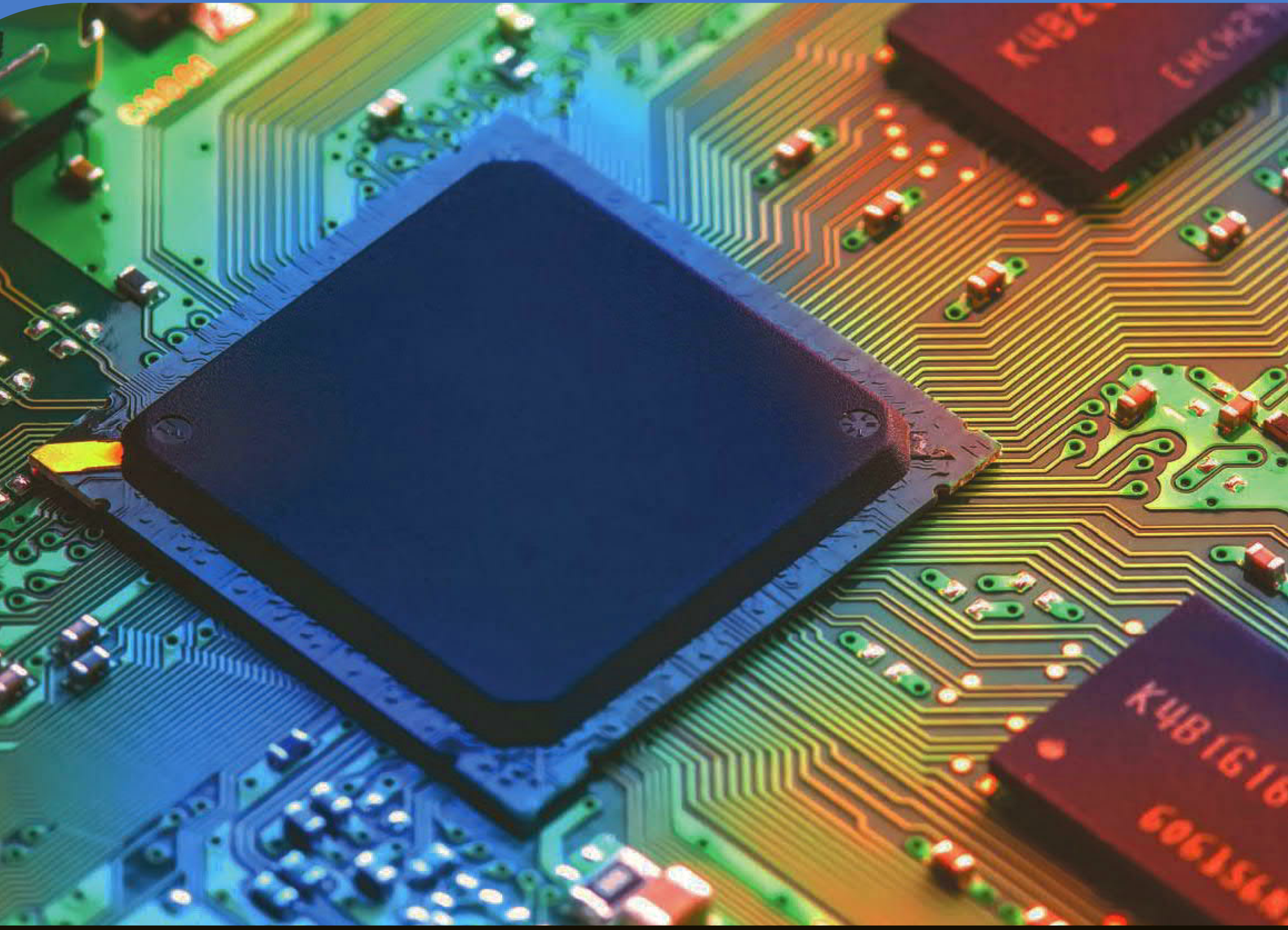
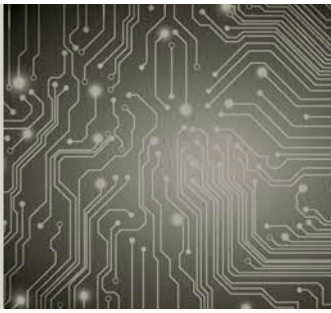
Third Edition

Digital Systems Design Using VHDL



Charles H. Roth, Jr. | Lizy Kurian John



DIGITAL SYSTEMS DESIGN USING VHDL®

Third Edition

Charles H. Roth, Jr.
The University of Texas at Austin

Lizy Kurian John
The University of Texas at Austin

CENGAGE Learning™

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

CENGAGE Learning™

***Digital Systems Design Using VHDL®*,**
Third Edition
Charles H. Roth, Jr. and Lizy Kurian John

Product Director, Global Engineering: Timothy L. Anderson

Associate Media Content Developer: Ashley Kaupert

Product Assistant: Alexander Sham

Marketing Manager: Kristin Stine

Director, Higher Education Production: Sharon L. Smith

Content Project Manager: Jana Lewis

Production Service: RPK Editorial Services, Inc.

Copyeditor: Harlan James

Proofreader: Lori Martinsek

Indexer: Shelly Gerger-Knechtl

Compositor: SPi Global

Senior Art Director: Michelle Kunkler

Internal Designer: Stratton Design

Cover Designer: Tin Box Studio

Cover Images: (upper image) stockchairatgfx/Shutterstock.com, (lower image) Raimundas/Shutterstock.com

Intellectual Property
Analyst: Christine Myaskovsky
Project Manager: Sarah Shainwald

Text and Image Permissions Researcher: Kristiina Paul

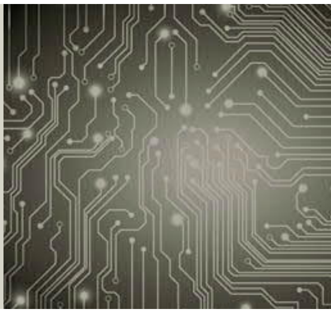
Manufacturing Planner: Doug Wilke

Library of Congress Control Number: 2016952395

ISBN: 978-1-305-63514-2

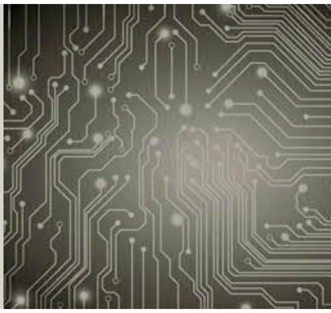
Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Printed in the United States of America
Print Number: 01 Print Year: 2016



CONTENTS

PREFACE



This textbook is intended for a senior-level course in digital systems design. The book covers both basic principles of digital system design and the use of a hardware description language, VHDL, in the design process. After basic principles have been covered, students are encouraged to practice design by going through the design process. For this reason, many digital system design examples, ranging in complexity from a simple binary adder to a microprocessor, are included in the text.

Students using this textbook should have completed a course in the fundamentals of logic design, including both combinational and sequential circuits. Although no previous knowledge of VHDL is assumed, students should have programming experience using a modern higher-level language such as C. A course in assembly language programming and basic computer organization is also very helpful, especially for Chapter 9.

This book is the result of many years of teaching a senior course in digital systems design at the University of Texas at Austin. Throughout the years, the technology for hardware implementation of digital systems has kept changing, but many of the same design principles are still applicable. In the early years of the course, we handwired modules consisting of discrete transistors to implement our designs. Then integrated circuits were introduced, and we were able to implement our designs using breadboards and TTL logic. Now we are able to use FPGAs and CPLDs to realize very complex designs. We originally used our own hardware description language together with a simulator running on a mainframe computer. When VHDL was adopted as an IEEE standard and became widely used in industry, we switched to VHDL. The widespread availability of high-quality commercial CAD tools now enables us to synthesize complex designs directly from the VHDL code.

All of the VHDL code in this textbook has been tested using the Modelsim simulator. The Modelsim software is available in a student edition, and we recommend its use in conjunction with this text. The companion website that accompanies this text provides a link for downloading the Modelsim student edition and an introductory tutorial to help students get started using the software. Students can access these materials by visiting https://login.cengage.com.

Structure

Because students typically take their first course in logic design two years before this course, most students need a review of the basics. For this reason, **Chapter 1** includes a review of logic design fundamentals. Most students can review this material on their own, so it is unnecessary to devote much lecture time to this chapter.

