World Wide Web, 837

X

Xilinx Virtex FPGA, 5, 834
Xilinx XC4000 FPGA, 830
XNOR (Exclusive-NOR) gate (*see* Gates)
XOR (Exclusive-OR) gate (*see* Gates)
XOR and XNOR operators (Verilog), 704
XOR synthesis, 216

FUNDAMENTALS OF DIGITAL LOGIC WITH VERILOG DESIGN

provides a desirable balance between classical and modern design approaches. Basic concepts are introduced using simple logic circuits, which are designed by using both manual techniques and modern CAD-tool-based methods. Having established the fundamental concepts, more complex, realistic circuits are then designed with the CAD tools.

KEY FEATURES OF THIS TEXT ARE:

Numerous detailed examples, ranging from circuits with only a few basic logic elements to digital systems such as simple processor.

The IEEE standard Verilog language is an integral part of design techniques used throughout the book. The language is introduced gradually in a way that is easily understood by beginning designers.

An in-depth presentation of modern digital circuit technology with emphasis on CMOS circuits and Programmable Logic Devices (CPLDs and FPGAs)

Altera's MAX+plusII CAD system (on CD-ROM) and a series of step-by-step tutorials are included. The CD-ROM also contains all Verilog examples presented in the book.

An Online Learning Center for the book contains PowerPoint figures from the text, solutions for instructors to access, and web links. The URL is **www.mhhe.com/brownvranesic2.**

The McGraw-Hill Companies

Tata McGraw-Hill
Publishing Company Limited
7 West Patel Nagar, New Delhi 110 008

Visit our website at : www.tatamcgrawhill.com

9 780070 495098 90000