Chapter 2 starts with an overview of modern design flow. It also summarizes various technologies for implementation of digital designs. Then, it introduces the basics of VHDL, and this hardware description language is used throughout the rest of the book. Additional features of VHDL are introduced on an as-needed basis, and more advanced features are covered in Chapter 8. From the start, we relate the constructs of VHDL to the corresponding hardware. Some textbooks teach VHDL as a programming language and devote many pages to teaching the language syntax. Instead, our emphasis is on how to use VHDL in the digital design process. The language is very complex, so we do not attempt to cover all its features. We emphasize the basic features that are necessary for digital design and omit some of the less-used features. Use of standard IEEE VHDL libraries is introduced in this chapter and only IEEE standard libraries are used throughout the test. Chapter 2 also provides coding tips and strategies on how to write VHDL code that can lead to the intended hardware quickly.

VHDL is very useful in teaching top-down design. We can design a system at a high level and express the algorithms in VHDL. We can then simulate and debug the designs at this level before proceeding with the detailed logic design. However, no design is complete until it has actually been implemented in hardware and the hardware has been tested. For this reason, we recommend that the course include some lab exercises in which designs are implemented in hardware. We introduce simple programmable logic devices (PLDs) in **Chapter 3** so that real hardware can be used early in the course if desired. Chapter 3 starts with an overview of programmable logic devices and presents simple programmable logic devices first, followed by an introduction to complex programmable logic devices (CPLDs) and Field Programmable Gate Arrays (FPGAs). There are many products in the market, and it is useful for students to learn about commercial products. However, it is more important for them to understand the basic principles in the construction of these programmable devices. Hence we present the material in a generalized fashion, with references to specific products as examples. The material in this chapter also serves as an introduction to the more detailed treatment of FPGAs in Chapter 6.

Chapter 4 presents a variety of design examples, including both arithmetic and non-arithmetic examples. Simple examples such as a BCD to 7-segment display decoder to more complex examples such as game scoreboards, keypad scanners, and binary dividers are presented. The chapter presents common techniques used for computer arithmetic, including carry look-ahead addition and binary multiplication and division. Use of a state machine for sequencing the operations in a digital system is an important concept presented in this chapter. Synthesizable VHDL code is presented for the various designs. A variety of examples are presented so that instructors can select their favorite designs for teaching.

Use of sequential machine charts (SM charts) as an alternative to state graphs is covered in **Chapter 5**. We show how to write VHDL code based on SM charts and how to realize hardware to implement the SM charts. Then, the technique of microprogramming is presented. Transformation of SM charts for different types of microprogramming is discussed. Then, we show how the use of linked state machines facilitates the decomposition of complex systems into simpler ones. The design of a dice-game simulator is used to illustrate these techniques.

Chapter 6 presents issues related to implementing digital systems in Field Programmable Gate Arrays. A few simple designs are first hand-mapped into FPGA building blocks to illustrate the mapping process. Shannon's expansion for decomposition of functions with several variables into smaller functions is presented. Features of modern FPGAs like carry chains, cascade chains, dedicated memory, dedicated multipliers, etc., are then presented. Instead of describing all features in a selected commercial product, the features are described in a general fashion. Once students understand the fundamental general principles, they will be able to understand and use any commercial product they have to work with. This chapter also introduces the processes and algorithms in the software design flow. Synthesis, mapping, placement, and routing processes are briefly described. Optimizations during synthesis are illustrated.

Basic techniques for floating-point arithmetic are described in **Chapter 7**. We present a simple floating-point format with 2's complement numbers and then the IEEE standard floating-point formats. A floating-point multiplier example is presented starting with development of the basic algorithm, then simulating the system using VHDL, and finally synthesizing and implementing the system using an FPGA. Some instructors may prefer to cover Chapters 8 and 9 before teaching Chapter 7. Chapter 7 can be omitted without loss of any continuity.

By the time students reach **Chapter 8**, they should be thoroughly familiar with the basics of VHDL. At this point we introduce some of the more advanced features of VHDL and illustrate their use. The use of multi-valued logic, including the IEEE-1164 standard logic, is one of the important topics covered. A memory model with tri-state output busses illustrates the use of the multi-valued logic.

Chapter 9 presents the design of a microprocessor, starting from the description of the instruction set architecture (ISA). The processor is an early RISC processor, the MIPS R2000. The important instructions in the MIPS ISA are described and a subset is then implemented. The design of the various components of the processor, such as the instruction memory module, data memory module, and register file are illustrated module by module. These components are then integrated together, and a complete processor design is presented. The model can be tested with a test bench, or it can be synthesized and implemented on an FPGA. In order to test the design on an FPGA, one will need to write input-output modules for the design. This example requires understanding of the basics of assembly language programming and computer organization. After presenting the MIPS design, the chapter progresses to a design with the ARM ISA. A simplified introduction to the ARM ISA is first presented, followed by an implementation of a subset of the ISA. This is a significant addition to the previous MIPS design. The coverage is augmented with relevant example questions, solutions, and exercise problems on the ARM ISA.

Chapter 10 is a new chapter, presenting new material on verification, a concept central to the design of complex systems. A good understanding of timing in sequential circuits and the principles of synchronous design is essential to the digital system design process. Functional verification is introduced, explaining jargon in verification, validation, emulation, and distinction with testing. Self-testing test benches are explained. Concept of coverage is introduced. Timing verification is presented with static timing analysis of circuits. Clock skew, clock gating, power gating, and asynchronous design are introduced.

The important topics of hardware testing and design for testability are covered in **Chapter 11.** This chapter introduces the basic techniques for testing combinational and sequential logic. Then scan design and boundary-scan techniques, which facilitate the testing of digital systems, are described. The chapter concludes with a discussion of built-in self-test (BIST). VHDL code for a boundary-scan example and for a BIST example is included. The topics in this chapter play an important role in digital system design, and we recommend that they be included in any course on this subject. Chapter 11 can be covered any time after the completion of Chapter 8.

Chapter 12, available only online via https://login.cengage.com, presents three complete design examples that illustrate the use of VHDL synthesis tools. First, a wristwatch design shows the progress of a design from a textual description to a state diagram and then a VHDL model. This example illustrates modular design. The test bench for the wristwatch illustrates the use of multiple procedure calls to facilitate the testing. The second example describes the use of VHDL to model RAM memories. The third example, a serial communications receiver-transmitter, should easily be understood by any student who has completed the material through Chapter 8.

New to the Third Edition

For instructors who used the second edition of this text, here is a mapping to help understand the changes in the third edition. The IEEE numeric-bit library is used first until multi-valued logic is introduced in Chapter 8. The multi-valued IEEE numeric-std library is used thereafter. All code has been converted to use IEEE standard libraries instead of the BITLIB library.

Chapter 1	Logic hazard description is improved. More detailed examples on static hazards are added. Students are introduced to memristors. The sequential circuit timing section is kept to an introductory level because more elaborate static timing analysis is presented in a new chapter on verification, Chapter 10.
Chapter 2	Coding examples to improve test bench creation are introduced in Chapter 2. Coding tips and strategies for synthesizable code are presented. Multiple debugging examples are presented towards the end of the chapter.
Chapter 3	Information on commercial chips updated to reflect state of the art. Added introduction to programmable System on a Chip (SoC).
Chapter 4	General introduction to parallel prefix adders with details of Kogge Stone adder. New exercise problems including those on Kogge Stone and Brent-Kung adders.
Chapter 5	Added historical perspective on microprogramming. New example problems and new exercise problems.
Chapter 6	Information on commercial chips updated to reflect state of the art. Xilinx Kintex chips described. New problems added to make use of the new types of FPGA architectures.
Chapter 7	Several new example problems on IEEE floating point standards illustrated in detail. Rounding modes in IEEE standard and Microsoft Excel illustrated with examples. Several new exercise problems.
Chapter 8	Functions and procedures from the prior edition's Chapter 2 moved to here. Many sections from old Chapter 8 are still here. A memory model previously in old Chapter 9 presented as example of multi-valued logic design in new Chapter 8. New examples on functions and procedures added. VHDL function NOW is introduced. New exercise questions on Kogge-Stone and Brent-Kung adder to utilize advanced VHDL features such as generate are added.
Chapter 9	This chapter covers ARM processor design. A simplified introduction to the ARM ISA is first presented followed by an implementation of a subset of the ISA. This is a significant addition to the MIPS design that was previously presented. Several example questions and solutions on the ARM ISA are presented. Several exercise problems are added.
Chapter 10	This is a new chapter on verification. It covers functional verification as introduced, explaining terminology in verification, validation, emulation, and distinction with testing. Self-checking test benches are explained. Concept of coverage is introduced. Timing verification is presented with static timing analysis of circuits. Clock skew, clock gating, power gating, and asynchronous design are briefly presented. Exercise problems cover functional and timing verification.

Chapter 11	The prior edition's Chapter 10 on testing is modified and retained as Chapter 11. Memory testing is introduced. Several new problems added. Tests such as the popular March 14 tests are introduced in the chapter and new exercise problems are included.
Chapter 12	This chapter will be available only electronically. The wristwatch design, the memory timing models, and the UART design will be available to interested instructors and students. This chapter may be accessed at https://login.cengage.com.

Instructor Resources

A detailed **Instructor's Solutions Manual** containing solutions to all the exercises from the text, **VHDL code** used in the book, and **Lecture Note PowerPoint slides** are available via a secure, password-protected Instructor Resource Center at https://login.cengage.com.

Acknowledgments

We would like to thank the many individuals who have contributed their time and effort to the development of this textbook. Over many years we have received valuable feedback from the students in our digital systems design courses. We would especially like to thank the faculty members who reviewed the previous edition and offered many suggestions for its improvement. These faculty include:

Lee Belfore, *Old Dominion University*
Gang Feng, *University of Wisconsin, Platteville*
K. Gopalan, *Purdue University, Calumet*
Miriam Leeser, *Northeastern University*
Melissa C. Smith, *Clemson University*
Aaron Striegel, *University of Notre Dame*
Don Thomas, *Carnegie Mellon University*

We also wish to acknowledge Dr. Nur Touba's comments on various parts of the book. Dr. Earl Swartzlander provided comments on the parallel prefix adder section.

We thank ARM Limited for providing the permission to include an example design based on the ARM ISA in Chapter 9. Special thanks go to Ian Burgess at Mentor Graphics for his work on the ModelSim code. We also take this opportunity to express our gratitude to the student assistants who helped with the word processing, VHDL code testing, and illustrations: Arif Mondal, Kevin Johns, Jae-Min Jo, Di Xie, Poulami Das, and Kangjoo Lee, who helped on this version, and Roger Chen, William Earle, Manish Kapadia, Matt Morgan, Elizabeth Norris, and Raman Suri, who helped on the previous edition.

We wish to acknowledge and thank our Global Engineering team at Cengage Learning for their dedication to this new book: Timothy Anderson, Product Director; Ashley Kaupert, Associate Media Content Developer; Jana Lewis, Content Project Manager; Kristin Stine, Marketing Manager; Elizabeth Brown and Brittany Burden, Learning Solutions Specialists; Alexander Sham, Product Assistant; and Rose Kernan of RPK Editorial Services, Inc. They have skillfully guided every aspect of this text's development and production to successful completion.

Charles. H. Roth, Jr.
Lizy K. John

ABOUT THE AUTHORS

Charles H. Roth, Jr. is Professor Emeritus of Electrical and Computer Engineering at the University of Texas at Austin. He has been on the UT faculty since 1961. He received his BSEE degree from the University of Minnesota, his MSEE and EE degrees from the Massachusetts Institute of Technology, and his PhD degree in EE from Stanford University. His teaching and research interests included logic design, digital systems design, switching theory, microprocessor systems, and computer- aided design. He developed a self-paced course in logic design, which formed the basis of his textbook, *Fundamentals of Logic Design*. He is also the author of *Digital Systems Design Using VHDL*, two other textbooks, and several software packages. He is the author or co-author of more than 50 technical papers and reports. Six PhD students and 80 MS students have received their degrees under his supervision. He received several teaching awards including the 1974 General Dynamics Award for Outstanding Engineering Teaching.

Lizy Kurian John is the B.N. Gafford Professor in the Electrical and Computer Engineering at University of Texas at Austin. She received her PhD in Computer Engineering from the Pennsylvania State University. Her research interests include computer architecture, performance evaluation, workload characterization, digital systems design, FPGAs, rapid prototyping, and reconfigurable architectures. She is the recipient of many awards including the NSF CAREER award, UT Austin Engineering Foundation Faculty Award, Halliburton, Brown, and Root Engineering Foundation Young Faculty Award 2001, University of Texas Alumni Association (Texas Exes) Teaching Award 2004, the Pennsylvania State University Outstanding Engineering Alumnus 2011, etc. She has co-authored a book on *Digital Systems Design using VHDL* (Cengage Publishers, 2007), a book on *Digital Systems Design using Verilog* (Cengage Publishers, 2014) and has edited four books including a book on *Computer Performance Evaluation and Benchmarking.* In the past, she has served as Associate Editor of *IEEE Transactions on Computers, IEEE Transactions on VLSI and IEEE Micro*. She holds 10 U.S. patents and is an IEEE Fellow (Class of 2009).

CHAPTER 1

REVIEW OF LOGIC DESIGN FUNDAMENTALS

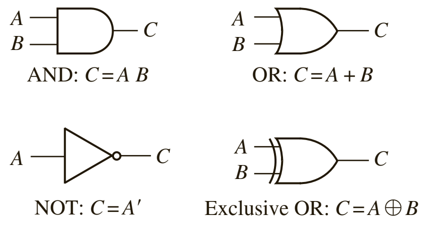
This chapter reviews many of the logic design topics normally taught in a first course in logic design. First, combinational logic and then sequential logic are reviewed. Combinational logic has no memory, so the present output depends only on the present input. Sequential logic has memory, so the present output depends not only on the present input but also on the past sequence of inputs. Various types of flip-flops and their state tables are presented. Example designs for Mealy and Moore sequential circuits are illustrated, followed by techniques to reduce the number of states in sequential designs. Circuit timing and synchronous design are particularly important, since a good understanding of timing issues is essential to the successful design of digital systems. A detailed treatment of sequential circuit timing is presented in Chapter 10 in a section on timing verification. For more details on any of the topics discussed in this chapter, the reader should refer to a standard logic design textbook such as Roth and Kinney, ***Fundamentals of Logic Design***, 7th Edition (Cengage Learning, 2014). Some of the review examples that follow are referenced in later chapters of this text.

1.1 Combinational Logic

Some of the basic gates used in logic circuits are shown in Figure 1-1. Unless otherwise specified, all the variables used to represent logic signals are two-valued, and the two values are designated 0 and 1. Normally positive logic is used, for which a low voltage corresponds to a logic 0 and a high voltage corresponds to a logic 1. When negative logic is used, a low voltage corresponds to a logic 1 and a high voltage corresponds to a logic 0.

For the AND gate of Figure 1-1, the output $C = 1$ if and only if the input $A = 1$ *and* the input $B = 1$. Use a raised dot or simply write the variables side by side to indicate the AND operation; thus $C = A$ AND $B = A \cdot B = AB$. For the OR gate, the output $C = 1$ if and only if the input $A = 1$ *or* the input $B = 1$ (inclusive OR). Use $+$ to indicate the OR operation; thus $C = A$ OR $B = A + B$. The NOT gate, or inverter, forms the complement of the input; that is, if $A = 1, C = 0$, and if $A = 0, C = 1$. Use a prime $(')$ to indicate the

FIGURE 1-1: Basic Gates

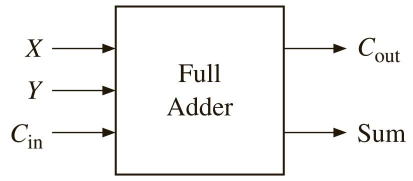


complement (NOT) operation, so $C = \text{NOT A} = A'$. The exclusive-OR (XOR) gate has an output $C = 1$ if $A = 1$ and $B = 0$ or if $A = 0$ and $B = 1$. The symbol $\oplus$ represents exclusive OR, so write

$$C = A \text{ XOR } B = AB' + A'B = A \oplus B \tag{1-1}$$

The behavior of a combinational logic circuit can be specified by a truth table that gives the circuit outputs for each combination of input values. As an example, consider the full adder of Figure 1-2, which adds two binary digits (X and Y) and a carry (C_{in}) to give a sum (Sum) and a carry out (C_{out}). The truth table specifies the adder outputs as a function of the adder inputs. For example, when the inputs are $X = 0$, $Y = 0$, and $C_{\text{in}} = 1$, adding the three inputs gives $0 + 0 + 1 = 01$, so the sum is 1 and the carry out is 0. When the inputs are 011, $0 + 1 + 1 = 10$, so $Sum = 0$ and $C_{\text{out}} = 1$. When the inputs are $X = Y = C_{\text{in}} = 1$, $1 + 1 + 1 = 11$, so $Sum = 1$ and $C_{\text{out}} = 1$.

FIGURE 1-2: Full Adder



(a) Full adder module

X	Y	C_{in}	C_{out}	Sum
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

(b) Truth table

Derive algebraic expressions for Sum and C_{out} from the truth table. From the table, $Sum = 1$ when $X = 0$, $Y = 0$, and $C_{\text{in}} = 1$. The term $X'Y'C_{\text{in}}$ equals 1 only for this combination of inputs. The term $X'YC'_{\text{in}} = 1$ only when $X = 0$, $Y = 1$, and $C_{\text{in}} = 0$. The term $XY'C'_{\text{in}}$ is 1 only for the input combination $X = 1$, $Y = 0$, and $C_{\text{in}} = 0$. The term XYC_{in} is 1 only when $X = Y = C_{\text{in}} = 1$. Therefore, Sum is formed by ORing these four terms together:

$$Sum = X'Y'C_{\text{in}} + X'YC'_{\text{in}} + XY'C'_{\text{in}} + XYC_{\text{in}} \tag{1-2}$$

Each of the terms in this sum of products (SOP) expression is 1 for exactly one combination of input values. In a similar manner, C_{out} is formed by ORing four terms together:

$$C_{\text{out}} = X'YC_{\text{in}} + XY'C_{\text{in}} + XYC'_{\text{in}} + XYC_{\text{in}} \tag{1-3}$$

Each term in Equations (1-2) and (1-3) is referred to as a ***minterm***, and these equations are referred to as ***minterm expansions***. These minterm expansions can also be written in m-notation or decimal notation as follows:

$$Sum = m_1 + m_2 + m_4 + m_7 = \Sigma m(1, 2, 4, 7)$$

$$C_{\text{out}} = m_3 + m_5 + m_6 + m_7 = \Sigma m(3, 5, 6, 7)$$

The decimal numbers designate the rows of the truth table for which the corresponding function is 1. Thus $Sum = 1$ in rows 001, 010, 100, and 111 (rows 1, 2, 4, 7).

A logic function can also be represented in terms of the inputs for which the function value is 0. Referring to the truth table for the full adder, $C_{\text{out}} = 0$ when $X = Y = C_{\text{in}} = 0$. The term $(X + Y + C_{\text{in}})$ is 0 only for this combination of inputs. The term $(X + Y + C_{\text{in}}')$ is 0 only when $X = Y = 0$ and $C_{\text{in}} = 1$. The term $(X + Y' + C_{\text{in}})$ is 0 only when $X = C_{\text{in}} = 0$ and $Y = 1$. The term $(X' + Y + C_{\text{in}})$ is 0 only when $X = 1$ and $Y = C_{\text{in}} = 0$. C_{out} is formed by ANDing these four terms together:

$$C_{\text{out}} = (X + Y + C_{\text{in}})(X + Y + C_{\text{in}}')(X + Y' + C_{\text{in}})(X' + Y + C_{\text{in}}) \qquad (1\text{-}4)$$

C_{out} is 0 only for the 000, 001, 010, and 100 rows of the truth table and, therefore, must be 1 for the remaining four rows. Each of the terms in the Product of Sums (POS) expression in Equation (1-4) is referred to as a ***maxterm***, and (1-4) is called a ***maxterm expansion***. This ***maxterm expansion*** can also be written in decimal notation as

$$C_{\text{out}} = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi M(0, 1, 2, 4)$$

where the decimal numbers correspond to the truth table rows for which $C_{\text{out}} = 0$.

1.2 Boolean Algebra and Algebraic Simplification

The basic mathematics used for logic design is Boolean algebra. Table 1-1 summarizes the laws and theorems of Boolean algebra. They are listed in dual pairs; for example, Equation (1-10D) is the dual of (1-10). They can be verified easily for two-valued logic by using truth tables. These laws and theorems can be used to simplify logic functions, so they can be realized with a reduced number of components.

A very important law in Boolean algebra is the ***DeMorgan's law***. DeMorgan's laws, stated in Equations (1-16, 1-16D), can be used to form the complement of an expression on a step-by-step basis. The generalized form of DeMorgan's law in Equation (1-17) can be used to form the complement of a complex expression in one step. Equation (1-17) can be interpreted as follows: To form the complement of a Boolean expression, replace each variable by its complement; also replace 1 with 0, 0 with 1, OR with AND, and AND with OR. Add parentheses as required to assure the proper hierarchy of operations. If AND is performed before OR in F, then parentheses may be required to assure that OR is performed before AND in F'.

EXAMPLE

Find the complement of F if

$$F = X + E'K(C(AB + D') \cdot 1 + WZ'(G'H + 0))$$
$$F' = X'(E + K' + (C' + (A' + B')D + 0)(W' + Z + (G + H') \cdot 1))$$

Additional parentheses in F' were added when an AND operation in F was replaced with an OR. The dual of an expression is the same as its complement, except that the variables are not complemented.

TABLE 1-1: Laws and Theorems of Boolean Algebra

Operations with 0 and 1:			
$X + 0 = X$	(1-5)	$X \cdot 1 = X$	(1-5D)
$X + 1 = 1$	(1-6)	$X \cdot 0 = 0$	(1-6D)
Idempotent laws:			
$X + X = X$	(1-7)	$X \cdot X = X$	(1-7D)
Involution law:			
$(X')' = X$	(1-8)		
Laws of complementarity:			
$X + X' = 1$	(1-9)	$X \cdot X' = 0$	(1-9D)
Commutative laws:			
$X + Y = Y + X$	(1-10)	$XY = YX$	(1-10D)
Associative laws:			
$(X + Y) + Z = X + (Y + Z) = X + Y + Z$	(1-11)	$(XY)Z = X(YZ) = XYZ$	(1-11D)
Distributive laws:			
$X(Y + Z) = XY + XZ$	(1-12)	$X + YZ = (X + Y)(X + Z)$	(1-12D)
Simplification theorems:			
$XY + XY' = X$	(1-13)	$(X + Y)(X + Y') = X$	(1-13D)
$X + XY = X$	(1-14)	$X(X + Y) = X$	(1-14D)
$(X + Y')Y = XY$	(1-15)	$XY' + Y = X + Y$	(1-15D)
DeMorgan's laws:			
$(X + Y + Z + \cdots)' = X'Y'Z'\cdots$	(1-16)	$(XYZ\ldots)' = X' + Y' + Z' + \cdots$	(1-16D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \ldots, X_n', 1, 0, \cdot, +)$			(1-17)
Duality:			
$(X + Y + Z + \cdots)^D = XYZ\cdots$	(1-18)	$(XYZ\cdots)^D = X + Y + Z + \cdots$	(1-18D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \ldots, X_n, 1, 0, \cdot, +)$			(1-19)
Theorem for multiplying out and factoring:			
$(X + Y)(X' + Z) = XZ + X'Y$	(1-20)	$XY + X'Z = (X + Z)(X' + Y)$	(1-20D)
Consensus theorem:			
$XY + YZ + X'Z = XY + X'Z$	(1-21)	$(X + Y)(Y + Z)(X' + Z) = (X + Y)(X' + Z)$	(1-21D)

Four ways of simplifying a logic expression using the theorems in Table 1-1 are as follows:

1. ***Combining terms***. Use the theorem $XY + XY' = X$ to combine two terms. For example,

$$ABC'D' + ABCD' = ABD' \; [X = ABD', Y = C]$$

When combining terms by this theorem, the two terms to be combined should contain exactly the same variables, and exactly one of the variables should appear complemented in one term and not in the other. Since $X + X = X$, a given term may be duplicated and combined with two or more other terms. For example, the expression for C_{out} in Equation (1-3) can be simplified by combining the first and fourth terms, the second and fourth terms, and the third and fourth terms:

$$\begin{aligned} C_{\text{out}} &= (X'YC_{\text{in}} + XYC_{\text{in}}) + (XY'C_{\text{in}} + XYC_{\text{in}}) + (XYC'_{\text{in}} + XYC_{\text{in}}) \\ &= YC_{\text{in}} + XC_{\text{in}} + XY \qquad (1\text{-}22) \end{aligned}$$

Note that the fourth term in Equation (1-3) was used three times.

The theorem can still be used, of course, when X and Y are replaced with more complicated expressions. For example,

$$(A + BC)(D + E') + A'(B' + C')(D + E') = D + E'$$
$$[X = D + E', Y = A + BC, Y' = A'(B' + C')]$$

2. ***Eliminating terms***. Use the theorem $X + XY = X$ to eliminate redundant terms if possible; then try to apply the consensus theorem $(XY + X'Z + YZ = XY + X'Z)$ to eliminate any consensus terms. For example,

$$A'B + A'BC = A'B \; [X = A'B]$$
$$A'BC' + BCD + A'BD = A'BC' + BCD \; [X = C, Y = BD, Z = A'B]$$

3. ***Eliminating literals***. Use the theorem $X + X'Y = X + Y$ to eliminate redundant literals. Simple factoring may be necessary before the theorem is applied. For example,

$$\begin{aligned} A'B + A'B'C'D' + ABCD' &= A'(B + B'C'D') + ABCD' && \text{(by (1-12))} \\ &= A'(B + C'D') + ABCD' && \text{(by (1-15D))} \\ &= B(A' + ACD') + A'C'D' && \text{(by (1-10))} \\ &= B(A' + CD') + A'C'D' && \text{(by (1-15D))} \\ &= A'B + BCD' + A'C'D' && \text{(by (1-12))} \end{aligned}$$

The expression obtained after applying 1, 2, and 3 will not necessarily have a minimum number of terms or a minimum number of literals. If it does not and no further simplification can be made using 1, 2, and 3, deliberate introduction of redundant terms may be necessary before further simplification can be made.

4. ***Adding redundant terms***. Redundant terms can be introduced in several ways, such as adding XX', multiplying by $(X + X')$, adding YZ to $XY + X'Z$ (consensus theorem),

or adding XY to X. When possible, the terms added should be chosen so that they will combine with or eliminate other terms. For example,

$$\begin{aligned}
& WX + XY + X'Z' + WY'Z' && \text{(Add WZ}' \text{ by the consensus theorem.)} \\
&\quad = WX + XY + X'Z' + WY'Z' + WZ' && \text{(Eliminate } WY'Z'.) \\
&\quad = WX + XY + X'Z' + WZ' && \text{(Eliminate } WZ'.) \\
&\quad = WX + XY + X'Z'
\end{aligned}$$

When multiplying out or factoring an expression, in addition to using the ordinary distributive law (1-12), the second distributive law (1-12D) and theorem (1-20) are particularly useful. The following is an example of multiplying out to convert from a product of sums to a sum of products:

$$\begin{aligned}
& (A + B + D)(A + B' + C')(A' + B + D')(A' + B + C') \\
&\quad = (A + (B + D)(B' + C'))(A' + B + C'D') && \text{(by (1-12D))} \\
&\quad = (A + BC' + B'D)(A' + B + C'D') && \text{(by (1-20))} \\
&\quad = A(B + C'D') + A'(BC' + B'D) && \text{(by (1-20))} \\
&\quad = AB + AC'D' + A'BC' + A'B'D && \text{(by (1-12))}
\end{aligned}$$

Note that the second distributive law (1-12D) and theorem (1-20) were applied before the ordinary distributive law. Any Boolean expression can be factored by using the two distributive laws (1-12 and 1-12D) and theorem (1-20). As an example of factoring, read the steps in the preceding example in the reverse order.

The following theorems apply to exclusive-OR:

$$X \oplus 0 = X \tag{1-23}$$

$$X \oplus 1 = X' \tag{1-24}$$

$$X \oplus X = 0 \tag{1-25}$$

$$X \oplus X' = 1 \tag{1-26}$$

$$X \oplus Y = Y \oplus X \qquad \text{(commutative law)} \tag{1-27}$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z \qquad \text{(associative law)} \tag{1-28}$$

$$X(Y \oplus Z) = XY \oplus XZ \qquad \text{(distributive law)} \tag{1-29}$$

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y' \tag{1-30}$$

The expression for ***Sum*** in Equation (1-2) can be rewritten in terms of exclusive-OR by using Equations (1-1) and (1-30):

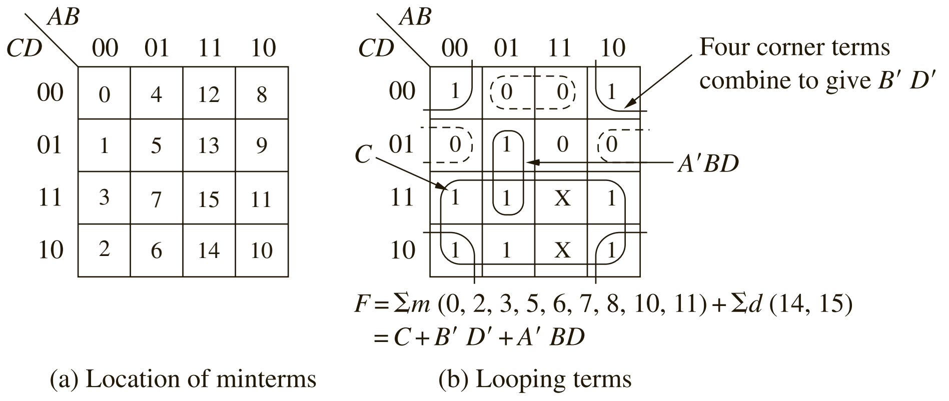
$$\begin{aligned}
Sum &= X'(Y'C_{\text{in}} + YC'_{\text{in}}) + X(Y'C'_{\text{in}} + YC_{\text{in}}) \\
&= X'(Y \oplus C_{\text{in}}) + X(Y \oplus C_{\text{in}})' = X \oplus Y \oplus C_{\text{in}}
\end{aligned} \tag{1-31}$$

The simplification rules that you studied in this section are important when a circuit has to be optimized to use a smaller number of gates. The existence of equivalent forms also helps when mapping circuits into particular target devices where only certain types of logic (e.g., NAND only or NOR only) are available.

1.3 Karnaugh Maps

Karnaugh maps (K-maps) provide a convenient way to simplify logic functions of three to five variables. Figure 1-3 shows a four-variable Karnaugh map. Each square in the map represents one of the 16 possible minterms of four variables. A 1 in a square indicates that the minterm is present in the function, and a 0 (or blank) indicates that the minterm is absent. An X in a square indicates that you don't care whether the minterm is present or not. ***Don't cares*** arise under two conditions: (1) The input combination corresponding to the don't care can never occur, and (2) the input combination can occur, but the circuit output is not specified for this input condition.

FIGURE 1-3: Four-Variable Karnaugh Maps



The variable values along the edge of the map are ordered so that adjacent squares on the map differ in only one variable. The first and last columns and the top and bottom rows of the map are considered to be adjacent. Two 1's in adjacent squares can be combined by eliminating one variable using $xy + xy' = x$. Figure 1-3 shows a four-variable function with nine minterms and two don't cares. Minterms $A'BC'D$ and $A'BCD$ differ only in the variable C, so they can be combined to form $A'BD$, as indicated by a loop on the map. Four 1's in a symmetrical pattern can be combined to eliminate two variables. The 1's in the four corners of the map can be combined as follows:

$$(A'B'C'D' + AB'C'D') + (A'B'CD' + AB'CD') = B'C'D' + B'CD' = B'D'$$

as indicated by the loop. Similarly, the six 1's and two X's in the bottom half of the map combine to eliminate three variables and form the term C. The resulting simplified function is

$$F = A'BD + B'D' + C$$

The minimum sum of products representation of a function consists of a sum of prime implicants. A group of one, two, four, or eight adjacent 1's on a map represents a prime implicant if it cannot be combined with another group of 1's to eliminate a variable. A prime implicant is essential if it contains a 1 that is not contained in any other prime implicant. When finding a minimum sum of products from a map, essential prime implicants should be looped first, and then a minimum number of prime implicants to cover the remaining 1's should be looped. The Karnaugh map shown in Figure 1-4 has five prime implicants and three essential prime implicants. $A'C'$ is essential because minterm m_1 is not covered by any other prime implicant. Similarly, ACD is essential because of m_{11}, and $A'B'D'$ is essential because of m_2. After looping the essential prime implicants, all 1's are covered except m_7. Since m_7 can be covered by either prime implicant $A'BD$ or BCD, F has two minimum forms:

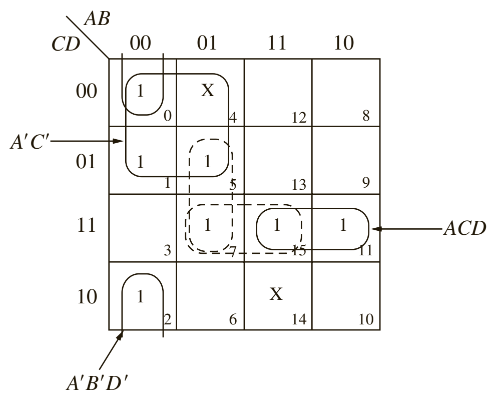
$$F = A'C' + A'B'D' + ACD + A'BD$$

and

$$F = A'C' + A'B'D' + ACD + BCD$$

When don't cares (X's) are present on the map, the don't cares are treated like 1's when forming prime implicants, but the X's are ignored when finding a minimum set of prime

FIGURE 1-4: Selection of Prime Implicants



Prime implicants: $A'C'$, ACD, $A'B'D'$, $A'BD$, BCD
Essential prime implicants: $A'C'$, ACD, $A'B'D'$

implicants to cover all the 1's. The following procedure can be used to obtain a minimum sum of products from a Karnaugh map:

1. Choose a minterm (a 1) that has not yet been covered.
2. Find all 1's and X's adjacent to that minterm. (Check the n adjacent squares on an n-variable map.)
3. If a single term covers the minterm and all the adjacent 1's and X's, then that term is an essential prime implicant, so select that term. (Note that don't cares are treated like 1's in steps 2 and 3 but not in step 1.)
4. Repeat steps 1, 2, and 3 until all essential prime implicants have been chosen.
5. Find a minimum set of prime implicants that cover the remaining 1's on the map. (If there is more than one such set, choose a set with a minimum number of literals.)

To find a minimum product of sums from a Karnaugh map, loop the 0's instead of the 1's. Since the 0's of F are the 1's of F', looping the 0's in the proper way gives the minimum sum of products for F', and the complement is the minimum product of sums for F.

For Figure 1-3, first loop the essential prime implicants of F' ($BC'D'$ and $B'C'D$, indicated by dashed loops) and then cover the remaining 0 with AB. Thus the minimum sum for F' is

$$F' = BC'D' + B'C'D + AB$$

from which the minimum product of sums for F is

$$F = (B' + C + D)(B + C + D')(A' + B')$$

1.3.1 Simplification Using Map-Entered Variables

Two four-variable Karnaugh maps can be used to simplify functions with five variables. If functions have more than five variables, *map-entered variables* can be used. Consider a truth table as in Table 1-2. There are six input variables (A, B, C, D, E, F) and one output variable (G). Only certain rows of the truth table have been specified. To completely specify the truth table, 64 rows will be required. The input combinations not specified in the truth table result in an output of 0.

TABLE 1-2: Partial Truth Table for a Six-Variable Function

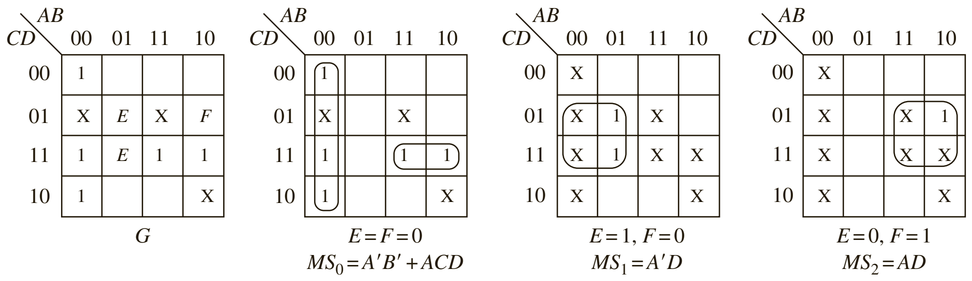
A	B	C	D	E	F	G
0	0	0	0	X	X	1
0	0	0	1	X	X	X
0	0	1	0	X	X	1
0	0	1	1	X	X	1
0	1	0	1	1	X	1
0	1	1	1	1	X	1
1	0	0	1	X	1	1
1	0	1	0	X	X	X
1	0	1	1	X	X	1
1	1	0	1	X	X	X
1	1	1	1	X	X	1

Karnaugh map techniques can be extended to simplify functions such as this using map-entered variables. Since E and F are the input variables with the most number of don't cares (X), a Karnaugh map can be formed with A, B, C, D and the remaining two variables can be entered inside the map. Figure 1-5 shows a four-variable map with variables E and F entered in the squares in the map. When E appears in a square, this means that if $E = 1$, the corresponding minterm is present in the function G, and if $E = 0$, the minterm is absent. The fifth and sixth rows in the truth table result in the E in the box corresponding to minterm 5 and minterm 7. The seventh row results in the F in the box corresponding to minterm 9. Thus, the map represents the six-variable function

$$G(A, B, C, D, E, F) = m_0 + m_2 + m_3 + Em_5 + Em_7 + Fm_9 + m_{11} + m_{15}$$
$$(+ \text{ don't care terms})$$

where the minterms are minterms of the variables A, B, C, D. Note that m_9 is present in G only when $F = 1$.

FIGURE 1-5: Simplification Using Map-Entered Variables



Next a general method of simplifying functions using map-entered variables is discussed. In general, if a variable P_i is placed in square m_j of a map of function F, this means that $F = 1$ when $P_i = 1$, and the variables are chosen so that $m_j = 1$. Given a map with variables $P_1, P_2, \ldots$ entered into some of the squares, the minimum sum of products form of F can be found as follows: Find a sum of products expression for F of the form

$$F = MS_0 + P_1 MS_1 + P_2 MS_2 + \cdots \tag{1-32}$$

where

- MS_0 is the minimum sum obtained by setting $P_1 = P_2 = \cdots = 0$.
- MS_1 is the minimum sum obtained by setting $P_1 = 1, P_j = 0\ (j \neq 1)$, and replacing all 1's on the map with don't cares.
- MS_2 is the minimum sum obtained by setting $P_2 = 1, P_j = 0\ (j \neq 2)$, and replacing all 1's on the map with don't cares.

Corresponding minimum sums can be found in a similar way for any remaining map-entered variables.

The resulting expression for F will always be a correct representation of F. This expression will be a minimum sum provided that the values of the map-entered variables can be assigned independently. On the other hand, the expression will not generally be a minimum sum if the variables are not independent (for example, if $P_1 = P_2'$).

For the example of Figure 1-5, maps for finding MS_0, MS_1, and MS_2 are shown, where E corresponds to P_1 and F corresponds to P_2. Note that it is not required to draw a map for $E = 1, F = 1$, because $E = 1$ already covers cases with $E = 1, F = 0$ and $E = 1, F = 1$. The resulting expression is a minimum sum of products for G:

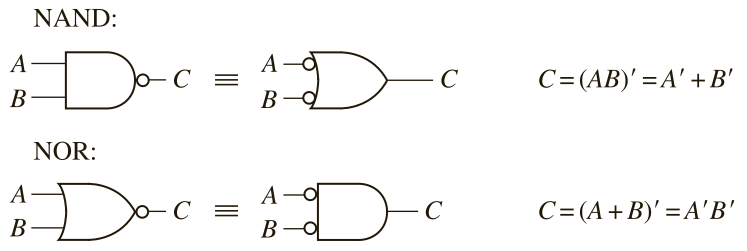
$$G = A'B' + ACD + EA'D + FAD$$

After some practice, it should be possible to write the minimum expression directly from the original map without first plotting individual maps for each of the minimum sums.

1.4 Designing With NAND and NOR Gates

In many technologies, implementation of NAND gates or NOR gates is easier than that of AND and OR gates. Figure 1-6 shows the symbols used for NAND and NOR gates. The ***bubble*** at a gate input or output indicates a complement. Any logic function can be realized using only NAND gates or only NOR gates.

FIGURE 1-6: NAND and NOR Gates

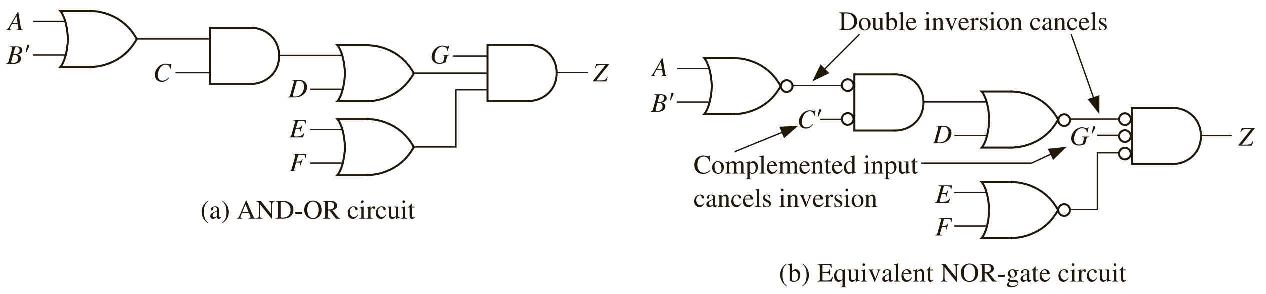


Conversion from circuits of OR and AND gates to circuits of all NOR gates or all NAND gates is straightforward. To design a circuit of NOR gates, start with a product-of-sums representation of the function (circle 0's on the Karnaugh map). Then find a circuit of OR and AND gates that has an AND gate at the output. If an AND gate output does not drive an AND gate input and an OR gate output does not connect to an OR gate input, then conversion is accomplished by replacing all gates with NOR gates and complementing inputs if necessary. Figure 1-7 illustrates the conversion procedure for

$$Z = G(E + F)(A + B' + D)(C + D) = G(E + F)[(A + B')C + D]$$

Conversion to a circuit of NAND gates is similar, except the starting point should be a sum of products form for the function (circle 1's on the map), and the output gate of the AND-OR circuit should be an OR gate.

FIGURE 1-7: Conversion to NOR Gates


A
B′
C
D
G
E
F
Z
(a) AND-OR circuit
Double inversion cancels
A
B′
C′
D
G′
E
F
Z
Complemented input cancels inversion
(b) Equivalent NOR-gate circuit

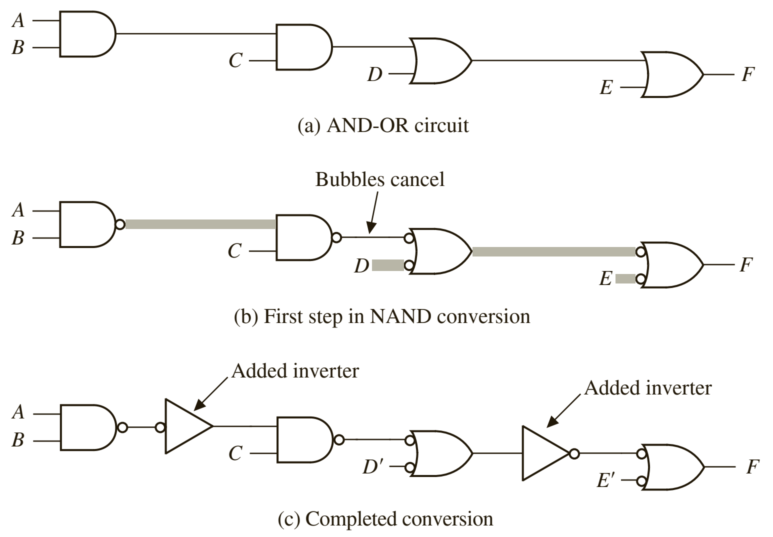
Even if AND and OR gates do not alternate, you can still convert a circuit of AND and OR gates to a NAND or NOR circuit, but it may be necessary to add extra inverters so that each added inversion is canceled by another inversion. The following procedure may be used to convert to a NAND (or NOR) circuit:

1. Convert all AND gates to NAND gates by adding an inversion bubble at the output. Convert OR gates to NAND gates by adding inversion bubbles at the inputs. (To convert to NOR, add inversion bubbles at all OR gate outputs and all AND gate inputs.)
2. Whenever an inverted output drives an inverted input, no further action is needed, since the two inversions cancel.

3. Whenever a noninverted gate output drives an inverted gate input or vice versa, insert an inverter so that the bubbles will cancel. (Choose an inverter with the bubble at the input or output, as required.)
4. Whenever a variable drives an inverted input, complement the variable (or add an inverter) so the complementation cancels the inversion at the input.

In other words, if we always add bubbles (or inversions) in pairs, the function realized by the circuit will be unchanged. To illustrate the procedure, you convert Figure 1-8(a) to NANDs. First, add bubbles to change all gates to NAND gates (Figure 1-8(b)). The highlighted lines indicate four places where you have added only a single inversion. This is corrected in Figure 1-8(c) by adding two inverters and complementing two variables.

FIGURE 1-8: Conversion of AND-OR Circuit to NAND Gates

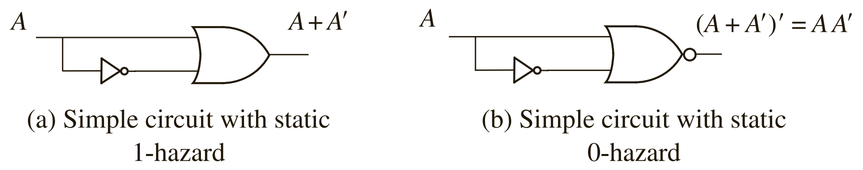


1.5 Hazards in Combinational Circuits

When the input to a combinational circuit changes, unwanted switching transients may appear in the output. These transients occur when different paths from input to output have different propagation delays. If, in response to an input change and for some combination of propagation delays, a circuit output may momentarily go to 0 when it should remain a constant 1, it is said that the circuit has a static ***1-hazard***. Similarly, if the output may momentarily go to 1 when it should remain a 0, it is said that the circuit has a static ***0-hazard***. If, when the output is supposed to change from 0 to 1 (or 1 to 0), the output may change three or more times, the circuit has a ***dynamic hazard***.

Consider the two simple circuits in Figure 1-9. Figure 1-9(a) shows an inverter and an OR gate implementing the function $A + A'$. Logically, the output of this circuit is expected to be a 1 always; however, a delay in the inverter gate can cause static hazards in this circuit. Assume a nonzero delay for the inverter and that the value of A just changed from 1 to 0. There is a short interval of time until the inverter delay has passed when both inputs of the OR gate are 0 and hence the output of the circuit may momentarily go to 0. Similarly, in the

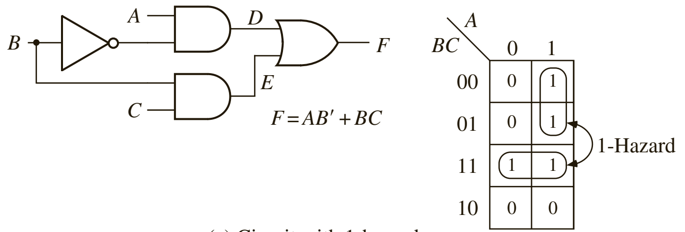
FIGURE 1-9: Simple Circuits Containing Hazards


A
A + A′
A
(A + A′)′ = A A′
(a) Simple circuit with static 1-hazard
(b) Simple circuit with static 0-hazard

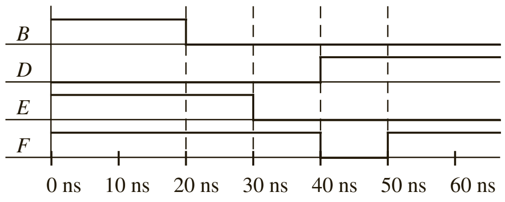
circuit in Figure 1-9(b), the expected output is always 0; however, when A changes from 1 to 0, a momentary 1 appears at the output of the inverter because of the delay. This circuit hence has a static 0-hazard. The hazard occurs because both A and A' have the same value for a short duration after A changes.

A static 1-hazard occurs in a sum of product implementation when two minterms differing by only one input variable are not covered by the same product term. Figure 1-10(a) illustrates another circuit with a static 1-hazard. If $A = C = 1$, the output should remain a constant 1 when B changes from 1 to 0. However, as shown in Figure 1-10(b), if each gate has a propagation delay of 10 ns, E will go to 0 before D goes to 1, resulting in a momentary 0 (a 1-hazard appearing in the output F). As seen on the Karnaugh map, there is no loop

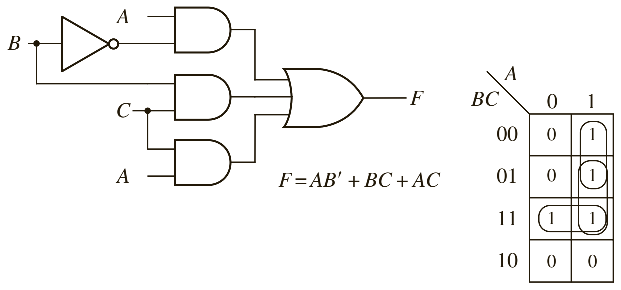
FIGURE 1-10: Elimination of 1-Hazard



(a) Circuit with 1-hazard



(b) Timing chart



(c) Circuit with hazard removed

that covers both minterm ABC and $AB'C$. So if $A = C = 1$ and B changes from 1 to 0, BC immediately becomes 0, but until an inverter delay passes, AB' does not become a 1. Both terms can momentarily go to 0, resulting in a glitch in F. If you add a loop corresponding to the term AC to the map and add the corresponding gate to the circuit (Figure 1-10(c)), this eliminates the hazard. The term AC remains 1 while B is changing, so no glitch can appear in the output. In general, nonminimal expressions are required to eliminate static hazards.

To design a circuit that is free of static and dynamic hazards, the following procedure may be used:

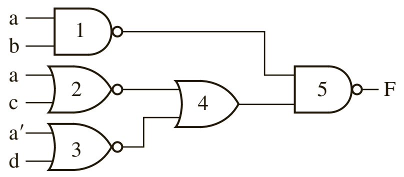
1. Find a sum of products expression (F^t) for the output in which every pair of adjacent 1s is covered by a 1-term. (The sum of all prime implicants will always satisfy this condition.) A two-level AND-OR circuit based on this F^t will be free of dynamic, 1-, and 0-hazards.
2. If a different form of circuit is desired, manipulate F^t to the desired form by simple factoring, DeMorgan's laws, and so on. **Treat each x_i and x_i' as independent variables to prevent introduction of hazards.**

Alternatively, you can start with a product-of-sums expression in which every pair of adjacent 0s is covered by a 0-term.

Given a circuit, one can identify the static hazards in it by writing an expression for the output in terms of the inputs exactly as it is implemented in the circuit and manipulating it to a sum of products form, treating x_i and x_i' as independent variables. A Karnaugh map can be constructed and all implicants corresponding to each term circled. If any pair of adjacent 1's is not covered by a single term, a static 1-hazard can occur. Similarly, a static 0-hazard can be identified by writing a product-of-sums expression for the circuit.

EXAMPLE

(a) Find all the static hazards in the following circuit. For each hazard, specify the values of the input variables and which variable is changing when the hazard occurs.



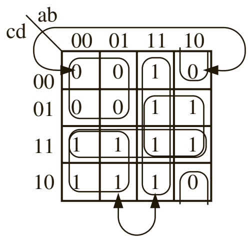
(b) Design a NAND-gate circuit that is free of static hazards to realize the same function.

Answer:

(a) Static-1 hazard: Write an expression for the output as it is implemented

$$\begin{aligned} F &= ((ab)' \cdot (a + c)' + (a' + d)')' \\ &= ab + ((a + c)' + (a' + d)')' \quad \textit{Simplify treating a and a' as independent variables} \\ &= ab + (a + c)(a' + d) \\ &= ab + aa' + ad + a'c + cd; \end{aligned}$$

Circle all these terms on the K-map; the arc shows nearby 1's that are not in the same product term, indicating a 1-hazard.



1-hazard occurs when *bcd* = 110, and *a* changes

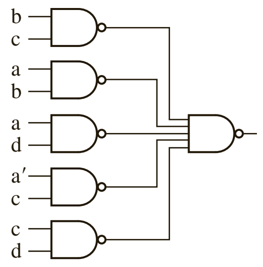
When bcd = 110 and *a* changes from 1 to 0, then the output of gate 1 changes from 0 to 1 while the output of gate 4 changes from 1 to 0. If the output of gate 1 changes before the output of gate 4 changes, then there is a short period of time where both inputs to gate 5 are 1, causing the output of gate 1 to go to 0 temporarily, thereby creating a glitch before the output of gate 4 changes to 0 and restoring the output of gate 5 back to 1. A glitch could also happen when *bcd* = 110 and *a* changes from 0 to 1, but gate 4 changes before gate 1.

Static-0 hazard:

$$F = ab + aa' + ad + a'c + cd \qquad \text{Equation derived above}$$
$$= ab + a(a' + d) + c(a' + d)$$
$$= ab + (a + c)(a' + d)$$
$$= (ab + a + c)(ab + a' + d) \qquad \text{see Table 1-1 for Boolean laws}$$
$$= (a + c)(a + a' + d)(a' + b + d) \qquad a + ab = a;\ a' + d + ab = (a' + d + a)(a' + d + b)$$

Circle all these terms of 0's in the K-map; the arc shows 0's not in same term.
0-hazard occurs when *bcd* = 000, and *a* changes

(b) We will design a 2-level sum of products circuit because a 2-level sum of products circuit has no 0-hazard as long as an input and its complement are not connected to the same AND gate. Avoid the 1-hazard by adding product term *bc*.



A logic hazard is said to exist in a logic network if some set of delays in the network could lead to a glitch. It does not mean that the network will necessarily have a glitch. A logic network can have a logic hazard, but an actual hardware implementation of the logic network may not show any glitches for its particular set of delay values. If there are two unlinked adjacent boxes in the K-map, the logic network has a static hazard that may result in a glitch for a transition in

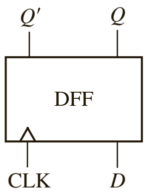
vias A via is an electrical connection between layers in an integrated circuit (IC).

either direction (irrespective of which box is the starting input vector and which box is the ending input vector). The presence or absence of a logic hazard depends only on the K-map and not on the actual delays in the final implementation. The idea is that you cannot easily predict delays in the final layout since it depends on how transistors in the gates are sized, how many vias a wire goes through, and so on. But if you design the logic network so that it is hazard-free, then it is guaranteed not to have glitches no matter what the final delays in the layout are.

1.6 Flip-Flops and Latches

Sequential circuits commonly use flip-flops as storage devices. There are several types of flip-flops, such as Delay (D) flip-flops, J-K flip-flops, Toggle (T) flip-flops, and so on. Figure 1-11 shows a clocked D flip-flop. This flip-flop can change state in response to the rising edge of the clock input. The next state of the flip-flop after the rising edge of the clock is equal to the D input before the rising edge. The ***characteristic equation*** of the flip-flop is therefore $Q^+ = D$, where Q^+ represents the next state of the Q output after the active edge of the clock and D is the input before the active edge.

FIGURE 1-11: Clocked D Flip-Flop with Rising-Edge Trigger

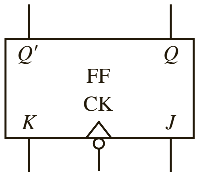


D	Q	Q+
0	0	0
0	1	0
1	0	1
1	1	1

Figure 1-12 shows a clocked J-K flip-flop and its truth table. Since there is a bubble at the clock input, all state changes occur following the falling edge of the clock input. If $J = K = 0$, no state change occurs. If $J = 1$ and $K = 0$, the flip-flop is set to 1, independent of the present state. If $J = 0$ and $K = 1$, the flip-flop is always reset to 0. If $J = K = 1$, the flip-flop changes state. The characteristic equation, derived from the truth table in Figure 1-12, using a Karnaugh map, is

$$Q^+ = JQ' + K'Q \tag{1-33}$$

FIGURE 1-12: Clocked J-K Flip-Flop



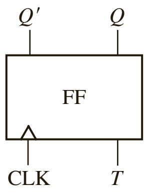
J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

A clocked T flip-flop (Figure 1-13) changes state following the active edge of the clock if $T = 1$, and no state change occurs if $T = 0$. T flip-flops are particularly useful for designing counters. The characteristic equation for the T flip-flop is

$$Q^+ = QT' + Q'T = Q \oplus T \tag{1-34}$$

A J-K flip-flop is easily converted to a T flip-flop by connecting T to both J and K. Substituting T for J and K in Equation (1-33) yields Equation (1-34).

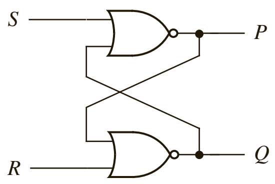
FIGURE 1-13: Clocked T Flip-Flop



T	Q	Q^+
0	0	0
0	1	1
1	0	1
1	1	0

Two NOR gates can be connected to form an unclocked S-R (set-reset) flip-flop, as shown in Figure 1-14. An unclocked flip-flop of this type is often referred to as an S-R latch. If $S = 1$ and $R = 0$, the Q output becomes 1 and $P = Q'$. If $S = 0$ and $R = 1$, Q becomes 0 and $P = Q'$. If $S = R = 0$, no change of state occurs. If $R = S = 1$, $P = Q = 0$, which is not a proper flip-flop state, since the two outputs should always be complements. If $R = S = 1$ and these inputs are simultaneously changed to 0, oscillation may occur. For this reason, S and R are not allowed to be 1 at the same time. For purposes of deriving the characteristic equation, assume that $S = R = 1$ never occurs, in which case $Q^+ = S + R'Q$. In this case, Q^+ represents the state after any input changes have propagated to the Q output.

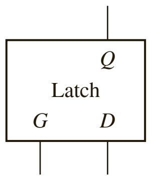
FIGURE 1-14: S-R Latch



S	R	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	–
1	1	1	–

A gated D latch (Figure 1-15), also called a transparent D latch, behaves as follows: If the gate signal $G = 1$, then the Q output follows the D input $(Q^+ = D)$. If $G = 0$, then the latch holds the previous value of $Q(Q^+ = Q)$. Essentially, the device will not respond to input changes unless G = 1; it simples "latches" the previous input right before G became 0. Some refer to the D latch as a level-sensitive D flip-flop. Essentially, if the gate input G is viewed as a clock, the latch can be considered as a device that operates when the clock level is high and does not respond to the inputs when the clock level is low. The characteristic equation for the D latch is $Q^+ = GD + G'Q$. Figure 1-16 shows an implementation of the D latch using gates. Since the Q^+ equation has a 1-hazard, an extra AND gate has been added to eliminate the hazard.

FIGURE 1-15: Transparent D Latch



G	D	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1